The Static Allocation is Not a Static: Optimizing SSD Address Allocation Through Boosting Static Policy

Yang Zhou, Fang Wang, Zhan Shi, and Dan Feng, *Fellow, IEEE*

***Abstract*—The address allocation policy in SSD aims to translate the logical address of I/O requests into a physical address, and the static address allocation is widely used in modern SSD. Through extensive experiments, we find that there are significant differences in the utilization of SSD parallelism among different static address allocation policies. We also observe that the fixed address allocation design prevents SSDs from continuing to meet the challenges posed by cloud workloads and misses the possibility of further optimization. These situations stem from our excessive reliance on SSD parallelism over time. In this paper, we propose `HsaP`, a hybrid static address allocation policy, that adaptively chooses the best static allocation policy to meet the SSD performance at runtime. `HsaP` is a *dynamic* scheduling scheme based on *static* address allocation policy. The static policy ensures that `HsaP` has stable performance and light-weight overhead, while dynamic scheduling can effectively combine different allocation policies, selecting the best-performing static mapping mode for a given SSD state. Meanwhile, `HsaP` can further improve the read and write performance of SSDs simultaneously through plane reallocation and data rewrite. Experimental results show that `HsaP` achieves significant read and write performance gain of a wide range of the latest cloud block storage traces compared to several state-of-the-art address allocation approaches.**

***Index Terms*—Address allocation policy, block-level i/o, SSD parallelism.**

I. Introduction

NAND-FLASH based Solid-State Disks (SSDs) are gradually replacing traditional disks in enterprise storage systems. In addition to NAND-flash media with faster read and write operations, the flash translation layer (FTL) within the SSD is also key to improving performance. The FTL is to manage the mapping between logical page addresses (LPA) and physical page addresses (PPA). The procedure for allocating PPA according to LPA is referred to as an address allocation policy, which determines data layout and impacts SSD performance.

There are two groups of allocation policies, namely, static and dynamic policies. The static policy is based on the SSD parallelism resources (channel, chip, die, and plane) to get the PPA by the mathematical formula, while the dynamic policy assigns LPAs to any free parallelism resource. Currently, the channel-priority allocation policy has been proven to be optimal in most cases by many works [1], [2], [3], [4], [5]. One type of static policy-CWDP (channel/way/die/plane) has become the default algorithm within modern SSDs [3], [6], [7] due to having optimal sequential parallelism. In fact, each parallelism level has its advantages and limitations [8]. For example, PCWD (plane/channel/way/die) is an allocation policy to minimize parallelism, but it can reduce request collisions. Static policy is widely used in SSD because of its stable performance and low overhead.

Although address allocation in SSD is a classic topic, which has been well studied and is also mature in industrial products, however, in the recent *FAST* 2024 conference, Jun et al. point out that the static address allocation mechanism (CWDP) still adversely affects the performance of SSDs [9]. This is because CWDP maximizes bandwidth and throughput by allowing requests to be spread across different locations in flash memory, at the cost of more serious file fragmentation. To this end, they proposed an NVMe (Non-Volatile Memory Express) command extension that enables the file system to provide hints about the write operation to SSDs, as well as a novel CWDP mapping scheme considering the *hint*s for the FTL. Although their experiments and analyses reveal that the primary cause of the degraded performance stems from a significant increase in SSD die-level collisions, in addition to improving the address allocation policy at the SSD device level, it also needs to modify the relevant code of the filesystem and block layer to achieve it. Another paper, also presented at *FAST* 2024, did not directly optimize address mapping for SSDs, but still argued that there are inherent flaws in the static policy, which are reflected in the data remapping process [10]. These studies inspire us that *it's time to make some changes to static policies*.

In fact, we can solve the collision problem at the flash level by using a policy such as PCWD that is opposite to the characteristics of CWDP. PCWD avoids collisions between requests by mapping write requests to smaller flash memory units separately, although this comes at the cost of losing parallelism. In addition, we find that CWDP fails to perform optimally under burst I/O (see Section III-B2 in detail), which is similar to optimizing cache capacity overloading [11]. In this way, we hope to maximally utilize all available parallelism, so the SSD can serve as many I/O requests (throughput) as possible. On the other hand, the request latency should be maintained stably (i.e., low tail latency). It is a challenge

Manuscript received 12 March 2024; revised 15 May 2024; accepted 24 May 2024. Date of publication 30 May 2024; date of current version 7 June 2024. This work was supported by NSFC under Grant U22A2027, Grant 2023YFB4502701, Grant 61832020, and Grant 61821003. Recommended for acceptance by J. Lofstead. *(Fang Wang and Zhan Shi are the co-corresponding authors.} (Corresponding author: Fang Wang; Zhan Shi.)*

The authors are with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China (e-mail: zhouyang1024cs@hotmail.com; wangfang@hust.edu.cn; zshi@hust.edu.cn; dfeng@hust.edu.cn).

Digital Object Identifier 10.1109/TPDS.2024.3407367

1045-9219 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.

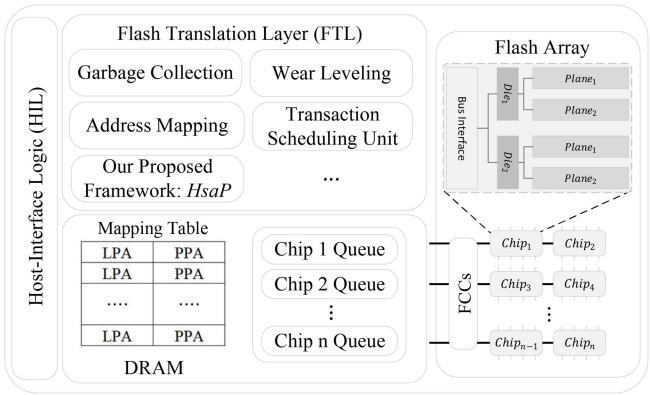


Fig. 1. The organization of SSDs.

to maximize all parallelism while preventing excessive tail latency.

In this study, we propose a hybrid static address allocation policy named HsaP, which fully exploits the advantages of different static allocation policies to enhance the adaptability and robustness in SSDs that a single static policy does not have. With the benefits of static policies, we hope that HsaP can further implement the dynamic trade-off of different static policies, thus obtaining better performance than dynamic address allocation policies. Furthermore, two key enabling techniques are introduced in HsaP to fully exploit both read and write parallelism within the hybrid policy. The first, *plane reallocation*, selectively allocates different block types to overcome the strict constraints of plane parallelism. The second, *data rewrite*, judiciously triggers data migration to effectively avoid the potential parallelism loss caused by hybrid scheduling by analyzing the request lifetime, without impacting the front-end user I/Os. In contrast to [9], which analyses it from a file system perspective, our work is done at the device level, although the reason for performance degradation is attributed to address mapping in the end. Our method avoids the need to modify all the modules in the I/O end-to-end path and focuses the coverage of the solution at the device level. In addition, the work of [9] is implemented on an emulator, which cannot reflect the real internal working mechanism of SSD. Our contributions in this work are listed as follows:

1) Through some preliminary studies in realistic enterprise-scale traces, we revisit the impact of different static policies on SSD parallelism. Through extensive benchmark experiments, we obtain four observations about traces and SSD parallelism, which motivates HsaP's design. We further analyze the reasons why CWDP can cause request collisions from a device level perspective.
2) According to the characteristics of the two static policies, we propose a hybrid static policy HsaP to adaptively and dynamically adjust different allocation policies. The proposed plane reallocation approach allows HsaP to fully exploit the last-level parallelism of SSDs for performance improvement by smartly satisfying the restrictions all the time. Combined with the data rewrite module, HsaP can simultaneously optimize the read and write performance of SSDs.
3) The experiment shows that HsaP is a light-weight address allocation policy and achieves encouraging performance and parallelism improvement of the SSD under various workloads.

The paper is organized as follows. Section II introduces the background, while Section III presents our motivation for this work. Section IV analyses the phenomena present in Section III. Section V presents the proposed HsaP policy. Sections VI and VII present the experimental setup and performance evaluation, with a discussion in Section VIII. We conclude in Section IX.

II. Background

A. Parallelism and Address Allocation

Modern SSDs usually contain host interface logic (HIL), flash translation layer (FTL), and flash chip array, as shown in Fig. 1. The HIL is responsible for dividing the requests in the waiting queue into sub-requests (also known as transactions) based on the physical page size. Then FTL contains various management units, such as address mapping, transaction scheduling, garbage collection, and wear leveling.[1] The underlying flash array offers multiple levels of parallelism: channel-, chip-, die-, and plane-level parallelisms, and the rest two levels are supported by advanced commands (*interleaving command* and *multi-plane command*). The *multi-plane command* has very strict read and write constraints, and only the same type of operations that access the same address in different planes can be processed in parallel.

SSD parallelism provides flexibility in the design of address allocation policies. Most existing SSD address allocation methods use static allocation, which employs a mathematical formula to convert an LPA to a PPA in flash memory. For example, the CWDP policy assigns new PPAs as follows:[2]

$$\begin{aligned} ID_{Channel} &= LPA\%N_{Channel} \\ ID_{Chip} &= (LPA/N_{Channel})\%N_{Chip} \\ ID_{Die} &= [LPA/(N_{Chip} \times N_{Channel})]\%N_{Die} \\ ID_{Plane} &= [LPA/(N_{Die} \times N_{Chip} \times N_{Channel})]\%N_{Plane} \end{aligned} \tag{1}$$

$N_{Channel}$ is the number of channels in the SSD, the rest is similar. Overall, there are 24 (4!) types of static allocation policies according to an SSD's four-level parallelism resources. This can be divided into four groups including *C-priority*,[3] *W-priority*, *D-priority*, and *P-priority*.

[1]Note that the flash firmware and internal DRAMs take half the cost of SSDs or more than that [12]. Many SSD vendors do not configure additional DRAM for SSD cache, so this paper does not consider optimization policies that require caching, which is consistent with the SSD configurations in [13].

[2]Inside a plane (block), blocks (pages) are programmed sequentially, and hence the block (page) ID can be deterministically assigned.

[3]CWDP, CWPD, CDWP, CDPW, CPWD, and CPDW

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.

TABLE I
BASIC INFORMATION OF ALICLOUD, TENCLOUD, AND MSR

	AliCloud	TenCloud	MSR
Source	Alibaba Cloud [17]	Tencent Cloud [18]	Microsoft [19]
#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

B. Cloud I/O Traces

We analyze the block-level I/O traces collected from two of the world's top five cloud providers over a long period, Alibaba Cloud and Tencent Cloud, as shown in Table I. These cloud service providers use virtualization technology to separate applications and underlying infrastructure, enabling them to exhibit I/O access characteristics that are different from the applications themselves. AliCloud and TenCloud comprise a variety of workloads (e.g., web services, key-value stores), and hence are representative to drive our analysis. The MSR traces, collected more than a decade ago in a data center composed of Microsoft's Windows servers, are still widely used in the field of SSD research [5], [13], [14], [15], [16]. The trace files record every I/O request issued by the cloud applications and each item of the trace file contains the *device_id*, *opcode*, *offset*, *length*, and *timestamp*.

Overall, AliCloud and TenCloud have much larger scales than MSR. AliCloud and TenCloud workloads are write-dominated due to the extensive use of read caches in cloud applications [20], while MSR workload is read-dominated. In addition, MSR is collected from traditional data centers, while AliCloud and TenCloud represent modern cloud environments, making them more representative.

III. MOTIVATION

In this section, we motivate our design by studying I/O characteristics and SSD performances of different configurations, which inspire us to propose `HsaP`.

A. Workload Analysis

As cloud platforms become increasingly popular, accurately understanding I/O behaviors in modern cloud storage is of paramount importance for system design and optimization. In this subsection, we conduct an in-depth comparative analysis of AliCloud, TenCloud, and MSR in two aspects: temporal patterns and I/O intensities, and make two key observations.

1) Temporal Access Patterns: In some research on system optimization by analyzing I/O characteristics, they usually pay more attention to the data locality [21], [22] or updating frequency [23], [24], which is a *first-order* feature oriented method. Considering the asymmetry of flash memory read and program, this first-order method is not suitable for SSD. We need a method that can reflect both locality and access frequency to characterize I/O, which guides us to optimize SSD. Following [25], [26], we examine four types of adjacent pages including read-after-read (RAR) page, read-after-write (RAW) page, write-after-write (WAW) page, and write-after-read (WAR) page. For example, RAR is the time interval between two consecutive reads to a page, and RAW is the time interval between a write and a subsequent read to the same page. The above method is a *second-order* combination of access time and access frequency so that it can be optimized for the deeper I/O characteristics of SSDs. We measure the time of four types as the elapsed time between the adjacent pages. Fig. 2 presents the cumulative distribution of the RAR, RAW, WAW, and WAR time intervals from 1 s to Inf for all I/O traces.

We notice that the page continues to read or write for a short time after the last read or write (RAR or WAW). For example, on average 60%-70% of the total overwrites or rereads occur within five minutes of the previous write or read, especially for AliCloud and MSR traces. In addition, we also notice that the written page is rarely or even not read in a short time. In TenCloud, only 10% of pages are RAW within 10 seconds, and this is almost rare for short periods of time in AliCloud and MSR. This is because the widespread use of cloud cache makes read requests (or hot data) triggered by the first-time writes absorbed by the upper cache, resulting in longer RAW times for the underlying storage. And WAR only represents an update of the data, which means a new write, we do not pay too much attention to it.

Furthermore, we count the percentage of the above four types of adjacent page, as shown in Fig. 3. We observe that WAW has the largest percentage in the write-dominant AliCloud and TenCloud. The second is RAR, which accounts for more than 80% of all types together with WAW. It has a similar conclusion on the production workload used in [20] (not open source). Apart from WAW and RAR, WAR and RAW also have non-negligible proportions and play important roles in our designs. For example, according to the access characteristics of RAW, we can distribute sub-requests in different parallel units on the first write, so that we can take advantage of the multiple parallel units when reading these data next time. In contrast, for WAW, we find that it is not optimal or necessary to distribute sub-requests across multiple parallel units (see Section III-B2 in detail). In a word, the first observation (Observation ❶) we make is that: the next read/write type of the same page will usually maintain the last read/write status, and for the new page just written, the probability of reading in a short time is low.

2) Burst I/O Characteristics: Burst I/Os are very common in many workloads, resulting in long-tail latency in SSDs. When the I/O traffic is light, SSD can increase concurrency to speed up the processing of sub-requests. However, when the I/O traffic is heavy, sub-requests are much likely to be blocked, which causes long latency. The deteriorating performance is attributed to the fact that the bandwidth of flash chips fails to handle burst I/O accesses swiftly. This is also related to the pattern of different address mapping policies. Ren et al. suggest that the bandwidth of more than ten times in peak workloads should be considered [20]. To further examine burst I/O duration and traffic in individual volumes, we categorise the burst I/O by bandwidth into four groups: **i**) more than 10 times average ($\geq$10), **ii**) more than 100 times average ($\geq$100), **iii**) more than 1000 times average ($\geq$1000), and **iv**) more than 10000 times average ($\geq$10000). We calculate the proportions for the four groups of

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.

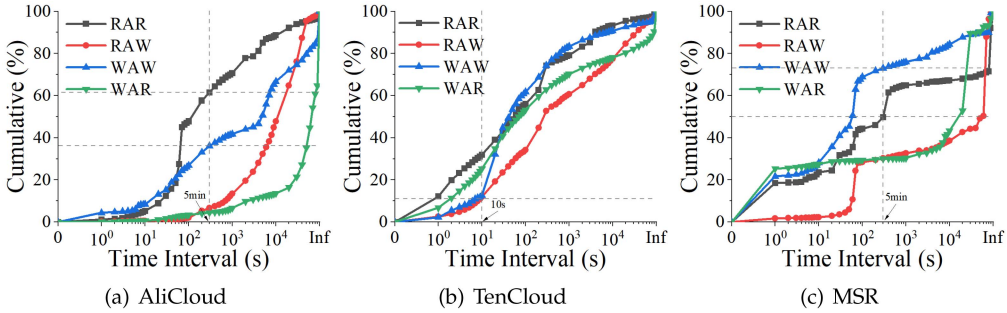


Fig. 2. Cumulative distributions of RAR, RAW, WAW, and WAR times across all traces.

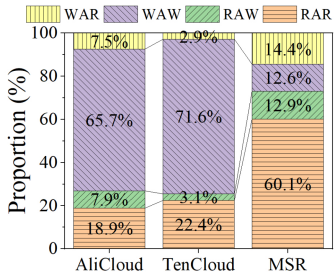


Fig. 3. Proportion of four types of adjacent pages.

TABLE II
PERCENTAGE OF TIME DURATION OF BURST I/O

	Read			Write		
	AliCloud	TenCloud	MSR	AliCloud	TenCloud	MSR
i	0.2133%	0.8891%	1.2273%	1.3473%	1.4400%	1.3598%
ii	0.0532%	0.0875%	0.1013%	0.0532%	0.0544%	0.0852%
iii	0.0078%	0.0101%	0.0108%	0.0035%	0.0016%	0.0045%
iv	0.0009%	0.0006%	0.0002%	0.0004%	<0.0001%	0.0002%

time duration and traffic for each volume, and represent the proportions across all volumes by boxplots in Fig. 4. Since the time duration of burst I/O for all traces is very short, we use Table II to show it directly.

Note that the duration of burst I/O is short in Table II, which is less than 1% in most cases. However, burst I/O contains a large proportion of traffic in Fig. 4. In read I/O, 75% of traces have more than 96.33%, 80.93%, and 91.12% of burst I/O traffic ($\geq$10) in AliCloud, TenCloud, and MSR, respectively (Fig. 4(a), (b), and (c)), while in write I/O, half of the volumes have more than 62.51%, 43.26%, and 73.76% of burst I/O traffic in these traces (Fig. 4(d), (e), and (f)). We draw a second observation (Observation ❷): SSDs in cloud environments usually receives a light I/O bandwidth most of the time, but a large amount of data traffic will arrive in the way of burst I/Os during a very short period.

B. SSD Parallelism

Rich SSD four-level parallelism is the key to improving SSD performance. In this subsection, we show via trace analysis that existing address allocation policies do not make full use of the four-level parallelism of SSDs, and investigate the influence of different address allocation policies on SSD parallelism. We also obtain two key observations from a series of benchmark experiments (see Section VI-A for the experimental setup).

1) How Many Plane Parallelism Can Be Achieved?: Since multiple planes can each carry out one operation in parallel, multi-plane parallelism only takes the time of one read, write, or erasure operation. However, strict constraints make it difficult for plane parallelism to occur frequently. We obtain the proportion of triggered transaction-level plane parallelism under different address allocation policies by replaying all traces and observe the impact of SSD physical page size on plane parallelism. Here we present the plane parallelism proportion for four representative policies in Fig. 5, including CWDP, WCDP, DCWP, and PCWD (representing *C-priority*, *W-priority*, *D-priority*, and *P-priority*, respectively).

It can be observed that as the physical page size increases (from 4 KB to 32 KB), the proportion of parallelism decreases. This is due to the decrease in the number of physical pages used when the total data traffic remains constant. Taking a typical 8 KB as an example, only about 2% of read transactions and write transactions trigger the plane parallelism of SSD. As QLC (quad-level-cell) flash becomes more popular, larger physical pages will also further reduce the proportion of plane parallelism, and this trend is significant as the physical page size increases. For example, the proportion of plane parallelism is less than 0.5% when the physical page size is 32 KB, which hardly uses the plane parallelism of SSD. At the same time, we also find that the *P-priority* PCWD policy can trigger more plane parallelism under the same conditions. For example, in AliCloud's read I/O, PCWD triggers a higher percentage of plane parallelism than the CWDP (WCDP and DCWP behave similarly to CWDP) 42.5%, 65.5%, 129.8%, and 173.2%, respectively, on physical pages ranging from 4 KB to 32 KB. While for write I/O, this is 3.6%, 5.1%, 31.1%, and 49.4%, respectively. The above result is related to the allocation principle of PCWD itself, i.e., planes are allocated in priority to channels, so that different sub-requests within I/O are allocated to different planes within the same die. We can get the same conclusion on the rest of the traces (especially for MSR). Therefore, we come to the third observation of SSD parallelism (Observation ❸): multi-plane parallelism is far from well utilized in SSD, and PCWD has the opportunity to achieve more multi-plane parallelism.

2) What are the Performance Differences Between Different Allocation Policies?: Different address allocation policies

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.

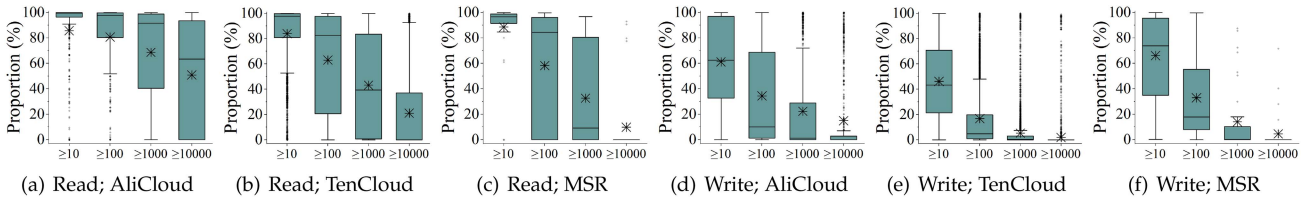


Fig. 4. Proportion for the four groups of burst I/O traffic across all traces.

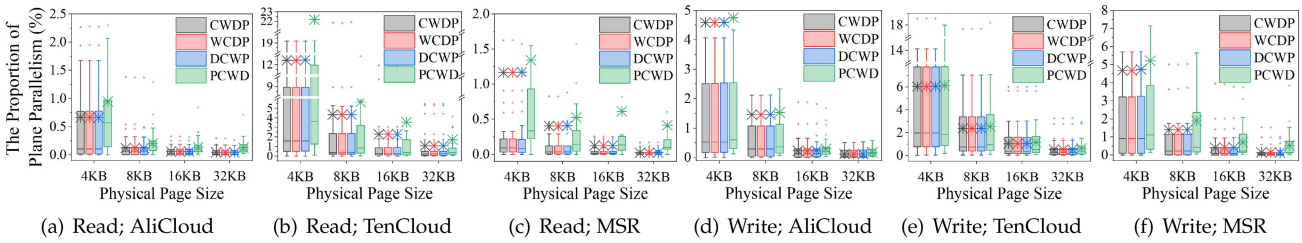


Fig. 5. Proportion of plane parallelism.

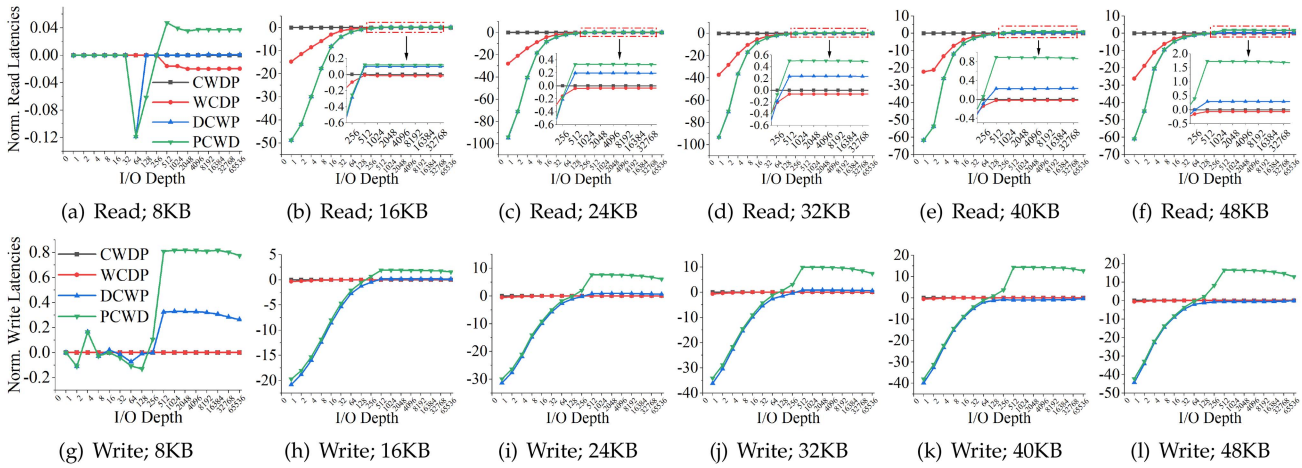
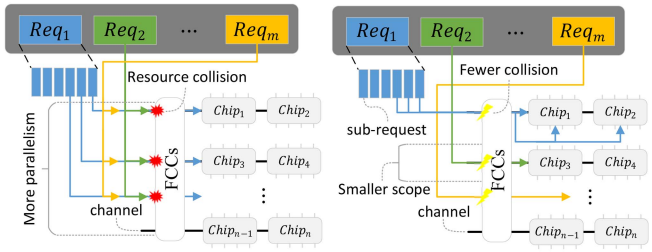


Fig. 6. The results of the normalized read and write latencies of the four policies by varying I/O depth and size. The performance is normalized to CWDP, and a positive value indicates #% better performance than CWDP (The CWDP curve will always be 0).

have different priorities in exploiting SSD parallelism. Existing researches confirms that *C-priority* static allocation policies can perform better in most cases due to better parallel utilization [1], [2], [3], [4], [5]. However, *is Higher Parallelism Always Beneficial?* Meanwhile, in the previous subsection, we also found that PCWD can achieve more plane parallelism, which means that *SSDs deployed with the PCWD policy will have better performance than other policies?* To this end, we run a 100%-random reads/writes synthetic flow where we vary the I/O intensity with a total I/O depth from 1 to 65536 (using powers of 2). I/O depth controls I/O intensity. We obtain the I/O latency for all 24 address allocation methods and also select the representative policies of four types of parallelism, including CWDP, WCDP, DCWP, and PCWD. For read requests, we test the read performance by writing data on the flash memory in advance according to the corresponding address allocation policy.

Fig. 6 shows how I/O intensity affects different address allocation policies. We find that the CWDP policy has the best performance when the I/O traffic is light (e.g., I/O depth < 64), both for read and write requests. This is because CWPD prioritizes sub-requests to different channels, and the chip on each channel is able to execute multiple sub-requests independently, thus maximizing the SSD parallelism. However, the PCWD can show better performance in handling burst write I/O when the I/O traffic is heavy (e.g., I/O depth > 256). And the performance improvement of SSD is more obvious as the I/O size increases. For example, when I/O depth is 512, PCWD improves 0.8%, 1.9%, 7.6%, 9.9%, 14.3%, and 16.5% compared to CWDP from 8 KB to 48 KB, respectively. In contrast, the burst read I/O has little impact on different allocation policies (<2%). The results are similar to other I/O performance metrics that we measured, including bandwidth, median latency, and tail latency, which we could not share in the paper due to limited space.

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.



(a) CWDP (b) PCWD

Fig. 7. An example of allocating resources for requests in the underlying flash memory.

We argue that although the channels can service I/O in parallel, CWDP will aggravate the resource collision between different sub-requests in the channel, resulting in sub-requests having to experience longer waiting times when the I/O traffic is heavy. In fact, Zhang et al.'s study [27] also show that CWDP not only fully utilize parallelism but also scatter data, which may result in more I/O interference. For example, the increased overhead of garbage collection and unfairness between users, *etc*. And PCWD can transfer collisions to a smaller scope (*P-priority* is less broad than *C-priority*) and maximize resource utilization, so it can achieve better performance. Therefore, the advantages and drawbacks of the two policies complement each other. Insight from [8], Fig. 7 shows an example of the resource allocation scenarios based on our analysis. This observation reveals a critical insight into the relationship between CWDP and PCWD in hybrid policy design. Here we get the fourth observation about SSD parallelism (Observation ❹): Using a fixed address allocation policy cannot guarantee the SSD performance with different I/O access patterns. As the I/O intensity increases, PCWD performs better than CWDP, especially for write I/O.

IV. Analysis

In this paper, we adopt CWDP and PCWD as two representative address allocation policies for SSDs, which are also the two policies with the largest performance difference observed in Fig. 6. Based on our observations, the crux of the problem is that resource collisions lead to longer queuing times for sub-requests, which in turn result in performance differences between different address allocation policies. We build a queuing model for the chip queue to find out the influencing factors.

Based on previous studies, such as [28], [29], it has been demonstrated that for an operation queue, it can be assumed that the arrival process of its data operations follows a *Poisson distribution*. Therefore, for simplicity, we use the *Poisson distribution* model to simulate the queuing process of data in the chip queue [30], i.e., $P(k) = \frac{\lambda^k e^{-\lambda}}{k!}$. In SSD, k represents the number of sub-requests queued in the chip queue, λ represents the average number of sub-requests queued, and $P(k)$ represents the probability that there are k sub-requests waiting in the queue. Upon the arrival of a request R at the SSD, it is divided into sub-requests r_1, r_2, to r_s (s is the number of sub-requests in a request) based on the physical page size. When using CWDP policy, the latency of R ($Lantency^R_{CWDP}$) depends on the slowest sub-request among all sub-requests (can be seen as *parallel*). At this time, the *expected value* of latency $E(Lantency^R_{CWDP})$ is

$$\begin{aligned} E(Lantency^R_{CWDP}) &= \lim_{n\to\infty}[1\times P(0) + 2\times P(1) + \cdots \cdots \\ &\quad + (1+n)\times P(n)] \times L \\ &= L \times \sum_{n=0}^{\infty}(1+n)\times P(n) \\ &= L \times \left[\sum_{n=0}^{\infty} P(n) + \sum_{n=0}^{\infty} nP(n)\right] \\ &= L \times \left[1 + \sum_{n=0}^{\infty} nP(n)\right] \end{aligned} \tag{2}$$

where L represents the operation time on the flash memory (all flash operations in the wait queue are of the same type). If the physical address is allocated according to the PCWD policy, $Lantency^R_{PCWD}$ depends on the first served sub-request, and the remaining sub-requests will be served immediately after the previous sub-request (can be seen as *serial*). In this case, the *expected value* $E(Lantency^R_{PCWD})$ is

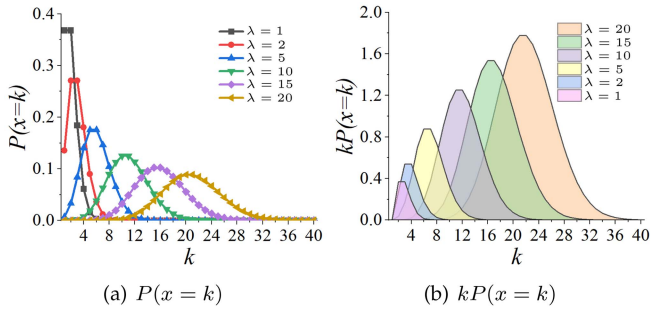
$$\begin{aligned} E(Lantency^R_{PCWD}) &= \lim_{n\to\infty}[s\times P(0) + (s+1)\times P(1) \\ &\quad + \cdots\cdots + (s+n)\times P(n)] \times L \\ &= L \times \sum_{n=0}^{\infty}(s+n)\times P(n) \\ &= L \times \left[s\times\sum_{n=0}^{\infty} P(n) + \sum_{n=0}^{\infty} nP(n)\right] \\ &= L \times \left[s + \sum_{n=0}^{\infty} nP(n)\right] \end{aligned} \tag{3}$$

When the number of sub-requests is 1 ($s = 1$), CWDP and PCWD get the same latency ($Lantency^R_{CWDP} = Lantency^R_{PCWD}$). If $s > 1$, we get $Lantency^R_{CWDP} < Lantency^R_{PCWD}$, which is satisfied when the I/O traffic is light. When the I/O traffic is heavy, the *Poisson distribution* corresponding to the two policies is not the same, i.e., $P_{CWDP}(k) \neq P_{PCWD}(k)$. In this case, the expected latency difference between the two policies on R is

$$\begin{aligned} &E(Lantency^R_{CWDP}) - E(Lantency^R_{PCWD}) \\ &= L \times \left[s - 1 + \sum_{n=0}^{\infty} nP_{CWDP}(n) - \sum_{n=0}^{\infty} nP_{PCWD}(n)\right] \end{aligned} \tag{4}$$

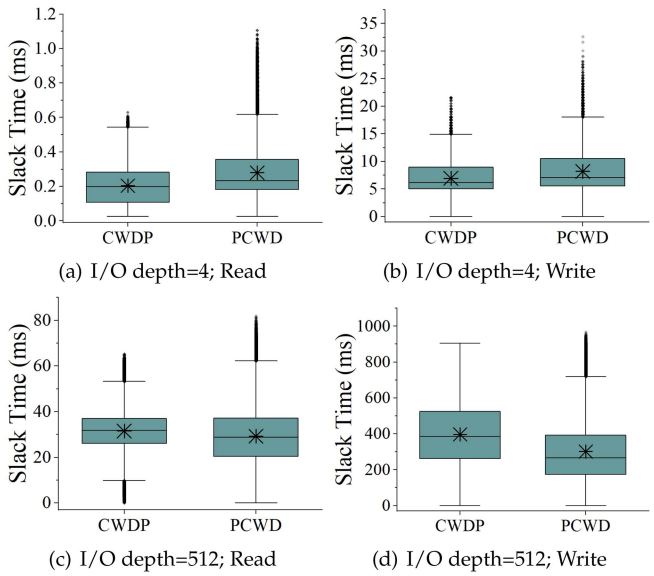
We know that the factor that affects the different Poisson distributions is λ, and a larger λ means that more sub-requests are queued in the chip queue, thus requiring longer wait times. Fig. 8 plots the changes in the Poisson distribution $P(k)$ and the composite function $kP(k)$ corresponding to different λ. Obviously, a larger λ also makes $\sum kP(k)$ larger ($\sum kP(k)$ represents the area under the curve in Fig. 8(b)). Since CWDP

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.



(a) $P(x = k)$ (b) $kP(x = k)$

Fig. 8. The change of $P(x = k)$ and $kP(x = k)$ with λ. The area of the region surrounded by the curve in Fig. 8(b) represents $\sum kP(k)$.



(a) I/O depth=4; Read (b) I/O depth=4; Write

(c) I/O depth=512; Read (d) I/O depth=512; Write

Fig. 9. The slack time distribution of CWDP and PCWD.

triggers more resource collisions under burst I/O, it leads to a serious queuing problem than PCWD, i.e., $\lambda_{CWDP} > \lambda_{PCWD}$. Furthermore, $\sum kP_{_{CWDP}}(k) > \sum kP_{_{PCWD}}(k)$, so $Lantency^{R}_{_{CWDP}} > Lantency^{R}_{_{PCWD}}$ (s is a fixed value). Next we experimentally confirm the possibility that $\lambda_{CWDP} > \lambda_{PCWD}$.

N. Elyasi et al. propose a concept to describe the parallel performance of SSD, namely *slack time* [31]. The slack time is obtained by accumulating the idle time due to waiting for unfinished sub-requests when sub-requests of the same request are working in parallel. This metric can display the parallel performance of the SSD in a numerical form, and directly reflect the difference in λ values between different policies. Fig. 9 shows the slack time distribution of CWDP and PCWD policies when both read and write requests are 48 KB. The heavy I/O traffic is generated when the I/O depth is 512, and the light I/O traffic is set to 4.

It is observed that when the I/O traffic is heavy, CWDP has a larger slack time than PCWD (especially for write requests, Fig. 9(d)). At this time, the waiting time of SSD sub-requests with CWDP policy is longer ($\lambda_{CWDP} > \lambda_{PCWD}$), resulting

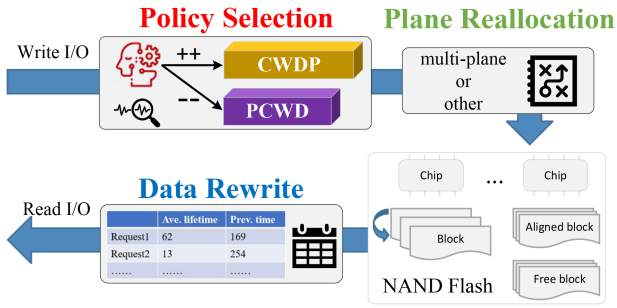


Fig. 10. Architecture of HsaP.

in longer latency than PCWD. When the I/O traffic is light, fewer resource collisions allow CWDP to make full use of SSD parallelism to make the slack time smaller, so the performance is better than PCWD.

V. Design

We design HsaP based on our observations in Section III. Fig. 10 depicts the high-level architecture of the HsaP system, a real-time load-aware and learning-based hybrid address allocation scheme deployed in the FTL layer. There are three primary modules at the heart of HsaP. First, the policy selection module selects the appropriate address allocation scheme for write requests based on SSD performance. This module uses learning and feedback mechanisms to find the optimal plan for write requests in two static address allocation policies (based on *Observation* ❷ and ❹). Second, the plane reallocation module maintains two active blocks for each plane, and the write pointer of one block is always at the same offset address as the write pointer of the block in the other plane under the same die, thus achieving multi-plane parallelism (based on **Observation ❸**). Third, the data rewrite module predicts the average lifetime of each request by recording the access time of the request, aiming at choosing the time to rewrite the data, so as to avoid unreasonable address allocation affecting the read performance (based on **Observation ❶**).

In fact, there are trade-offs and influences between the performance of different policies. Traditional adaptable control algorithms can solve some of the scheduling and allocation problems. However, the existing techniques use fixed design and do not update the relevant parameters of the cache at run time. Furthermore, it is hard to make certain rules that can effectively handle the various SSD configurations under dynamically changing I/O workloads. Consequently, we need a more intelligent algorithm to adjust the parameter considering the dynamically changing system states. We adopt a reinforcement learning-assisted hybrid policy which can learn how to find the optimal solution without any prior knowledge about I/O workload or SSD configuration. Reinforcement learning (RL) is valuable because of its inherent ability to adapt to the unknown dynamics of the system being learned, and it has been applied in various hybrid systems tasks, including hybrid SSD [32], [33], hybrid-gray-code encoding [34], and hybrid cache [35], [36] et al..

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: Hybrid Allocation Policy in HsaP.

Input: HIL_length threshold (hl_T), $allocation\_weight$ (aw)
1 **while** the address mapping of SSD is called **do**
2 **if** the number of write requests in HIL is less than hl_T **then**
3 HsaP uses CWDP policy // I/O traffic is light
4 **else**
5 A random number $rand \in [0, 1]$ is generated
6 **if** $rand \leq aw$ **then**
7 HsaP uses CWDP policy // $\in [0, aw]$ for CWDP
8 **else**
9 HsaP uses PCWD policy // $\in (aw, 1]$ for PCWD

Algorithm 2: Update Allocation Weight aw in HsaP.

Input: $target_{CWDP}$,$target_{PCWD}$, differential threshold ($diff_T$)
Output: Updated aw
1 **if** $\frac{|target_{CWDP} - target_{PCWD}|}{\min(target_{CWDP}, target_{PCWD})} > diff_T$ **then**
2 $reward = \ln(1 + |target_{CWDP} - target_{PCWD}|)$
3 /* +1 is to avoid negative infinitesimal of reward */
4 **if** $target_{CWDP} < target_{PCWD}$ **then**
5 $tmp = aw + \alpha * reward$
6 $aw = \frac{tmp}{tmp + aw}$ // Increase the probability of CWDP
7 **else**
8 $tmp = (1 - aw) + \alpha * reward$
9 $aw = 1 - \frac{tmp}{1 + tmp - aw}$ /* Decrease the probability of CWDP, $(1 - aw) = \frac{tmp}{tmp + (1 - aw)}$ */

A. Policy Selection Module

In HsaP, there are two address allocation policies, one is CWDP with *C-priority* and the other is PCWD with *P-priority*. HsaP needs to be both nimble and adaptive in choosing address allocation policies for different I/O access patterns. As shown in the previous section (**Observation ❹**), CWDP can improve read and write performance by taking advantage of SSD parallelism when the I/O traffic is light, while PCWD can improve write performance by maximizing resource utilization when the I/O traffic is heavy (the read performance improvement is limited). To this end, we adopt Q-Learning [37] but has been adjusted and improved to achieve self-learning and feedback in hybrid address allocation policy for HsaP. Our improvement is mainly in the ultra light-weight RL model that merges state, action, and reward into an integral model insight from [35], which is very suitable for time-critical and resource-constrained scenarios. In fact, one field similar to our scenario is the design of hybrid cache replacement algorithms [36]. There are two policies including LRU and LFU for replacing data out of cache in these works and choosing the right replacement policy is also the key to these works.

Algorithm 1 shows how HsaP selects the address allocation policy. Hybrid address allocation policy within HsaP performs this control with a threshold HIL_length (hl_T) and a weight $allocation\_weight$ (aw). The hl_T threshold controls whether to enable the scheduling of hybrid policy. When the number of write requests in the host interface is less than hl_T (I/O traffic is light), the address allocation policy of HsaP will adopt CWDP, thus exploiting the SSD parallelism. When the number of write requests is greater than hl_T, HsaP will enable hybrid scheduling to select CWDP or PCWD. The aw weight controls the probability of HsaP selecting different address allocation policies when the hybrid policy is enabled. Specifically, when a write request requires assigning a physical address, a random number $rand \in [0, 1]$ ($rand \in \mathbb{R}$) is generated. If $rand \leq aw$, HsaP uses CWDP policy for address allocation, otherwise the PCWD policy is used. The hl_T set to infinity or $aw = 1$ is the default option for modern SSDs. Using probability to select two policies randomly enables HsaP to have a certain probability of choosing any of these options, which also makes full use of the exploration & exploitation in RL. Many previous works [35], [36], [38], [39] rely on probability to determine the use of different policies in storage systems.

The hl_T is usually not set too small to avoid HsaP having a certain probability of using the PCWD when the I/O traffic is light, resulting in poor performance. The aw variable will be dynamically adjusted at runtime by the HsaP's performance and workload change. HsaP records the *ave. stack time* of requests in each period (e.g., $10ms$) after using CWDP and PCWD, and then combines the I/O intensity (IO_depth) to calculate the metric $target$ through (5) to reflect the performance of the two policies.

$$target = \frac{\sum ave.\,stack\,time}{number\,of\,requests \times IO\_depth} \tag{5}$$

The *ave. stack time* is based on the stack time divided by the number of sub-requests, which eliminates differences in request size. A larger $target$ means that the sub-requests wait longer for each other and therefore exhibits long request latency. HsaP will update aw based on the $target$ metrics of the two policies ($target_{CWDP}$ and $target_{PCWD}$), and the update process is described in Algorithm 2.

Because the requests have a wide range of stack time, HsaP will determine the difference between the policies based on the relative values of $target_{CWDP}$ and $target_{PCWD}$ (Line 1 in Algorithm 2). The adjustment process is triggered only when there is a large difference (larger than $diff_T$) between $target_{CWDP}$ and $target_{PCWD}$. If $target_{CWDP} > target_{PCWD}$, it means that the stack time between the corresponding sub-requests after the request has been allocated according to the PCWD policy is shorter and the performance of PCWD is better, so aw should be decreased (Line 9 in Algorithm 2), and vice versa. The $reward$ is the reward value in RL, which is used to update aw. The α is the learning rate. A large α leads to faster convergence but may get sub-optimal results. HsaP adjusts the α by 2% in each update.

We have already introduced that the Q-Learning algorithm used in HsaP is an improved light-weight version. Moreover, our improved version mainly refers to the process of Algorithm 2, which already includes adjusting and training the Q-Learning algorithm parameters. We eliminate the creation of Q-tables in Q-Learning, thus reducing space and time overheads. In fact, we can assume that the state is the $target$ metric and the action is the adjustment of the aw-value in Q-Learning. Q-Learning can

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.

be used by online training, so the training process is the update of aw by Algorithm 2.

In addition, several problems should be solved before the above hybrid scheduling process works. First, hl_T is a sensitive parameter. For a large hl_T, hybrid policy scheduling will rarely be triggered and the advantages of PCWD cannot be exploited over a wider range of I/O intervals. Meanwhile, a too small hl_T would misuse PCWD in some sensitive I/O intervals (PCWD performance is clearly poor when I/O traffic is light), which may impact the SSD performance. For example, the period of burst I/O is usually very short (**Observation ❷**), and when I/O traffic suddenly transitions from burst intervals to light intervals, aw may be relatively small. This leads to a situation that the address allocation policy will still choose PCWD and cause very poor SSD performance. To solve this issue, hl_T will be studied in the experiment section. Second, similar to the determination of hl_T, distinguishing the performance differences between CWDP and PCWD is also an important problem (based on threshold $diff_T$). When there is a significant performance difference between the two policies, aw should be updated and adjusted to match the new I/O and SSD states. However, if $diff_T$ is set too small, aw will be updated frequently, but the impact on SSD performance may be negligible. More importantly, in most cases, I/O traffic is light over long time intervals, so frequent updates are not necessary. In the experiment, a sensitive study will be presented.

B. Plane Reallocation Module

The deployment of PCWD creates more conditions for HsaP to meet plane parallelism (**Observation ❸**). To achieve plane parallelism, it is critical to ensure that the access addresses of writes on all planes in the same die are always aligned. Prior works have focused on increasing plane parallelism by proactively running GC on the remaining planes [40], as well as selecting dies that can accommodate multi-plane parallelism when allocating addresses [41] and implementing it through caching [5]. However, what all of these works do is to optimize plane parallelism passively. None of them is able to satisfy the strict restrictions all the time.

Traditional SSDs have only one appending point (open block) for SSD writes, and data pages with various lifetimes are very likely mixed in the same block. Insight from multiple active zones in the zoned namespace SSD (ZNS SSD), in HsaP, multiple active blocks is proposed to exploit the plane-level parallelism through two write pointers with different rules are opened in a block to satisfy the strict restrictions all the time. In order to achieve the goal, HsaP will maintain two active blocks in each plane, one of which maintains the same page offset address as the write pointer of the active block on the other plane within the same die. This active block can only use the advanced commands multi-plane command when performing write operations, which we call the *aligned block*. Another active block implements normal writes, which is used when the plane cannot achieve plane parallelism, and we call it a *free block*. Fig. 11 shows the difference between aligned blocks and free blocks. In addition, we also need to ensure that two sub-requests with the same die can be allocated to the aligned block when

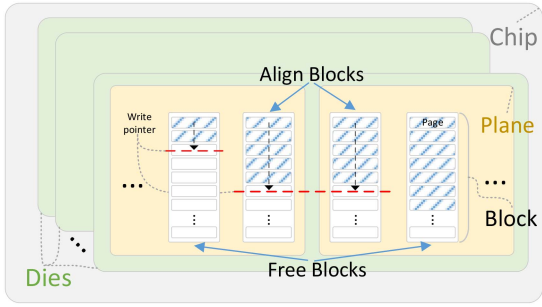


Fig. 11. Aligned block and free block in HsaP's flash array.

allocating the block address in the address mapping unit of SSD. Therefore, after allocating the address from channel to plane (following the CWDP or PCWD policy), HsaP will allocate the block address to the aligned block for the two sub-requests on different planes within the same die. The remaining sub-requests that cannot meet these conditions will be allocated to the free block. Through the above methods, HsaP can continuously realize plane parallelism.

C. Data Rewrite Module

There are three cases after data is written to flash memory, including no-operation, overwrite, and read (one or more times). Different address allocation policies affect not only the write performance but also the read performance. Based on **Observation ❹**, there is also a difference in the read performance of different address allocation policies. Especially when the I/O traffic is light, the data written by PCWD policy will have poor read performance. This is because data tend to be placed in fewer parallel resources (e.g., all sub-requests are aggregated within a channel or chip), reducing read parallelism for subsequent reads and deteriorating read performance. Therefore, HsaP needs to rewrite the frequently read data according to the CWDP policy to meet the read performance. HsaP creates a table structure for requests written by the PCWD policy. Each request requires a 4-byte structure to record its lifetime information, including the request's average lifetime ($avelife_{time}$) and the time of the last read ($prev_{time}$). The above information is updated whenever the request is read, and the update process is described in (6). It also needs to be updated when only the sub-requests are read, and the record is destroyed when it is overwrite.

$$avelife_{time} = (curr_{time} - prev_{time}) \times discount\_ratio + avelife_{time} \times (1 - discount\_ratio) \tag{6}$$

The $avelife_{time}$ on the right side of the equation is the average lifetime before updating, while the $curr_{time}$ and $prev_{time}$ indicates the current timestamp and the last access timestamp, respectively. The $discount\_ratio$ represents the discount rate of the current lifetime and the past lifetime. We use a $discount\_ratio$ of 0.8 for the new lifetime as it performs the best overall by quickly adapting to changing workload characteristics. The newly calculated average lifetime will be used as the expected lifetime of this read and be recorded as the new

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.

TABLE III
CONFIGURATION OF THE SIMULATED SSD

SSD Host Interface	PCIe 3.0 (NVMe 1.2)
SSD Capacity / Data Cache	512GB / DRAM-less
Page Size / Mapping Size	8KB / 8KB
Flash Communication Interface	ONFI 3.1 (NV-DDR2) Width: 8 bit, Rate: 333 MT/s
Flash Latencies (TLC)	Read latency: 100 μs Program latency: 1600 μs Erase latency: 3.8 ms
Channel / Chip / Die / Plane	8 / 4 / 2 / 2
Block / Page	2048 / 256
Flash Translation Layer	GC Policy: RGA [43] GC Threshold: 0.05 Address Mapping: DFTL [44] TSU Policy: Sprinkler [45] Over Provisioning Ratio: 0.07

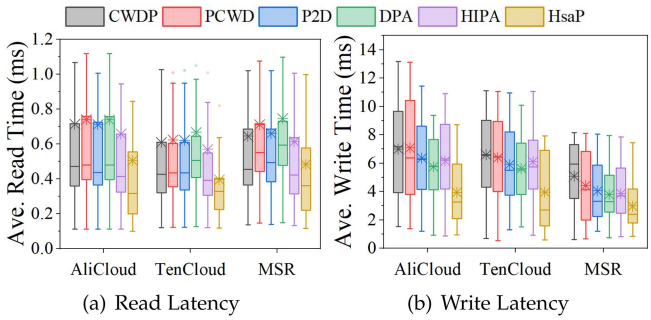


Fig. 12. Performance comparison for different policies.

history lifetime. To perform data rewrite, we need to predict when in the future a page written by the host will be read or overwritten. HsaP decides whether to start a rewrite after each average lifetime update, and the effectiveness of the data rewrite stems from the characteristics of RAR vs. RAW in **Observation ❶**. This allows us to isolate a time interval whose the probability of RAR is greater than the RAW to take advantage of the read performance of CWDP. Based on the findings of Fig. 2, we have specified rewrite intervals for the traces of three sources, including 10 to 10^3s, 10^2 to 10^3s, and 10 to 10^3s, respectively. The dynamic rewrite intervals design is beyond the scope of our paper and is left to future research. Once the updated average lifetime falls within the above range, the rewrite will be triggered to meet future read performance requirements. To avoid the impact of rewriting on the current user I/Os, it has a low priority.

VI. Experimental Settings

A. SSD Configuration and Traces

Following [13], [15], and [16] (the settings for these methods are almost the same), we use MQSim [7], an open-source multi-queue NVMe SSD simulator, to test the effectiveness of hybrid static address allocation policy. The detailed simulation parameters are tabulated in Table III based on Samsung PM9A3 SSD [42]. The workloads studied in this work include AliCloud, TenCloud, and MSR. These workloads have been introduced in Section II-B.

B. Schemes for Comparison

Five schemes are implemented to show the effectiveness of HsaP.

- CWDP: MQSim [7] is implemented to represent the traditional SSD design. The address allocation policy is CWDP, which has been proven to be one of the best static allocation policies in a prior study [1], [2], [3], [4], [5].
- PCWD: Similar to CWDP, the only difference is that the address allocation policy uses PCWD.
- P2D [9]: P2D is a page-to-die allocation policy designed to ensure that pages are allocated to contiguous dies based on their order in the request. For overwrites, the P2D sets a flag in the write I/O to indicate that the write operation is for overwriting an existing file block. By assigning the new page to the same die where the original flash page was located, P2D can achieve in-place updates of blocks in SSD.
- DPA [4]: DPA designates requests to idle planes according to the status of the channel and chip (idle or busy) in a round-robin manner, aiming to alleviate I/O load imbalance. This is a greedy dynamic allocation policy, which is implemented in the TSU.
- HIPA [16]: It is also a dynamic address allocation scheme for load balancing. By monitoring the load status in each die, HIPA dynamically allocates PPA for requests to achieve load balancing. We follow the suggestion of the author for HIPA with the die-level workload prediction mechanism using the number of write requests to gauge workload metrics.

VII. Performance Evaluation

A. Overall Performance

Fig. 12 shows the boxplots of per-volume average latency over all traces under read and write, respectively. For AliCloud/TenCloud/MSR, HsaP can reduce 32.7%/35.1%/29.0%, 32.8%/35.4%/29.1%, 31.5%/40.7%/35.2%, and 27.1%/33.4%/26.1% the average read latency compared with CWDP, P2D, DPA, and HIPA, respectively. And for write latency, HsaP is reduced by 43.7%/40.1%/41.4%, 37.2%/34.7%/31.6%, 35.5%/32.8%/30.6%, and 36.1%/35.4%/26.8%, respectively. Overall, our method achieves the lowest average latency and 75th percentiles for all comparison policies, confirming the effectiveness of our new hybrid static address allocation policy for HsaP. Besides, we make three key observations from the figure. First, CWDP performs better than PCWD in read performance. This is because CWDP can evenly distribute write data to different parallel units, thus maximizing the use of SSD parallelism when reading data. However, in terms of write performance, P2D can reduce collisions with other requests without losing parallelism, thus outperforming CWDP and PCWD. Second, DPA tends to improve write performance. In fact, DPA can achieve better load balancing when allocating addresses for write requests. However, this allocation method does not consider the read performance, which depends on the data layout in the underlying flash. Third, HIPA is similar to our approach, taking into account the features of both the freedom of dynamic policy

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.

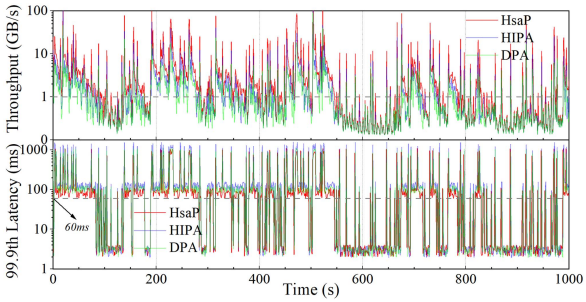


Fig. 13. The adaptability of HsaP and other policies.

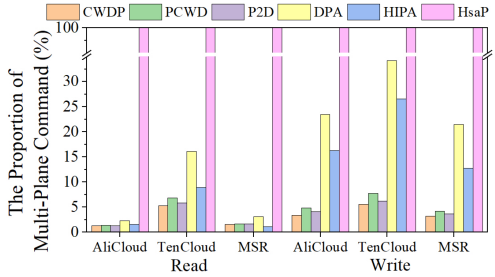


Fig. 14. Percentages of multi-plane command, normalized to that of HsaP.

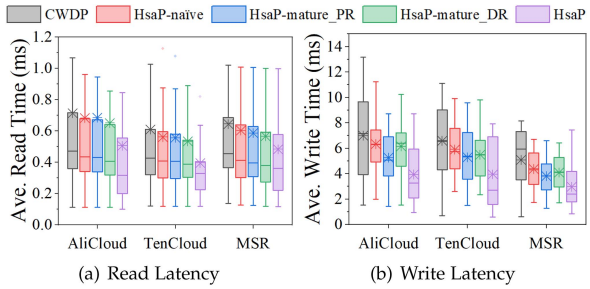


Fig. 15. Performance comparison for HsaP, CWDP, and three variants of HsaP to verify the effectiveness of our proposed module.

in write performance and the parallelism of static policy in read performance. These features allow HIPA to perform well in both read and write performance. However, HIPA is not optimized for plane parallelism in the dynamic policy. In addition, various mechanisms such as program or erase-pause make it difficult for HIPA to establish accurate queue prediction models, resulting in worse write performance than DPA and our approach.

B. Adaptability Comparison

To verify the adaptability of HsaP, we compare it with DPA and HIPA under the constantly changing workload. Fig. 13 shows how HsaP and the other two policies perform under a period of changing I/O traffic extracted from AliCloud. The results show that for the constantly-changing workload, the performance of HsaP exhibits higher throughput and lower tail latency (especially for $\geq 60\,ms$) compared to DPA and HIPA. We attribute this to the fact that HsaP can quickly adjust the probability of using different policies through self-learning and feedback from RL. This adjustment is not a step-by-step process in HsaP, but rather a pattern of large strides. Furthermore, we also find that our method performs similarly to several other dynamic policies in the light-bandwidth scenario ($< 1\,GB/sec$). In this case, HsaP degrades to CWDP. This shows the limited effect of dynamic policies in general scenarios and confirms the benefits of CWDP as a default allocation policy within SSDs. Although we only present the results for one of the traces, we did perform the same analysis on all remaining workloads, and they do not change our conclusions. Besides, HIPA performs better than DPA in the metrics of throughput, and DPA can outperform HIPA in tail latency. This means that a fully dynamic address allocation method is indeed more effective for load balancing.

C. Parallelism Performance

In Fig. 14, we collected the percentages of read and write operations processed by multi-plane command. Two observations can be concluded from the results: First, compared with other schemes, HsaP can improve the proportion of read and write operations in plane parallelism. On average, the percentages of multi-plane write are increased by 26.2×, 19×, 22.7×, 3.9×, and 5.9× compared with CWDP, PCWD, P2D, DPA, and HIPA in all traces. Second, the plane parallelism achieved on write requests also further enhances the plane parallelism of read requests. For instance, HsaP shows 52.8×, 48.2×, 50.1×, 27.3×, and 54.9× better multi-plane read for CWDP, PCWD, P2D, DPA, and HIPA. The reason is that HsaP will use the PCDW policy when allocating addresses for write requests, and the plane reallocation module also allows more write data to be distributed according to plane parallelism, thus providing more opportunities for read requests to trigger plane parallelism. In conclusion, the results show that the proposed plane reallocation module is able to maintain aligned write points for most write operations.

D. Effectiveness of Proposed Module

To verify the benefits of policy selection, plane reallocation, and data rewrite module, a comprehensive ablation evaluation has been performed. We split and combine the individual modules in HsaP to get three variants including

- HsaP-naïve: It only contains the policy selection module, which is an unconstrained hybrid scheduling model.
- HsaP-mature_PR: An upgraded version of HsaP-naïve that includes the plane reallocation module.
- HsaP-mature_DR: Another upgraded version of HsaP-naïve that includes the data rewrite module (plane reallocation module is not included).

We make a comparison between the three variants and our designed HsaP, and CWDP is treated as the baseline. As shown in Fig. 15, different modules have different biases in improving reading and writing performance. Although it is the most basic hybrid policy model of HsaP, the HsaP-naïve's performance is better than that of CWDP. This suggests that the hybrid policy has the full potential to perform better than CWDP or PCWD alone. In addition, HsaP-mature_PR is close to HsaP in write performance, while HsaP-mature_DR is similar to HsaP in read performance. This is because the plane reallocation module

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.

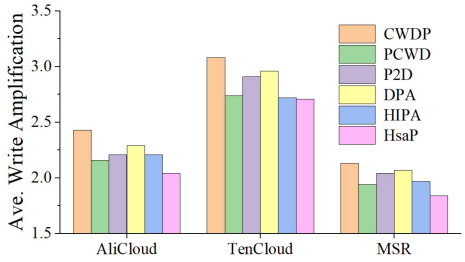


Fig. 16. Write amplification of different address allocation schemes.

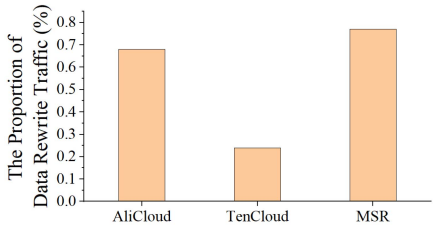


Fig. 17. The proportion of data rewrite traffic in HsaP to all traces traffic.

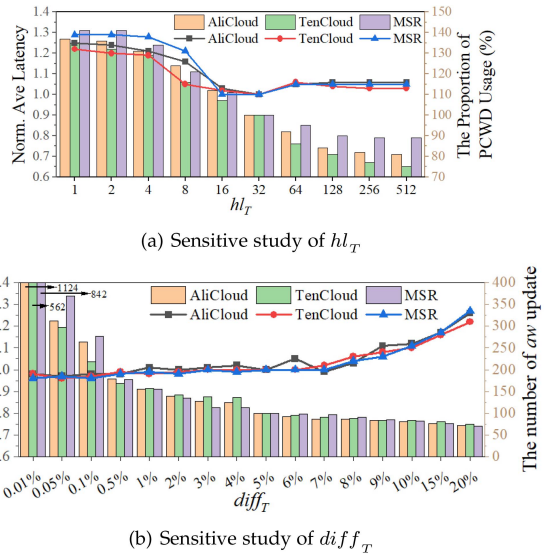


Fig. 18. Sensitive study of hl_T and $diff_T$. The line graph represents the normalized average latency, and the bar graph represents the proportion of PCWD usage in Fig. 18(a) and the normalized number of updates in Fig. 18(b). The results of Fig. 18(a) is normalized to $hl_T = 32$, and the results of Fig. 18(b) is normalized to $diff_T = 5\%$.

is mainly for write optimization, while the data rewrite is for read optimization. Although HsaP-mature_PR can also increase the proportion of read parallelism to some extent (Fig. 15(a)), since the read execution time is shorter than the write execution time, multi-plane read may not have noticeable performance gain. In summary, by including plane reallocation and the data rewrite module into HsaP-naïve, we can effectively increase the plane parallelism of write requests and further improve read performance.

E. Write Amplification

The layout of data in the underlying flash memory determines the read performance and the efficiency of garbage collection. We evaluate the write amplification of the six SSD optimization schemes after multiple consecutive writes on all traces. Write amplification plays an important role in measuring SSD overhead (small write amplification signifies low SSD overhead). Since HsaP performs data rewrite, we also count the size of the rewritten data as a percentage of all traffic. Fig. 16 shows the write amplification ratios of the address allocation schemes and Fig. 17 shows the proportion of additional traffic introduced by HsaP.

The results reveal that the write amplification of CWDP and DPA is much higher than those of the other schemes. This is because these policies tend to stripe data to different channels to increase parallelism. However, excessive parallelism inadvertently amplifies the garbage collection overhead due to the larger unit of space reclamation [27]. In contrast, PCWD clusters data into fewer parallel units by sacrificing parallelism, resulting in a smaller the write amplification. Further observation reveals that our method lowers write amplification of the other schemes. The reason is that HsaP will use the PCWD policy when I/O intensity is high, which means that most of the traffic is concentrated in fewer parallel units (high-intensity I/O traffic accounts for a very large proportion of the total traffic). In addition, the improvement of plane parallelism also makes garbage collection more efficient. This is why even though HsaP switches between CWDP and PCWD, its write amplification is lower than both. Fig. 17 reveals that the additional traffic caused by data rewrite in HsaP accounts for less than 1% of the total traffic. Overall, HsaP provides the lowest write amplification among the six address allocation schemes and well maintains the balance between performance and SSD lifetime.

F. Sensitivity Study

There are two parameters in HsaP, the hl_T threshold for triggering hybrid scheduling and the differential threshold $diff_T$ for distinguishing the performance of two static policies. To understand their characteristics, sensitive studies are presented to show their performance impact. hl_T and $diff_T$ are varied from 1 to 512 and 0% to 20%, respectively. Fig. 18 shows how hl_T and $diff_T$ affect the performance of HsaP. In order to more intuitively reflect the differences, all metrics are normalized to the suggested values we derived, i.e., results at $hl_T = 32$ and $diff_T = 5\%$. In addition, when we are testing the impact of hl_T, $diff_T$ is set to 5%. In contrast when testing $diff_T$, hl_T is 32.

In Fig. 18(a), the condition that triggers hybrid scheduling is higher by increasing hl_T. In this case, the proportion of PCWD used will gradually decrease and eventually stabilize. This is because in most cases, there is a significant difference in the traffic between burst I/O and normal I/O in these traces, so the usage of PCWD is also similar when hl_T is set to a large value (512, the maximum number of requests that can be accommodated in the HIL). This phenomenon is further reflected in latency, especially when hl_T is larger or smaller, respectively. And in Fig. 18(b), updating aw is more frequent as $diff_T$ continues to decrease. The reason is that a smaller $diff_T$ makes HsaP more sensitive to the performance differences between

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.

CWDP and PCWD, resulting in triggering more update aw. At the same time, we also find that frequent updates have less impact on latency, so there is no need to set the threshold too sensitive to trigger meaningless updates. In conclusion, hl_T should be set to an appropriate value to maximize the benefits of hybrid scheduling, while $diff_T$ should ensure that HsaP can achieve adaptive changes while reducing unnecessary parameter updates. For simplicity, hl_T and $diff_T$ are set to 32 and 5%, respectively.

G. Overhead Analysis

The overhead of HsaP mainly comes from the space overhead of table structure in data rewrite and the time overhead of model training, and we will analyze the impact of these two overheads in detail.

Space Overhead: In the data rewrite module, each request requires a 4-byte structure to record the information written to the flash memory in PCWD mode. The worst-case scenario assumes that all 512 G of capacity is written with PCWD write requests, while the average write request size is set to 24 KB. This requires a total of 85.3 MB extra space. In summary, the storage requirement is modest for modern SSDs.

Time Overhead: In HsaP, the two most time-consuming are computing cost and training time. Since updating the aw and other variables takes only 2.7 ms every second, the computing overhead is almost negligible ($2.7ms$/1 s $<1\%$). For training time, HsaP takes around 5 to 20 minutes for model training. Note that HsaP only needs training once and will subsequently be used and updated to implement scheduling between CWDP and PCWD. Such training time for a RL model is acceptable in SSD.

VIII. Discussion

Although we have seen the effectiveness of HsaP in the experimental section, there are still the following limitations. First, HsaP is somewhat parameter-dependent, which is reflected in the need to empirically set some critical parameters before deploying HsaP. For example, the data rewrite module of HsaP needs to set appropriate rewrite intervals, which may require relevant testing of the currently collected workload. Second, although the experiment confirms the relationship between CWDP and PCWD, we do not consider the combination of other address mapping schemes. This combination is not just a combination of two options, but can also be three or more. Finally, we only evaluate HsaP on the simulator, and further validation is needed for more realistic environment testing.

As for further work, we have several research directions to enhance the performance of HsaP. First, a more efficient and flexible scheduling scheme for CWDP and PCWD is key to improving HsaP performance. HsaP does not activate scheduling during periods of light I/O traffic, leaving room for further optimization. Second, HsaP can optimize and test flash devices in zone mode, such as ZNS SSDs. Finally, we can explore the integration of HsaP with other SSD optimization techniques and investigate its scalability and performance in larger-scale or more diverse storage environments [9], [10], [11], [20], [46].

IX. Conclusion

High-performance SSDs have always been a goal for researchers to follow. HsaP serves this goal by creating a new class of light-weight and adaptive, hybrid allocation policy. The hybrid framework provides a solution to find a trade-off between the two policies. Furthermore, HsaP also utilizes a plane reallocation method to enhance write performance while deploying data rewrite mechanisms to mitigate the impact on read performance. We demonstrate the effectiveness of HsaP by evaluating it against other state-of-the-art designs across various real-world workloads. We believe that static policies using hybrid frameworks can hold great promise and hope for future complex SSD systems.

References

[1] J.-Y. Shin et al., "FTL design exploration in reconfigurable high-performance ssd for server applications," in *Proc. 23 rd Int. Conf. Supercomput.*, 2009, pp. 338–349.

[2] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," in *Proc. Int. Conf. Supercomput.*, 2011, pp. 96–107.

[3] 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. 2012.

[4] A. Tavakkol, P. Mehrvarzy, M. Arjomand, and H. Sarbazi-Azad, "Performance evaluation of dynamic page allocation strategies in ssds," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 1, no. 2, pp. 1–33, 2016.

[5] 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 *Proc. 35th Symp. Mass Storage Syst. Technol.*, 2019, pp. 172–184.

[6] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "FlashSim: A simulator for NAND flash-based solid-state drives," in *Proc. 1st Int. Conf. Adv. Syst. Simul.*, 2009, pp. 125–131.

[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 *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 49–65.

[8] M. Jung and M. T. Kandemir, "An evaluation of different page allocation strategies on high-speed SSDs," in *Proc. HotStorage*, 2012, pp. 1–5.

[9] Y. Jun, S. Park, J.-U. Kang, S.-H. Kim, and E. Seo, "We ain't afraid of no file fragmentation: Causes and prevention of its performance impact on modern flash SSDS," in *22nd USENIX Conf. File Storage Technol.*, 2024, pp. 193–208.

[10] Z. Jiao, X. Zhang, H. Shin, J. Choi, and B. S. Kim, "The design and implementation of a capacity-variant storage system," in *Proc. 22nd USENIX Conf. File Storage Technol.*, 2024, pp. 159–176.

[11] J. Zhang et al., "L-QOCO: Learning to optimize cache capacity overloading in storage systems," in *Proc. ACM/IEEE 59th Des. Automat. Conf.*, 2022, pp. 379–384.

[12] H. Bae, J. Kim, M. Kwon, and M. Jung, "What you can't forget: Exploiting parallelism for zoned namespaces," in *Proc. 14th ACM Workshop Hot Topics Storage File Syst.*, 2022, pp. 79–85.

[13] A. Tavakkol et al., "FLIN: Enabling fairness and enhancing performance in modern NVME solid state drives," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 397–410.

[14] S. Wu, B. Mao, Y. Lin, and H. Jiang, "Improving performance for flash-based storage systems through GC-aware cache management," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2852–2865, Oct. 2017.

[15] H. Sun, S. Dai, J. Huang, and X. Qin, "Co-active: A workload-aware collaborative cache management scheme for NVME SSDs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 6, pp. 1437–1451, Jun. 2021.

[16] H. Sun, X. Cheng, C. Zhang, Y. Yue, and X. Qin, "HIPA: A hybrid load balancing method in SSDs for improved parallelism performance," *J. Syst. Archit.*, vol. 131, 2022, Art. no. 102705.

[17] Alibaba Cloud, 2020. [Online]. Available: https://github.com/alibaba/block-traces

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.

[18] Tencent Cloud, 2018. [Online]. Available: http://iotta.snia.org/traces/27917
[19] Microsoft, 2017. [Online]. Available: http://iotta.snia.org/traces/block-io/388
[20] Y. Ren et al., "Dissecting the workload of cloud storage system," in *Proc. IEEE 42nd Int. Conf. Distrib. Comput. Syst.*, 2022, pp. 647–657.
[21] Y. Luo, M. Lin, Y. Pan, and Z. Xu, "Dual locality-based flash translation layer for NAND flash-based consumer electronics," *IEEE Trans. Consum. Electron.*, vol. 68, no. 3, pp. 281–290, Aug. 2022.
[22] Z. Chen, G. Wang, Z. Shi, Y. Guan, and T. Wang, "Region-based flash caching with joint latency and lifetime optimization in hybrid SMR storage systems," in *Proc. Des. Automat. Test Eur. Conf. Exhib.*, 2023, pp. 1–6.
[23] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "AutoStream: Automatic stream management for multi-streamed SSDs," in *Proc. 10th ACM Int. Syst. Storage Conf.*, 2017, pp. 1–11.
[24] Z. Sha, Z. Cai, F. Trahay, J. Liao, and D. Yin, "Unifying temporal and spatial locality for cache management inside SSDs," in *Proc. Des. Automat. Test Eur. Conf. Exhib.*, 2022, pp. 891–896.
[25] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD lifetimes with disk-based write caches," *FAST*, vol. 10, pp. 101–114, 2010.
[26] J. Li, Q. Wang, P. P. Lee, and C. Shi, "An in-depth comparative analysis of cloud block storage workloads: Findings and implications," *ACM Trans. Storage*, vol. 19, no. 2, pp. 1–32, 2023.
[27] X. Zhang, S. Pei, J. Choi, and B. S. Kim, "Excessive SSD-internal parallelism considered harmful," in *Proc. 15th ACM Workshop Hot Topics Storage File Syst.*, 2023, pp. 65–72.
[28] R. W. Wolff, "Poisson arrivals see time averages," *Operations Res.*, vol. 30, no. 2, pp. 223–231, 1982.
[29] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proc. 38th Annu. Int. Symp. Comput. Archit.*, 2011, pp. 319–330.
[30] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.
[31] N. Elyasi, M. Arjomand, A. Sivasubramaniam, M. T. Kandemir, C. R. Das, and M. Jung, "Exploiting intra-request slack to improve SSD performance," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 375–388.
[32] S. Yoo and D. Shin, "Reinforcement learning-based SLC cache technique for enhancing SSD write performance," in *Proc. 12th USENIX Workshop Hot Topics Storage File Syst.*, 2020, pp. 1–7.
[33] Q. Wei, Y. Li, Z. Jia, M. Zhao, Z. Shen, and B. Li, "Reinforcement learning-assisted management for convertible SSDs," in *Proc. ACM/IEEE 60th Des. Automat. Conf.*, 2023, pp. 1–6.
[34] 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.*, 2023, pp. 122–136.
[35] G. Vietri et al., "Driving cache replacement with ML-based lecar," in *Proc. 10th USENIX Workshop Hot Topics Storage File Syst.*, 2018, pp. 1–6.
[36] L. V. Rodriguez et al., "Learning cache replacement with {CACHEUS}," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021, pp. 341–354.
[37] C. J. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, no. 3, pp. 279–292, 1992.
[38] Y. Zhou and K. Xiao, "Extracting prerequisite relations among concepts in wikipedia," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, 2019, pp. 1–8.
[39] K. Wu et al., "The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021, pp. 307–323.
[40] N. Shahidi, M. T. Kandemir, M. Arjomand, C. R. Das, M. Jung, and A. Sivasubramaniam, "Exploring the potentials of parallel garbage collection in SSDs for enterprise storage systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 561–572.
[41] A. Tavakkol, P. Mehrvarzy, and H. Sarbazi-Azad, "TBM: Twin block management policy to enhance the utilization of plane-level parallelism in SSDs," *IEEE Comput. Archit. Lett.*, vol. 15, no. 2, pp. 121–124, Feb. 2015.
[42] PM1725b SamsungEnterpriseSSD, 2022. [Online]. Available: https://semiconductor.samsung.com/ssd/enterprise-ssd/
[43] Y. Li, P. P. Lee, and J. C. Lui, "Stochastic modeling of large-scale solid-state storage systems: Analysis, design tradeoffs and optimization," in *Proc. ACM SIGMETRICS/Int. Conf. Meas. Model. Comput. Syst.*, 2013, pp. 179–190.
[44] 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.
[45] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 524–535.
[46] 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 *Proc. ACM/IEEE 60th Des. Automat. Conf.*, 2023, pp. 1–6.

Yang Zhou received the BE degree in computer science and information engineering from Hubei University (HUBU), Wuhan, China, in 2020. He is currently working toward the PhD degree majoring in computer science and technology in Huazhong University of Science and Technology (HUST), Wuhan, China. He is a professional membership of the China Computer Federation (CCF). His research interests include NAND Flash, dependable system, and storage system. He has more than 10 publications in major journals and international conferences as the first author, including IEEE-TC, IEEE-TPDS, ACM-ToS, ICAPS, IJCNN, DAC, MSST, ICPP, ICCD, MSN, NCSC, ICITIA, Application Research of Computers and Journal of Chongqing University, etc. He has served as reviewer for several ACM/IEEE international conferences. He has received many honors, including the China National Scholarship for Doctoral Students, the China Optics Valley Scholarship, and the AAAI ICAPS Student Scholarship Award, etc.

Fang Wang received the BE and master's degrees in computer science, in 1994, 1997, respectively, and the PhD degree in computer architecture from Huazhong University of Science and Technology (HUST), in 2001, China. She is a professor and PhD supervisor of computer science and engineering with HUST. Her interests include massive storage systems, parallel file systems, and non-volatile storage. She has more than 90 publications in major journals and conferences, including *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE/ACM Transactions on Networking*, TCAD, ACM-TACO, INFOCOM, ICDE, MSST, DATE, HPDC, ICDCS, ICPP, COLING, ICCD, IWQoS, etc.

Zhan Shi received the BS degree and master's degree in computer science, and the PhD degree in computer Engineering from Huazhong University of Science and Technology (HUST), China. He is working with the Huazhong University of Science and Technology (HUST) in China, and is an associate professor in Wuhan National Research Center for Optoelectronics. His research interests include distributed storage, cloud storage and graph storage. His papers have been published in major journals and international conferences, including ICDE, EuroSys, ICPP, ICDCS, CloudCom, CCGrid, SC, ICC, ACM ToS, IEEE-TSC, JPDC, FGCS.

Dan Feng (Fellow, IEEE) received the BE, ME, and PhD degrees in computer science and technology, in 1991, 1994, and 1997, respectively, from Huazhong University of Science and Technology (HUST), China. She is a professor, PhD supervisor, and dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, file systems, and non-volatile storage. She has more than 200 publications in major journals and international conferences, including *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *ACM Transactions on Storage*, JCST, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, ICPP, MSST, etc. She serves on the program committees of multiple international conferences, including SC 2011, 2013 and MSST 2012.

Authorized licensed use limited to: Jinan University. Downloaded on August 18,2025 at 10:23:23 UTC from IEEE Xplore. Restrictions apply.