Learning to Coordinate Read-Write Cache Policies in SSD

Yang Zhou, Fang Wang, Zhan Shi, Dan Feng
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
zhouyang1024cs@hotmail.com, {wangfang, zshi, dfeng}@hust.edu.cn

***Abstract*—Caching is one of the most effective methods to optimizing SSD performance. This work finds that the existing cache management policies are inefficiently used due to two issues. First, since most SSD caches are designed as write caches, and do not account for reads I/O accesses that are critical to SSD performance. Second, lack of effective coordination between read and write caching policies leads to the inefficient utilization of cache and flash memory bandwidth. This leaves us room to significantly improve SSD performance. To this end, we present a Learning-based Coordinated Read-Write policy (`L-CoRW`) for SSD cache management. First, `L-CoRW` adaptively controls read caches based on observed access patterns and SSD internal state to efficiently utilize and optimize SSD cache and bandwidth resources. Second, `L-CoRW`, adopts the active write-through policy, creates a sufficient amount of clean data during low-intensity and poor data locality to mitigate the SSD performance cliff caused by burst I/O. Experimental results on the latest block-level I/O traces show that `L-CoRW` significantly reduces the response time and tail latency by up to 48.6% and 33.7%, respectively, over the previous schemes with negligible cost. Meanwhile, `L-CoRW` can reduce write amplification factor (WAF) by 9.6% to 24.7%.**

***Index Terms*—Cache Management, Solid State Disks, Block-level I/O**

I. Introduction

Currently, solid-state drives (SSDs) have been widely deployed in modern storage systems. Modern SSDs are usually configured with an on-board cache which plays a crucial role in improving SSD throughput and lifetime. Some SSD caches apply the write cache mode due to flash read/write asymmetry.

Many existing approaches propose cache management methods for write optimization [1]–[8]. These existing research works combine the workload characteristics with flash memory to reorder the cache queue to keep hot data in the cache as much as possible. For example, Chen *et al.* propose eviction-cost-aware (ECR) cache management policy for page-level flash-based SSDs [6]. ECR organizes the write data in the cache according to the underlying flash chip, and expects to free up the cache space by writing back the dirty pages on the idle chip. Similar to ECR, Liu *et al.* propose load-aware cache replacement algorithm (LCR) [5], which trades off the miss ratio and the miss penalty by narrowing down the range of victim data (dirty or clean data) to be selected in cache. The basic idea is to give a higher priority to cache the data on the flash that are in high I/O state. CFLRU (Clean-First LRU) [4] evicts clean written data preferentially but does not consider the trade-off between write-through and write-back in different workload phases, resulting in the uneconomical eviction of clean data. Write-through means that the data is also synchronously written to the flash layer when it is written to the cache. At this time, the data in the cache is clean data. On the contrary, write-back means that data only needs to be written to the cache. In this case, there is dirty data in the cache, which needs to be written to the flash memory when the dirty data is replaced out of the cache. Note that this paper does not explore the impact of different cache replacement or prefetching algorithms on SSDs.

In fact, the core of the above methods is to keep an appropriate proportion of clean pages in the SSD cache, so as to reduce the impact of burst I/O on SSD. However, these methods adopt passive update plans, which are reflected in the fact that corresponding strategies are only taken when the system experiences abnormal performance. Frequent replacement of dirty data in the face of burst I/O under write intensive workloads often leads to a sharp SSD *performance cliff* [2]. Besides, these caching scheme need to allocate the detailed physical address for all write data in advance, which undoubtedly increases the SSD complexity. Recently, Co-Active [2] solves the above problem through the *queuing theory* model. Unfortunately, various mechanisms such as program or erase-pause make it difficult for Co-Active to establish accurate queue prediction models. Moreover, we also found that the existing techniques use heuristically determined fixed design and do not adjust the factors at run time. In addition, some aggressive caching policies also affect the SSD related properties, like WAF. For example, Huang *et al.* [9] design a cache management scheme that periodically write-back data to flash in active mode. CFLRU, LCR, and ECR also do not consider flash memory wear during write-through, resulting in these methods continuously writing a large I/O traffic to flash memory, which raises write amplification and shortens the lifespan of SSDs. Limited by the fact that the frequency of access only distinguishes between once and many times, it also does not filter out hot data very well, so the wear problem remains as well.

Through some preliminary studies in realistic enterprise scale traces (see Section III), we find that the differences in I/O traffic and locality of the read and write requests for modern block-level I/O traces can be complementary. This allows us to improve SSD performance by taking advantage of the read and write I/O on the relevant features, which involves the coordination of the read and write cache policies. Furthermore, burst I/Os are also very common in enterprise traces, which can cause the SSD performance cliff in the cache. One reason is that

existing cache policies use write-back method to handle data updates [7], which makes SSD cache unable to quickly buffer I/O traffic. Therefore, we need to consider how to alleviate the impact of burst I/O on SSDs. At the same time, setting a reasonable caching scheme for data with better locality is also important to further reduce flash wear and improve SSD lifespan [10]. Aiming to augment and coordinate these practical policies in SSD (e.g., read cache, write-through), in this paper, we propose a coordinated read-write cache management policy based on online reinforcement learning called L−CoRW which makes the advantages and drawbacks of the different caching modes complement each other. The contributions of this paper are as follows:

- We provide an in-depth experimental analysis of the latest realistic enterprise scale traces. We find significant differences between the read and write requests in the I/O traffic and locality, forming the key component of our proposed novel cache policy.
- We propose L−CoRW, a light-weight learning-based cache control framework for modern SSDs that mitigates the performance degradation and performance cliff triggered by burst I/O under write-dominated workloads.
- We evaluate L−CoRW using a wide variety of storage workloads consisting of diverse applications in an open-source SSD simulator and demonstrate the efficiency of L−CoRW.

The paper is organized as follows. Section II introduces the background, while Section III presents our motivation for this work. Section IV clarifies our design goals and explains why we used a learning-based approach. Section V presents the proposed L−CoRW policy. The experimental evaluation is presented in Section VI, with a discussion in Section VII. Section VIII concludes the paper.

II. Background

A. SSDs Architecture

Fig. 1 illustrates the architecture of an SSD. Inside the SSD, requests fetched from multiple host-side submission queue (SQ) are first placed at the host interface logic (HIL). Each request is divided into multiple transactions at the granularity of a single page. Next, all transactions will access the on-board DRAM. Some transactions that do not hit the cache or trigger a dirty data write-back will be translated by the address mapping table from the logical page address (LPA) into the physical page address (PPA). Finally, the transaction scheduling unit (TSU) schedules these transactions to achieve four level parallelism, from channel, chip, die to plane [11], [12].

Most current SSD controller consists of embedded DRAM and processors to process the I/O requests and manage the flash. Due to the read and program NAND flash-based asymmetry, SSD internal cache is usually designed as a write cache mode. More specifically, read requests will not be cached when cache miss, while write requests are cached in the SSD cache (when write cache miss) [7]. To ensure data consistency, dirty pages replaced out of the cache are written back to the underlying flash memory synchronously (passive policy). In addition, there

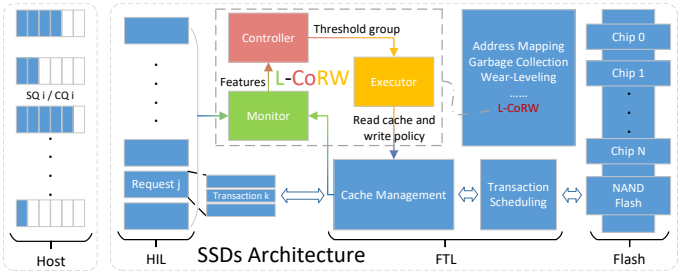


Fig. 1. The overview of a modern SSD and L−CoRW.

is a proactive write-through policy to synchronize write data to cache and flash memory. However, either write-back or write-through will compete with the read request for limited bandwidth resources at the TSU, which is unfair to the read request.

B. Block-level I/O Traces

We use realistic enterprise scale workloads to study cache management policy. To gain meaningful insight into the I/O characteristics of the storage system, we consider the public block-level I/O traces from Alibaba Cloud [13], Tencent Cloud [14], and Microsoft [15]. Table I shows different categories of basic statistics of AliCloud, TenCloud, and MSR, where MSR has been used in most previous studies related to SSD caching [1], [2], [6], [16].

TABLE I
Basic statistics of AliCloud, TenCloud, and MSR

	AliCloud	**TenCloud**	**MSR**
Source	Alibaba Cloud	Tencent Cloud	Microsoft
#Volumes	1000	4966	36
Duration (days)	31 (Jan. 2020)	7 (Oct. 2018)	7 (Feb. 2007)
Total requests	20,232,973,351	25,998,077,615	868,424,016
Write (%)	75.00	69.84	29.77

We can observe that both AliCloud and TenCloud have a significantly higher percentage of write requests. This is because front-end application servers absorb the vast majority of reading requests by caching hot data in storage system, resulting in many storage back-ends experiencing a write-dominant workload behavior. Since MSR traces are over 10 years old and the number of volumes is small, we mainly focus on the AliCloud and TenCloud traces in this paper.

C. Reinforcement Learning

In reinforcement learning (RL), an agent interacts with its environment to learn how to adapt to the current state. The goal of RL is to learn the optimal action function and thus maximize the reward. In L−CoRW, we adopt Q-Learning [17] as the learning algorithm. Q-Learning is a popular RL algorithm that is one of the most classic *value*-based RL methods and does not require any prior knowledge of the system. Many research works [16], [18], [19] have confirmed that Q-Learning can adapt well to discrete state spaces, which meets our goal of a light-weight learning algorithm. The core of Q-Learning

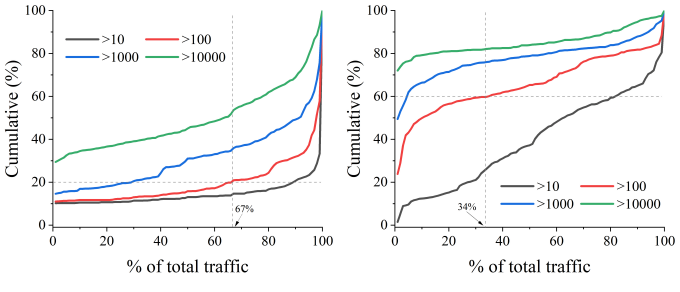
is the calculation of Q-tables, which are defined as $Q(s,\ a)$. The s and a represent state and action. The rows of Q-table represent Q-value of the state while the columns of Q-table represent action, which measures how good it will be if the current state takes this action. $Q(s,\ a)$ updates follow *Bellman Equation* which is defined below:

$$Q(s,\ a) \longleftarrow Q(s,\ a) + \lambda[r + \gamma \max_a Q(s^{'},\ a^{'}) - Q(s,\ a)] \quad (1)$$

where λ, r, and γ are the learning rate, the reward, and the discount factor, respectively. The $a^{'}$ is the action in the next state $s^{'}$. For the use of Q-table, we will choose the action a that maximises $Q(s,\ a)$ ($\mathrm{argmax}_a Q(s, a)$) at state s, thus obtaining the policy recommended by RL.

III. Motivation

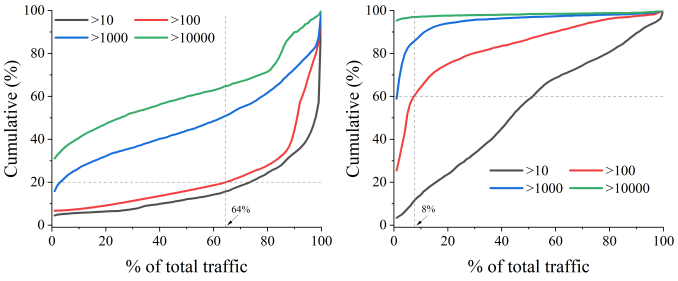
In this section, we present a high-level analysis of AliCloud and TenCloud. Even though there are more write requests in AliCloud and TenCloud, we find some interesting differences between the read and write requests, creating more challenges for designing an optimal cache management policy. Currently, there are studies [2] pointing out that SSD performance is more affected by burst I/Os. In addition to limited hardware resources, stereotypical resource scheduling also reduces the utilisation of hardware resources. For example, although sometimes the read data traffic is greater than the write traffic, the SSD cache is always used as the write cache at this time. According to Ren *et al.* [20], tens of times disparity in peak workloads on cloud storage servers should be considered when load balancing. We pre-process the workloads for our analysis and evaluation as follows. We remove some workloads that have almost no read requests (e.g., <0.1% of reads) or a too small number of requests (e.g., <8MB) to avoid biasing our analysis. We analyze the percentage of data traffic in the workloads when the I/O traffic is 10 times (>10), 100 times (>100), 1000 times (>1000), and 10000 times (>10000) larger than the average I/O bandwidth. The results are shown in Fig. 2 and Fig. 3.



(a) read (b) write

Fig. 2. The cumulative distribution of the burst I/O traffic for AliCloud.

We find that burst I/O contains a large proportion of traffic, especially for read I/O. For example, in about 80% of AliCloud's read traces, the data volume generated by >100 times traffic accounts for over 67% of the total data volume of each workload, which is 64% in TenCloud. In contrast, the >100 times write traffic in 60% of AliCloud traces accounts for less



(a) read (b) write

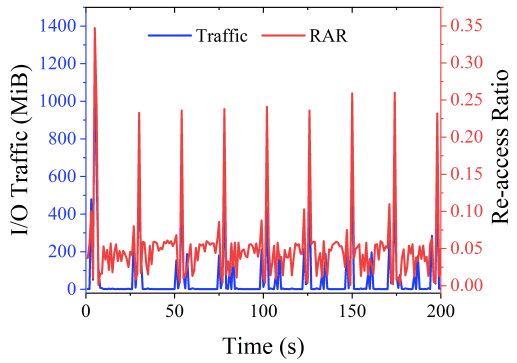
Fig. 3. The cumulative distribution of the burst I/O traffic for TenCloud.

than 34% of the total data volume of each workload, which is even less in TenCloud (<8%). This means that burst read I/O is more common, and the write traffic is more stable and smooth than the read traffic in AliCloud and TenCloud.

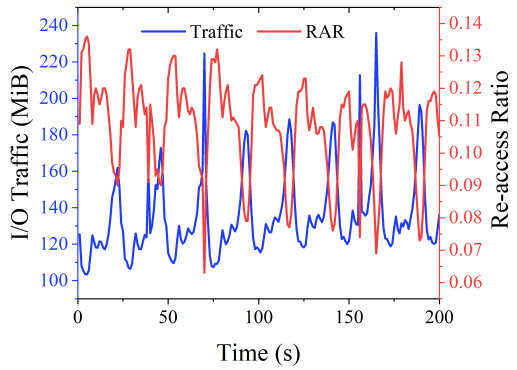
For SSD caches, smooth I/O traffic can reduce the possibility of blocking and SSD performance cliff. Moreover, another important factor that affects the cache performance is the data locality. A vast majority of cache replacement algorithm favor LRU (Least Recently Used) in SSD because it can exploit locality features. LRU performs well when data that was used recently is likely to be reused in the near future [21]. Some metrics generally used to characterise data locality include *footprint* [22], *re-access ratio* [14], and *reuse distance* [23], etc. Here we use the re-access ratio to show the locality because the algorithm overhead is small, and it facilitates us to build the corresponding model inside the SSD for analysis. Re-access ratio (RAR $\in [0, 1)$) is defined as a ratio of the re-access data to the total data and a higher RAR-value indicates better data locality. We speculate on the possible relationship by observing the change in I/O traffic versus locality. Fig. 4 plots I/O traffic versus locality for one of the traces selected in AliCloud, with similar cases for the other traces. We can see that the locality of read requests and I/O traffic show a positive correlation (Fig. 4 (a)), and the trend of the read traffic is not stable (from 0 to 1400 MiB), while the write traffic's behavior is the opposite of the read (Fig. 4 (b)). However, a question is whether the above pattern still holds during the whole time period of the workload.

Based on the observation of I/O traffic and locality, we conduct experiments to evaluate the potential correlations between them. By calculating the *Pearson correlation coefficient* ρ ($\rho \in [-1, 1]$) between the real-time I/O traffic and the RAR, we can quantify the correlation between them. A higher $|\rho|$-value indicates higher correlation, where the ρ is positive for positive correlation and negative for the opposite. Fig. 5 shows the correlation analysis of the read and write request I/O traffic and locality for all volumes.

We can see that more than 80% / 87% of volumes have a positive correlation ($\rho > 0$) between RAR and I/O traffic for the read, and nearly 50% / 56% have a moderate correlation ($\rho > 0.4$) in AliCloud / TenCloud. In contrast, more than 80% / 84% of volumes have a negative correlation ($\rho < 0$) between RAR and I/O traffic for the write, and nearly 70% / 66% have a



(a) read



(b) write

Fig. 4. Motivating examples, trends in I/O traffic and locality.

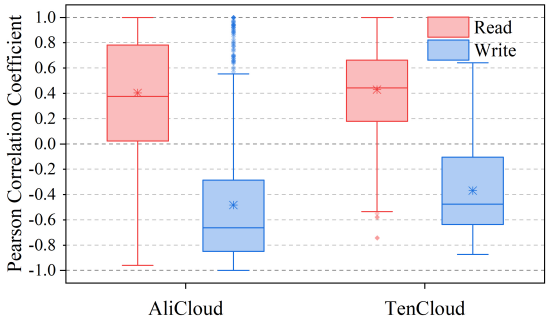


Fig. 5. Correlation analysis of I/O traffic and locality.

moderate correlation ($\rho < -0.4$) in AliCloud / TenCloud. This means that a higher read I/O traffic exhibits better locality, and a higher write I/O traffic exhibits poor locality. These observations provide key insight for us to design novel cache management policy.

IV. Design Goals and Challenges

A. Our Goal

We present L-CoRW, a learning-based coordinate read-write cache policy that balances SSD cache resources to improve read and write performance. L-CoRW has the following goals:

- Perform better than other cache management policies.
- Require no special knowledge or configuration.
- Be robust to dynamic workloads.

To achieve these goals, two concerns need to be considered. The first is the allocation of read cache, which will affect the benefit trade-off when the cache space is used by read and write requests. Although traditional SSD caches are used as write caches, since read and write requests have differences in locality characteristics, a reasonable allocation of cache resources is able to optimise both read and write performance. In AliCloud and TenCloud workloads, burst read I/O exhibit better locality, it is possible to improve overall SSD performance by caching some read requests when the write data has a poor locality. Suppose a part of cache space is allocated to the read requests with little impact on write requests. Especially when the write locality is poor, the cache will trigger frequent dirty data write-back. In this case, can we allocate cache space to read requests with better locality to obtain higher cache gains? This can reduce the number of read requests that enter the TSU due to SSD cache misses and possibly boost write performance (more available TSU bandwidth released from read requests) while improving read performance (release some SSD cache space from write cache as read cache).

The second is write cache policy in SSD cache. There are two ways to synchronise the cache, *write-through* and *write-back*. Both methods affect the clean data ratio within the SSD cache. Considering the write overhead of SSDs, SSD cache usually uses write-back mechanism by default. Nevertheless, write-back results in a large amount of dirty data in the cache, leading to frequent flushing of dirty data back to flash when the cache is full. Evicting dirty data requires triggering a write-back flash operation which is the most cost-effective way for cache. But furthermore, it is not wise to always evict clean data preferentially like CFLRU, as obtaining clean data requires consideration of the cost of write-through. To reduce the write latency, existing research projects [2], [7] reduce passive dirty data write-back when the cache is full by setting an active write-through. However, this method does not consider the data locality. In fact, implementing a write-through policy for write requests with better locality does not improve write performance. On the contrary, excessive write-through will consume some TSU bandwidth, which will affect the read performance. One possible solution is to stop frequent write-through when the write traffic exhibit better locality, better write data locality will allow previously written data to be updated (rewrite) in a short time. In this case, using a write-back policy does not affect the SSD performance because the written data is constantly updated in the cache and does not trigger a lot of dirty page flushes back. Therefore, the allocation of read cache and write cache policy must be determined by considering the workload patterns and the internal behavior of SSD cache such as data locality, clean data ratio etc. Besides, these factors must be adjusted according to the system state change (e.g., insufficient cache space or high amount of dirty data).

B. Challenges

With a determined goal, the key of our design includes the allocation of read cache and the selection of write policy. Since the adjustment is not very complicated, can other conventional methods be useful in doing so? To explore the challenges of

constructing problems in this paper and show the advantages of the learning-based approach, we have performed several preliminary experiments. The evaluation environments and device parameters are described in Section VI-A. We fix the relevant parameters in Algorithm 1 to obtain eight schemes respectively, and the parameter thresholds of each scheme are shown in Table II. tt and rt thresholds control the allocation of read cache, while ct controls the selection of write policy. Although these parameters and algorithm have not been introduced in detail, here we only need to know that these eight schemes (CoRW$_1$-CoRW$_8$) are the models of L−CoRW without learning. We conduct experiments with the trace used in Fig. 4, where the conventional SSD (Baseline) and the proposed learning-based scheme are used for comparison, and the results are shown in Fig. 6.

TABLE II
DIFFERENT MODEL PARAMETERS FOR CoRW

policy	threshold group		
	tt	rt	ct
CoRW$_1$	100	100	0.66
CoRW$_2$	100	100	0.33
CoRW$_3$	100	10	0.66
CoRW$_4$	100	10	0.33
CoRW$_5$	10	100	0.66
CoRW$_6$	10	100	0.33
CoRW$_7$	10	10	0.66
CoRW$_8$	10	10	0.33

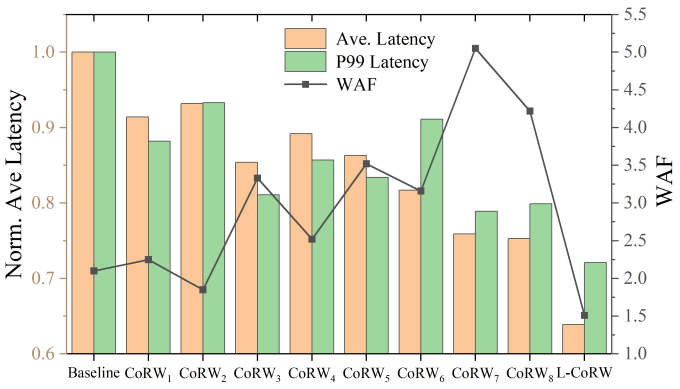


Fig. 6. Average latency, tail latency, and WAF comparison (normalized to Baseline).

Overall, CoRW performs better than Baseline in both average latency and tail latency, which suggests that the insight we observed is meaningful. However, CoRW is still far from L−CoRW in all metrics, and CoRW is unable to obtain trade-offs on different metrics. In detail, CoRW$_1$ and CoRW$_2$ require high values of I/O intensity and locality to allocate read cache for read I/O (tt and rt have higher thresholds), resulting in limited performance gains. In contrast, CoRW$_7$ and CoRW$_8$ have more relaxed read cache allocation conditions, but this further reduces the write cache space resulting in triggering more cache replacements and causing higher WAFs. In addition, a larger ct threshold means that more write requests will be write-through to maintain a higher proportion of clean data. This can mitigate the impact of burst I/O, so CoRW schemes with larger ct-value will perform better in terms of tail latency, but this also results in higher WAF. In general, the constantly changing workloads (Fig. 4) make it difficult to set optimal values to adapt to all types of workloads, which is not only a challenge for CoRW. In the experimental section, we will compare the performance differences between existing studies and learning-based approaches.

V. DESIGN

A. Architectural Overview

As shown in Fig. 1, L−CoRW consists of three main components, including the **monitor**, **controller**, and **executor**. The monitor extracts I/O features from the HIL and cache features from the FTL, while collecting the end-to-end latency of all requests. The controller is a light-weight RL algorithm based on Q-Learning and the internal Q-table can be updated according to the workloads and the SSD cache state. The executor completes the adjustment of the read/write policy and the update of the threshold according to the controller. To summarize, L−CoRW learns policies such as favoring read traffic with better locality over write traffic and keeps some clean data free to service future arriving burst I/O traffic directly from experience.

B. Reinforcement Learning in L−CoRW

In this work, we formulate SSD cache management as a reinforcement learning problem, as shown in Fig. 7. With every new request, L−CoRW observes the state of the workload and the SSD cache and takes a chosen action (read cache allocation and write cache policy). For every choice, L−CoRW receives a reward related to the latency that evaluates the chosen action's accuracy and timeliness by taking into account current I/O average latency and tail latency. Based on the reward, L−CoRW will be able to improve its decision policies by updating the model parameters. We hope that L−CoRW can find the optimal cache management policy for read and write requests respectively that can balance SSD resources such as cache space to minimise the I/O latency. The status space, action space, and reward function for RL-based L−CoRW are defined as follows.

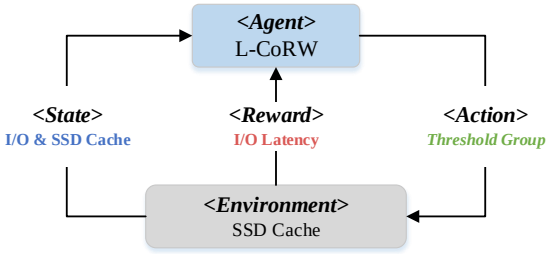


Fig. 7. Formulating the L−CoRW as an RL-agent.

1) State: In Section III, we reported that I/O traffic and locality are important factors affecting cache performance. So the I/O traffic T and RAR-values Ra of the read (T_r, Ra_r) and write (T_w, Ra_w) traffic are the main characteristics reflecting

TABLE III
CATEGORIZATION OF STATE

Category	State	Metrics		
		Low	Mid	High
Workload	$\frac{T_w}{T_r}$	$[0, 10)$	$[10, 100)$	$[100, \infty)$
	$\frac{Ra_r}{Ra_w}$	$[0, 10)$	$[10, 100)$	$[100, \infty)$
SSD	Cr	$[0, 0.33)$	$[0.33, 0.66)$	$[0.66, 1]$

the workload state. What's more, it is also important to choose between write-through or write-back for write requests, as replacing clean pages has less overhead, while replacing dirty pages requires writing data to flash memory (need to consume TSU bandwidth). However, when the I/O traffic is light, the flushing back of dirty pages does not have an impact on the I/O of the HIL because the SSD is experiencing very light traffic pressure at this time and there are sufficient bandwidth resources on the TSU side. Therefore, L−CoRW also needs to maintain a certain ratio of clean pages Cr in the cache according to the current workload and SSD state (clean data ratio in SSD cache). For workload characteristics, we use the *proportion* (e.g., A ÷ B) to reflect the difference between read and write requests in I/O traffic ($\frac{T_r}{T_w}$) and RAR ($\frac{Ra_r}{Ra_w}$). For the SSD cache state, we also classify the ratio of clean data Cr in the cache. Table III shows the workload and cache state, each of which is divided into multiple bins (low, mid, and high) to limit the total number of different states, inspired by [16], [18], [24]. For the workload state, we divide it into $[0, 10)$, $[10, 100)$, and $[100, \infty)$, while the clean data ratio is divided into $[0, 0.33)$, $[0.33, 0.66)$, and $[0.66, 1]$. There are a total of 3×3×3=27 state combinations for L−CoRW. The threshold settings of 10 and 100 times are based on the description in [20] and the observation of the results in Fig. 2 and Fig. 3. We can see from Fig. 2 and Fig. 3 that for read I/O traffic the distinction is most obvious at ×100 and ×1000, while this value is ×10 and ×100 for write I/O. This means that the proportion of read and write I/O intensity is around 10 to 100 times (100 ÷ 10 = 10, 1000 ÷ 100 = 10, 1000 ÷ 10 = 100). Therefore, we simply divide the workload state interval into two thresholds of 10 and 100, while the clean data ratio is based on the principle of uniform division (1 ÷ 3 ≈ 0.33, 2 ÷ 3 ≈ 0.66). We intend to investigate the effect of fine-grained division in our future work.

2) Action: L−CoRW needs to adjust two cache policies based on the state, including read cache allocation and write cache policy. When allocating a cache for a read request, we do not directly specify a precise cache size, but rather decide whether the read request needs to be cached (bypass or caching) if read cache miss. In this process, L−CoRW requires two thresholds including the I/O traffic threshold $tt \in \{tt_l, tt_h\}$ $(tt_l < tt_h)$ and the RAR threshold $rt \in \{rt_l, rt_h\}$ $(rt_l < rt_h)$, which will be used to determine whether the current workload state meets the conditions for caching read requests. For the write cache policy, L−CoRW decides whether to use write-through or write-back policy for write requests based on the clean data threshold $ct \in \{ct_l, ct_h\}$ $(ct_l < ct_h)$ within the SSD cache. In summary, the action of L−CoRW is to select an appropriate *threshold group* (tt, rt, ct), and we will describe how to use these thresholds in Section V-C. There are a total of 2×2×2=8 action combinations for L−CoRW.

3) Reward: The goal of L−CoRW is to learn an optimal cache management policy that can minimize the end-to-end request latency. L−CoRW needs to determine how good the current policy is based on the current average latency ($latency_{ave}$) and tail latency ($latency_{tail}$) together with the I/O traffic (T). The reward function is given as follows:

$$r = -[w \times \frac{latency_{ave}}{T} + (1 - w) \times \alpha \times \frac{latency_{tail}}{T}] \quad (2)$$

where α is the scaling factor ($\alpha = \frac{latency_{ave}}{\mathbf{MAX}(latency)}$) for tail latency, used to normalize tail latency to the same value range as the average latency. The w represents the weight when L−CoRW optimizes the two objectives of average latency and tail latency. By default, w is set to 0.5, which means that average latency and tail latency are equally important in the optimization objectives of L−CoRW. The minus sign means that we punish long latencies because the reward is typically maximized. The reward function is independent of changes in the hardware environment and workload, but depends only on the state. Therefore, adapting the reward function to other state definitions is relatively easy.

4) Update and Use: L−CoRW needs to select the action a with the maximum $Q(s, a)$-value corresponding to the current state s in the Q-table. The selected a-value is used as the new threshold group (tt, rt, ct), which will be used to implement the cache management policy for L−CoRW. Then, L−CoRW obtains the reward r-value and uses the current and past states ($s^{'}$ and s) and actions ($a^{'}$ and a) to update the Q-table with Equation 1. In Equation 1, the λ and γ are set to 0.9 and 0.1. We plan to update the Q-table every 0.2 second.

C. Cache Management Policy in SSD

Algorithm 1 shows the pseudo-code of the cache management policy, which describes how L−CoRW uses the threshold group (tt, rt, ct).

By default, the SSD cache inside L−CoRW is a write cache. However, for the read I/O, they can be cached if the current write I/O traffic is less than the read I/O traffic and the read locality is better (Algorithm 1, lines 4-5). This ensures that read performance is improved by allocating some of the write cache to the read I/O without affecting the write performance. For the write I/O, L−CoRW needs to choose to write-through or write-back based on the current write I/O traffic, clean data ratio, and locality (Algorithm 1, lines 10-14). L−CoRW will only apply a write-through policy for worse locality write data ($Ra_w < 0.01$, empirically) when the I/O traffic is light and the ratio of clean data is low. This allows L−CoRW to have sufficient clean data in the cache when it encounters burst I/O write, while reducing unnecessary write-through when write requests have better locality. In Algorithm 1, $\overline{T_w}$ is calculated based on the I/O traffic in the past time, which is because AliCloud and TenCloud collect data daily.

In addition, several problems should be solved before the above Coordinated Read-Write policy works. The threshold

Algorithm 1: Cache management policy in L-CoRW

Input: I/O traffic threshold (tt), RAR threshold (rt), clean data threshold (ct), the current average write I/O traffic ($\overline{T_w}$), the current clean data ratio (C_r)

```
for every request do
    if request is read I/O then
        if there is a cache miss for request then
            if Tr/Tw > tt and Ra_r/Ra_w > rt then
                Put request into cache and Fetch from cache
            else
                Fetch request from flash memory
        else
            Fetch request from cache
    else if request is write I/O then
        if T_w < T_w (avg) and C_r < ct and Ra_w < 0.01 then
            Write-through request
        else
            Write-back request
```

used by tt and rt are sensitive parameters. Taking tt as an example, for a large tt threshold, L-CoRW needs to reach a higher condition for I/O traffic in order to trigger cache space allocation for read requests, which will limit L-CoRW to achieve better performance. However, for a small tt threshold, L-CoRW will make cache allocation for read I/O easier, but this will reduce write cache space, which may impact the SSD performance. The above analysis apply equally to the rt. Second, similar to the determination of tt and rt, maintaining an appropriate proportion of clean pages for burst I/O is also an important problem (based on threshold ct). L-CoRW wants to maintain enough clean pages in the cache, but this requires write-through more write requests and causing write amplification issues. However, write amplification can be mitigated if the ct is set very small, but the performance improvement on SSD will be very limited. Finally, the update frequency for Q-table is also an issue. Updating too slowly will fail to accommodate I/O changes, and updating too quickly is completely unnecessary. In the experiment, a sensitive study will be presented (see Section VI-E in detail).

VI. Evaluation

A. Experimental Settings

We use MQ-SSD [7], an open-source multi-queue NVMe (Non-Volatile Memory Express) SSD simulator [25], to test the effectiveness of RL-based cache management policy. MQ-SSD has been widely used in the exploration of SSDs [2], [26]–[28]. Table IV presents the SSD configuration in this paper. We apply the block-level I/O traces from Table I and these traces have already been introduced in previous section.

TABLE IV
Configuration of the simulated SSD

SSD Host Interface	PCIe 3.0 (NVMe 1.2)
SSD Capacity / Data Cache	512GB / 128MB
Data Cache DRAM row size	8KB
Flash Communication Interface	ONFI 3.1 (NV-DDR2) Width: 8 bit, Rate: 333 MT/s
Flash Latencies (TLC) [2]	Read latency: 75 μs Program latency: 750 μs Erase latency: 3.8 ms
Channel / Chip / Die / Plane	8 / 4 / 2 / 2
Block/Page	2048 / 256
Page Capacity	8KB
Flash Translation Layer	GC Policy: RGA [29] GC Threshold: 0.05 Address Mapping: DFTL [30] TSU Policy: Sprinkler [31] Over Provisioning Ratio: 0.07

We compare L-CoRW with the following five alternative solutions, including the CFLRU [4], MQ-SSD [7], LCR [5], ECR [6], and Co-Active [2]. CFLRU is an LRU queue that prioritises the expulsion of clean data. MQ-SSD represents the conventional SSD design for cache management, in which write cache mode combined with a cold-and-hot separation unit are applied. LCR and ECR are both load-aware cache management algorithms, the only difference is that ECR chooses between dirty and clean pages in order to minimize the eviction cost in write-dominant applications. Co-Active is a cache management policy that proactively triggers write-backs based on hot or cold data, which is collaboratively aware of I/O access patterns and the status in flash chips. We run these methods on a local Inspur server equipped with a six-core 2.10GHz Intel(R) Xeon(R) E5-2620, 64GiB RAM, and an SMC 512GB hard disk.

B. Performance Evaluation

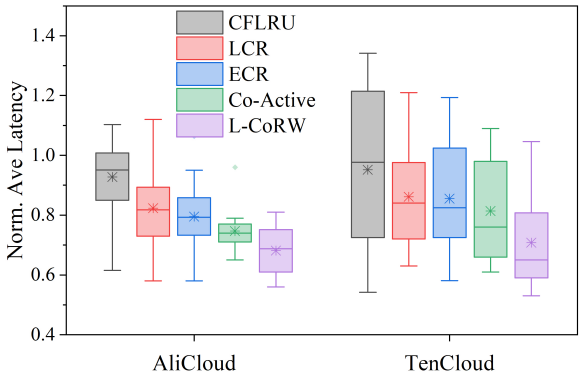


Fig. 8. Average latency comparison (normalized to MQ-SSD).

Fig. 8 compares average latency of different cache policies with AliCloud and TenCloud traces. The performances are normalized to MQ-SSD (baseline). To summarize the findings, L-CoRW is distinctly the best performing policy in 87% / 83% of the AliCloud / TenCloud traces ranging from 0.5 to 0.9 normalized average latency. A head-to-head comparison shows that L-CoRW reduces the average latency of CFLRU, ECR, and Co-Active by 48.6%, 34.5%, and 25.8% among all the

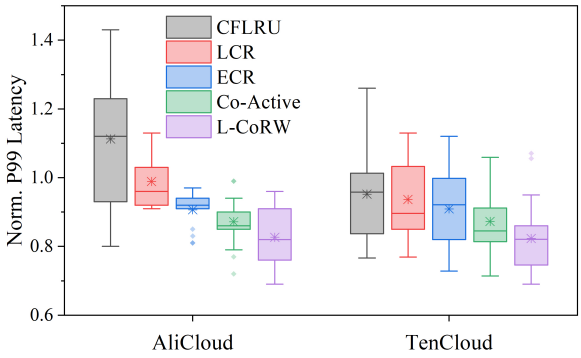


Fig. 9. Tail latency comparison (normalized to MQ-SSD).

workloads, respectively. Since CFLRU prioritizes the eviction of clean data in all cases, which leads to a case where no clean data to choose in burst I/O accesses. Thus, CFLRU is worse than MQ-SSD in some workloads. ECR has more options for evicting dirty pages than LCR, thus obtaining slightly better performance than LCR by balancing write-back overhead. Co-Active allows the pressure of burst I/O to be released in TSU and achieves better performance in write-dominant traces. However, compared with L-CoRW, ECR and Co-Active do not work until passive write-back is triggered, whereas L-CoRW tends to be proactive.

Fig. 9 shows the P99 tail latency for all traces, reflecting the ability of different caching policies to mitigate SSD performance cliffs. The results show that L-CoRW has lower tail latency with 22.6% to 36.7% reduction compared to other caching policies. L-CoRW maintains more clean data for the cache by writing cold data and worse locality data to flash memory in advance (write-through) under low I/O traffic, so as to cope with future burst I/O. In this way, L-CoRW can have more clean data to evict in case of burst I/O, thus avoiding the need to consume the TSU bandwidth resources when replacing dirty data. In contrast, other caching policies, especially CFLRU and LCR, do not have sufficient clean data in the cache, thus leading to performance cliff in SSDs.

C. Performance Analysis

To demonstrate how L-CoRW can reduce the end-to-end latency and SSD performance cliffs, we analyze the real-time internal metrics of L-CoRW. We extract two million I/O requests from AliCloud traces as simulator inputs. We set every 10000 I/Os as an episode and count the metrics of each episode. Fig. 10 shows the change of cache hit rates for MQ-SSD (baseline) and L-CoRW and cache space allocated by L-CoRW for read I/O, with similar results for other volumes. We can observe that L-CoRW improves the read cache hit rate by caching read requests at the appropriate episode. Although caching these read requests occupies the write cache space, it is possible to improve the write cache hit rate simultaneously in most cases (e.g., episode in 29∼32, 52∼55). This is because L-CoRW evaluates the differences in I/O traffic and locality between the current read and write requests. Thus, caching read requests is likely to benefit when the read I/O traffic is small

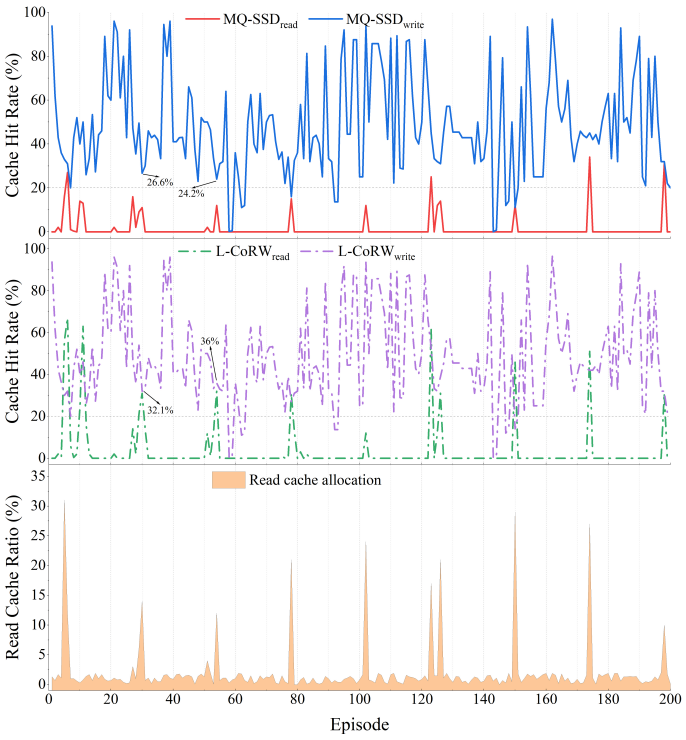


Fig. 10. The change of cache hit rates for MQ-SSD (baseline) and L-CoRW and cache space allocated by L-CoRW for read I/O. (MQ-SSD$_{read}$/MQ-SSD$_{write}$: MQ-SSD's read/write cache hit rate; L-CoRW$_{read}$/L-CoRW$_{write}$: L-CoRW's read/write cache hit rate)

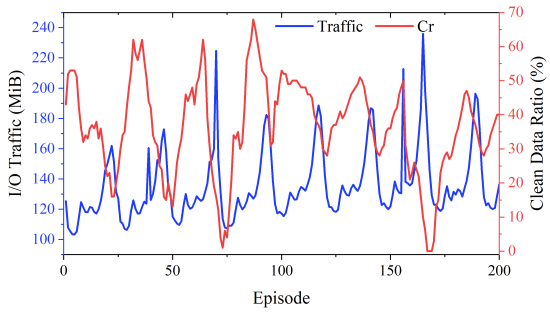


Fig. 11. The adaptability of the write I/O traffic and the clean data ratio.

relative to the write I/O traffic and the read requests have better locality (at this time, the write locality and hit rate are low). Note that L-CoRW does not achieve better performance by increasing cache hit rates, which is not a feature or optimization objective used by L-CoRW. We only demonstrate this process to analyze why L-CoRW is better than other solutions.

Furthermore, Fig. 11 depicts the real-time write I/O traffic and clean data ratio in L-CoRW. It is observed that L-CoRW can reserve a sufficient clean data ratio in the cache for burst I/O, thus reducing cache blocking caused by flushing dirty data to flash memory. Besides, when the write I/O traffic is light, L-CoRW utilises the TSU's bandwidth resources to write-through infrequently data to the underlying flash memory timely, so as to prepare for the next burst I/O by increasing the clean data ratio in the cache.

D. Write Amplification Factor

WAF is very important to understand SSD performance and lifetime. WAF is calculated as $Flash\_write/User\_write$, where $Flash\_write$ and $User\_write$ are flash write size and user write size, respectively [32]. Low write amplification value signify good performance. Fig. 12 presents the experiment result of the WAF in AliCloud and TenCloud workloads. On average, L−CoRW reduces the overall WAF by 24.3% compared with conventional SSDs and 9.6%-24.7% compared with other schemes. MQ-SSD and CFLRU lack effective recognition mechanisms for hot and cold data and locality, leading to high WAF. ECR only selects and distinguishes data for load balancing, and this method only have limited effectiveness in reducing WAF. Co-Active still has uncertainty in prediction accuracy, and passive replacement strategies can easily affect the prediction of future data, so it does not perform better than L−CoRW. By contrast, L−CoRW improves adaptability and timeliness using RL, aiming to keep more better locality write requests in the cache as much as possible. Thus, the write amplification is improved by reducing the amount of data written into the flash memory, allowing it to achieve the lowest overall WAF. In addition, we also notice that AliCloud has a larger WAF value than TenCloud. This is because TenCloud in general shows a higher randomness ratio than AliCloud [33], resulting in the need to move more valid pages during garbage collection.

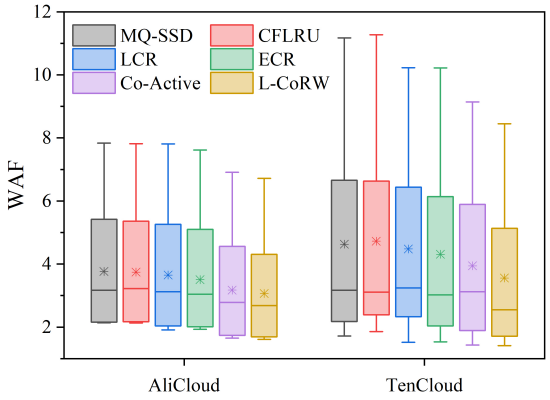


Fig. 12. Write amplification factor comparison.

E. Sensitivity Study

At the end of Section V-C, we mention several key thresholds that need to be determined, which affect the average latency, tail latency, and WAF of L−CoRW. To understand their characteristics, sensitive studies are presented to show their performance impact. In sensitivity experiments, the two thresholds tt_l and tt_h for tt are varied from 2 to 512 and 4 to 1024, respectively (tt_l and tt_h must meet the condition $tt_l < tt_h$). We only discuss the results for the tt as our observations apply equally to the rt. Fig. 13 (a) shows how tt_l and tt_h affect the average latency of L−CoRW (normalized to the optimal value of L−CoRW). We can observe that both tt_l and tt_h will result in poor performance of L−CoRW in both large and small cases. This is because L−CoRW allocates too much read cache when the threshold is small. However, when the threshold is large it will misallocate read cache resources, resulting in limited performance improvement. This case is especially worst when tt_l and tt_h are at extreme values ($tt_l = 2$ and $tt_h = 1024$). But we also find that there is a appropriate combination of tt_l and tt_h thresholds ($tt_l = 8$ and $tt_h = 128$) that allow L−CoRW to perform optimally. For rt, we obtain optimal values of $rt_l = 16$ and $rt_h = 64$, as shown in Fig. 13 (b).

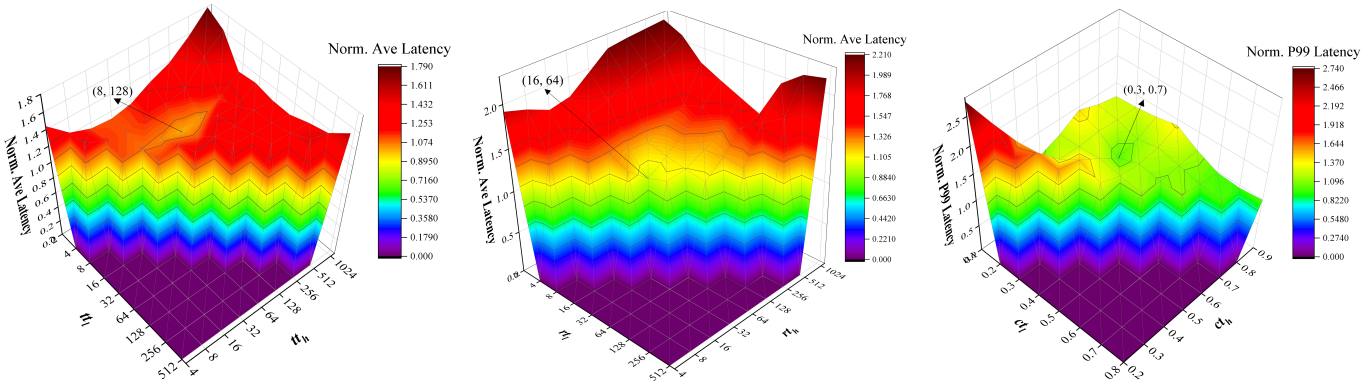
We use the same method as analysing tt thresholds to investigate the effect of ct threshold (including ct_l and ct_h) on L−CoRW, with the difference that we focus on the tail latency metric as in Fig. 13 (c). Note that the lowest tail latency occurs when both ct_l and ct_h have high values, while the performance of L−CoRW is the worst when ct_l and ct_h are very small. This is because L−CoRW eliminates the long-tail latency impact of burst I/O on SSDs when L−CoRW reserves enough clean pages for the cache. However, reserving too many clean pages can trigger more write-through in the SSD cache, leading to a larger WAF. Fig. 14 shows the impact of different ct_l values on WAF when ct_h is equal to 0.9, 0.7, and 0.5. It can be observed that when the values of ct_l and ct_h are large, the WAF of L−CoRW is $\times 2$ to $\times 3$ times higher than that of conventional SSD (MQ-SSD), which is obviously unacceptable. Therefore, we choose the most appropriate ct threshold ($ct_l = 0.3$ and $ct_h = 0.7$) to satisfy both tail latency and WAF.

Furthermore, we investigate the impact of updating Q-table frequency on L−CoRW. As observed in Fig. 15, there is no significant improvement in performance when updates are very frequent. This is because we divide each state into multiple bins to limit the total number of different states of the Q-table, resulting in only a limited number of optional values. Frequent updating of Q-table at this time does not allow L−CoRW to update the values in the threshold group. When Q-table updates too slowly, L−CoRW may miss the best optimisation time due to its inability to adapt to the rapid change of I/O workload, resulting in no performance improvement. Therefore, we set the update interval for Q-table to 0.2 second. Finally, we also analyze the impact of different SSD cache sizes, as shown in Fig. 16. We find that when the cache is small, the cache space resource becomes valuable at this time, so L−CoRW can have the opportunity to exploit the I/O locality as much as possible. When the cache is large, the performance of L−CoRW decreases because the SSD has enough cache space to hold more data. In summary, L−CoRW can maximize performance gains in the case of small SSD cache, which is in line with our design goal.

F. Implementation Overhead

The overhead of L−CoRW mainly comes from the space overhead of Q-table structure and the time overhead of RL model training and updating the *threshold group*, and we will analyze the impact of these two overheads in detail.

Space Overhead. The size (# of entries) of Q-table is # of states × # of actions. From the number of bins of each state and action in Section V-B, the total number of combinations is 216 (=27×8), and the Q-table size is 16byte×216=3456byte<4KB,



(a) Sensitive study of tt_l and tt_h (b) Sensitive study of rt_l and rt_h (c) Sensitive study of ct_l and ct_h

Fig. 13. Sensitive study of *threshold group* (tt, rt, ct) (all cases where $tt_l \geq tt_h$, $rt_l \geq rt_h$, and $ct_l \geq ct_h$ are replaced by 0, indicating that the purple area of the figure is meaningless).

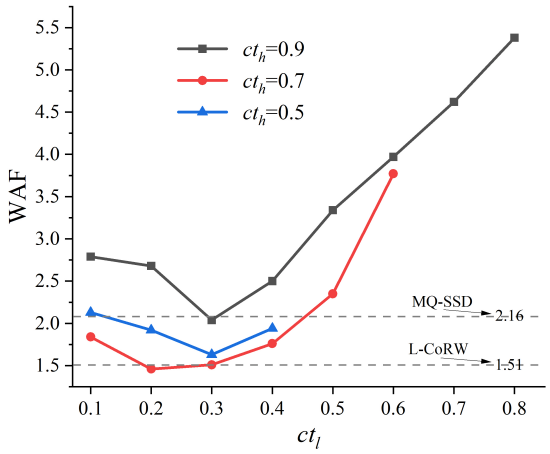


Fig. 14. The impact of ct_l and ct_h on WAF (there are missing parts of the curve due to $ct_l < ct_h$).

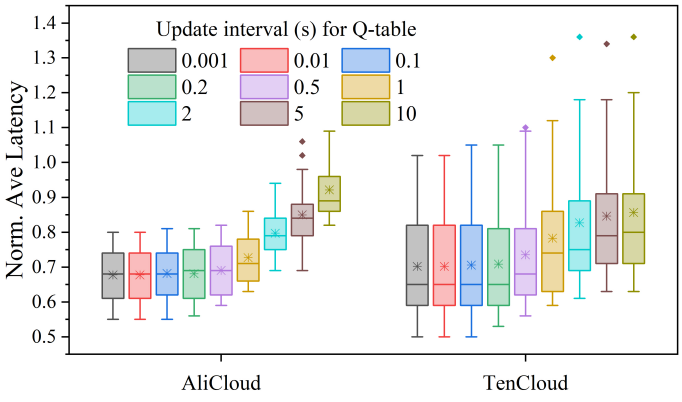


Fig. 15. Sensitive study of updating Q-table frequency.

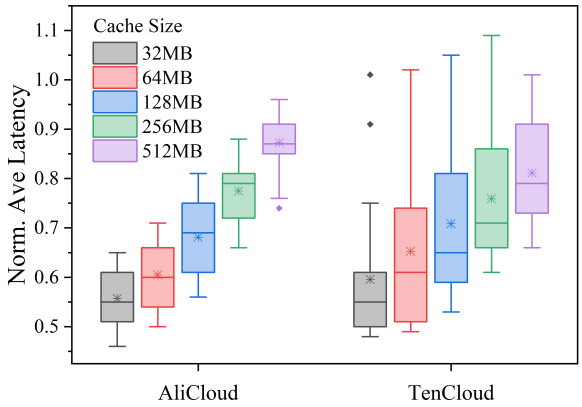


Fig. 16. Sensitive study of SSD cache sizes.

which is less than 0.003% of the cache spaces. The number 27 (=3×3×3) is the number of state combinations, 8 (=2×2×2) is the number of action combinations. Thus, the small Q-table can be cached in the internal DRAM of the SSD. Moreover, L−CoRW can be implemented in the firmware of a modern SSD, and does not require specialized hardware.

Time Overhead. Since updating the Q-table takes only 1.6ms every 0.2 second, the latency overhead is almost negligible (2.7ms/0.2s <2%). The time overhead of different policies are MQ-SSD : CFLRU : ECR : Co-Active : L−CoRW = 1 : 1.06 : 1.23 : 1.46 : **1.17**. Overall, considering the significant performance improvement of the L−CoRW on both latency and tail latency, the above small overhead is acceptable.

VII. Discussion

Two issues need to be considered during the actual deployment of L−CoRW. The first one is the impact on the cache persistence model. In fact, L−CoRW's design is entirely orthogonal to the cache persistence model, and L−CoRW does not modify the consistency mechanism of dirty data and metadata in SSDs. On the contrary, L−CoRW reduces the risk of data loss in the event of SSD power failure. This is because L−CoRW reduces the proportion of dirty data in the SSD cache. In traditional

SSDs, the write cache mode within modern SSDs is write-back mode by default [7], which means that in the worst-case scenario, all data in the current cache will be dirty. In contrast, due to the existence of the *ct* threshold, L−CoRW will always have a portion of clean pages in the cache, which will prevent L−CoRW from introducing additional dirty pages after a power outage. Therefore, the capacitance inside the existing SSD is sufficient for L−CoRW to flush dirty data to the underlying flash memory. In addition, L−CoRW requires only a limited number of parameters (threshold group) to run, and these parameters are not required to be persisted in flash memory. Therefore, the working mechanism of L−CoRW does not affect the data persistence and consistency in the cache. Another issue is that some studies [34], [35] have pointed out that traditional write-back can evict dirty data out of order (relative to the original write sequence), resulting in the networked storage being inconsistent after host-level failures. Network storage inconsistency compromises both data availability after a flash or host failure and the correctness of network storage level solutions such as replication and backup. However, since each module of L−CoRW does not involve network storage and host-side caching, the above issues do not exist for L−CoRW. If L−CoRW is deployed in ZNS SSD, because some functions need to be implemented in the host side memory, this issue needs to be further studied in future work.

Another explanation is why L−CoRW chose to use reinforcement learning instead of other machine learning or deep learning algorithms. There are several reasons for this. 1) RL-based cache management policy can learn to find the optimal solution without prior knowledge about I/O workload or SSD configuration. Therefore, it can be easily deployed and used in any SSD cache. 2) Traditional machine learning or deep learning algorithms can solve some of the above problems. Although these methods can have high accuracy, they require large datasets for training in offline mode. 3) Due to parameter updates requiring retraining, making specific rules that can effectively handle the various SSD configurations under dynamically changing user workloads is hard. There have been some studies using RL in real-world system environments [18].

VIII. Conclusion

In this paper, inspired by our analysis of diverse real enterprise scale workloads, we proposed an RL-based cache management policy for the SSDs. The proposed method dynamically coordinates read and write cache policy based on the I/O characteristics and the SSD cache status. Our extensive evaluations show that L−CoRW effectively improves system performance compared to state-of-the-art SSD cache policies. Further, we significantly demonstrate that L−CoRW improves the efficiency of write amplification during garbage collection and sensitivity.

IX. Acknowledgment

This work was supported by NSFC (61832020, 61821003, No. U22A2027). Fang Wang and Zhan Shi are the co-corresponding authors.

References

[1] Y. Zhang, K. Zhou, P. Huang, H. Wang, J. Hu, Y. Wang, Y. Ji, and B. Cheng, "A machine learning based write policy for ssd cache in cloud block storage," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1279–1282.

[2] H. Sun, S. Dai, J. Huang, and X. Qin, "Co-active: a workload-aware collaborative cache management scheme for nvme ssds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 6, pp. 1437–1451, 2021.

[3] S. Wu, Y. Lin, B. Mao, and H. Jiang, "Gcar: Garbage collection aware cache management with improved performance for flash-based ssds," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12.

[4] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "Cflru: a replacement algorithm for flash memory," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, 2006, pp. 234–241.

[5] C. Liu, M. Lv, Y. Pan, H. Chen, Y. Li, C. Li, and Y. Xu, "Lcr: Load-aware cache replacement algorithm for flash-based ssds," in *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 2018, pp. 1–10.

[6] H. Chen, Y. Pan, C. Li, and Y. Xu, "Ecr: Eviction-cost-aware cache management policy for page-level flash-based ssds," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 15, p. e5395, 2021.

[7] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "Mqsim: A framework for enabling realistic studies of modern multi-queue ssd devices," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 49–66.

[8] Q. Wei, C. Chen, and J. Yang, "Cbm: A cooperative buffer management for ssd," in *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2014, pp. 1–12.

[9] M. Huang, O. Serres, V. K. Narayana, T. El-Ghazawi, and G. Newby, "Efficient cache design for solid-state drives," in *Proceedings of the 7th ACM international conference on Computing frontiers*, 2010, pp. 41–50.

[10] H. Wang, P. Huang, S. He, K. Zhou, C. Li, and X. He, "A novel i/o scheduler for ssd with improved performance and lifetime," in *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2013, pp. 1–5.

[11] J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, R. Gao, X.-F. Cai, S. Maeng, and F.-H. Hsu, "Ftl design exploration in reconfigurable high-performance ssd for server applications," in *Proceedings of the 23rd international conference on Supercomputing*, 2009, pp. 338–349.

[12] C. Gao, L. Shi, C. J. Xue, C. Ji, J. Yang, and Y. Zhang, "Parallel all the time: Plane level parallelism exploration for high performance ssds," in *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2019, pp. 172–184.

[13] J. Li, Q. Wang, P. P. Lee, and C. Shi, "An in-depth analysis of cloud block storage workloads in large-scale production," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 37–47.

[14] Y. Zhang, P. Huang, K. Zhou, H. Wang, J. Hu, Y. Ji, and B. Cheng, "Osca: An online-model based cache allocation scheme in cloud block storage systems," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 785–798.

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

[16] S. Yoo and D. Shin, "Reinforcement learning-based slc cache technique for enhancing ssd write performance," in *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[17] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[18] J. Zhang, X. Li, X. Zhou, M. Yuan, Z. Cheng, K. Huang, and Y. Li, "L-qoco: learning to optimize cache capacity overloading in storage systems," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 379–384.

[19] Y. Zhou and K. Xiao, "Extracting prerequisite relations among concepts in wikipedia," in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–8.

[20] Y. Ren, X. Sun, K. Li, J. Lin, S. Feng, Z. Ren, J. Yin, and Z. Qi, "Dissecting the workload of cloud storage system," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2022, pp. 647–657.

[21] A. Jain and C. Lin, "Cache replacement policies," *Synthesis Lectures on Computer Architecture*, vol. 14, no. 1, pp. 1–87, 2019.

[22] X. Xiang, C. Ding, H. Luo, and B. Bao, "Hotl: a higher order theory of locality," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, 2013, pp. 343–356.
[23] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient mrc construction with shards," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 95–110.
[24] Y. Zhou, F. Wang, and D. Feng, "A disk failure prediction method based on active semi-supervised learning," *ACM Transactions on Storage*, vol. 18, no. 4, pp. 1–33, 2022.
[25] Available at https://github.com/CMU-SAFARI/MQSim, 2018.
[26] H. Sun, X. Cheng, C. Zhang, Y. Yue, and X. Qin, "Hipa: A hybrid load balancing method in ssds for improved parallelism performance," *Journal of Systems Architecture*, vol. 131, p. 102705, 2022.
[27] R. Nadig, M. Sadrosadati, H. Mao, N. M. Ghiasi, A. Tavakkol, J. Park, H. Sarbazi-Azad, J. G. Luna, and O. Mutlu, "Venice: Improving solid-state drive parallelism at low cost via conflict-free accesses," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–16.
[28] Y. Zhou, F. Wang, Z. Shi, D. Feng, and Y. Du, "Fair will go on: A collaboration-aware fairness scheme for nvme ssd in cloud storage system," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.
[29] Y. Li, P. P. Lee, and J. C. Lui, "Stochastic modeling of large-scale solid-state storage systems: Analysis, design tradeoffs and optimization," in *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*, 2013, pp. 179–190.
[30] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," *Acm Sigplan Notices*, vol. 44, no. 3, pp. 229–240, 2009.
[31] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 524–535.
[32] B. Li, W. Tong, J. Liu, D. Feng, Y. Feng, J. Qin, P. Li, and B. Liu, "Maximizing bandwidth management ftl based on read and write asymmetry of flash memory," in *Proceedings of the 36th International Conference on Massive Storage Systems and Technology (MSST'20)*, 2020.
[33] J. Li, Q. Wang, P. P. Lee, and C. Shi, "An in-depth comparative analysis of cloud block storage workloads: Findings and implications," *ACM Transactions on Storage*, vol. 19, no. 2, pp. 1–32, 2023.
[34] D. Qin, A. D. Brown, and A. Goel, "Reliable writeback for client-side flash caches," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 451–462.
[35] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 45–58.