Page Type-Aware Full-Sequence Program Scheduling via Reinforcement Learning in High Density SSDs

Jun Li, Zhigang Cai, Balazs Gerofi, *Member, IEEE*, Yutaka Ishikawa, *Member, IEEE*, and Jianwei Liao

***Abstract*—Full-sequence program (FSP) can program multiple bits simultaneously, and thus complete a multiple-page write at one time for naturally enhancing write performance of high density 3-D solid-state drives (SSDs). This article proposes an FSP scheduling approach for the 3-D quad-level cell (QLC) SSDs, to further boost their read responsiveness. Considering each FSP operation in QLC SSDs spans `four` different types of QLC pages having dissimilar read latency, we introduce matching four pages of application data to the suited QLC pages and flush them together with the one-shot program of FSP. To this end, we employ reinforcement learning to classify the (cached) application data into `four` categories on the basis of their historical access frequency and the associating request size. Thus, the frequently read data can be mapped to the QLC pages having less access latency, meanwhile the other data can be flushed onto the slow QLC pages. Then, we can group four different categories of data pages and flush them together into a four-page unit of 3-D QLC SSDs with an FSP operation. In addition, a proactive rewrite method is also triggered for grouping the hot read data with the cached data to form an FSP unit. Through a series of emulation tests on several realistic disk traces, we show that the proposed mechanisms yields notable performance improvement on the read responsiveness.**

***Index Terms*—3-D NAND flash, full-sequence program (FSP), quad-level cell (QLC), read performance, reinforcement learning (RL), scheduling.**

I. Introduction

NAND flash memory-based solid-state drives (SSDs) have been widely employed in smartphones, laptops, and data centers [1], [2], [3]. Thanks to the cell density development and 3-D stacked technology, 3-D high density SSDs become a mainstream in the market [4], [5], [6]. For example, modern 3-D quad-level cell (QLC) can store up to 4-bits information per cell, and with 128 layers or even 176 layers [7], [8]. Although high density SSDs can greatly contribute to the reduction of per unit price, the program (write) latency significantly prolongs, due to the fact that the small-sized high density cells must store more cell states to hold more information, leading to narrow margins for distinguishing between these states.

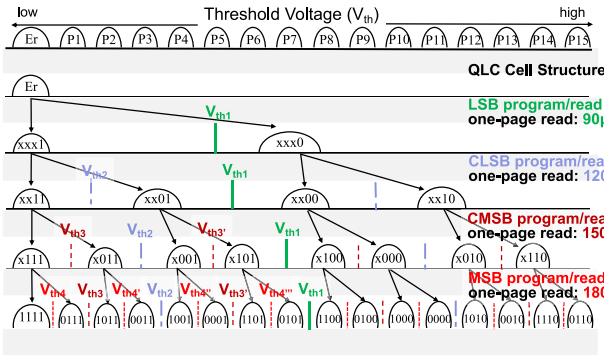

Fig. 1. Four-step programming and read procedures in QLC NAND flash [16], [46], and the read latency of different types of QLC page is referred to [11] and [36].

To accelerate the degraded program throughput in high density SSDs, the full-sequence program (FSP) mechanism is advanced for the high density flash memory [9]. It can program multiple bits simultaneously, and thus complete a multiple-page write at one time [10], [11]. As seen in Fig. 1, FSP can synchronously write `four` *data pages* to `four` *QLC pages*[1] with one program operation, so it can speed up write throughput by up to 4 × in QLC SSDs [16]. In fact, the flash memory vendors, such as Toshiba [12] and Hynix [13] have already enabled the advanced FSP functionality in their SSD products, to enhance programming efficiency.

When scheduling an FSP operation, it couples multiple data pages from the buffered data in the dynamic random access memory (DRAM) cache of SSDs, by following the cache management policies, such as first input first output (FIFO) and least recently used (LRU). However, grouping the data

Manuscript received 6 August 2024; accepted 10 August 2024. This work was supported in part by the Natural Science Foundation Project of CQ CSTC under Grant 2022NSCQ-MSX0789, and in part by the Opening Project of State Key Laboratory for Novel Software Technology under Grant KFKT2024B47. This article was presented at the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Jianwei Liao.)*

Jun Li and Zhigang Cai are with the College of Computer and Information Science, Southwest University, Chongqing 400715, China (e-mail: lijun951111@gmail.com; czg@swu.edu.cn).

Balazs Gerofi is with the Supercompute Platforms Group, Intel Corporation, Santa Clara, CA 95054 USA, and also with the High Performance Artificial Intelligence Systems Research Team, RIKEN Center for Computational Science, Kobe 650-0047, Japan (e-mail: balazs.gerofi@intel.com).

Yutaka Ishikawa is with the Information Systems Architecture Science Research Division, National Institute of Informatics, Chiyoda 101-8430, Japan (e-mail: yutaka ishikawa@nii.ac.jp).

Jianwei Liao is with the College of Computer and Information Science, Southwest University, Chongqing 400715, China, and also with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China (e-mail: liaojianwei@il.is.s.u-tokyo.ac.jp).

Digital Object Identifier 10.1109/TCAD.2024.3444718

[1] In this article, we use the term of the *data page* representing the application data that is buffered in the SSD cache with an SSD page size, and utilize the term of the *QLC* or *SSD page* indicating a basic storage cell of QLC SSDs.

pages in a sequential order and flushing them onto the flash cells with an FSP operation may degrade the read performance. Wu et al. [35] proposed logical address distance-aware FSP scheduling for efficiently utilizing internal parallelism of SSD channels, after observing that read access exhibits a sequential feature. Specifically, they utilize 5% of the internal DRAM cache as the write back cache, and then group the application data that do not have adjacent addresses into a storage unit for each FSP operation, for improving the read access parallelism.

More specifically, a read access toward an SSD page mainly consists of two parts of latency, i.e., page read time and error correction code (ECC) decoding time. Apart from the fixed decoding time, each FSP storage unit has four QLC pages on the flash arrays, including a least significant bit (LSB) page, a central LSB (CLSB) page, a central most significant bit (CMSB) page, and a most significant bit (MSB) page, with varied read access latency. As the example demonstrated in Fig. 1, the latency of the LSB page that is the fastest QLC page, is 90μs, but it takes 180μs to read the slowest QLC page (i.e., the MSB page), which is 2 $\times$ latency of reading the LSB page [36].

On the other side, real-world applications commonly have varied read frequencies on different parts of their data. Thus, we argue that the one-shot program of FSP indiscriminately groups multipage data and flushes them together onto an FSP storage unit that has four types of QLC pages, must impact the performance of the read accesses.

To our knowledge, no existing work focuses on intelligently grouping user data based on their access feature, and mapping them onto the suited SSD pages in FSP scheduling. To address this issue and further optimize the read performance in FSP-supported QLC SSD devices, we propose a new FSP scheduling approach, by using reinforcement learning (RL) to this end. In brief, this article makes the following contributions.

1) We propose a delayed rewarding-enabled RL model, to classify the best-fit type of the QLC page for a given data page, by considering factors of the historical access frequency and the associated request size. To support the feature of delayed rewarding of the RL model, we introduce a new *q-table* design, called as *across-episode q-table*. Instead of an unique *q-value*, each element of the *q-table* consists of two values, including the *active* and *shadow q-value*, used for, respectively, making decisions in the current episode and updating in the next episode.
2) We group four pages of the user data in the DRAM cache of QLC SSDs by basically following the LRU manner, on the basis of the outputs of RL. Then, it can evict the grouped four data pages from the DRAM cache, and simultaneously program them onto the suited QLC pages with an FSP process.
3) We proactively trigger the rewrite operation for relocating the hot read data that were originally mapped in slow pages. It treats such hot read data as to-be-written data and puts them into the tail of the LRU list. After that, these hot read data can be grouped with in-cache write data, and to be flushed together with an FSP operation.
4) We offer comprehensive evaluation on several disk traces of real-world applications. As measurements indicate, our proposal can increase the read accesses on the fastest QLC pages (i.e., LSB pages) by 2.5 $\times$ and thus improve read performance by 36.1% on average, compared to the state-of-the-art methods.

II. Background and Motivation

A. Background Knowledge

1) SSD Background and FSP: The basic read/write unit of SSDs is the flash page, and a large I/O request of the user application must be split into multiple page-size data segments (i.e., data pages) [14]. Because the SSD device does not allow *in-place update*, the update operation is completed by flushing the new data on another flash page, indicating the original data page will be marked as invalid. To this end, SSD devices incorporate the flash translation layer (FTL), which maintains a page-level mapping table to record the mapping relationships between the logic page addresses of the user requests and the physical page addresses on underlying flash arrays [24].

Furthermore, FTL supports garbage collection (GC) [15], which is used to reclaim the space occupied by the invalid data (i.e., outdated pages) caused by the out-place updates, when the available capacity of SSD becomes lower than a predefined threshold. Since, each SSD block affords a limited number of erases, SSD devices commonly utilize a DRAM cache for buffering the part of the frequently accessed user data, to avoid the flush operations on the SSD blocks and extend the lifetime of the SSD devices.

More importantly, QLC flash stores four bits of information in each cell, thus increasing the capacity of the SSD blocks. Specifically, each QLC block has many four-type pages, including LSB pages, CLSB pages, CMSB pages, and MSB pages, and the number of each type of pages is equivalent. The issue is that the different types of SSD pages have the same capacity but dissimilar read latency.

The FSP approach programs multiple data pages in a word line at a time, such as four data pages for the QLC SSDs, to allow the fast programming speed in high density SSDs [35], [36]. In this article, we study organizing four pages of the user data matching the types of four QLC pages, and programming them as an unit simultaneously into QLC flash with FSP.

2) Reinforcement Learning: Different from the most machine learning and deep learning approaches, the RL method is a lightweight machine learning model. It incurs low space and computational overhead, e.g., requiring a few KBs of memory and less than 0.31% of I/O processing time in the experiment that will be detailedly described in Section IV-D3, as it merely maintains and updates a *q-table* to associate the states with the actions (in the Q-learning cases). We use it as an online model without offline training in advance that the other classification models need, e.g., the existing artificial neural network (ANN) model. Because the RL model has the feature of an online model, it can learn the specific rule. However, the other offline classification model can only learn an universe one without updating on the workloads. As a result, the RL model has been successfully

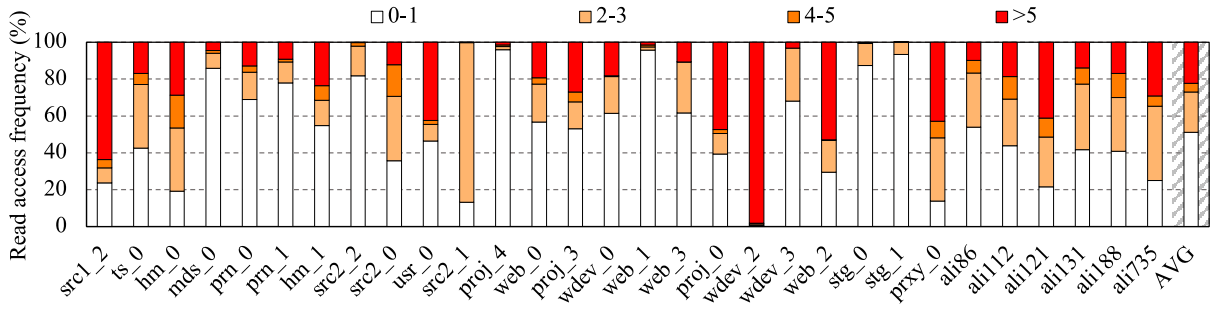

Fig. 2. Read access distribution of different levels of access frequency after running the selected block I/O traces.

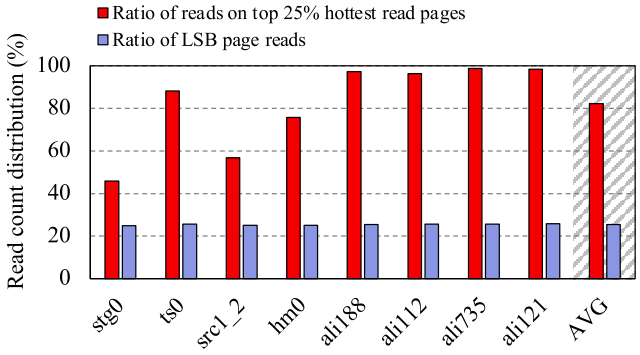

Fig. 3. Access distribution of the hottest read addresses after running the selected I/O traces. In which, the values of read bars represent the ratios of the read count on the top 25% hottest read addresses to the read count on all read addresses, whereas those of blue bars stand for the ratios of read counts for LSB pages dividing that of all pages.

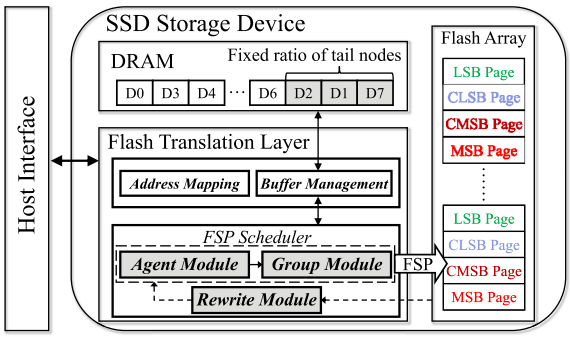

Fig. 4. High level overview of the proposed FSP scheduler implemented in QLC SSD devices.

applied to the resource-limited SSD devices, for guiding the GC scheduling [28], read refreshing scheduling [29], and other aspects [31], [32], [33], [34].

Especially, Q-learning is a widely used model-free RL algorithm, to learn the value of an action in a particular state [21]. To this end, it holds a data structure of the *q-table* and their values are referred to as the *q-values*, corresponding pairs of state and action, for directing the best-fit action according to the given state in the future.

B. Motivations

We replayed 30 block I/O traces of real-world applications that are from *Microsoft Research Cambridge* [26] and *Alibaba cloud* [27], and then recorded the read access frequency of different parts of the data. Fig. 2 shows the distribution results of varied read frequencies of the data, and the frequency setting is referred to [17]. In cases of the most block I/O traces, some pieces of the data are intensively read, while the others are not. This fact verifies that the phenomenon that different parts of application data endure varied read accesses at different phases are common.

Moreover, we collected the distribution of reads on the top 25% frequently (hot) read pages and the fastest QLC pages (i.e., LSB pages). The experiments were conducted on the 2 MB cache setting, and the other SSD-related experiment settings and the detailed specifications of selected traces will be described in Section IV-A. As seen in Fig. 3, the read access to the top 25% of the frequently read addresses accounts for a major part of the total read accesses with 82.2% on average. This ratio represents the best-case scenario, where all these data can be read from the fastest QLC pages. The ratio of LSB page reads to all the reads is, however, only around 25%. That is to say, there is room for optimization on allocating the hot read data to the suitable type of the SSD pages.

Such observations motivate us to group the data pages having different levels of read hotness into a storage unit of FSP (i.e., four-page data), corresponding to the read latency of four types of pages of QLC SSDs. After that, we can commit the unit of four data pages to four types of QLC pages with an FSP operation. Consequently, the read performance of SSDs can be significantly enhanced, as the hottest read data are preferably kept on the fastest QLC pages.

III. RL-Based FSP Scheduling

A. Overview

Fig. 4 demonstrates the high-level overview architecture of the proposed FSP scheduling scheme. First, the FTL of SSD takes charge of serving host requests. In which, our proposed method takes charge of dispatching the four cached pages as an FSP unit, when it needs to evict the data and makes room for the new data in the SSD cache. In our design, we introduce a component of the FSP scheduler running at FTL, that consists of three modules of *Agent*, *Group*, and *Rewrite Module*. Specifically, *Agent Module* makes use of RL to categorize the cached data pages of applications into four types, according to the factors of the read hotness, the data size, and the size of the relevant request that initially writes the data page. *Group Module* couples four data pages that have different categories to form a storage unit, so that it can be

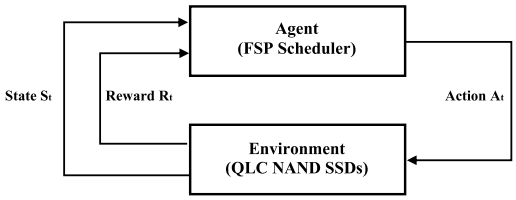

Fig. 5. Framework of online training-based RL in our proposal of FSP scheduling.

flushed onto the QLC pages with an FSP operation. Note that, *Agent* and *Group Modules* are activated when evicting the FSP units of the user data pages from the data cache, and *Rewrite Module* is activated when the hot read data are retrieved from the slow pages. Specifically, *Rewrite Module* primarily aims to proactively redistribute some hot read data to the fast pages by utilizing the functionalities of both the *Agent* and *Group Modules*.

Besides, the native functionality of cache management running at FTL is modified for supporting the newly proposed FSP scheduling approach. To be specific, a fixed ratio of tail data pages in the LRU list of the cached data pages instead of only four tail data pages, are treated as the evicted candidates, when grouping a four-page FSP unit needed by *Group Module*.

B. Agent Module: RL-Based Classification

The basic idea is to utilize the RL model categorizing data pages and then recommending four of them to be flushed together with an FSP process, for better matching four QLC pages that have different read latency, in an FSP storage unit.

Fig. 5 illustrates the framework of RL, and it shows the interaction between the *Agent* (the FSP scheduler in our context) and *Environment* (QLC NAND SSDs in our context). As seen, *Agent* maintains a data structure to make the best-fit decisions (e.g., *Action* A_t). Then, it needs iterating and updating the corresponding values of the data structure according to the future reward (i.e., Reward R_t), in the learning process. Note that, in this work, the feature of *online training* in RL makes it possible to approximate the optimal policies with not much overhead. This section mainly describes our RL implementation, including action, state, reward, and other details.

Actions: The available selections for the *Agent* in the RL model. The purpose of our RL-based model is to determine the type of data pages, corresponding to the four types of QLC pages. Consequently, *Actions* are set consisting of LSB, CLSB, CMSB, and MSB.

States: The observations related to action decisions from the *Environment* in the RL model. Considering the basic granularity of I/O scheduling inside SSDs is the data page, we define states in the RL model associating with the given data page. In the context of our RL-based model, the states are the current situations of the cached data pages, which are the candidates of FSP scheduling, to be classified into four categories. Besides, the historical access information of the data pages is crucial for accurate classification. Thus, we consider the following three aspects when defining the states.

1) The short-term and long-term history information of access frequency on the data page should be considered first. Specifically, the long-term access frequency is categorized into four states, as analysed in the observation of Section II-B, namely, `0 and 1`, `2 and 3`, `4 and 5`, and `>5`. Meanwhile, the short-term access frequency, is determined by whether or not the data page has been accessed. In our design, the long-term information refers to the accumulated read count of histories, while the short-term one is based on the page read in the recent episode. In brief, 2 bits plus 1 bit are, respectively, used to record the long-term and short-term information.
2) Apart from read requests that need accessing the data saved in QLC pages, the small update (write) requests expect fetching the original data from the SSD pages as well, and such update operations are called as read-modify-writes (RMWs) [22]. Thus, the size information is also referred to as a factor when defining the states. Considering the small size of the I/O request and page-level application data may require the update operation with RMW, we define a small I/O request if the size is not larger than one page; otherwise, the request is defined as a large one.
3) The type of QLC page that holds the data associating with the previous write operation on the same logical page address can be divided into four states (i.e., four types of QLC pages; to record this information, we use 2 bits to represent the LSB, CLSB, CMSB, and MSB page, by referring to the previous action of classifying the type of the data page. For example, data with the page address `368` was ever flushed into the MSB page before it re-entered the DRAM cache. We record this information in the page address `368` as `11`. Note that, if the pages of data are the new data that first appear, we set this information as the LSB page (i.e., `00`) in default.

In summary, we define `four` groups of the long-term access frequency, `two` types of the short-term access frequency, `two` kinds of the size of I/O requests, `two` categories of the size of page-level data chunks, and `four` page types of previous write request on the same logical page address. That is, we have $4 \times 2 \times 2 \times 2 \times 4 = 128$ states in total in our RL-based model.

Rewards: The feedback from the *Environment* after specific actions have been completed regarding the given states. In the context of FSP scheduling, the reward cannot be well defined as instant feedback, since the read requests on the current data page may not immediately occur. Thus, we propose a method of delaying reward updates during the training process. To this end, an *across-episode q-table* (will be described later) is designed to effectively support this functionality. First, it randomly explores the rule of the state-action in the first episode, and each episode is set as `1000` steps/actions in this article, by also referring to [28] and [29]. Next, it observes the data pages committed to the QLC pages in the previous episode. After that, the feedback can be given for updating the RL policy with relative rewards in the following episode.

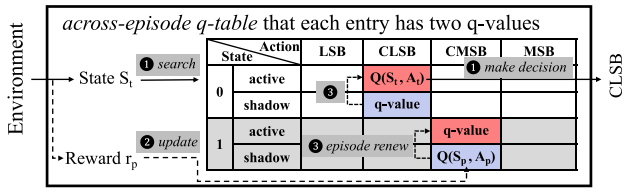

Fig. 6. $Q-table$ design that supports the feature of delayed rewarding in our scenario of the RL model. Each element of the state-action pair consists of two $q-values$ (the active and shadow $q-value$) for separately supporting decision making and periodically $q-value$ renewing. There are three kinds of operations on our $q-table$: ❶*search* the *active q-value* based on the current state and then *make a decision* on the page type for the data page; ❷*update* the *shadow q-value* for the state-action pair, according to the delayed reward; and ❸*renew* all *active q-values* by copying relevant *shadow q-values*, after starting a *new episode*. Note that, we only present two states in the figure for illustration simplicity.

Specifically, we use the Q-learning implementation in our RL model, and the q function is the rule of updating *q-value*, as defined in

$$Q'(S_t, A_t) = (1-\alpha)Q(S_t, A_t) + \alpha\big[r + \gamma Q(S_{t+1}, A_{t+1})\big] \quad (1)$$

where $Q(S_t, A_t)$ and $Q(S_{t+1}, A_{t+1})$ are, respectively, the value in *q-table* when the action A_t/A_{t+1} is taken at the state S_t/S_{t+1} and time $t/t+1$. Specifically, in the left-hand side of (1), $Q'(S_t, A_t)$ represents the new *q-value* after updating. In addition, the parameters of α and γ are the step size and the discount factor, which are set as the typical values, i.e., `0.3` and `0.8` [20], [28]. The parameter of r means the reward, and the reward function is correspondingly given the feedback of read accesses in the next episode. According to the parameters above in (1), the new $Q'(S_t, A_t)$ can be updated, by referring to the old *q-value* $Q(S_t, A_t)$, and the rewards that include the reward r and the policy decision $Q(S_{t+1}, A_{t+1})$ in next time point $t+1$. The updated time point is the next time period after the decision occurs (will be detailedly described in Fig. 6).

On the one hand, the reward of our design of the RL model is defined, on the basis of the access information of different types of the data pages. Specifically, ± 1, ± 0.5, ∓ 0.5, and ∓ 1 are, respectively, used to reflect the LSB, CLSB, CMSB, and MSB page is accessed or not in the next episode. On the other hand, the factor of I/O response time is also considered when deciding the reward [28], [29]. While the response time of the current read request (normalized to the unit of `1` KB) is lower than the `70`th, `90`th, and `99`th percentiles of the normalized completion time of all the historical read requests, the reward r will be, respectively, set as `1`, `0.5`, and `0`, for giving positive feedback to the RL policy. Otherwise, it will be given a negative value of `-1` [29]. In the end, we sum these two parts as the final reward, for iteratively updating *q-table*.

To support the functionality of delaying reward update in our RL-based model, we propose a new design of *across-episode q-table*, as shown in Fig. 6. In which, each entry (i.e., a state-action pair) keeps two values of the *active q-value* and the *shadow q-value*. The *active q-value* is used to determine the action based on the current state S_t in the current episode, and the *shadow q-value* is used to reflect the real-time value of the state-action pair with delayed rewarding.

Algorithm 1: RL-Based Decision in FSP Scheduling

Input: States for four pages, State-action pairs in the previous episode
Output: A_t

```
1  Initialize q table as an empty set;
2  Function q_table(S)
3      Random ζ between 0 to 99;
4      if (ζ < ε) then
5          Return random action;
6      else
7          Return argmax(Q(S));
8  if episode < 4 then
9      ε = 80;
10 else
11     ε = 1;
12 A_t = q_table(S_t);
13 /*Update q-table from 2nd episode*/
14 if episode > 0 then
15     r = reward(lpn.p);
16     Q' = r+ γ Q(S_{p+1},A_{p+1});
17     Q'(S_p,A_p) = (1 − α) Q(S_p,A_p)+α Q';
18 if new episode then
19     Copy active q_values to shadow q_values;
```

As seen, in the step of ❶, it first searches the *q-table* with the state of S_t, and all *active q-values* associating with *State 0* (i.e., *Row #0* in the figure) should be retrieved. Assuming that, $Q(S_t, A_t)$ is the maximum value associated with *State 0*, the corresponding action of CLSB is decided. In the step of ❷, the *q-table* refines the *shadow q-values* with delayed rewarding. Note that, p and t represent the same offset of step in the previous episode and the current episode, and when the step in S_t is completed, the reward in the previous S_p is then used to update the $Q(S_p, A_p)$ based on (1). When a new episode starts, it copies the corresponding *shadow q-values* to the *active q-values* that is called episode renewing, to avoid the interference of determining actions and updating rewards, as illustrated in the step of ❸.

For clearly illustrating the workflow of the proposed RL-based classification model, Algorithm 1 shows the implementation details. First, the proposed *across-episode q-table* is initialized as all zeros, shown in line `1`. Lines `2`–`7` show the *q-table* decision policy. Specifically, the basic principle is determining the corresponding action having the maximum *q-value* in *q-table* under the specific state, when the random value ζ is lower than ϵ; otherwise, it aggressively determines the actions by the random strategy. In order to construct a stable *q-table*, we randomly select the actions by utilizing ϵ-greedy initialization in the initial period, and preserving the chance to test the other actions during the remainder of the periods when the decision policy is stable. That is, the first 4×1000 actions are randomly selected with a large ϵ (`80%`) and a small ϵ (`1%`) during the remainder of the period [20]

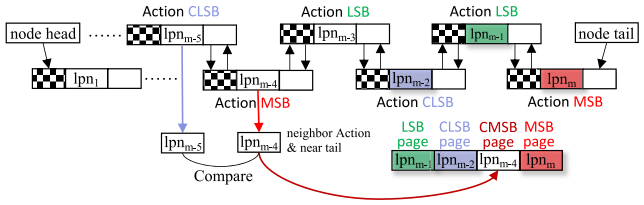

Fig. 7. Illustration on grouping the four page types of data in the case of CMSB does not have the candidate data page.

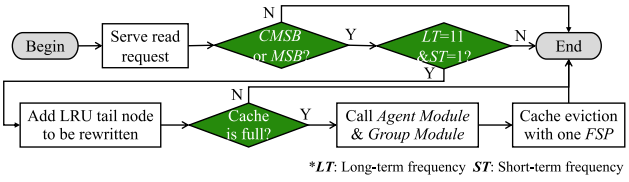

Fig. 8. Workflow of proactive hot read data with *Rewrite Module*.

in our experiments when running the benchmarks, shown in lines 8–11. Then, line 12 is to show the main operation by calling the Function *q_table*(*S*) to determine the action. Lines 14–17 present the process of updating *q-table* based on the defined reward, which has been described in the previous discussions. When a new episode starts, the *active q_values* should duplicate to the *shadow q_values*, as seen in lines 18 and 19.

C. Group Module: FSP Unit Management

This section depicts coupling four pages of cached data to form a storage unit of FSP, on the basis of their best-fit page type, output by the RL-based classification model. We adopt an intuitive solution analysing a fixed number of the cached data pages from the tail of the LRU list, and then select four data pages having varied categories (i.e., *Actions* offered by the RL-based classification model) to build an FSP storage unit, corresponding to the four-type QLC pages.

We emphasize that it needs matching four data pages with four-type QLC pages from the candidate cached data pages, and it also allows selecting the other types of data pages and replacing the missed type of data pages in the candidates, when constructing an FSP storage unit. For example, in the case that the CMSB page does not have a recommended data page, according to the outputs of the RL-based classification model, the data page has the neighboring type (i.e., MSB or CLSB) can be alternatively used, as a part of the FSP storage unit.

Specifically, the RL-based classification model checks the fixed ratio of the cached data pages from the tail node, and temporally records the determined actions (i.e., the best-fit type of the QLC page for the data pages). Fig. 7 shows the policy of grouping an FSP storage unit from the cached data pages, on the basis of their actions output by the RL-based model. As seen, we do not have the data page whose determined action is CMSB. Then, we need selecting a data page that has a neighboring action, i.e., the cached data page has either the MSB or the CLSB action. Specifically, our selection rule is to use the data page that has a neighboring action, and is closer to the tail of the cached page list. Finally, we evict four data pages from the cache and flush them together with an FSP operation, when we need cache space for the new data.

D. Rewrite Module: Proactive Hot Data Redistribution

It is inevitable that some frequently read accessed data pages are not updated by host I/O requests, while these data are stored in slow pages (i.e., CMSB and MSB pages). This section introduces a *Rewrite Module*, for proactively placing these data into fast pages. When the data page types of the current read request is slow pages, it checks the history information of access frequency that has been described in Section III-B. If the long-term and short-term frequency are, respectively, the highest value (i.e., 11 and 1), it awakes the functionality of the proactive *Rewrite Module*.

Fig. 8 shows the workflow of *Rewrite Module* in our FSP scheduler. As seen, it triggers rewriting a hot read data page that was originally stored in a slow flash page, and temporarily loads into the cache as the dirty data. Meanwhile, the data page will be treated as a new tail node in the LRU list. Since, the goal of *Rewrite Module* is to relocate the hot read data to appropriate pages, it calls *Agent* and *Group Module* to group cached data into a to-be-written FSP unit. Note that, the hot read data should be temporarily buffered as a to-be-written dirty data if the SSD cache is not full. They hold their location in the LRU list, avoiding impacting the cache utilization of the write data.

TABLE I
CHARACTERISTICS OF THE EVALUATED SSDS

Parameters	Values	Parameters	Values
Channel size	8	*Program latency*	1.3ms
Chip size	2	*Erase latency*	10ms
Plane size	2	*LSB read latency*	0.09ms
Block per plane	4096	*CLSB read latency*	0.12ms
Page per block	1024	*CMSB read latency*	0.15ms
Page size	8KB	*MSB read latency*	0.18ms
FTL scheme	Page	*Erase limit*	500
Cache size	2/8/32MB	*GC Threshold*	5%

IV. EXPERIMENTS AND EVALUATION

A. Experimental Setup

We have performed trace-driven simulation with *SSDsim* [23] to model the 3-D QLC NAND flash-based SSDs. The *SSDsim* simulator has been utilized for several SSD-related optimization researches, such as [32], [35], [36], and [47]. The evaluated SSD parameters are based on [36] with the detailed specifications described in Table I. We have extended *SSDsim* for supporting the FSP scheme, referred to [36]. Dynamic page allocation [35], greedy GC [24], and static wear-leveling [25] are employed by default.

SSDs commonly use the DRAM data cache to absorb hot write data for two reasons as follows.

1) The write latency of the flash memory is around 10 × longer than the read latency, indicating a write hit can lead to more I/O reduction.
2) Absorbing more write requests in the data cache can reduce the number of flushing operations to the flash array, thus extending the lifetime of SSDs.

TABLE II
SPECIFICATIONS ON SELECTED DISK TRACES
(ORDERED BY READ RATIO)

Trace	# of Req.	Read ratio	Int. footprint	Int. ratio
*stg*0	2,030,915	15.2%	4567.8MB	72.3%
*ts*0	1,801,734	17.6%	52.8MB	9.3%
*src*1_2	1,907,773	25.4%	836.8MB	51.3%
*hm*0	3,993,316	35.5%	650.0MB	31.3%
*ali*188	6,198,540	46.6%	29.0MB	2.4%
*ali*112	3,116,126	63.0%	61.6MB	4.9%
*ali*735	963,179	70.1%	8.1MB	5.1%
*ali*121	18,565,868	78.2%	77.4MB	4.5%

Then, we use the DRAM cache as a write cache by default [37], [38], [40]. The varied sizes of DRAM cache range from 2 to 32 MB [39], [42], [48], to evaluate the efficiency of the proposed method under different scales of the SSD cache. In addition, a fixed number of data nodes in the cache list tail is set as 5% of the total number of nodes, referred to [35].

We employed eight commonly used disk traces. Specifically, the first four traces are from the block I/O trace collection of *Microsoft Research Cambridge* [26], which have been used as benchmarks in several recent SSD optimized methods [32], [35], [36], [47]. The remaining four recent traces are obtained from *Alibaba cloud* [27], representing the first 72 data traces. Table II shows the detailed specifications of the block I/O traces. Specially, the metrics of *Int. footprint* and *Int. ratio* mean the footprint of the hottest read addresses that endure 80% of all the read accesses and the ratio of the corresponding footprint to the total footprint, for reflecting the read *locality* of the benchmarks.

In addition, we selected the following comparison counterparts in our evaluation tests as follows.

1) *Baseline* adopts the default scheduling based on the LRU replacement policy in the cache management [19]. It evicts the final four data pages at the tail of the LRU list from the cache and flushes them together with an FSP operation.
   Besides, we introduce another baseline that buffers both the read and write data in the cache, labeled as *Baseline-RW*.
2) Access frequency-based scheduling, (labeled as *Frequency*) is based on the work proposed by Lv et al. [44]. It employs the intuitive factor of access frequency as the basis of the page type classification, though it is designed for the traditional four-step programming.
   For evaluating *Frequency* in the context of FSP, it groups FSP units with LRU tail nodes, by directly considering the history information of read counts to match the suited QLC page for a specific data page. In addition, the rewrite procedure also groups the to-be-written hot read data with the other three data nodes in the LRU tail list, as an FSP unit, by following an immediate refresh fashion.
3) Distance-based scheduling (labeled as *Distance*) proposed by Wu et al. [35], groups inconsistent access addresses into a storage unit of FSP, to maximize the internal parallelism of QLC SSD. In our implementation, the distance threshold is set as half of the page number of the total write back cache, by following the design principle of *Distance* [35].
4) The RL-based FSP scheduling (labeled as RL) is our proposed method. It utilizes the RL model classifying the type of data pages, and groups four of them that have different types into a storage unit of FSP. For further redistributing the location of hot read data that have been stored in slow flash pages, it proactively groups such hot read data with the cached data as an FSP unit.

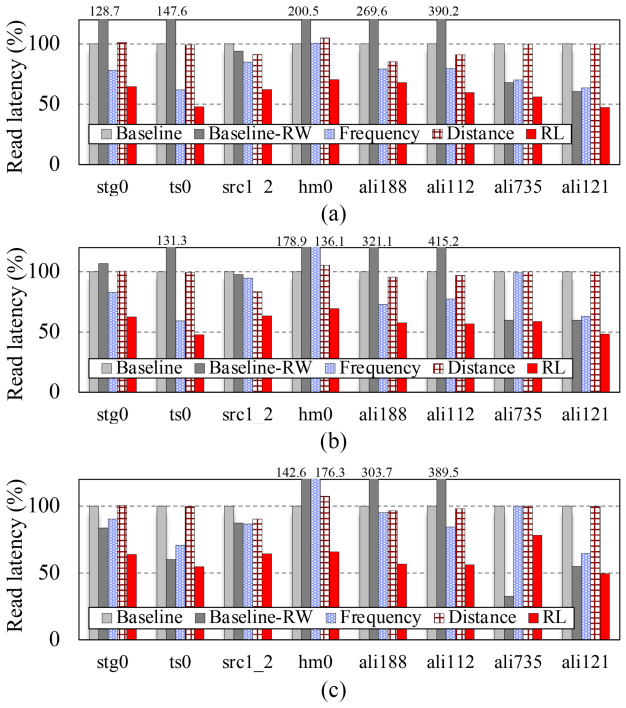

Fig. 9. Read performance comparison after running the selected block I/O traces. (a) Cache size: 2 MB. (b) Cache size: 8 MB. (c) Cache size: 32 MB.

B. Tests and Benefit Illustration

To measure the validity of the proposed mechanism that aims to boost read access performance by considering the dissimilarity of page types, we use the following metrics in our tests: 1) *I/O latency*, and 2) *read ratio of the fastest QLC pages*.

1) I/O Performance: Reducing the read latency is the primary target of the proposed scheme. Figs. 9 and 10 show the comparison of the normalized read and total I/O latency after running the selected block traces, while the cache sizes are set as 2, 8, and 32 MB.

As seen, our proposed RL approach outperforms the comparison counterparts on the measurements of read response time and total I/O response time. This is because, it can make use of the RL-based policy to find suited types of the page data in a storage unit of FSP. In other words, the RL-based model can learn the rule of grouping what four data pages to corresponding QLC pages, by considering both the read frequency and the size of the data pages and

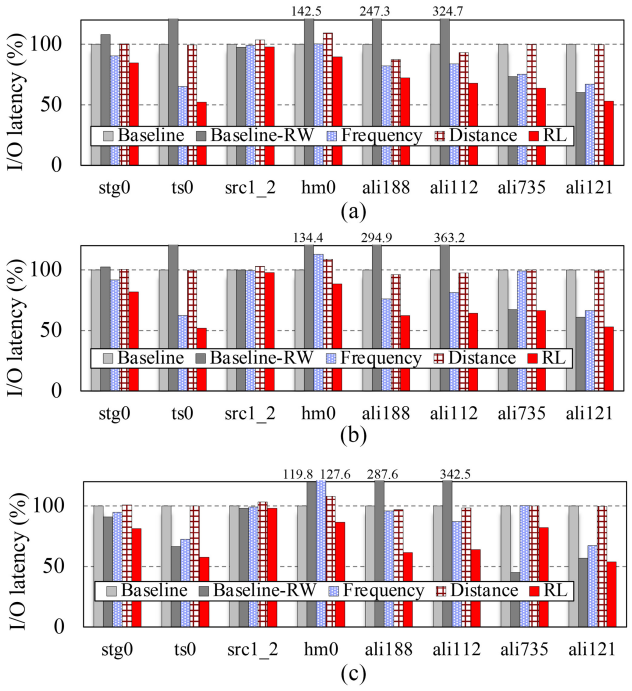

Fig. 10. Comparison of the overall I/O performance after running the selected block I/O traces. (a) Cache size: 2 MB. (b) Cache size: 8 MB. (c) Cache size: 32 MB.

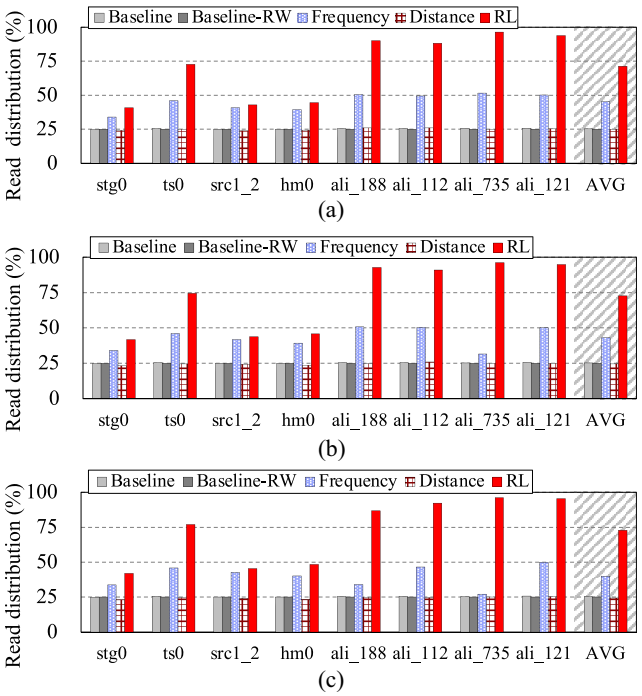

Fig. 11. Comparison of read accesses on LSB pages after running the selected block I/O traces. (a) Cache size: 2 MB. (b) Cache size: 8 MB. (c) Cache size: 32 MB.

relevant requests. Specifically, RL decreases the read latency by 40.4%, 28.3%, and 38.8%, in contrast to *Baseline*, *Frequency*, and *Distance*, respectively. Consequently, RL cuts down the total I/O latency by 27.8%, 17.4%, and 28.2%, when comparing with three counterparts.

Baseline-RW performs the worst in the most workloads, and this is because *Baseline-RW* buffers hot read data in the cache, which must affect the write performance, since it cannot use all the cache space for efficiently absorbing the write data. When running two read-intensive traces of *ali735* and *ali121*, however, we see *Baseline-RW* does yield attractive I/O performance. For example, in the case of *ali735* with the configuration of 32 MB cache size, *Baseline-RW* perform the best. This is because the trace of *ali735* has very intensive read accesses on a limited portion of address space (i.e., the size of its *Int. footprint* is only 8.1 MB), indicating the major part of the hot read data can be buffered in the data cache. We emphasize that *Baseline-RW* brings about more flush operations on the flash array and more erase operations, impacting the lifetime of SSDs, which will be presented in Section IV-D2.

Another clue is that, the *Frequency* method improves the read performance by 13.6% on average across three cache configurations in contrast to *Baseline*, since it only considers the factor of historical access frequency information, thus inaccurately directing FSP scheduling. In addition, *Distance* can only slightly improve the read performance by 1.9% on average, compared to the *Baseline*. This is due to the fact that, it does not take the access feature into consideration, which offsets the performance gains from the access parallelism.

2) Read Distribution on Fast Pages: We further analyse the read access count on the fastest QLC pages after running the selected block I/O traces, which matters the overall I/O latency. Fig. 11 presents the comparison results with four selected FSP scheduling methods. As seen, *Frequency* and RL can increase the read count from the fastest QLC pages (i.e., LSB pages), compared to the *Baseline* scheme. Specifically, *Frequency* and RL increase the read counts from the LSB pages by 1.7 and 2.9 × on average, respectively. As a result, RL can yield the best I/O performance improvements after running the selected block traces. On the contrary, *Distance* cannot better utilize the LSB pages to response the read requests. Consequently, it limits the read performance improvement that has been analysed in Section IV-B1.

The interesting observation is, the tendency of the increase in the ratio of LSB page reads is similar to that of the reduction in I/O latency. Specifically, in some cases of write-intensive workloads, e.g., *stg0*, *src1_2*, and *hm0*, RL in contrast to the *Baseline*, can increase the LSB page read count by 75.8% on average, but the increase is limited to 14.9%, compared to *Frequency*. Since, the rewrite module is enabled based on the recent read frequency information, there is a low probability of triggering rewrite operations during the specific episode (window) of such write-intensive workloads. Nevertheless, the good point is that, in these three traces, RL achieves 34.8%, 33.2%, and 33.4% of reductions in read latency, as well as 10.4%, 11.1%, and 14.0% of decreases in I/O latency, compared to *Baseline*, *Frequency*, and *Distance*, respectively.

Another interesting clue is, our proposed method RL can yield the read ratio of 73.1% from LSB pages on average under the three cache settings. Take the 2M setting as an illustration. RL can achieve 71.2% of the read ratio after running the evaluated workloads. This performance is approaching the ratio of 82.2% in the best-case scenario, where all the data

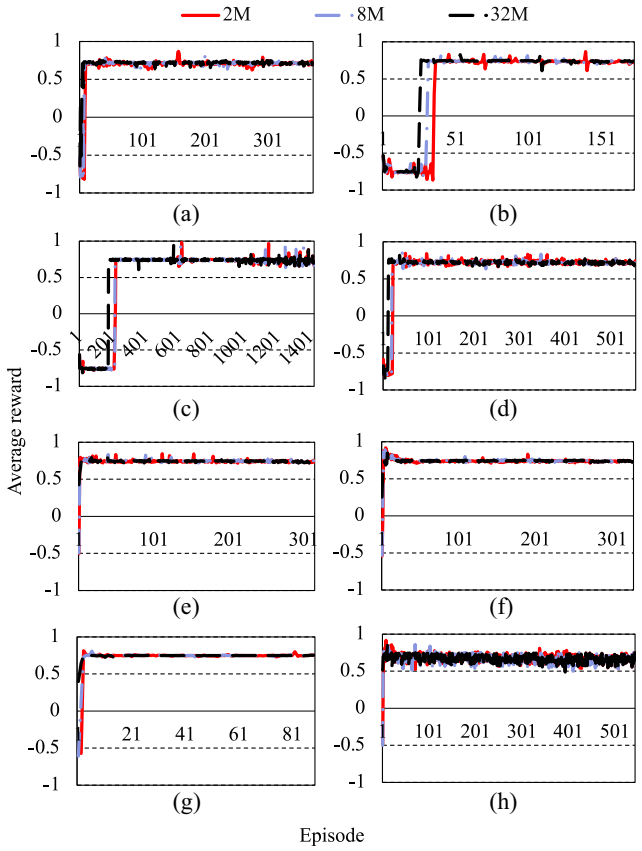

Fig. 12. Reward analysis on RL after running the selected block I/O traces. Note that, the unit of the *X*-axis is an episode, i.e., 1000 steps. (a) trace: stg0. (b) trace: ts0. (c) trace: src1_2. (d) trace: hm0. (e) trace: ali188. (f) trace: ali112. (g) trace: 735. (h) trace: 121.

from the top 25% most frequently read addresses are allocated on the LSB pages, as observed in Section II-B. This fact illustrates that the proposed method RL can effectively utilize the fastest QLC pages matched with the frequently read data. This is the root cause of the performance benefits from our proposed RL.

C. Analysis on Reinforcement Learning

This section focuses on the learning process of RL in settings of three cache sizes. Specifically, the average reward in each episode is an indicator of the training performance, and the general goal of the *Agent* is to maximize its total cumulative reward.

Fig. 12 shows the average reward in each episode, which is also a metric demonstrating convergence. As seen, it yields a low positive or negative average reward and reveals apparent fluctuations in the first few episodes, but the average reward gradually yields an attractive convergence tendency in the following episodes. For example, in the first 30 episodes of running *ts0*, RL aggressively starts to exploit the decisions and provides further feedback to fix the policy of *q-table* used by *FSP scheduler*. After that, the average reward remains stable. This verifies the policy is trained as a convergence rule, to determine the suitable action for each specific state, in our RL scenario.

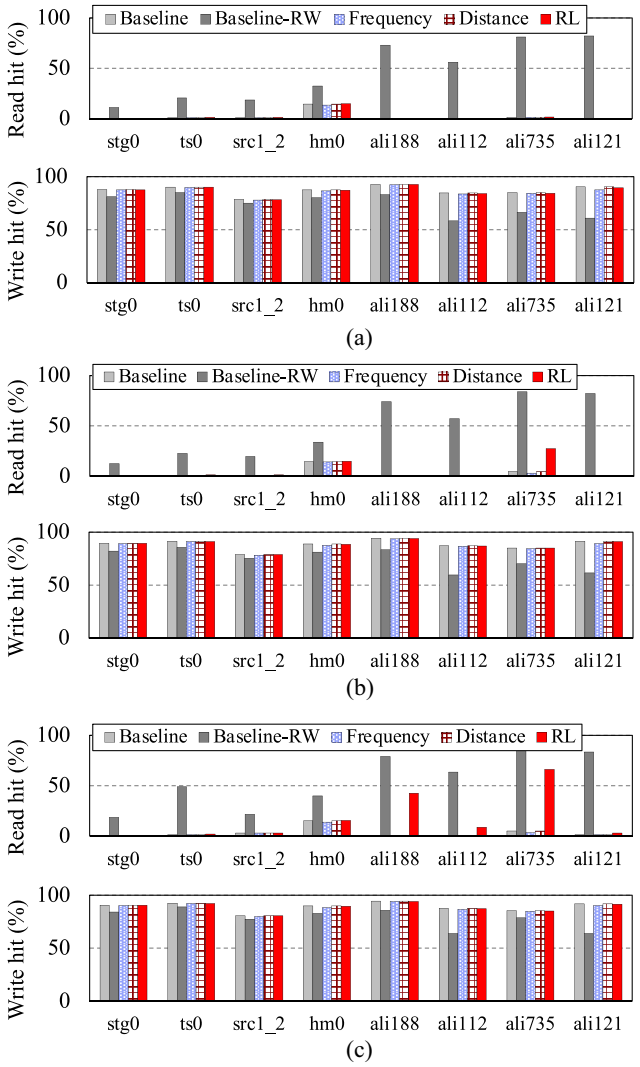

Fig. 13. Comparison of the read/write cache hit ratio after running the selected block I/O traces. (a) Buffer size: 2 MB. (b) Buffer size: 8 MB. (c) Buffer size: 32 MB.

D. Overhead Analysis

This section depicts overhead analysis on cache use efficiency, erase count, as well as time and space consumption.

1) Cache Use Efficiency: The DRAM data cache inside SSDs is used to minimize the I/O latency by absorbing the read/write requests on the frequently accessed data, on the basis of the principle of data locality. To evaluate whether the proposed method affects cache use efficiency or not, we record the results of cache hit ratios, with configurations of three cache sizes after running all the evaluated benchmarks. Fig. 13 presents the relevant results.

Obviously, *Baseline-RW* can greatly improve the read hit ratio, but yields the worst write hit ratio. This fact verifies the read/write data cache can contribute to more read hits, but it impacts the number of write hits. Note that, the write latency of the flash memory is around 10 × more than the read latency, and *Baseline-RW* shows the worst I/O performance when running the most of selected benchmarks, as the previously described in Section IV-B1.

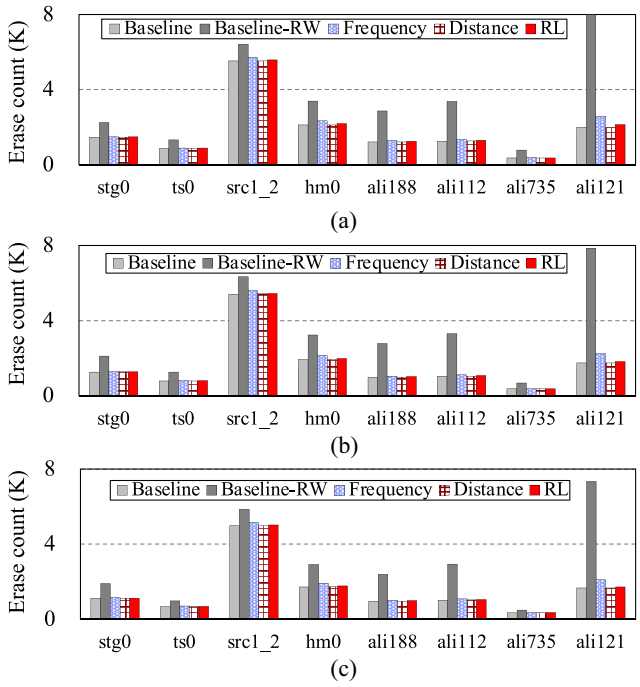

Fig. 14. Number of erase operations induced by GC after running the selected block I/O trace. (a) Buffer size: 2 MB. (b) Buffer size: 8 MB. (c) Buffer size: 32 MB.

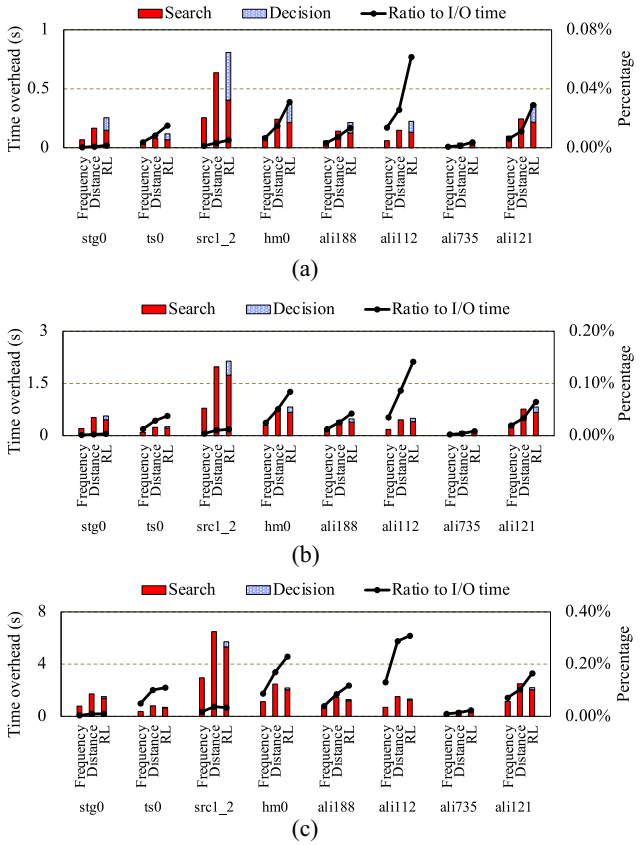

Fig. 15. Time overhead after running the selected traces with advanced FSP scheduling methods of *Frequency*, *Distance*, and RL. (a) Cache size: 2 MB. (b) Cache size: 8 MB. (c) Cache size: 32 MB.

The most important thing is that, our RL method shows a similar level of cache use efficiency as the *Baseline* method. That is to say, our method does not pose a noticeable impact on the metric of cache use efficiency. Note that, the read performance gain of the proposed approach primarily benefits from the effective alignment with the four types of QLC pages, that is, more frequently read data can be served by LSB pages that exhibit the lowest read latency.

2) Erase and Lifetime Statistics: We use the metric of erase count induced by GC to reflect the endurance of the SSD. Fig. 14 reports the result of erase numbers after running the selected traces, when implementing different FSP scheduling methods under varied cache settings.

As discussed, *Baseline-RW* yields the least number of write hits in the cache, resulting in an average increase of 2.2 × in terms of the erase count. In addition, Fig. 14 shows that *Frequency* brings about more erase operations by 7.6% on average, compared to *Baseline*. On the one hand, the *Frequency* mechanism aggressively triggers rewrite operations for the hot read data, since it only holds the recent frequency information whose trigger condition is less stringent than that of our proposed mechanism. On the other hand, it fails to smartly guide the matching between the cached data and QLC page types, thereby further increasing the number of subsequently triggered rewrite operations.

RL slightly increases the erase number by 2.5% on average, compared to the *Baseline*. This is because it only groups candidates of the evicted data pages from a limited number of tail pages in the cache, which negligibly impacts cache use efficiency, as well as the measure of erase count. In addition, it can successfully direct FSP scheduling to the appropriate pages, which also limits the erase number induced by the subsequent rewrites.

To further explore the impacts of the proposed method on the SSD lifetime, we collect the total amount of the processed write data before the appearance of 5% wear-out blocks inside SSDs [49], [50]. Specifically, we run the selected workloads in a repeated manner until the number of wear-out blocks approaches 5% of all the blocks, and then record the amount of write data. Our proposed RL scheme slightly reduces the lifetime by 0.7% on average, compared to the *Baseline*. We suggest that the static wear-leveling [25] is employed to balance the erase distribution inside SSDs, for boosting the SSD's lifetime.

3) Time and Space Overhead: Three optimized FSP scheduling approaches need to traverse a fixed 5% of cache data pages (i.e., the candidates of data pages), so that they consume computation time in the process of cache eviction. Considering SSD controllers usually have limited computation power and memory capacity, we use a local ARM-based machine equipped with an ARM Cortex A7 Dual-Core with 800 MHz. Fig. 15 shows the comparison results of extra time cost, compared to the *Baseline*.

The measurements demonstrate that our proposed method only takes between 1.6E-05% and 0.31% of the total I/O processing time. Obviously, all the three optimized FSP scheduling methods do need more time consumption in the search process, while the number of evicted data page candidates increases with the DRAM cache size varying from 2 to 32 MB. Note that, the time overhead is collected in a resource-limited ARM platform that was described in

Section IV-A. The experimental results in Section IV-B have considered the impact of the time overhead on the I/O latency.

It is worth mentioning that our proposed RL approach incurs an extra part of the time overhead caused by decision making and policy updating in the RL-based model. However, this part of the overhead does not increase as the cache capacity becomes larger. This is because the update process is independent of the cache size, only depending on the size of the *q-table*. We suggest that our RL-based FSP scheduling scheme does not result in noticeable time overhead, which is consistent with the other RL-based optimization methods in SSDs [28], [29], [30], [31].

Our proposed RL method needs to hold the history read frequency information, i.e., 1 bit for short-term and 2 bits for long-term access frequency on the data page. It needs 48 MB for simulated 1 TB SSD, while the *Frequency* requires 16 MB. Besides, different from the other FSP scheduling methods that do not need extra data structures, our RL method results in the space overhead. To be specific, RL holds a *q-table* with two types of the *q-values*, to direct FSP scheduling and update the *q-values* based on the rewards from the system environment. They consume 4 KB (= 128 (states) * 4 (actions) * 4B (one entry needed for *q-value*) * 2 (table number)). In addition, the logical addresses of the data pages that are evicted in the previous episode should always be recorded for getting the reward and the read frequency information. Except for the logical addresses, it also requires 1 bit to represent the data is a small I/O request or not, and 2 bits to record the previously written page type. Therefore, it totally requires 2.32 KB (= 1000 (steps in an episode) * (2B (logical address number) + 3bits)). In summary, RL takes acceptable memory space in SSDs for directing the FSP optimization.

V. Related Work

A. Reinforcement Learning-Based Optimization

RL is a lightweight method which can make decisions for the system scheduling without heavy overheads. For the internal GC scheduling in NAND-based SSDs, Kang et al. [28] proposed an RL-assisted method for GC to reduce the long-tail latency and ensure the quality of services. Similarly, Li et al. [29] proposed an RL-based scheduling on read refresh operations, caused by read disturb errors, for mitigating the negative impacts of normal I/O requests. Fan et al. [30] proposed a Q-learning-based backup strategy to efficiently execute the program by utilizing the residual energy. Pan et al. [31] utilized RL for the cache cleaning on DM-SMR, and thus mitigate long-tail latency. These scheduling methods focus on the topic of quality of services, and do internal operations at the suitable time periods based on the RL decisions. In addition, Wu et al. [32] proposed an RL-based I/O merging for SSDs. It considers the operations and size of queued I/O requests as the states of RL, and thus fine merging methods can improve the system throughput. Different from these RL-based mechanisms, our proposed method faces the topic of varied read latency of high density SSD pages. Considering the factors of access frequency and size, data pages can be grouped and simultaneously written through FSP into specific QLC pages, for yielding better read performance.

B. FSP Technology-Based Proposals

For the purpose of adapting to the software layer management of FSP technology, Wu et al. [35] proposed an FSP-aware data allocation policy for improving the read performance. It considers grouping the data whose access addresses are not adjacent. Then, the internal parallelism can be exploited to yield better read latency. Liu et al. [36] proposed a page allocation scheme in the SSD firmware, which supports the smaller program granularity of the FSP operation, to employ the irregular number of written data pages. It can eliminate the space fragmentation and thus improve the page utilization in the SSD storage system. However, while the FSP operations make the program latency about four page types uniform, the read latency of them has significantly varied values. Our proposal of FSP scheduling considers the dissimilarity of page types, to classify the best fit of the cached data pages, and flush them into specific QLC pages, thus improving the read access performance.

C. Read-Optimized Scheduling Proposals

For the purpose of speedup the read performance of high density SSDs, Chang et al. [43] proposed a read performance improvement method for the TLC flash memory, by utilizing a list to record the read count, and a bitmap to keep track of recently unread fast pages. Lv et al. [44] introduced a read data hotness identification via an LRU list, and move them to the page having corresponding read latency. However, these methods target the traditional four-step programming routine, and we propose an RL-based scheduling method under the default FSP-enabled 3-D QLC NAND flash-based SSDs.

VI. Conclusion and Future Work

This article proposes a page type-aware FSP scheduling, through RL in 3-D QLC SSDs. Our goal is to map the frequently read data to the QLC pages with lower access latency, and flush other data to the slow QLC pages in an FSP operation, for boosting read performance of high density SSDs. To this end, we employ RL to classify the (cached) application data into four categories on the basis of their historical access frequency and the associated request size. After that, we can match four data pages in cache that have different output actions of the RL-based classification model, to the suited QLC pages and flush them together with one-shot program of FSP. In addition, we proactively trigger rewriting the hot read data that was previously stored in slow pages, by grouping it with the cached data as an FSP unit. Experimental results show that our proposal improves the read responsiveness by between 14.2%-62.7%, in contrast to the state-of-the-art methods.

Our proposal of RL-based scheduling is initially designed for the SSD devices adopting page-level mapping. In the future, we will further investigate the applicability of our approach in the scenarios of data interleaving across the QLC pages.

REFERENCES

[1] R. Micheloni, "Solid-state drive (SSD): A nonvolatile storage system," *Proc. IEEE*, vol. 105, no. 4, pp. 583–588, Apr. 2017.

[2] J.-W. Im et al., "7.2 A 128Gb 3b/cell V-NAND flash memory with 1Gb/s I/O rate," in *Proc. Int. Solid-State Circuits Conf. (ISSCC)*, 2015, pp. 1–3.

[3] J. Liao, F. Zhang, L. Li, and G. Xiao, "Adaptive wear-leveling in flash-based memory," *IEEE Comput. Archit. Lett.*, vol. 14, no. 1, pp. 1–4, Jan.–Jun. 2015.

[4] S. Lee et al., "A 1Tb 4b/cell 64-stacked-WL 3D NAND flash memory with 12MB/s program throughput," in *Proc. Int. Solid-State Circuits Conf. (ISSCC)*, 2018, pp. 340–342.

[5] C. Chang, "Is 3D NAND the right technology for removable devices?" in *Proc. Flash Memory Summit*, 2017, pp. 1–17.

[6] T. Hsiao, "3D flash leads to more powerful embedded applications," in *Proc. Flash Memory Summit*, 2017,pp. 1–11.

[7] "Toshiba and western digital readying 128-layer 3D NAND flash." 2019. [Online]. Available: https://www.techpowerup.com/253373/toshiba-and-western-digital-readying-128-layer-3d-nand-flash

[8] "Micron ships the industry's first 176-Layer QLC NAND." 2022. [Online]. Available: https://investors.micron.com/news-releases/news-release-details/micron-ships-industrys-first-176-layer-qlc-nand-volume-and

[9] H. Maejima et al., "A 512Gb 3b/Cell 3D flash memory on a 96-word-line-layer technology," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2018, pp. 336–338.

[10] J. K. Kim, "Multipage program scheme for flash memory," U.S. Patent 9 484 097, Nov. 2016.

[11] C. Kim et al., "11.4 A 512Gb 3b/cell 64-stacked WL 3D V-NAND flash memory," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2017, pp. 202–203.

[12] (Toshiba, Tokyo, Japan). *3D Flash Memory 'BiCS Flash^{TM}'*. 2024. [Online]. Available: https://business.kioxia.com/en-us/memory/bics.html

[13] "TLC NAND flash toggle," Data Sheet H27QFG8NNM8R BCG V3 256Gb, SK Hynix, Icheon-si, South Korea, 2016.

[14] W. Zhang, Q. Cao, H. Jiang, and J. Yao, "PA-SSD: A page-type aware TLC SSD for improved Write/Read performance and storage efficiency," in *Proc. Int. Conf. Supercomput. (ICS)*, 2018, pp. 22–32.

[15] J. Cui, Y. Zhang, J. Huang, W. Wu, and J. Yang, "ShadowGC: Cooperative garbage collection with multi-level buffer for performance improvement in NAND flash-based SSDs," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2018, pp. 1247–1252.

[16] N. Shibata et al., "13.1 A 1.33 Tb 4-bit/cell 3D-flash memory on a 96-word-line-layer technology," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2019, pp. 210–212.

[17] C. Ji et al., "Pattern-guided file compression with user-experience enhancement for log-structured file system on mobile devices," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2021, pp. 127–140.

[18] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drivers," in *Proc. 7th ACM Int. Conf. Embed. Softw. (EMSOFT)*, 2009, pp. 295–304.

[19] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," in *Proc. Int. Conf. Compil., Archit. Synth. Embed. Syst. (CASES)*, 2006, pp. 234–241.

[20] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT press, 2018.

[21] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, no. 3, pp. 279–292, 1992.

[22] Z. Sha, J. Li, Z. Cai, M. Huang, J. Liao, and F. Trahay, "Degraded mode-benefited I/O scheduling to ensure I/O responsiveness in RAID-enabled SSDs," *ACM Trans. Design Autom. Electron. Syst. (TODAES)*, vol. 27, no. 6, pp. 1–24, 2022.

[23] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1141–1155, Jun. 2013.

[24] C. Gao et al., "Constructing large, durable and fast SSD system via reprogramming 3D TLC flash memory," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2019, pp. 493–505.

[25] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Improving flash wear-leveling by proactively moving static data," *IEEE Trans. Comput.*, vol. 59, no. 1, pp. 53–65, Jan. 2010.

[26] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Trans. Storage*, vol. 4, no. 3, pp. 1–23, 2008.

[27] "Alibaba block traces." Accessed: Mar. 10, 2024. [Online]. Available: https://github.com/alibaba/block-traces

[28] W. Kang, D. Shin, and S. Yoo, "Reinforcement learning-assisted garbage collection to mitigate long-tail latency in SSD," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 1–20, 2017.

[29] J. Li et al., "Mitigating negative impacts of read disturb in SSDs," *ACM Trans. Design Autom. Electron. Syst.*, vol. 26, no. 1, pp. 1–24, 2020.

[30] W. Fan, Y. Zhang, W. Song, M. Zhao, Z. Shen, and Z. Jia, "Q-learning based backup for energy harvesting powered embedded systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2020, pp. 1247–1252.

[31] Y. Pan, Z. Jia, Z. Shen, B. Li, W. Chang, and Z. Shao, "Reinforcement Learning-Assisted Cache Cleaning to Mitigate Long-Tail Latency in DM-SMR," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, 2021, pp. 103–108.

[32] C. Wu et al., "Maximizing I/O throughput and minimizing performance variation via reinforcement learning based I/O merging for SSDs," in *Proc. IEEE Trans. Comput.*, vol. 69, no. 1, pp. 72–86, Jan. 2020.

[33] Q. Wei, Y. Li, Z. Jia, M. Zhao, Z. Shen, and B. Li, "Reinforcement Learning-Assisted Management for Convertible SSDs," in *Proc. 60th ACM/IEEE Design Autom. Conf. (DAC)*, 2023, pp. 1–6.

[34] M. Li, C. Wu, C. Gao, C. Ji, and K. Li, "RLAlloc: A deep reinforcement learning-assisted resource allocation framework for enhanced both I/O throughput and QoS performance of multi-streamed SSDs," in *Proc. 60th ACM/IEEE Design Autom. Conf. (DAC)*, 2023, pp. 1–6.

[35] F. Wu, Z. Lu, Y. Zhou, X. He, Z. Tan, and C. Xie, "OSPADA: One-shot programming aware data allocation policy to improve 3D NAND flash read performance," in *Proc. Int. Conf. Comput. Design (ICCD)*, 2018, pp. 51–58.

[36] C.-Y. Liu, Y. Lee, W. Choi, M. Jung, M. T. Kandemir, and C. Das, "GSSA: A Resource Allocation Scheme Customized for 3D NAND SSDs," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2021, pp. 426–439.

[37] G. Wu, X. He, and B. Eckart, "An adaptive write buffer management scheme for flash-based SSDs," *ACM Trans. Storage*, vol. 8, no. 1, pp. 1–24, 2012.

[38] R. Liu, Z. Tan, L. Long, Y. Wu, Y. Tan, and D. Liu, "Improving fairness for SSD devices through DRAM over-provisioning cache management," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 10, pp. 2444–2454, Oct. 2022.

[39] W. H. Kang, S. W. Lee, B. Moon, Y. S. Kee, and M. Oh, "Durable write cache in flash memory SSD for relational and NoSQL databases," in *Proc. ACM Int. Conf. Manag. Data (SIGMOD)*, 2014, pp. 529–540.

[40] S. Pang, Y. Deng, G. Zhang, Y. Zhou, Y. Huang, and X. Qin, "PSA-Cache: A page-state-aware cache scheme for boosting 3D NAND flash performance," *ACM Trans. Storage*, vol. 19, no. 2, pp. 1–27, 2023.

[41] J. Li et al., "Pattern-based prefetching with adaptive cache management inside of solid-state drives," *ACM Trans. Storage*, vol. 18, no. 1, pp. 1–25, 2022.

[42] S. Pang, Y. Deng, G. Zhang, J. Huang, and Z. Wu, "Minato: A read-disturb-aware dynamic buffer management scheme for NAND flash memory," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 43, no. 7, pp. 1930–1943, Jul. 2024.

[43] D.-W. Chang, W.-C. Lin, and H.-H. Chen, "FastRead: Improving read performance for multilevel-cell flash memory," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 9, pp. 2998–3002, Sep. 2016.

[44] Y. Lv, L. Shi, C. J. Xue, Q. Zhuge, and E. H. M. Sha, "Latency variation aware read performance optimization on 3D high density NAND flash memory," in *Proc. Great Lakes Symp. VLSI (GLSVLSI)*, 2020, pp. 411–414.

[45] C. Gao et al., "Exploiting chip idleness for minimizing garbage collection—Induced chip access conflict on SSD," *ACM Trans. Design Autom. Electron. Syst. (TODAES)*, vol. 23, no. 2, pp. 1–2, 2017.

[46] H. Huh et al., "13.2 A 1Tb 4b/cell 96-stacked-WL 3D NAND flash memory with 30mb/s program throughput using peripheral circuit under memory cell array technique," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2020, pp. 220–221.

[47] Y. Lv et al., "MGC: Multiple-gray-code for 3D NAND flash based high-density SSDs," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2023, pp. 122–136.

[48] W. Liu, J. Cui, T. Li, J. Liu, and L. T. Yang, "A space-efficient fair cache scheme based on machine learning for NVMe SSDs," *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, vol. 34, no. 1, pp. 383–399, Jan. 2023.

[49] B. Schroeder, R. Lagisetty, and A. Merchant, "Flash reliability in production: The expected and the unexpected," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 67–80.

[50] S. Wang, F. Wu, C. Yang, J. Zhou, C. Xie, and J. Wan, "WAS: Wear aware superblock management for prolonging SSD lifetime," in *Proc. 56th Annu. Design Autom. Conf. (DAC)*, 2019, pp. 1–6.