Rep-RAID: An Integrated Approach to Optimizing Data Replication and Garbage Collection in RAID-Enabled SSDs

Jun Li, Balazs Gerofi, François Trahay, Zhigang Cai, Jianwei Liao

▶ **To cite this version:**

Jun Li, Balazs Gerofi, François Trahay, Zhigang Cai, Jianwei Liao. Rep-RAID: An Integrated Approach to Optimizing Data Replication and Garbage Collection in RAID-Enabled SSDs. 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2023), May 2023, Orlando (FL), United States. pp.99 - 110, 10.1145/3589610.3596274 . hal-04769843

HAL Id: hal-04769843
https://hal.science/hal-04769843v1

Submitted on 12 Nov 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Rep-RAID: An Integrated Approach to Optimizing Data Replication and Garbage Collection in RAID-Enabled SSDs

JUN LI, Southwest University, China National Institute of Informatics, the Graduate University for Advanced Studies, Japan, China
BALAZS GEROFI, Intel Corporation, USA
FRANCOIS TRAHAY, Telecom SudParis, France
ZHIGANG CAI, Southwest University, China
JIANWEI LIAO*, Southwest University, China

Redundant Array of Independent Disks (RAID) technology has been recently introduced to flash memory based SSDs to enhance their data reliability. Although RAID increases reliability, it doubles the number of write operations and requires additional parity computation as every write operation on a data chunk leads to another update on the corresponding parity chunk. Data replication has been proposed to mitigate the overhead of write requests in RAID enabled SSDs, however, replication increases the cost of garbage collection (GC), which in turn limits the improvement of I/O performance compared to the baseline RAID implementation.

This paper introduces *Rep-RAID*, an improved data replication management scheme accompanied with optimized GC for RAID-enabled SSDs. Guided by a mathematical model, *Rep-RAID* only replicates frequently updated data chunks. Furthermore, *Rep-RAID* reorganizes new data stripes during the GC process by utilizing replicated data to replace invalid data chunks caused by data replication in old stripes. As a result, it decreases I/O latency for both read and write requests and significantly reduces the GC overhead induced by data movement. Experimental results show that the proposed scheme can improve I/O performance by 16.7%, and reduce tail latency by up to 17.9% at the 99.99th percentile, when compared to the state-of-the-art RAID-enabled SSDs.

CCS Concepts: • **Computer systems organization → Embedded software**.

Additional Key Words and Phrases: SSDs, RAID-5, Replication, Garbage Collection, Stripe Reorganization.

1 INTRODUCTION

NAND Flash memory-based SSDs have become the dominant storage devices due to their attributes of small size, high energy efficiency, low latency, and collectively massive parallelism [1, 2]. In order to cut down unit price, the cell density of flash has been increased from SLC (1 bit/cell) to MLC (2 bit/cell), TLC (3 bit/cell), and even to QLC (4 bit/cell). On the other hand, high density SSDs are prone to errors caused by disturbs [3]. Error correction codes (ECCs) (e.g., low density parity code) have been applied to SSDs for correcting read errors and for preventing uncorrectable bit errors [4].

As ECCs cannot recover corrupted data from device-level failures, the Redundant Array of Independent Disks (e.g., RAID-5) technology has been recently applied to SSDs [5], or called Redundant Array of Independent NAND (RAIN) [6]. The vendors have supported RAID technology in SSD products [7, 8] and several researches have studied RAID-enabled SSDs to optimize access performance and endurance [9–12]. Specifically, *RAID-5* organizes the data as stripes, where each stripe consists of N data chunks and 1 parity chunk that is XORed with the corresponding data chunks. Although enabling *RAID-5* increases SSD reliability, it doubles the number of writes and causes additional XOR computations, as every write on a data chunk leads to another update on the corresponding parity chunk[1] (termed as *write penalty*) [13].

*Corresponding author, e-mail: liaotoad@gmail.com.

[1]A (data/parity) chunk is normally referred to as a page in RAID-enabled SSDs. Namely, the size of a chunk is equal to that of a page.

More importantly, each SSD block can only bear a limited number of erase operations, and the extra parity updates cause unexpected erases which impacts the endurance of the SSD device [14]. Furthermore, the XOR computations for generating parity chunks occupy the main controller of the SSD and thus delay the scheduling of normal I/O requests. In order to mitigate *write penalty*, RAID implementations are often accompanied by a DRAM buffer to absorb overwrites or writes to a single data chunk [9, 15, 16]. Certain data updates can be fulfilled in the buffer, to avoid direct data and parity writes in SSD blocks [17]. As a consequence, I/O responsiveness can be improved and SSD lifetime can be extended. On the other hand, DRAM is expensive, which limits the size of the DRAM buffer inside SSDs [18], rendering parity updates on the SSD pages inevitable.

Due to the excess writes in RAID systems, the average and tail latency of I/O operations indicate performance degradation compared to raw SSDs without a RAID module [20]. The software overhead of the RAID controller becomes a major component in the latency of write requests, which may postpone I/O scheduling and thus affect I/O responsiveness [24]. Thus, mitigating the impact of servicing write operations in RAID-enabled SSDs is a major concern [25].

Recently, introducing replication functionality inside RAID SSDs or SSD RAID [24, 25] has been proposed. These designs store the data of small requests temporally in the underlying flash cells and mirroring them into multiple cells to ensure data reliability. Such approaches can avoid certain XOR computations and also reduce software overhead caused by updating RAID stripes in I/O intensive periods [24]. When the storage system becomes idle, it periodically converts the replicated data into RAID stripes. However, such replication scheme introduces additional page moves during garbage collection (GC), as dirty (obsolete) data chunks must be retained in the original stripe to protect the rest of the data chunks. Consequently, the large number of obsolete data chunks heavily affect GC, adversely impacting the tail latency of I/O requests of application [28].

To address the aforementioned issue that alleviates tail latency and improves I/O performance, we propose an integrated approach of a model-based replication method for boosting I/O performance of RAID-enabled SSD system, and a novel stripe reorganization scheme for decreasing the number of dirty (obsolete) data chunks that are only held to protect the rest data chunks in the original stripe. In brief, this paper makes the following three contributions:

- We propose a model-based data replication scheme that directs the replication of frequently updated data chunks. In particular, we construct a cost-evaluation model to determine whether or not making a replica is favorable by comparing its benefits with the overhead of garbage collection in RAID-enabled SSDs. Moreover, to improve the flexibility of the model, we adjust the update frequency threshold based on the I/O features of user applications.
- We propose a stripe reorganization scheme to minimize the overhead of page moves in GC operations. The scheme reorganizes a new data stripe by coupling the cold replicated data with the valid data chunks in existing stripes when performing garbage collection. Consequently, the number of page moves in GC can be greatly reduced, as certain data chunks no longer need to be migrated when their (cold) replicas have been reorganized as new stripes.
- We present a comprehensive evaluation on several disk traces of real-world applications. The experimental results show that our method can not only cut down the overall I/O time by `14.7%`, but it also decreases the number of page moves by `20.9%` during GC process and thus reduces tail latency by `4.1%` at the `99.99`th percentile on average, compared to state-of-the-art methods.

2 BACKGROUND AND MOTIVATION

2.1 Background Knowledge

Figure 1 illustrates an example of GC process. It first selects the victim SSD block that has the least number of valid pages (e.g. $Block_i$), by following the greedy policy [38]. Then it migrates all valid

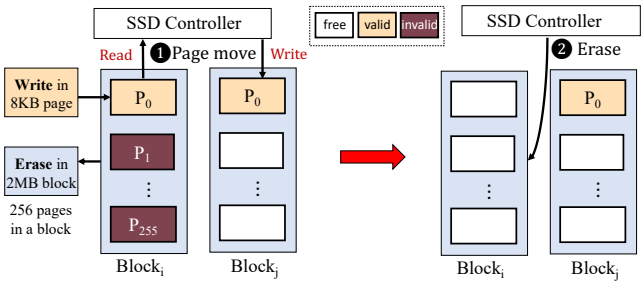

Fig. 1. The work flow of GC process in an SSD device. In which the data page size is 8KB [26] and each block holds 256 pages [27].

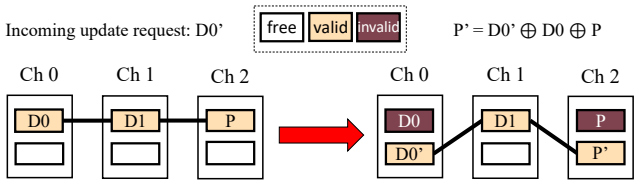

Fig. 2. The illustration of stripe update process in an RAID-enabled SSD device. Note that we use three channels (`Ch0`–`Ch2`) and each data stripe has two data chunks and one parity chunk, for simplicity of illustration.

pages in the GC block to another available block (e.g., the data of P_0 in $Block_i$ is moved to the free page in $Block_j$). Finally, the victim block of $Block_i$ is erased to reclaim space. The issue is, however, that I/O requests targeting at the same channel cannot be served when performing GC, which places negative effects on I/O responsiveness of incoming requests [22, 23, 33].

On the other hand, while the increasing cell density of flash memory chip cuts down the unit price, it brings more bit errors and poses increased threats to data reliability. Error correction codes (ECCs) are widely used to deal with read errors and to ensure data correctness, but they cannot help in chip/channel/SSD-level errors [10, 25].

In response to the above limitations, RAID has been successfully applied in SSD devices, and many corrective RAID implementations have been proposed [9, 17]. In principle, all (data/parity) chunks that belong to the same data stripe will be distributed across all associated chips/channels of the SSD device. In case a chip/channel becomes unavailable, the device can still restore all the lost data/parity chunks by reading other relevant RAID components accompanying with certain XOR computations.

When an update request comes, the corresponding stripe needs updating both data chunk and parity chunk on their original RAID channels. As the example illustrated in Figure 2, besides renewing the data chunk to $D0'$, it computes the newest parity chunk of P', through XORing the updated data chunk of $D0'$, the old data chunk of $D0$, and the old parity chunk of P.

2.2 Replication-Based RAID SSD

Because all updates on data chunks and parity chunks have to be completed on their original RAID components, which may cause imbalanced I/O workloads across all RAID components [25]. Figure 3 shows the difference of write latency across channels in conventional RAID5-enabled SSDs. As seen, the tail latency of write latency unveils significant dissimilarity among all RAID channels. We also recorded the standard deviation of tail latencies at `99.99`th percentile in multiple channels.

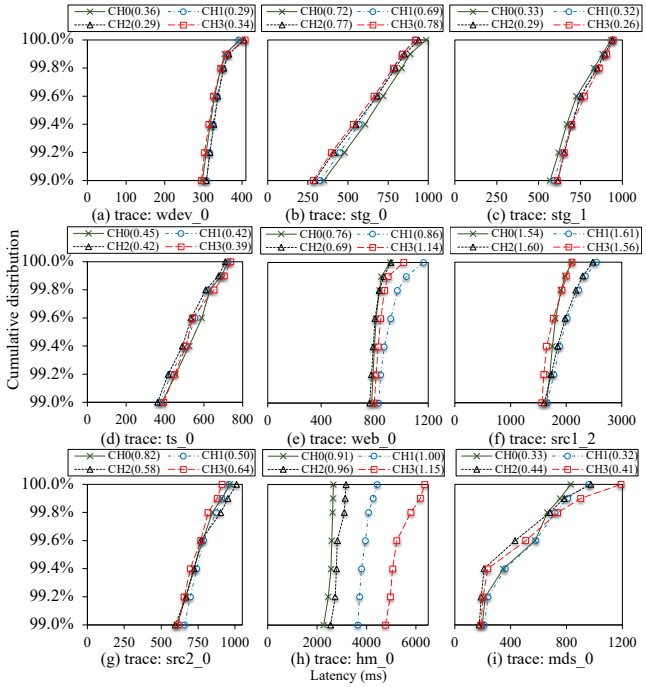

Fig. 3. CDF of write latency on multiple channels of the same stripes in conventional RAID5-enabled SSDs. The unit of write in this figure is the size of page. The number of channel written chunks is shown in the legend (unit: 10^6).

It shows that the values of standard deviation of tail latencies at the `99.99`th percentile are ranged between `7.0` and `1644.2`, proving that the latency dissimilarity of multiple channels exists in certain application workloads. We argue that the frequently updated chunks congest their located channels, especially when the I/O accesses are extremely intensive, thus significantly impacting the tail latency. Furthermore, it has been verified that the software and XOR overhead account for a major part of write latency when updating a data stripe in SSD-based RAID systems [24].

With respect to the aforementioned issues, replication-based RAID implementations, *CR5M* [25] and *FusionRAID* [24], have been proposed in both RAID-enabled SSDs and SSD RAID systems. In which, *CR5M* prefers replicating the data chunk in a dedicated chip of the same channel associating with the original data chunk, meanwhile *FusionRAID* distributes the replicated data chunks into multiple units with the highest parallelism that can benefit both write and read. More specifically, replication-based RAID does not renew the stripe immediately for servicing **small requests**. It temporarily organizes the updated chunks as replication pairs, for avoiding from exacerbating I/O intensity, as well as reducing the software and XOR overhead.

Figure 4 shows an example of data replication in RAID-enabled SSD systems, in which the stripe consists of data chunks (*D0*, *D1*) and a parity chunk *P*. When the update request of *D0* arrives, the newest data chunk is selected to make replicas, i.e., *R0* and *R0'* in two independent channels that are the highest parallel unit in SSDs, and then the old data chunk *D0* will be labelled as `semi-invalid`, in both *CR5M* and *FusionRAID* implementations.

For recording physical addresses of the replication data and the `semi-invalid` data, a replica table is introduced in such RAID-enabled SSDs. The replica table holds the old physical page address (*OPPN*) and replica physical page address (*RPPN*). When a data chunk is replicated, the replica address (*RA*) will be pointed to the position of relevant entry in the replica table. As the example

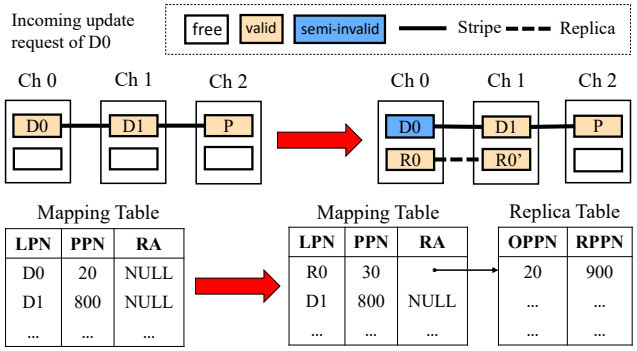

Fig. 4. Data replication and mapping table illustration in channel-level RAID-enabled SSD storage [24, 25]. In which, it creates a mirror pair of $R0$ and $R0'$ for servicing the update request on the data chunk of $D0$. Then, the obsolete chunk of $D0$ is marked as `semi-invalid` to protect the original data stripe, though it is not the latest.

shown in Figure 4, the *RA* of *D0* is NULL, and indicates the data chunk of *D0* does not have any replica. If the data chunk of *D0* is replicated for the first time, a new entry consists of *OPPN* and *RPPN* will be appended to the replica table. In addition, the physical page address (*PPN*) of *D0* is copied to *OPPN*, since this obsolete data chunk is still a component of the stripe of (*D0*, *D1*, *P*), and should not be reclaimed in the process of garbage collection. Once the replicated data chunk is updated again, only the information of *PPN* and *RPPN* should be renewed, and the relevant entry in the replica table will be removed if the data chunk is transferred into the RAID format.

2.3 Motivation

Enabling the replication functionality in RAID systems can boost the I/O performance and achieve a balanced workload across all RAID components. However, existing replication-enabled RAID implementations only consider the size information of update request, and fail to take other natures of data chunks, such as update frequency into account. As discussed, the frequency difference of data chunks induces imbalanced data access and software overhead over all RAID components, and thus degraded tail latency.

More importantly, we understand `semi-invalid` chunks are not up-to-date, but they are still useful for protecting other chunks in the same stripe. Thus, we need cope with them in GC processes that must increase the GC overhead and thus impact I/O processing. In order to disclose how many negative effects in GC processes, caused by migrating `semi-valid` chunks in replication-enabled RAID implementations in SSD contexts, we performed an experimental study. In the study, we compare the conventional RAID-5 implementation with two existing replication-enabled RAID implementations of *CR5M* and *FusionRAID*.

Figures 5(a) and 5(b) show the number of page moves during GC and its overall runtime overhead, respectively. As illustrated, *CR5M* and *FusionRAID* yield more page moves by over than `4.4X` and `2.5X` in contrast to *RAID-5*, which is the main cause of the increase in GC overhead, as shown in Figure 5(b). In other words, increased page move operations delay the processing of regular application I/O requests [28].

Such observations drive us to propose a new replication policy by considering the update frequency of data chunks to boost replication efficiency, as well as a new stripe re-organization scheme to decrease the number of `semi-valid` data chunks during GC processes, for replication-enabled RAID implementations in SSD scenarios.

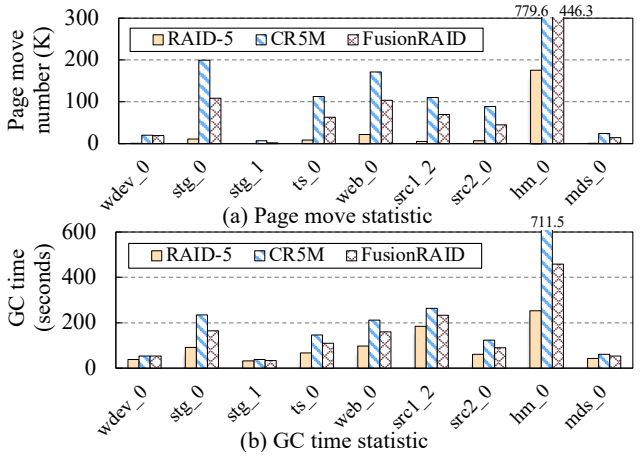

Fig. 5. The increased page moves and GC time in two typical replication-enabled RAID implementations of *CR5M* and *FusionRAID*.

3 DESIGN AND IMPLEMENTATION OF *REP-RAID*

3.1 Overview of *Rep-RAID*

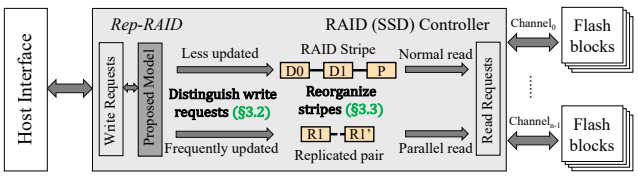

Fig. 6. Architectural overview on *Rep-RAID*.

We propose a new replication scheme for RAID-enabled SSDs called *Rep-RAID*, where the basic idea is to replicate **the most frequently written chunks**. Figure 6 illustrates the architecture of *Rep-RAID*. As seen, we first construct a model to direct the replication of data chunks when servicing an update request. We primarily refer to the factors of the update frequency of data chunks and the implied GC overhead (see Section 3.2 for more information). We then propose a stripe reorganization scheme that adaptively rebuilds data stripes during the GC process by combining the replicated chunks with valid data chunks of existing stripes in GC-targeted blocks (explained in Section 3.3).

To guarantee the reliability of data pairs that are not protected by RAID, *Rep-RAID* distributes the chunks of a data pair to different RAID components (i.e., channels). To put it another way, the original data stripe has both valid and invalid chunks, so we refer to it as a **semi-stripe**, and its valid and invalid data chunks are labelled as *semi-valid* and *semi-invalid*, respectively. Note that, *semi-invalid* data chunks should be kept for protecting the rest of valid data chunks in a **semi-stripe**, even though their data are not up-to-date.

3.2 Model-Based Replication

Replicating data chunks can save the time needed for XOR computation and the software overhead of the SSD controller caused by updating data stripes in a RAID system. In addition, read parallelism can be exploited by accessing the data replica that resides on an idle SSD channel for enhancing read performance. However, replicating data chunks may adversely impact the endurance of SSDs

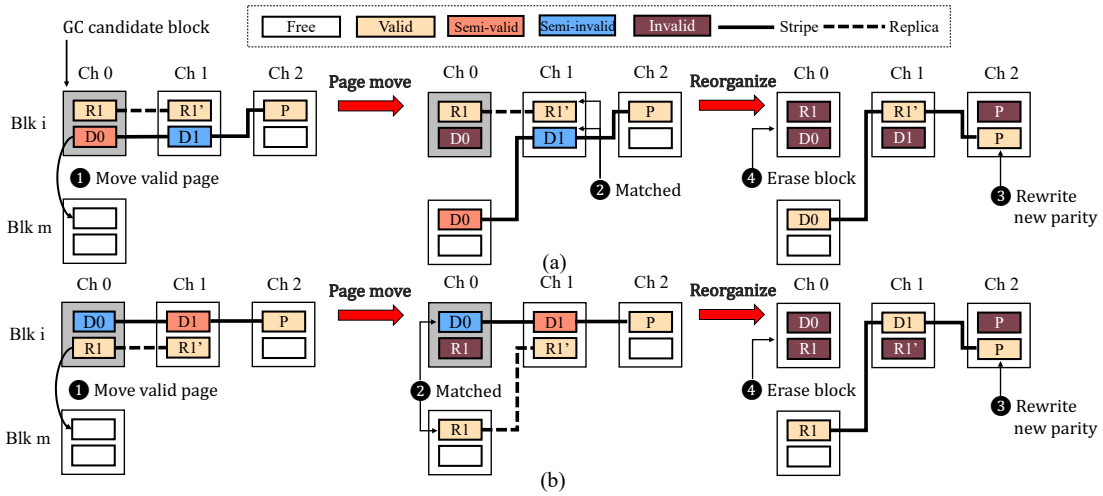

Fig. 7. Example of stripe reorganization with the *replicated chunks* and valid chunks of **semi-stripe**. (a) the reorganization starting from *semi-valid* chunks. (b) the reorganization starting from *replicated* chunks. Note that we use three channels (`Ch0`-`Ch2`) and each data stripe has two data chunks and one parity chunk, for simplicity of illustration.

since more space is needed for storing replicas, which in turn leads to more erase operations on SSD blocks.

To balance the benefits and the overhead of data replication in RAID-enabled SSDs, we construct a mathematical model to steer replication on update requests. Using the notations in Table 1, we trigger the replication process for servicing an update request when the following formula holds:

$$\sum_{i \in A} f(i) \cdot (t_{XOR} + t_{Soft}) + \bar{t}_R \cdot |A| \geq \delta \cdot |A| \cdot \bar{t}_{GC} + \alpha \cdot |A| \cdot \bar{t}_{Reorg} \tag{1}$$

Although each replication can save $t_{XOR}+t_{Soft}$, the parameter $\bar{t}_R$ cannot be accurately determined that is related to the idle/busy degree of the RAID system and its hardware configuration (e.g., the total number of SSD channels).

Equation 1 unveils that *Rep-RAID* only replicates data chunks with high update frequency for maximizing the benefits of the replication functionality in the context of RAID-enabled SSDs. In other words, we only replicate data chunks that meet the requirement of access frequency according to Equation 2:

$$f(i^*) = \frac{\delta \cdot \bar{t}_{GC} + \alpha \cdot \bar{t}_{Reorg} - \bar{t}_R}{t_{XOR} + t_{Soft}} \tag{2}$$

The primary principle of *Rep-RAID* is to replicate frequently written data for eliminating $t_{XOR} + t_{Soft}$, and the perfect situation of GC is about no replicated chunks in the target block (i.e. the cost of $\delta \cdot \bar{t}_{GC}$), and thus no stripe reorganization (i.e. the cost of $\alpha \cdot \bar{t}_{Reorg}$). We analyze the history information in the last time window to dynamically estimate the value of $f(i^*)$ in Equation 2, for directing data replication in the current time window [29, 37].

To periodically adjust the value of $f(i^*)$ in a real-time manner, we adaptively tune the size of time window for counting the access frequency of data chunks by following Equation 3. Consequently, only the accesses that occur in the time window of T^* are considered for determining the value of $f(i^*)$.

Table 1. Notation descriptions in the replication model

Notations	Explanation Descriptions
$f(i)$	The write access frequency of the block i during the fixed time window T
A	Set of replicated data chunks
$\|A\|$	Element number of set A
δ	The increasing GC number induced by replication data
α	The increasing number of reorganizing stripes induced by replication data
t_{XOR}	Time of each XOR computation
t_{Soft}	Software time by managing update chunks
$\bar{t}_{GC}$	Average time of each GC processing
$\bar{t}_R$	Average time saved by reading replicated data chunks
$\bar{t}_{Reorg}$	Average time of reorganizing stripes
$f(i^*)$	The least value of $f(i)$ for replication

$$T^* = \frac{T}{f(i^*)} \tag{3}$$

where T is the size of the default time window, whose unit is the number of request. We use `1024` in our design by referring to [36, 37]. Note that, the access frequency counter should be renewed when entering in the next time window.

3.3 Stripe Reorganization in GC

Because of the replication feature in RAID-enabled SSDs, the storage system holds **data stripes** that contain valid chunks, **semi-stripes** that contain *semi-valid chunks* and *semi-invalid chunks*, and **replication pairs** that contain valid chunks and their replicas at the same time. Based on this classification, we propose a stripe reorganization scheme to build new data stripes during garbage collection by combining the valid chunks of **semi-stripes** and the *replicated chunks* to confine the number of **replication pairs** and **semi-stripes**. By referring to Figure 7(a), we describe these terms with examples. Because of previously being updated as replicas, *D1* is now an obsolete *semi-invalid* data chunk, whose stripe is a **semi-stripe**, and other valid data chunks (e.g., *D0*) are labelled as *semi-valid*. The **replication pair** of *R1* and *R1'* that are stored in the different channel, to ensure data reliability.

After determining a GC candidate block by following the GC policy, (e.g. greedy policy [38]), *Rep-RAID* performs page moves in the following order:

(1) ***Semi-valid chunks***. *Rep-RAID* first selects to migrate *semi-valid chunks*. Then it searches *replicated chunks* in the GC candidate block, and the components of stripe reorganization matches when the mirror data of *replicated chunk* and the *semi-invalid chunk* in the **semi-stripe** are in the same channel. Finally, it can build the new stripe through combining the mirror data of *replicated chunks* and the other chunks in the matched **semi-stripe**.
Figure 7(a) shows a reorganization example triggered by migrating the *semi-valid chunk* of $D0$. It first migrates $D0$ to a new block, and then searches data chunks on other channels for building a new stripe. *Rep-RAID* prefers to find a replicated data chunk (e.g. $R1'$) in the block that has a *semi-invalid* data chunk (e.g. $D1$) belonging to the same stripe as $D0$. Then, $D0$ and $R1'$ can be reorganized as a new data stripe, and $R1$ in the GC block can be directly set as invalid

data, to avoid migration in GC. After the new data stripe is built, the *semi-invalid chunks* $D1$ on the other channel can be converted into invalid chunks, which can be reclaimed in future GC execution.

(2) ***Replicated chunks***. After all *semi-valid chunks* in the GC block have been moved out, *Rep-RAID* handles the *replicated chunks*. We can obtain a match for the *replicated chunk*, once there is a *semi-invalid chunk* in the GC candidate block. Then, we can construct a new data stripe by utilizing the *replicated chunk* to replace the position of *semi-invalid chunk* in the GC candidate block.
Figure 7(b) demonstrates the reorganization case triggered by migrating the *replicated chunk* of $R1$. Different from the case in (1), the matching process only searches the *semi-invalid* data chunk (e.g., $D0$) in the GC block, and it can be directly reclaimed without migration once the new stripe is built. At last, the mirror chunk of *replicated chunks* $R1'$ can be reclaimed in a future GC process.

(3) **The rest of not invalid chunks**. After migrating the *semi-valid chunks* and the *replicated chunks*, both *semi-invalid chunks* and *valid chunks*, are moved out from the GC block with the default migration scheme.

Note that, it traverses *semi-valid chunks* and *replicated chunks* in the GC target block, and performs two matched cases of stripe reorganization, as illustrated in Figure 7. At last, the rest of unmatched data chunks are mandatorily moved as default.

3.4 Implementation

Algorithm 1 shows the implementation details of the model-based replication and stripe reorganization of *Rep-RAID*. Lines 18-21 depict the workflow of servicing an update request. If the update frequency of target data is not smaller than the threshold of $f(i^*)$, *Rep-RAID* conducts a replication process by calling Function `replica_write()`. Otherwise, it carries out a write process by calling Function `stripe_write()`.

The value of $f(i^*)$ is initially set to 1, and the write requests will be replicated at the first update. Moreover, *Rep-RAID* enables adjusting the value of $f(i^*)$ in an adaptive manner after the GC process according to I/O characteristics of applications. Line 16 indicates the rules of modifying the value of $f(i^*)$. Note that the update count of data chunks will be cleared when the time window T^* is reached.

Lines 1-14 describe the process of stripe reorganization during the GC process. It traverses the *replicated chunks* and the *semi-valid chunks* in the GC target block, and performs two matched cases of stripe reorganization that are previously introduced in Figure 7.

4 EXPERIMENTAL EVALUATION

4.1 Experimental Environment

We have performed experimental evaluation using the widely studied SSD simulator SSDSim [42], which has been modified to support RAID-5. We use a local ARM-based machine as SSD controllers have usually limited computation power and memory capacity [11]. The machine has an ARM Cortex A7 Dual-Core with 800MHz, 128MB of memory and 32-bit Linux (*ver 3.1*). We simulated an 128-GB SSD and configured the performance parameters based on the values used in [40]. To reflect the impact of garbage collection before replaying traces, the simulated SSD is aged so that valid data and invalid data occupy 80% and 10% of its capacity, respectively. We obtain these settings by referring to previously published studies such as [11, 41]. The value of XOR latency is tested by the local ARM-based machine, which has been used for evaluation in RAID-enabled SSDs [11], and

Algorithm 1: I/O and GC processes in *Rep-RAID*

```
Input: args of pages, replicated pages, semi-stripe;
Output: null;
Function stripe_reorganization()
    for semi-valid_page in block do
        for replica_page in block do
            if semi-invalid_page.channel
            ==replica_page.mirror_channel then
                /*the case in Figure 7(a)*/
                page_migrate (semi-valid_page);
                build_new_stripe();
                break;
    for replica_page in block do
        /*the case in Figure 7(b)*/
        if semi-invalid_page then
            page_migrate (replica_page);
            build_new_stripe();
/*update f(i*)*/
f(i*) = (δ·t̄_GC + α·t̄_Reorg − t̄_R) / (t_XOR + t_Soft);
/*main function*/
if update request & req.update_num ≥ f(i*) then
    replica_write();
else
    stripe_write();
if GC then
    stripe_reorganization();
    page_migrate(); //the rest valid pages
    erase();
if t > T* then
    reset page.update_num;
```

the setting of software latency is referred to [24]. Table 2 shows the values of parameters in our evaluation.

With respect to the benchmarks, we utilized a set of *MSR Cambridge* block I/O traces [43], which are often used in the domain of SSD optimization [38, 39] as workloads for RAID-5 system. Table 3 presents the detailed specifications on the traces, where the metric of *Update R* indicates the update ratio of write requests.

We compare *Rep-RAID* to three RAID based mechanisms:

- *RAID-5* indicates the conventional RAID-5 implementation inside SSDs without replication, as the baseline method.

- *CR5M* [25] is a mirroring-assisted channel-level RAID5 architecture for a single SSD. It mirrors the requests with small sizes into a dedicated chip (called mirroring chip) in the same channel of

Table 2. Experimental SSD-related parameters

Parameters	Values	Parameters	Values
Channel Size	8	*Read latency*	0.045ms
Chip Size	4	*Write latency*	0.7ms
Plane Size	4	*Erase latency*	3.5ms
Block per plane	512	*GC threshold*	10%
Page per block	256	*XOR latency*	0.019ms
Page size	8KB	*Software latency*	1.82ms
FTL scheme	Page	*Wear-leveling*	Static

Table 3. The Characteristics of Evaluated Workloads

Trace	Write Ratio	Write Volume	Update R
wdev_0	79.9%	7.1GB	82.7%
stg_0	84.8%	5.1GB	81.9%
stg_1	36.3%	6.0GB	69.2%
ts_0	82.4%	11.3GB	65.9%
web_0	70.1%	11.7GB	79.0%
src1_2	74.6%	44.1GB	66.5%
src2_0	88.7%	9.3GB	70.2%
hm_0	64.5%	20.5GB	67.1%
mds_0	88.1%	7.4GB	79.6%

original data chunk. *CR5M* is the most related work to our proposal that supports replication in channel-level RAID5-enabled SSDs.

- *FusionRAID* [24] temporally replicates data chunks of small write requests to underlying flash memory and later converses the replicated chunks into RAID stripes during idle periods. To evaluate FusionRAID in the context of RAID-enabled SSDs, we apply *FusionRAID* in the default channel-level RAID-5 implementation.

4.2 Results and Discussions

To characterize our proposed mechanism, we use the following two performance metrics: (a) *I/O latency* that is the average I/O duration and (b) *long tail latency* that is the distribution of the duration of the slowest 1% of the I/O requests.

4.2.1 I/O Latency. We replayed the I/O traces and collected results on I/O latency with the selected RAID implementations. Because I/O latency greatly varies from case to case, we count the normalized I/O latency of all selected traces. Figure 8 presents the average latency of write requests, read requests, and all I/O requests.

Figure 8(a) shows the results of average write latency. As seen, *CR5M*, *FusionRAID* and *Rep-RAID* yield better performance on write responsiveness by 6.5%, 22.5% and 34.7%, respectively, when compared to the baseline of *RAID-5*. This fact verifies that replicating data chunks can avoid XOR computations and software overhead of updating the relevant data stripes, ultimately benefiting the write latency.

Another observation is that, while *CR5M* and *FusionRAID* both utilize the replication method to boost the responsiveness of write requests, *CR5M* cannot perform better than *FusionRAID*, even worse than *RAID-5* in the some cases (i.e., *src1_2* and *hm_0*). This is because *CR5M* replicates the small update requests but stores them in two chips of the same channel in channel-RAID5 SSDs.

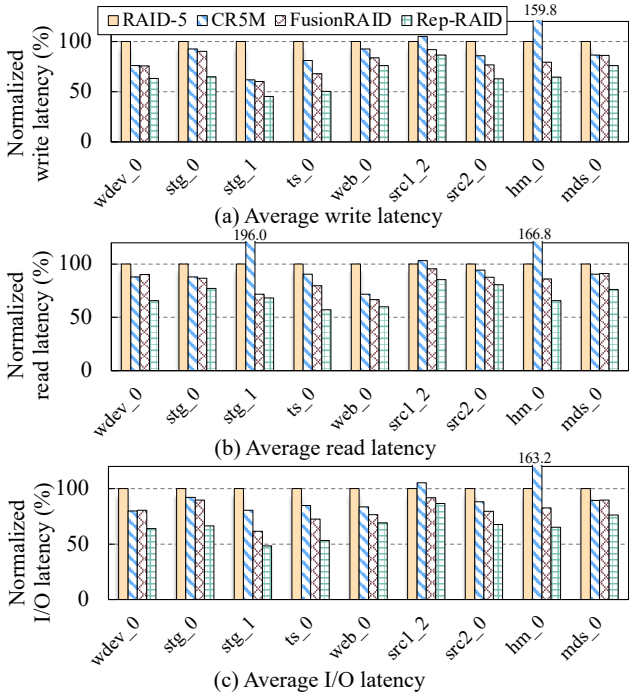

Fig. 8. I/O latency of evaluated RAID implementations.

Although this replication layout can protect the replication in the chip level, the access to these replication cannot exploit the parallelism of channel level inside SSDs.

More importantly, *Rep-RAID* decreases the write latency by 16.3% on average compared to *FusionRAID*. This is because *Rep-RAID* methods selectively replicate frequently written chunks by referring to the replication threshold and support organizing data stripes by combining the replicated chunks and the *semi-invalid chunks* of existing stripes. As a result, it can minimize the number of *semi-invalid* data chunks and thus reduce the GC time that is the major factor in the I/O processing delay.

Figure 8(b) presents the results of average read latency. As seen, *CR5M* does not perform well in the most cases, in contrast to *FusionRAID*, even worse than the baseline of *RAID-5*. This is because the setting of all experimental methods are based on channel-level RAID-5, but the design of *CR5M* utilizes different chips in the same channel to hold the replication pair, which results in not only the lower parallelism, but also the worse load balance at the channel level.

Similar to the results of write latency, *FusionRAID* and *Rep-RAID* outperform the baseline *RAID-5* in read performance as well. Moreover, *Rep-RAID* does better than *FusionRAID*, since the different selection of replication data and the decreased GC time reduce the interference towards the response of read requests, i.e., read/write interference [19] and read/GC interference [20].

In summary, *Rep-RAID* achieves the best I/O performance across the selected workloads. Specifically, *Rep-RAID* reduces the total I/O latency by 33.7%, 28.7% and 16.7% (see Figure 8(c)), when compared to conventional *RAID-5*, replication-enabled *CR5M* and *FusionRAID*, respectively.

4.2.2 Long-Tail Latency. The measure of long-tail latency is another critical indicator of SSD I/O performance, which is commonly expressed in the form of the Cumulative Distribution Function

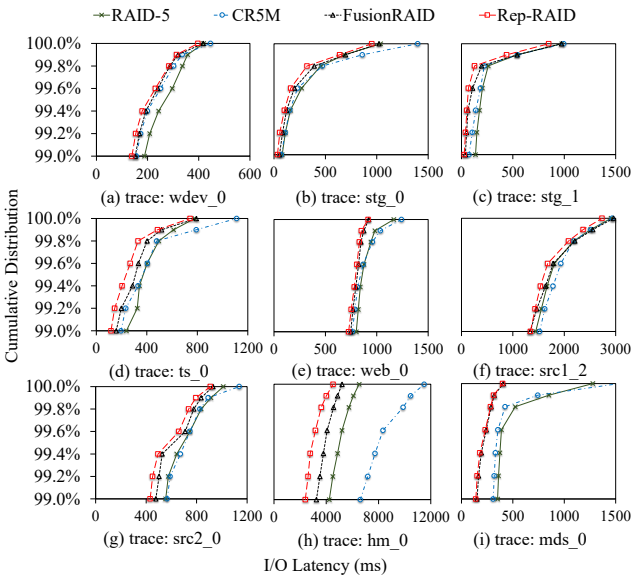

Fig. 9. CDF of long-tail latency of the selected methods.

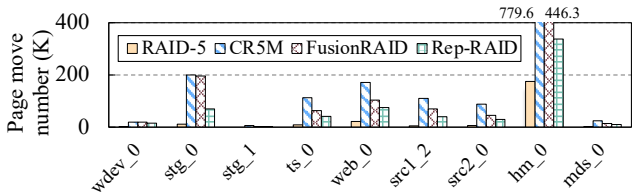

Fig. 10. The comparison of valid page migrations.

(*CDF*). We collected the results of long-tail latency after replaying the selected traces. Figure 9 reports the CDF of the slowest 1% of the I/O requests.

As seen, though the I/O performance of *CR5M* performs similarly as *FusionRAID* when running the most workloads, it does not yield an attractive tail latency. Specifically, *CR5M* shows the worse tail latency by 39.6% on average at the 99.99th percentile, compared with *RAID-5*. This is because, the access parallelism cannot be well utilized especially when the I/Os are intensive, and the increased number of page move heavily impacts the I/O response.

On the other hand, *FusionRAID* can improve the tail latency compared to *RAID-5*, since it can mitigate the negative impact of I/O requests caused by bursty requests. Thus, the tail latency can be further improved. On the other hand, *Rep-RAID* further reduces long-tail latency by 7.8% on average at the 99.99th percentile in contrast to *FusionRAID*. This is because *Rep-RAID* relieves the negative impact of page migration during garbage collection.

4.3 GC Statistics

This section further profiles the GC operations and each GC consists of multiple page moves and one erase. GC is a significant cause of delaying I/O processing in the SSD device [28], though replication methods can mitigate the software overhead that becomes the major component of postponing I/O servicing in such RAID-enabled SSD. We record the number of valid page movements, the number of erase operations caused by all GC operations, and the time required for completing GC.

4.3.1 Valid Page Moves. Figure 10 presents the amount of page moves during the GC process. *RAID-5* yields the least number of valid page moves, but it suffers from the overhead of software and xor in RAID-enabled SSDs when servicing I/O requests. The RAID implementations that utilize replication (i.e., *CR5M*, *FusionRAID* and *Rep-RAID*) require more page moves. This is because all these schemes induce a number of *semi-invalid chunks* in existing stripes after making replicas for them. Such *semi-invalid chunks* are still useful to protect other valid chunks in the stripe, and thus, they need to be migrated during GC. *CR5M* has the most number of valid page moves in case of the most workloads, caused by the placement of replication data in the same channel, that leads to an imbalanced data distribution.

More importantly, *Rep-RAID* reduces the number of valid page moves by 31.9% on average compared to *FusionRAID*. This is because *Rep-RAID* combines the valid chunks in **semi-stripes** with the replicated chunks to reorganize new data stripes, which, in turn, minimizes the numbers of *semi-invalid chunks* and *replicated chunks*.

Table 4. The number of erase operations induced by GC

Trace	*RAID-5*	*CR5M*	*FusionRAID*	*Rep-RAID*
wdev_0	10932	11023	10965	10961
stg_0	23657	24394	24042	23874
stg_1	9326	9342	9327	9324
ts_0	17509	17921	17729	17655
web_0	23365	23966	23689	23583
src1_2	51429	51836	51675	51567
src2_0	15975	16302	16120	16059
hm_0	35011	37361	36068	35638
mds_0	12219	12306	12262	12258

4.3.2 Erase Statistics. Table 4 reports the results of erase numbers when using the four RAID implementations. As seen, *RAID-5* unveils the least number of erases, and *Rep-RAID*, *FusionRAID* and *CR5M* increase the erase number by 0.69%, 1.08% and 2.15% in contrast to the baseline of *RAID-5*. This is because although replication-enabled methods can distribute the chunks more evenly that can reduce the erase number, less space can be reclaimed after each GC operation in replication-enabled schemes and more GC operations are triggered. Furthermore, *Rep-RAID* leads to a slight reduction in erase operations compared to *FusionRAID* and *CR5M*, as it can minimize the number of **semi-stripes** by reorganizing stripes during garbage collection, i.e., a fewer page move number presented in Figure 10.

4.3.3 GC Time Overhead. Figure 11 demonstrates the overall time needed for completing all GC operations and stripe reorganization processes. Both *RAID-5* and *CR5M* do not have the process of stripe reorganization, so they do not has the organization part of time overhead. As seen, *Rep-RAID* reduces the GC time by 12.1% compared to *FusionRAID*, though it proactively does athe stripe organization associated with GC processes.

4.4 Overhead Analysis

This section presents the spatial and runtime overhead caused by our proposal. Because *Rep-RAID* focuses on the replication of frequently accessed data chunks, it requires recording the update count of data chunks. More specifically, the spatial overhead is 2bit $\times$ 16,777,216 = 4MB, which we believe is an acceptable amount of memory space in the case of a 128-GB SSD. Note that 16,777,216 is the total number of chunks in the 128-GB SSD.

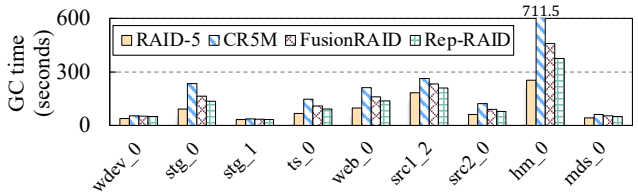

Fig. 11. The comparison of GC time.

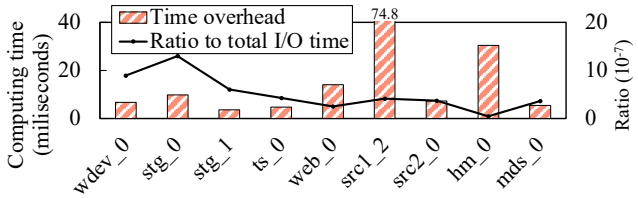

Fig. 12. Time overhead in the proposed *Rep-RAID*.

In addition, the replication-enabled methods, i.e., *CR5M*, *FusionRAID* and our proposed *Rep-RAID* need holding the replica table. As the entries in replica table are dynamically appended and removed, the related work needs `452.3`KB at most, while our proposed *Rep-RAID* requires a maximum of `485.8`KB, when running the evaluated workloads.

Except for these spatial overhead, *Rep-RAID* needs recording three parameters in Equation 2, i.e., increased GC cost $\delta \cdot \bar{t}_{GC}$, stripe reorganization cost $\alpha \cdot \bar{t}_{Reorg}$, and replica read benefit $\bar{t}_R$, which consumes $3 \times 4\text{B} = 12\text{B}$.

For adjusting the replication threshold of $f(i^*)$ and reorganizing data stripe, *Rep-RAID* methods need renewing the update count of chunks in every time window of T, determine the GC candidate block in a plane followed by proposed GC policy, and traversing all *semi-invalid chunks* and *replicated chunks* associated with the GC candidate block.

We recorded the time overhead while running the benchmarks, and Figure 12 shows the results. As seen, the computation overhead in *Rep-RAID* methods accounts for less than `(1.3e-4)`% of the total I/O processing time, varying from `3.6` to `74.8` milliseconds, which we believe is negligible, even on a resource-limited ARM platform.

5 RELATED WORK

Write penalty is the main issue of RAID systems, which becomes more severe in the context of RAID-enabled SSDs due to the nature of `out-of-place` updates of SSDs [30]. To address this shortcoming, a RAID buffer can be introduced to reduce the quantity of writes forwarded to the underlying flash cells. Lee et al. [15] proposed flash-aware redundancy array (FRA) to cut down parity updates in RAID-enabled SSDs. Their technique can absorb a number of write requests by holding the updated data in the RAID buffer. For absorbing even more write requests, Li et al. [11] introduced a patch-based data management scheme for dual-copy buffers in RAID-enabled SSDs, which only stores the updated part of data chunks in the buffer. Besides, a patch-based read/write mode is proposed for correctly servicing I/O requests. By further considering the stripe information, Tang et al. [34] presented a stripe-aware buffer policy to coalesce consecutive writes to the same file. Their proposal prefers evicting the data chunks that belong to the same stripe, by taking the factor in the reduction of parity updates into account.

Considering that the parity chunks are the most frequently updated parts in RAID stripes, a dedicated parity buffer, which decreases the write operations on parity chunks, is introduced in [31]. Moreover, Im and Shin [32] presented a partial parity technique to minimize the number

of read operations required for calculating a parity chunk by exploiting the implicit redundant data of flash memory. Based on the partial parity update approach, Kim et al. [10] introduced a RAID scheme that allows flexible stripe sizes and parity placement to minimize parity updates. Their technique, however, comes at the expense of more SSD pages for parity chunks. For further eliminating updates on the parity chunks, Li et al. [14] proposed to form data stripes by using the same level hotness of data chunks. Then it may have more full-stripe writes in a short period, and thus multiple data chunks in the same stripe can update the parity chunks only once. These methods succeed in utilizing the DRAM buffer to absorb writes of data and parity chunks. However, limited size of DRAM buffer cannot avoid these writes onto the underlying flash blocks.

RAID requires XOR computations and software overhead, which inevitably impact the scheduling of normal I/O requests. Some recent studies introduce the replication (mirror) technique into the SSD-related RAID systems for relieving the additional write traffic caused by RAID. Pan et al. [25] proposed a mirroring method for small write/update requests using an extra mirror chip per channel to avoid parity calculations and updates. In particular, it writes the updated data chunks in the original SSD chip and makes a replicated chunk in the reserved mirror chip to ensure data reliability. However, it needs to reserve extra chips of SSD for holding the replicated data. In addition, it will reorganize all replicated (up-to-date) chunks into data stripes *if-and-only-if* all relevant chunks in the original data stripe have been replicated or updated.

To accelerate the write responsiveness in bursty application workloads, Jiang et al. [24] have proposed a similar replication method (called *FusionRAID*) targeting small write requests. Instead of directly updating the data stripe, *FusionRAID* temporally replicates data chunks of small write requests to lighten I/O congestion, and later it converts the replicated chunks into RAID stripes according to various configurable thresholds. Because of the nature of `out-of-place` update, obsolete data chunks must be kept as 'valid' to protect other valid data chunks in the original stripe. After a certain period of time, replicated data chunks will be reorganized as new data stripes when predefined conditions are met. However, data chunks in new stripes may have different levels of access frequency which renders some data chunks obsolete but 'valid' and thus it increases the overhead of GC.

6 CONCLUSION

We have proposed and evaluated an integrated data replication and garbage collection scheme for RAID-enabled SSDs that mitigates the negative impact caused by write penalty. To this end, our proposal enables a mathematical model to guide the replication of frequent update requests. Furthermore, it reorganizes the cold replicated chunks with the semi-stripes to build the new data stripes, which in turn decreases data movement during garbage collection. Experimental results show that the proposed approach reduces the overall I/O latency and tail latency by `16.7%` and `7.8%`, respectively, when compared to state-of-the art methods.

ACKNOWLEDGMENTS

This work was partially supported by Chongqing Graduate Research and Innovation Project (No. CYB21110), and China Scholarship Council (No. 202106990042).

REFERENCES

[1] Bryan S. Kim, Jongmoo Choi and Sang Lyul Min. Design tradeoffs for SSD reliability. In *USENIX Conference on File and Storage Technologies (FAST)*, 2019: 281–294.

[2] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min and Sam H. Noh. Alleviating garbage collection interference through spatial separation in all flash arrays. In *USENIX Annual Technical Conference (ATC)*, 2019: 799-812.

[3] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential raid: Rethinking raid for ssd reliability. *ACM Transactions on Storage*, Vol. 6(2): 1-22, 2010. https://doi.org/10.1145/1807060.1807061

[4] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, Vol. 105(9): 1666-1704, 2017. https://doi.org/10.1109/JPROC.2017.2713127

[5] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash reliability in production: The expected and the unexpected. In *USENIX Conference on File and Storage Technologies (FAST)*. 67–80.

[6] NAND Flash Media Management Through RAIN. Retrieved from https://www.micron.com/-/media/client/global/documents/products/technical-marketing-brief/brief_ssd_rain.pdf

[7] P320h 2.5-Inch PCIe NAND SSD Features. Retrieved from https://www.micron.com/-/media/client/global/documents/products/data-sheet/ssd/p320h_2_5.pdf

[8] Huawei Tecal ES3000. Retrieved from https://www.storagereview.com/review/huawei-tecal-es3000-application-accelerator-review

[9] Soojun Im, and Dongkun Shin. Improving SSD reliability with RAID via elastic striping and anywhere parity. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013. https://doi.org/10.1109/DSN.2013.6575359.

[10] Jaeho Kim, Eunjae Lee, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Chip-level raid with flexible stripe size and parity placement for enhanced ssd reliability. *IEEE Transactions on Computers*, Vol. 65(4): 1116-1130, 2016. https://doi.org/10.1109/TC.2014.2375179

[11] Jun Li, Zhibing Sha, Zhigang Cai, François Trahay, and Jianwei Liao. Patch-Based Data Management for Dual-Copy Buffers in RAID-Enabled SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3956-3967, 2020. https://doi.org/10.1109/TCAD.2020.3012252

[12] Zhibing Sha, Jun Li, Zhigang Cai, Min Huang, Jianwei Liao, and Francois Trahay. Degraded Mode-benefited I/O Scheduling to Ensure I/O Responsiveness in RAID-enabled SSDs. *ACM Transactions on Design Automation of Electronic Systems*, 2022. https://doi.org/10.1145/3522755

[13] Jiguang Wan, Wei Wu, Ling Zhan, Qing Yang, Xiaoyang Qu, and Changsheng Xie. DEFT-Cache: A cost-effective and highly reliable SSD cache for RAID storage. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 102-111, 2017. https://doi.org/10.1109/IPDPS.2017.54

[14] Yongkun Li, Biaobiao Shen, Yubiao Pan, Yinlong Xu, Zhipeng Li, and John C. S. Lui. Workload-aware elastic striping with hot data identification for SSD RAID arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 36(5): 815-828, 2016. https://doi.org/10.1109/TCAD.2016.2604292

[15] Yangsup Lee, Sanghyuk Jung, and Yong Ho Song. FRA: a flash-aware redundancy array of flash storage devices. In *IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, pp. 163-172, 2009. https://doi.org/10.1145/1629435.1629459

[16] Soojun Im and Dongkun Shin. Delayed partial parity scheme for reliable and high-performance flash memory SSD. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1-6, 2010. https://doi.org/10.1109/MSST.2010.5496997

[17] Yubiao Pan, Yongkun Li, Yinlong Xu, and Zhipeng Li. Grouping-based elastic striping with hotness awareness for improving SSD raid performance. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 160-171, 2015. https://doi.org/10.1109/DSN.2015.51

[18] Youmin Chen, Youyou Lu, Pei Chen, and Jiwu Shu. Efficient and Consistent NVMM Cache for SSD-Based File System. *IEEE Transactions on Computers*, Vol. 68(8): 1147-1158, 2018. https://doi.org/10.1109/TC.2018.2870137

[19] Suzhen Wu, Weiwei Zhang, Bo Mao, and Hong Jiang. HotR: Alleviating Read/Write Interference with HotRead Data Replication for Flash Storage. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019: 1367-1372. https://doi.org/10.23919/DATE.2019.8715100

[20] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *USENIX Conference on File and Storage Technologies (FAST)*, pp. 15-28, 2017.

[21] Myoungsoo Jung, Ramya Prabhakar, and Mahmut Taylan Kandemir. Taking garbage collection overheads off the critical path in SSDs. In *International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2012: 164-186.

[22] Wonkyung Kang, Dongkun Shin, and Sungjoo Yoo. Reinforcement Learning-Assisted Garbage Collection to Mitigate Long-Tail Latency in SSD. In *ACM Transactions on Embedded Computing Systems*, 2017, 16(5s): 1-20. https://doi.org/10.1145/3126537

[23] Wonil Choi, Myoungsoo Jung, Mahmut Kandemir, and Chita Das. 2018. Parallelizing garbage collection with I/O to improve flash resource utilization. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. https://doi.org/10.1145/3208040.3208048

[24] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays. In *USENIX Conference on File and Storage Technologies (FAST)*, 2021: 355-370.

[25] Wen Pan and Tao Xie. A mirroring-assisted channel-RAID5 SSD for mobile applications. *ACM Transactions on Embedded Computing Systems*, 2018, 17(4): 1-27. https://doi.org/10.1145/3209625

[26] Wenhui Zhang, Qiang Cao, Hong Jiang, and Jie Yao. Improving overall performance of TLC SSD by exploiting dissimilarity of flash pages. *IEEE Transactions on Parallel and Distributed Systems*, 2019, 31(2): 332-346. https://doi.org/10.1109/TPDS.2019.2934458

[27] Congming Gao, Liang Shi, Kai Liu, Chun Jason Xue, Jun Yang, and Youtao Zhang. Boosting the performance of SSDs via fully exploiting the plane level parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2020, 31(9): 2185-2200. https://doi.org/10.1109/TPDS.2020.2987894

[28] Wonkyung Kang and Yoo Sungjoo. Dynamic management of key states for reinforcement learning-assisted garbage collection to reduce long tail latency in SSD. In Proceedings of *the 55th Annual Design Automation Conference (DAC)*, pp. 1-6, 2018. https://doi.org/10.1145/3195970.3196034

[29] Zhibing Sha, Jun Li, Lihao Song, Jiewen Tang, Min Huang, Zhigang Cai, Lianju Qian, Jianwei Liao, and Zhiming Liu. 2021. Low I/O Intensity-aware Partial GC Scheduling to Reduce Long-tail Latency in SSDs. *ACM Transactions on Architecture and Code Optimization*, 18, 4, Article 46 (December 2021), 25 pages. https://doi.org/10.1145/3460433

[30] Sergey Hardock, Petrov Ilia, and Gottstein Robert et al. From in-place updates to in-place appends: Revisiting out-of-place updates on flash. In proceedings of *the 2017 ACM International Conference on Management of Data (SIGMOD)*, pp. 1571-1586, 2017. https://doi.org/10.1145/3035918.3035958

[31] Soojun Im and Dongkun Shin. Delayed partial parity scheme for reliable and high-performance flash memory SSD. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010: 1-6. https://doi.org/10.1109/MSST.2010.5496997

[32] Soojun Im and Dongkun Shin. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Transactions on Computers*, 2010, 60(1): 80-92. https://doi.org/10.1109/TC.2010.197

[33] Suzhen Wu, Weidong Zhu, Guixin Liu, Hong Jiang, and Bo Mao. GC-aware request steering with improved performance and reliability for SSD-based RAIDs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018: 296-305. https://doi.org/10.1109/IPDPS.2018.00039

[34] Chenlei Tang, Jiguang Wan, Yifeng Zhu, Zhiyuan Liu, Peng Xu, Fei Wu, and Changsheng Xie. RAFS: A RAID-Aware File System to Reduce the Parity Update Overhead for SSD RAID. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019: 1373-1378. https://doi.org/10.23919/DATE.2019.8714938

[35] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. IEEE Symposium on Mass Storage Systems and Technologies (MSST). IEEE, 2011: 1-12. https://doi.org/10.1109/MSST.2011.5937224

[36] Jun Li, Xiaofei Xu, Xiaoning Peng, and Jianwei Liao. Pattern-based write scheduling and read balance-oriented wear-leveling for solid state drivers. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2019: 126-133. https://doi.org/10.1109/MSST.2019.00-10

[37] Xiaofei Xu, Zhigang Cai, Jianwei Liao, and Yutaka Ishiakwa. Frequent access pattern-based prefetching inside of solid-state drives. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020: 720-725. https://doi.org/10.23919/DATE48585.2020.9116382

[38] Congming Gao, Min Ye, Qiao Li, Chun Jason Xue, Youtao Zhang, Liang Shi, and Jun Yang. 2019. Constructing Large, Durable and Fast SSD System via Reprogramming 3D TLC Flash Memory. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Association for Computing Machinery, New York, NY, USA, 493–505. https://doi.org/10.1145/3352460.3358323

[39] Chun-Yi Liu, Yunju Lee, Myoungsoo Jung, Mahmut Taylan Kandemir, and Wonil Choi. 2021. Prolonging 3D NAND SSD lifetime via read latency relaxation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 730–742. https://doi.org/10.1145/3445814.3446733

[40] Jinhua Cui, Junwei Liu, Jianhang Huang, and Laurence T. Yang. SmartHeating: On the Performance and Lifetime Improvement of Self-Healing SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020, 40(1): 52-65. https://doi.org/0.1109/TCAD.2020.2990896

[41] Wenhui Zhang, Qiang Cao, Hong Jiang, and Jie Yao. 2018. PA-SSD: A Page-Type Aware TLC SSD for Improved Write/Read Performance and Storage Efficiency. In *International Conference on Supercomputing (ICS)*. Association for Computing Machinery, New York, NY, USA, 22–32, 2018. https://doi.org/10.1145/3205289.3205319

[42] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. *IEEE Transactions on Computers*, 2013, 62(6): 1141-1155. https://doi.org/10.1109/TC.2012.60

[43] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. 2009. Migrating server storage to SSDs: analysis of tradeoffs. In *European conference on Computer systems(EuroSys)*. Association for Computing Machinery, New York, NY, USA, 145–158. https://doi.org/10.1145/1519065.1519081

Received 2023-03-16; accepted 2023-04-21