Improving Read Performance of SSDs via Balanced Redirected Read

Jie Liang[1,3], Yinlong Xu[1,2], Dongdong Sun[1], Si Wu[1]
1. School of Computer Science and Technology, University of Science and Technology of China
2. Collaborative Innovation Center of High Performance Computing, National University of Defense Technology
3. AnHui Province Key Laboratory of High Performance Computing, Hefei, China
Email: {jieliang, sunddd, wusi}@mail.ustc.edu.cn, ylxu@ustc.edu.cn

***Abstract*—Modern SSDs have been a competitive alternative to traditional hard disks because of higher random access performance, lower power consumption and less noise. Although SSDs usually have multiple channels with each channel connected to multiple chips to improve their performance with parallel channels and chips, some recent studies show that contentions among I/O requests make SSDs read performance degrade notoriously, even worse than random write performance. Meanwhile, MLC/TLC flash memory technology increases modern SSDs' capacity while sacrificing their reliability. So fault tolerance, such as chip-level RAID, is also a necessity for SSDs. We propose a Balanced Redirected Read (BRR), which redirects some read requests from busy chips to relatively idle chips by decoding target data chunks with the data/parity chunks in a parity group. We also design a new layout of RAID-5 in SSDs, which distributes adjacent chips to different parity groups such that the degrees of loads on different chips in a group are more likely different. So BRR has more chances to redirect read requests from busy chips to idle chips. Compared with the recently proposed Parallel Issue Queuing (PIQ) using reordering I/O requests technique, BRR schedules the requests among chips more balanced. We implement BRR atop a trace driven simulator, and extensive experiments with real-world workloads show that BRR reduces the waiting time of read requests over PIQ (FIFO) by as much as 38.4% (77.1%) and averagely 14.2% (23.8%), and BRR can also slightly improve write performance due to the improvement of read performance.**

***Keywords*-RAID; SSD read latency; I/O contentions; load balance;**

I. Introduction

Because of higher shock resistance, lower power consumption, and higher random access performance, solid-state drives (SSDs) have progressively replaced hard-disk drives (HDDs), and been widely deployed in kinds of personal computers and large-scale storage servers.

An SSD is usually composed of multiple channels, where each channel connects to multiple NAND flash chips. Although there is abundant parallelism provided by these multiple chips, SSDs can not fully utilize it. One of the reasons is that there are lots of contentions for various shared resource among I/O requests, such as link contention, i.e., different I/O requests competing for I/O bus to transfer data, and storage contention, i.e., different I/O requests accessing the same chip at a short interval [6, 11]. Considering the speed of I/O bus is nearly dozens of times faster than the speed of chips' I/O [12], the main contentions can be regarded as storage contentions up until now. Suffering from contentions, SSDs random read performance may degrade notoriously, even worse than random write performance [13].

Meanwhile, MLC/TLC flash memory technology increases SSDs' capacity while sacrificing their reliability [3, 7]. Fault tolerance such as chip-level RAID [21] within SSDs is also a necessity for enhancing reliability. Chip-level RAID-5 have been widely studied by recent works [4, 14, 17], due to its high storage efficiency and fairish reliability. SSDs with chip-level RAID-5 usually divide all SSD chips into different parity groups and they can tolerate single chip failure within each group due to the redundant data called parity. For write requests of RAID-5, we should write original data chunks, and at the same time update (or generate parity when the data is written at the first time) their corresponding parity chunks to keep data consistency and availability. While for read requests, we can read the target data chunks directly or by degraded read, i.e., decode them by reading the data/parity chunks in the same parity group in case of chip failure. Parity update and degraded read further increase I/O contention. So how to alleviate various contentions on SSDs with chip-level RAID-5 becomes a key issue for nowadays.

There are many studies on SSDs I/O schedule which are proposed either at host system level or SSD controller level to alleviate contentions. Approaches implemented at host side do not need to make any hardware modification, and can utilize resources such as main memory and CPU [6]. While those deployed at SSD controller level can make full use of the real physical data allocation [20], to make I/O schedule more efficient and more accurate [18, 13]. Kim et al. [15] first propose an I/O scheduler in Linux system for SSDs, which bundles write requests to boost write performance and delays issuing them to give favor to read performance. Gao et al. [6] separate I/O requests according to the contentions in host side pending queue, and put requests with contentions into different batches to alleviate contentions. Jung et al. [13] reduce read contentions by classifying requests into multiple groups based on the physical SSD resource they shared. They implement the works in SSD controller by moving the queue in hardware interface to the beneath of FTL. Li et al. [18] design a device-level scheduler in SSD controller to mitigate the write-caused interference by dispatching as many outstanding requests as possible to a reordering set.

978-1-5090-3315-7/16/$31.00 ©2016 IEEE

However, existing works only focus on alleviating I/O contentions of SSDs without chip-level parity. Almost all of them do not exploit the chip-level parity of RAID-5 within SSDs. On the other hand, almost all of them only use reordering method to avoid I/O contentions. But it helps little in reducing the congestion of read requests on multiple chips. This is because read requests must be served by the target flash chips where requested data have been written before, and reordering can not change the target chips. While for write requests, the problem can be easily solved by redirecting write requests to other flash chips through modifying the mapping table within SSD controller [8].

In this paper, we modify Parallel Issue Queuing (PIQ) [6] to be applicable to RAID-5 and propose a balanced redirected read (BRR) strategy to improve read performance of SSDs with chip-level RAID-5. We conduct I/O contentions detecting, queue condition monitoring, and balanced redirected read to realize BRR. Meanwhile, BRR does not interfere with write requests. Note that the main idea of BRR is applicable to SSDs with any erasure code. While for nowadays, RAID-5 is the one which will be most likely used at chip level. So we only present BRR with RAID-5 in this paper. The main contributions of this paper are as follows.

- We propose an *I/O contentions detecting* method, which checks I/O requests contentions according to their target chip numbers and putting requests with contentions into different batches.
- We set a *queue condition monitoring* module, which gets status of queues on chips to predict the waiting time of each chip roughly, further decides how to read data chunks according to status of queues.
- We develop a *balanced redirected read* strategy, which redirects some read requests from busy chips to idle chips, on which the data/parity chunks have been encoded into a parity group of a RAID-5, to achieve parallelism more efficiently.
- We design a new layout of RAID-5 to discretize the chips in a parity group. The new data layout of RAID-5 makes the redirected chips of requests not successive, and further improves the read performance of BRR.
- We implement BRR atop a trace-driven simulator and evaluate the performance of BRR with extensive experiments. The experimental results show that BRR reduces the read waiting time over PIQ (FIFO) by as much as 38.4% (77.1%) and averagely 14.2% (23.8%).

The remainder of this paper is organized as follows. Section II reviews background on chip-level SSD contentions and motivates our BRR scheme. Section III presents the detailed design of BRR. Simulation studies and experimental evaluations are presented in Section IV. Finally, Section V concludes this paper.

II. Problem Formulation

In this section, we first introduce chip-level RAID-5 in SSDs, then explain the contentions among the I/O requests and the state-of-the-art approaches of reordering I/O requests to chip-level contention, and motivate our work at last.

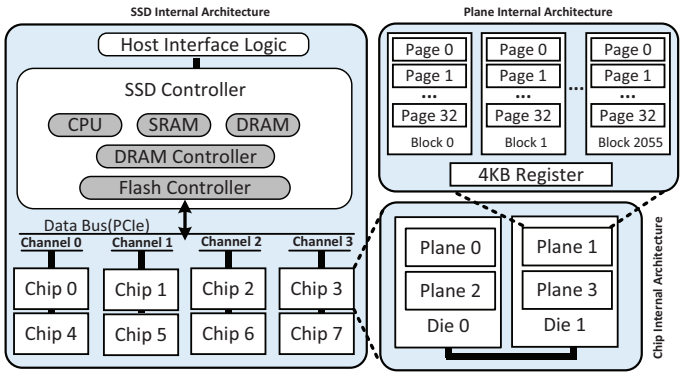

Fig. 1. Physical architecture of a modern SSD.

A. SSD

As illustrated in Fig. 1, modern SSDs mainly consist of three components, Host Interface Logic (HIL), SSD controller and many NAND flash chips. HIL is responsible for communicating between the host system and SSD storage devices. It translates host system requests to NAND flash aware requests and vice versa. SSD controller takes charge of an array of storage devices, including an embedded CPU with SRAM, a DRAM controller and some flash controllers. An embedded software, called Flash Translation Layer (FTL), is employed in SSD controller to manage address translation. The FTL also achieves wear leveling and garbage collection to extend flash chips' lifetime and provide more usable storage space, respectively. The flash controller issues read, write, and erasure operations to NAND flash chip according to the requests from HIL. Many NAND flash chips connecting to multiple I/O channels performs SSDs' channel-level and chip-level parallelism. There are multiple dies composed of multiple planes within each chip. Every plane usually has one register cache of size 4KB and many flash blocks. A block (128KB or 256KB) further consists of some pages. To summarize, there are six levels including channels, chips, dies, planes, blocks, pages in SSDs. Because the 4KB register locates at plane-level, I/O parallelism can be realized within an SSD at four levels, i.e., channel, chip, die, and plane levels. There are three basic operations, read, write and erase in SSDs to perform data access, where erase is performed in the unit of block, while read and write are performed in the unit of page. Fig. 1 shows an example of an SSD consisting of 4 channels, 2 chips per channel, 2 dies per chip, 2 planes per die, 2056 blocks per plane, and 32 pages per block.

B. Chip-level RAID-5

Chip-level RAID in SSDs provides tolerance against chip fault [14, 10, 16, 17]. We illustrate a chip-level RAID-5 in SSDs with Fig. 2. An SSD is divided into stripes, each of which consisting of multiple chunks. Each stripe should be across all chips and is commonly subdivided into different

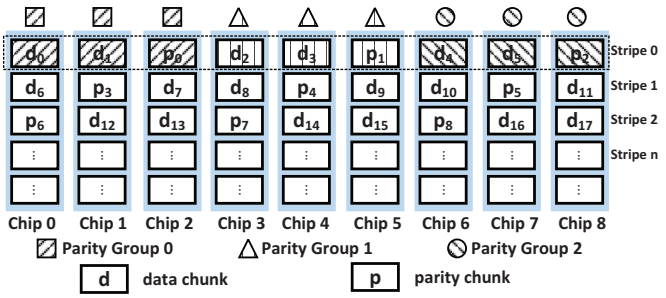

Fig. 2. Overview of a traditional chip-level RAID-5.

parity groups [19] which can shorten recovery cost and boost the robustness of the flash chips array. In Fig. 2, the chips marked square, triangle, and circle are divided into parity group 0, parity group 1 and parity group 2, respectively. There is one parity chunk generated from the data chunks in the same parity group of a stripe. Parity chunks are distributed among all chips in round-robin for load balance. When one data chunk is lost, we can use the remaining data chunks and the corresponding parity chunk in a parity group to recover the lost one. When a data chunk is updated, the corresponding parity chunk would be modified accordingly with either read-modify-write (RMW) or read-construct-write (RCW).

C. I/O Contentions

In this subsection, we explain how to detect I/O contentions with the target chips of requests via an example. Then we present our experimental results to show the low chip utilization of SSDs in practical systems due to I/O contentions.

1) Target chips and I/O contentions: As shown in previous studies [2], data chunks are stored on all chips in round-robin by modulo operation. Therefore, we can get the first target chip of a request with its logical block number (LBN), and then get the following target chips according to its size. If there are N chips in an SSD, the first target chip is the $LBN\%N\_th$, and the following target chips are from the $(LBN+1)\%N\_th$ to the $(LBN+size-1)\%N\_th$. For example, in Fig. 3, the LBN and size of R_1 are 3 and 2 respectively, and there are 9 chips. Thus the target chips of R_1 are Chip 3 and Chip 4.

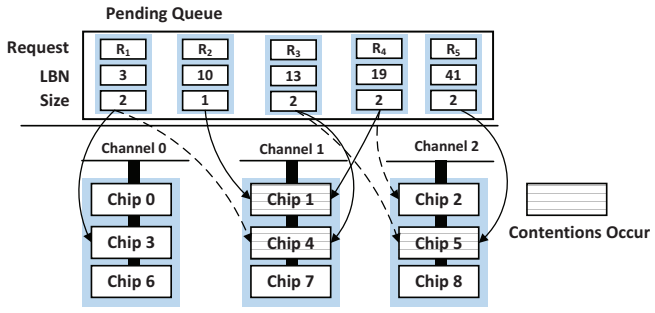

Fig. 3. Contentions among I/O requests.

I/O contention comes from multiple requests accessing the same chip within a short time interval. Therefore, the number of the same target chip accessed by multiple requests can be considered as the flag of contentions. As shown in Fig. 3, there are five I/O requests in the pending queue, e.g., where R_1 contends with R_3 on Chip 4, etc. I/O contentions will induce long waiting time, which degrades system performance.

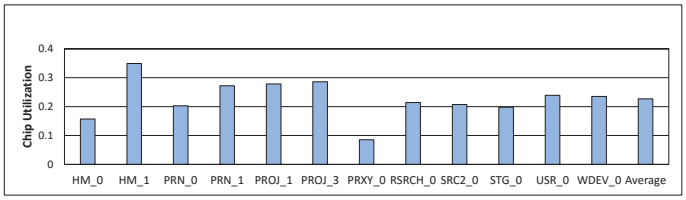

Fig. 4. Chip utilization on a 64 chips SSD of MSR Cambridge traces.

2) Low chip utilization of SSDs: Though there are multiple levels of components to achieve parallelism in modern SSDs, it is difficult to be exploited efficiently. We run twelve intensive application traces, which are selected from real-world servers [1], on an SSD of 64 chips with FIFO to show the chip utilization, and present the results in Fig. 4. As shown in Fig. 4, the chip utilizations of most traces are below 30%, and averagely 22.7%. One of the key issues of the low chip utilization is I/O contentions among requests in the pending queue. Gao et. al's work [6] also validates the similar results.

D. I/O Reordering

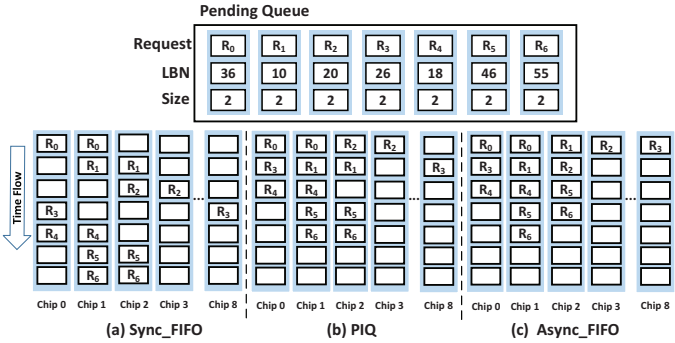

Fig. 5. Reordering requests to mitigate contentions.

There are three commonly used I/O scheduling methods, synchronous FIFO (abbr. as Sync_FIFO), PIQ, and asynchronous FIFO (abbr. as Async_FIFO), as shown in Fig. 5 with an example.

Sync_FIFO: With Sync_FIFO, requests are processed one by one. If there are contentions, only when all sub_requests of the first request in the pending queue are completely processed, the second one begins to be processed. As shown in Fig. 5(a) , R_2's sub_requests, which are on Chip 1 and Chip 2, will begin to be processed synchronously after R_1's sub_request on Chip 1 has been completed, even though Chip 2 has nothing to do when R_1 is being processed.

PIQ: PIQ detects contentions among requests, batches some requests without conflicts together, and processes these requests with higher priority. For example, in Fig. 5(b), R_2 will

be processed before R_1, because R_1 contends I/O with R_0 on Chip 1, while R_2 does not contend with R_0 on any chip. As shown in Fig. 5(a) and Fig. 5(b), Sync_FIFO takes 7 time slots to serve the 7 I/O requests, while PIQ only takes 5 time slots.

Async_FIFO: In real-world system, Async_FIFO is more commonly used than Sync_FIFO. Usually each chip in a modern SSD has its own I/O queue to serve requests, and it schedules its own queue respectively to improve I/O throughput. As shown in Fig. 5(c), the R_1's sub_request on Chip 2 would be processed before another one on Chip 1, which means that the R_1's sub_requests are processed asynchronously. So Async_FIFO takes 5 time slots to serve these requests. We can summarize that, compared with Async_FIFO, PIQ can not improve the chip utilization and can not reduce I/O latency for some traces.

E. Problem Formulation

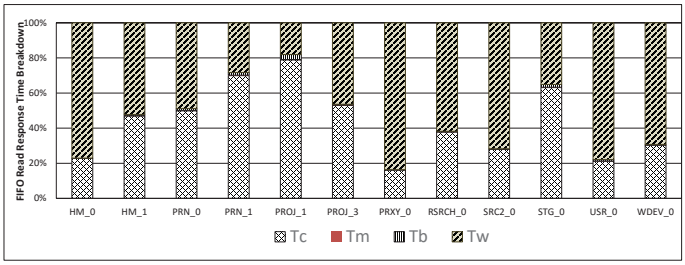

Fig. 6. Read response time breakdown of Async_FIFO.

Response time to I/O requests: The response time to an I/O request consists of 5 components, latency of waiting in host system queue (T_{ws}), delivering time on data bus (T_b), latency of waiting in chip queue (T_{wc}), processing time on chip (T_c), and latency of searching or updating metadata of data chunk (T_m). Given a configured SSD device, due to the hardware property, T_b and T_c are usually fixed for accessing a data chunk. For sake of simplicity, we combine T_{ws} and T_{wc} as T_w, which means the latency of waiting in the pending and outstanding queue. Compared with T_c, T_m is much shorter for read I/Os because metadata commonly keeps in memory and needs not be updated. While for write I/Os, it may nearly equal to T_c because metadata chunk (as big as data chunk) would be updated to SSD for data consistency. For I/O intensive workloads, T_w is almost the bottleneck of performance improvement as shown in Fig. 6. In real systems, I/O requests usually have multiple sub_requests, e.g., R_3 in Fig. 3 has 2 sub_requests. The response time to a request is determined by the longest one of its sub_requests. So if we can redirect the longest sub_requests from busy chips to idle chips, the waiting time of the I/O requests will decrease, even though redirected read may introduce extra I/Os which will increase the delivering time on data bus.

Redirected read with chip-level RAID-5: In a parity group, some data chunks are XOR-summed into a parity chunk to protect data loss. For example, in Stripe 0 of Fig. 2, there are three parity groups, where Group 0 consisting of data chunks d_0, d_1, and parity chunk p_0, i.e., $p_0 = d_0 \oplus d_1$. To read d_0, we can either get it directly from Chip 0 (*direct read*), or we can first read d_1 from Chip 1 and p_0 from Chip 2 and then get d_0 with $d_0 = p_0 \oplus d_1$ (*redirected read*). We call the reads, such as reading d_1 and p_0, as redirected sub_requests. Usually we should choose direct read which induces less reads. But in real systems, as in Fig. 4, due to I/O contentions, there are lots of idle chips when an SSD is processing I/O requests. If we can take advantage of redirected read to migrate some I/O requests from busy chips to idle chips, then chip utilization and read performance would be improved. On the other hand, because of temporal locality and spatial locality of read I/Os, some data chunks in a parity group may reside in memory. So redirected read may not induce to too many extra reads in some cases.

Spatial locality resulting in adjacent busy chips: Recent studies show that skewness, temporal locality and spatial locality exist in real system workloads [9, 8]. SSDs usually distribute data chunks among chips in round-robin. I/O localities make successive chunks be read simultaneously with high probability. So if a chip is busy, its neighbor chips are very likely to be busy too. Traditional chip-level RAID-5 encodes data chunks into a parity group according to the arriving order, which induces the chips of a parity group tending to be busy or idle at the same time. Therefore, redirected read has a poor performance for most practical applications with traditional RAID-5.

From the analysis above, we see that the I/O contentions leading to low chip utilization of SSD. However, the existing reordering methods do not take redirected read into consideration to balance the loads on different chips. In this paper, we propose a redirected read scheme which migrates I/Os with contentions from busy chips to idle chips to reduce the response time to I/O requests. On the other hand, redirected read performs poor with traditional RAID-5 layout. We will design a new RAID-5 layout to make the encoding data chunk as scattered as possible to achieve load balance. With redirected read on our new RAID-5 layout, both chip utilization and read performance of SSDs with chip-level RAID-5 are improved, as validated in our experiments in Section IV.

III. System Design

In this section, we first present our design objectives, then describe the overall architecture of our scheme, and finally detail each component of BRR from a practical perspective.

A. Design Objectives

We mainly focus on the improvement of read performance of SSDs with chip-level RAID-5. Our design is motivated by the following three objectives.

1) *Minimizing I/O response time.* I/O contentions increase the waiting time of I/O requests. Reordering I/O requests to avoid some contentions can reduce waiting latency to some extent. However, reordering can not shorten the longest pending queue of a request effectively, which

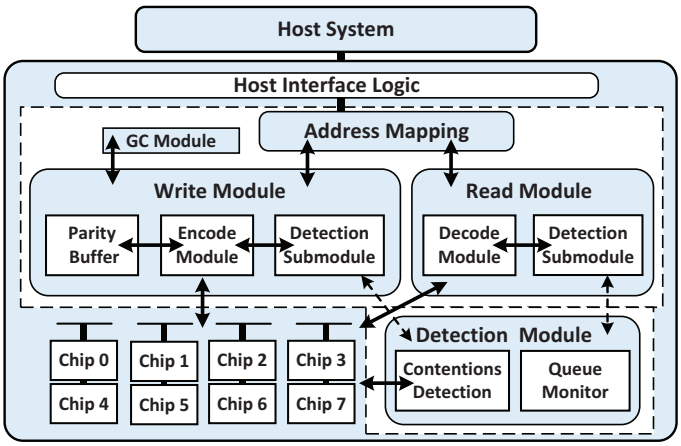

Fig. 7. Overall architecture of system design.

determines its response time. So we are to redirect some I/O sub_requests on busy chips to idle chips to alleviate I/O contentions and further reduce I/O response time.

2) *Scattering data chunks in a parity group.* Traditional RAID-5 usually encodes data chunks on adjacent chips into a parity group. Due to temporal and spatial localities, adjacent chips tend to be busy or idle at the same time. So redirected read performs poor with traditional chip-level RAID-5. We are to design a new layout of RAID-5 to scatter the data chunks in a parity group and distribute adjacent chips into different parity groups.
3) *Minimizing extra I/Os interference.* Usually redirected read will introduce extra I/Os, which may induce heavy load on a RAID. So we also need to limit extra I/Os when we redirect I/Os with BRR.

B. Overview of System Architecture

We implement a simulation system to realize and evaluate BRR. The system is composed of three key modules, write module, read module, and detection module, as depicted in Fig. 7. Note that we aim to improve read performance of SSDs with chip-level RAID-5. So we do not need to change SSD's internal except RAID coding module compared with existing I/O schedulers.

Host interface logic (HIL) receives I/O requests from host system, delivers them into write/read modules, and returns accessed data and/or related information to host system. An I/O request would be divided into multiple sub_requests of page size and each sub_request accesses one data chunk on its target chip.

Write module buffers write requests, encodes data chunks and generates parity chunks. We use a write buffer module to divide incoming write requests into parity groups. We also design a data layout of RAID-5 with longest code interval (LCI) to discretize the chips of storing the data/parity chunks in a parity group. Combined with detection module, we can reschedule some write requests according to the status of pending queues on chips.

Read module consists of decode module and detection submodule. The decode submodule serves redirected read by locating parity group, reading corresponding chunks, and performing XOR operations to get target chunks. The detection submodule is designed to limit extra I/Os induced by redirected read, i.e., if the extra I/Os induced by a redirected read need more time slots than direct read, we will use direct read regardless of contentions.

Detection module consists of I/O contentions detection and queue monitor. We can separate requests with contentions into different batches according to I/O contentions in the pending queue at SSD controller. And we can get all of the chips' queue status from queue monitor to estimate the waiting time of each sub_request.

C. New Data layout of RAID-5

To improve the performance of BRR on SSDs with chip-level RAID-5, we design a new data layout of RAID-5. Suppose that there are n chips C_0, C_1, ..., C_{n-1} in an SSD. We divide all chips into m parity groups, each with $n/m = t$ chips. From the view of practical implementation, n should be divisible by m, i.e, t is an integer. The i-*th* parity group consists of chips $C_{0+i}, C_{t+i}, ..., C_{(m-1)\times t+i}$, where $0 \leq i \leq t-1$. For example, there are 9 chips divided into 3 parity groups in Fig. 8, where Group G_0 consists of C_0, C_3, C_6, Group G_1 consists of C_1, C_4, C_7, Group G_2 consists of C_2, C_5, C_8.

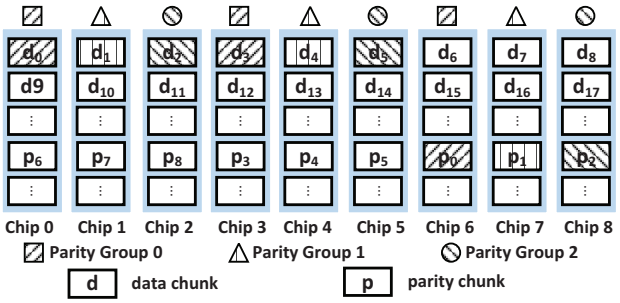

Fig. 8. Overall architecture of LCI RAID-5.

Due to spatial locality of I/O access, adjacent chips in an SSD tend to be busy or idle at the same time. With our new data layout of chip-level RAID-5, adjacent chips are divided into different parity groups such that chips in a parity group are more likely with different levels of workloads. So we can more likely migrate I/O requests from busy chips to idle chips with redirected read and improve the performance of BRR.

To realize chip-level RAID-5 in an SSD, we should buffer the data chunks in the pending queues of all chips. When each chip in a parity group (except for the chip to store the parity chunk in this group) has a data chunk in the buffer, we will encode the data chunks into a parity chunk and write it to the corresponding chip.

The oncoming data chunks of write requests are written into different chips of an SSD in round-robin to efficiently realize

parallelism of different chips. But our new data layout of chip-level RAID-5 distributes adjacent chips into different parity groups, which make chunks of a write request be written into different parity groups. So it may last long for all chips in a parity group have data chunks in buffer. To avoid data chunks in some chips waiting too long for being encoded, they will be encoded regardless of the data chunks in other chips if their waiting time exceeds a predefined threshold. But for a request to update a data chunk, it will be processed as usual, i.e., reading old data chunks and old parity chunk, writing new data chunk and new parity chunk. Note that we apply this technique to all of FIFO, PIQ, and BRR.

To relieve I/O contentions, we can also buffer parity chunks temporarily and write them into chips during idle time as shown in Lee's study [17]. Delaying write requests behind read requests in the pending queues can also boost SSDs' performance because the latency of a write operation is much longer than a read operation. In our experiments, we also delay write requests to reduce waiting time of read requests. Note that, we only delay write requests behind read requests in pending queues, and always process the delayed write requests prior to new coming read requests.

Algorithm 1 I/O Batch

Input:
I/O Request: Req
Output:
$BatchList$

```
1: Calculate target chip numbers of Req: TCNs;
2: Set Batched ← False;
3: Choose Read/Write Batch List according Req's type;
4: while Batch is not empty do
5:    Compare TCNs with current batch's TCNs;
6:    if No contention then
7:       Add Req into current Batch;
8:       Update Batch's TCNs;
9:       Set Batched ← True;
10:      Break;
11:   else
12:      Compare with next Batch;
13:   end if
14: end while
15: if Batched is False then
16:   Create a new batch and add it to BatchList;
17: end if
```

D. Detecting I/O Contentions

Detection module analyzes the requests in the pending queue of SSD controller and detects the contentions among them. It also estimates the waiting time of the requests in the queues of all chips, which serves for request scheduling.

Detecting contentions in the pending queue of SSD controller. According to the LBN and size of a request, we can get its target chips which will be accessed. If two requests target the same chip, they contend with each other. We group the requests without contentions into a *batch* and set their target chips as the target chips of the batch. When requests do have contentions with all previous batches, we create another new batch, and add the batch to batch-list. The batch algorithm is shown in Alg. 1. Requests in the pending queue of SSD controller are dispatched to chips in batch by detecting contentions.

Garbage collection (GC), wear leveling (WL) and normal updating across different chips may introduce out-of-place writes, and change some LBNs' mapping table. Therefore, the target chips would be changed due to data movement across chips, and further requests contention could not be detected by LBN and size. However GC, WL and normal updating in many modern SSDs are performed within a chip [2, 6]. Thus, the LBNs' target chips remain unchanged under this situation. Furthermore, dependencies between I/O requests in the pending queue do not exist, since they have been avoided when requests are added to the queue using traditional approaches as described in [6].

Estimating the expected waiting time on chips. We can estimate the expected waiting time on a chip as $L_r \times C_r + L_w \times C_w$, where L_r, L_w are the latencies of reading a chunk and writing a chunk respectively, and C_r, C_w are the numbers of read sub_requests and write sub_requests in the queue of the chip respectively. We use an array WT recording the expected waiting times of all chips. Meanwhile, we use an array AT recording the next earliest arrival time of each chip relative to current sub_request by searching the queue monitor. Once sub_requests reach chips, we update WT by getting C_r and C_w from the queue monitor. Note that there may be an error for estimating the expected waiting time when one request is completed in a short time while we may still consider it in the queue. Whereas the maximum error would be the latency of writing a chunk, and for most applications with heavy workloads, the error is tolerable because of relatively large queue length.

Since response time to a request is determined by the longest one of its sub_requests, it is necessary to estimate the expected waiting time for balancing workloads on different chips, which has not been considered by PIQ.

E. Balanced Redirected Read

A simple example. We first use a simple example in Fig. 9 to explain how BRR deals with the pending queue described in Fig. 5 of Section II. There are nine chips in an SSD with the new data layout of chip-level RAID-5. The pending I/O requests R_0, ..., R_6 are to access chips C_0 to C_8. With FIFO or PIQ, C_4, C_5, C_6, C_7 are always idle. PIQ takes five time slots to complete I/O requests. Note that in Fig. 9(a) asynchronous PIQ is different from PIQ in Fig. 5(b) in processing R_3 and R_5. While BRR redirects R_5 and R_6 to C_4, C_5, C_7 and C_8, such that the redirected sub_requests R_{5_b}, R_{6_b} are processed concurrently with R_0, R_1, R_2, and R_3. Thus BRR can complete all requests within three time slots.

Algorithm Design of BRR. We design BRR to reduce the response time to read requests. When a read request comes,

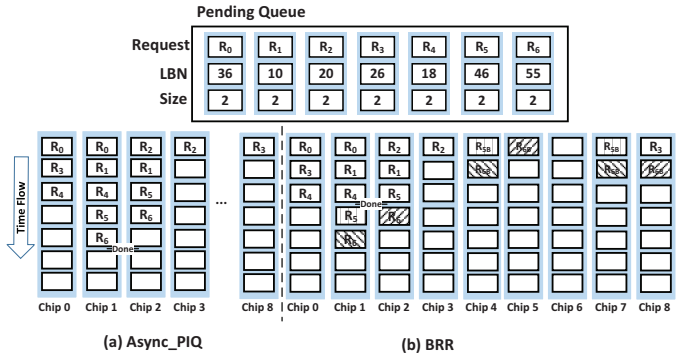

Fig. 9. PIQ and BRR are applied to the queue respectively.

Algorithm 2 BRR: Balanced Redirected Read

Input: $I/ORequest : Req$
Parity Group Length : L

1: Call I/O batch API to avoid I/O contentions;
2: **for** Each original sub_request **do**
3: Calculate original target chip number OCN and its redirected target chips numbers RCNs;
4: Estimate WT and get AT from queue monitor;
5: set $i \leftarrow 0$
6: **while** $i \neq L-1$ **do**
7: **if** $WT_j + L_r \leq WT_{OCN}$ && $WT_j + L_r \leq AT_j - Req.time$, $(j = RCN_i)$; **then**
8: $i \leftarrow i+1$, and Continue;
9: **else**
10: Break;
11: **end if**
12: **end while**
13: **if** $i = L-1$ **then**
14: Do redirected read to access encoded chips;
15: **else**
16: Access original target chip;
17: **end if**
18: **end for**

BRR decides whether to issue this request with direct read or redirected read. The details of BRR are shown in Alg. 2. In Line 1, we first reorder and group read requests into batches to avoid contentions in the pending queue of SSD controller. Then for each sub_request, we get its target chips with direct read and redirected read, and the expected waiting time on all of its target chips in Line 2 and 3. From Line 5 to 16, we compare its waiting time with direct read and redirected read, and select the one with less waiting time to process the request. Meanwhile, we can have no interference with following requests due to Line 7. Note that it takes time to decode data chunk with redirected read. But it is executed in memory and takes negligible time compared with processing SSD I/O operation.

F. Implementation Issues

In this subsection, we analyze the storage overhead and computational overhead of BRR, etc.

Storage overhead. Suppose there are n chips in an SSD. BRR needs to record the target chips of I/O requests, each request with n bits, and to record the expected waiting time on all chips with an array of n double variables. Because n bits are much smaller than the size of I/O requests and the array is a global variable, the extra storage overhead to execute BRR is negligible compared with FIFO and PIQ methods.

Computational overhead. BRR needs to know the target chips of read requests and the expected waiting times of all chips, and also needs decoding data chunks with redirected read. Because all of the computation can be realized in memory with operations, such as MOD, XOR and ADD. So the computational overhead of BRR is negligible compared with I/O delay.

Modification of FIFO, PIQ and RAID-5. To validate the advantage of BRR, we also implement asynchronous FIFO and asynchronous PIQ as the baseline in our experiments. Asynchronous scheduling reduce processing time for most applications, so it is often deployed in practical systems even though it requires higher management complexity. We modified PIQ to be asynchronous and applicable to RAID-5 (PIQ intends for serving RAID-0 originally) by changing the mapping scheme. Compared with traditional RAID-5, our new data layout of chip-level RAID-5 needs negligible modification of mapping table which can be configured at system initialization.

IV. Performance Evaluation

In this section, we evaluate the performance of BRR atop a simulated system with real-world workloads. Our design focuses on alleviating I/O contention on SSDs with chip-level RAID-5, which is also studied by recent works [6, 18]. Therefore, we choose asynchronous FIFO and the state-of-the-art reordering method, PIQ, as baseline algorithms to validate the efficiency of BRR.

In the following subsections, we first introduce our system configurations and the workloads used in the experiments, then present the experimental results, and discuss the impact of different parameter settings on the performance of BRR.

A. System Configuration

We use a trace driven simulator, DiskSim [5] with SSD extension [2]. DiskSim with SSD extension has been widely applied in the exploration of parallelism of SSDs. We modelled an SSD of 256GB, which is configured with 8 channels, and each channel connecting to 8 chips. The other main parameters of SSD configuration are listed in Table I. Greedy garbage collection and dynamic wear leveling are implemented. The over-provisioning ratio is set to 15% of the SSD's capacity, and for data allocation, the most widely used round-robin method is applied in page-level address mapping [2].

We use 64 as the default queue length for SSD chips and 8 as the default length of parity group which coding layout is set

TABLE I
DISKSIM CONFIGURATION

Parameter	Value
Page Size	4KB
# of pages per block	64
# of blocks per chip	16384
# of chips per SSD	64
Page read latency	0.025ms
Page write latency	0.200ms
Block erase latency	1.500ms

as LCI RAID-5. In the experiments we vary the length of I/O queue and the length of parity group to gain further insight into how the proposed approach behaves under various system settings.

The workloads in our experiments are chosen from MSR Cambridge traces on servers [1], which are widely used in previous works on improving SSDs' performance [6]. Table II shows the statistics of these traces, where chip utilization is collected with Async_FIFO schedule method. Note that we pre-process the traces to make all requests' size be aligned with 4KB.

B. Performance Evaluation

The response time to a request consists of 4 parts, T_w, T_b, T_c and T_m, as presented in Section II. Note that T_c, T_b and T_m are linearly related to the size of a request, whose transfer rates are determined by hardware property. We believe that T_w is the only part that can be improved with optimized requests scheduling. Thus we evaluate the performance of BRR with T_w. We record the response time to each read and write request in simulations and present the average read and write waiting time.

1) The average waiting time of read requests. Fig. 10 shows the results of the normalized read waiting time with Async_FIFO, PIQ, and BRR. On average, BRR reduces the read waiting time by 23.8% (14.2%) compared to Async_FIFO (PIQ). BRR performs the best under HM_0 workload with a reduction of waiting time by 77.1% (38.4%), while BRR performs the worst under PROJ_3 workload, only with a reduction by 8.6% (1.5%). The main reason is that only very few read requests can be redirected under PROJ_3 workload. Redirected read induces extra read I/Os (reading seven corresponding chunks in a parity group instead of reading the original chunk). These extra I/Os should locate on relatively idle chips and they can be processed in parallel, so we limit the triggering of redirected read with Line 6-14 in Alg. 2, which makes the extra I/Os of redirected read only lightly interfere with other I/Os.

2) The average waiting time of write requests. Fig. 11 shows the average waiting time of write requests with Async_FIFO, PIQ and BRR. From Fig. 11, we can see that BRR performs the best with all workloads, but only a little reduction of write waiting time by 3.2% (0.6%) compared with Async_FIFO (PIQ). This is because we only redirect read requests to improve read performance, while use the same schedule scheme for write requests with PIQ. BRR reduces the waiting time of write requests only from the reduction of read time such that some write requests can be processed earlier.

C. Chip Utilization of Flash Array

We define chip utilization as the average ratio of busy chips to all chips when a request is sent to the SSD. Fig. 12 shows the comparison of chip utilization of Async_FIFO, PIQ and BRR, where chip utilization of Async_FIFO is normalized as 1. On average, BRR improves the chip utilization by 7.3% (4.3%) compared with Async_FIFO (PIQ). Intuitively the improvement of chip utilization will reduce the response time to read requests. In our experiments, we do validate this phenomenon, but we do not find any further relationship between read performance and chip utilization. We believe that there are three reasons why we can not find further relationship: (1) Both write requests and read requests access chips and contribute to chip utilization, but BRR only redirects read requests. (2) The response time is determined by the longest latency of sub_requests. If BRR redirects some sub_requests on the chips not with the longest queue, it will improve chip utilization but will not reduce the response time. (3) If BRR redirects a read request to some chips with shorter pending queues and all of these pending queues are not empty, it will not improve chip utilization, but will reduce the response time to the read request.

D. Sensitivity to Queue Size and Parity Group Size

To validate the advantage of BRR with different system settings, we present the experiment results by varying the size of I/O queues from 1 to 256 and the number of chips in a parity group (*abbr. parity group size*) from 4 to 16 in this subsection.

Fig. 13 shows the results with different queue sizes and fixed parity group size of 8. On average, from 1 to 256, the read waiting time can be reduced by 24.4%. We can find that the waiting time becomes shorter as queue size increases. This is because that the larger queue size becomes, the larger variance of queue lengths on different chips is. So redirected read will more likely shorten the longest queue of sub_requests. When queue size increases from 1 to 32, the read waiting time decreases evidently. But when the queue size increases from 32 to 256, the waiting time further reduces by 2.9% only. The reason is that when queue size reaches a threshold, the I/O requests are not intensive enough to get more parallelism.

Fig. 14 shows the results with different parity group size and fixed queue size of 64. We can find that as the parity group size increases, the waiting time becomes longer. When the parity group size increases from 4 to 16, the average read waiting time increases by 17.6%. We believe that there are two main reasons: (1) The more chips are in a parity group, the more extra I/Os will be induced by redirected read. (2) It becomes more difficult to get enough relatively idle chips under the limitations of Line 6-14 in Alg. 2, when the parity group size is large. Due to the storage overhead and fault-tolerance in practical systems, the parity group size should

TABLE II
STATISTICS OF I/O WORKLOADS.

Trace	Total # of requests	Read Ratio	Average Size	ASync_FIFO Chip Utilization
HM_0	3993316	0.35503	8.887619	0.157154
HM_1	609311	0.953365	15.166593	0.349508
PRN_0	5585886	0.107858	12.537686	0.202753
PRN_1	11233411	0.753449	19.807669	0.272018
PROJ_1	23639742	0.894376	34.6365015	0.278648
PROJ_3	2244644	0.94817	9.749589	0.285729
PRXY_0	12518968	0.030635	7.073009	0.085106
RSRCH_0	1433655	0.093206	9.0778855	0.214134
SRC2_0	1557814	0.113447	7.594609	0.20702
STG_0	2030915	0.151871	11.9593995	0.197961
USR_0	2237889	0.404168	22.815757	0.239402
WDEV_0	1143261	0.200767	9.281464	0.235306

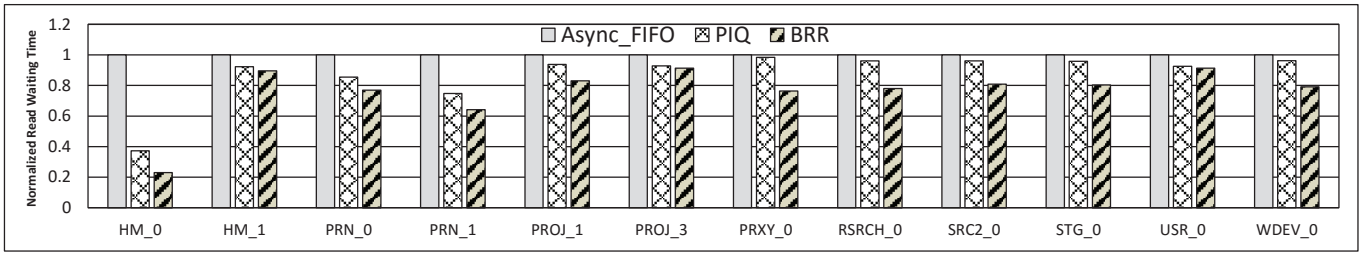

Fig. 10. Normalized read waiting time of Async_FIFO, PIQ, and BRR

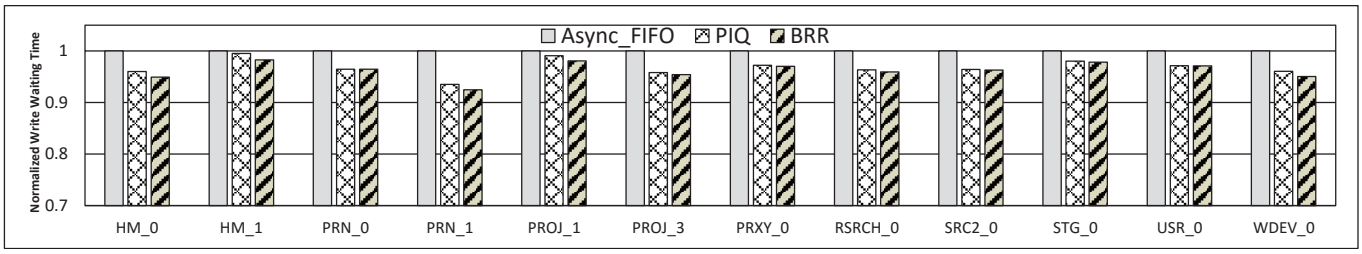

Fig. 11. Normalized write waiting time of Async_FIFO, PIQ, and BRR

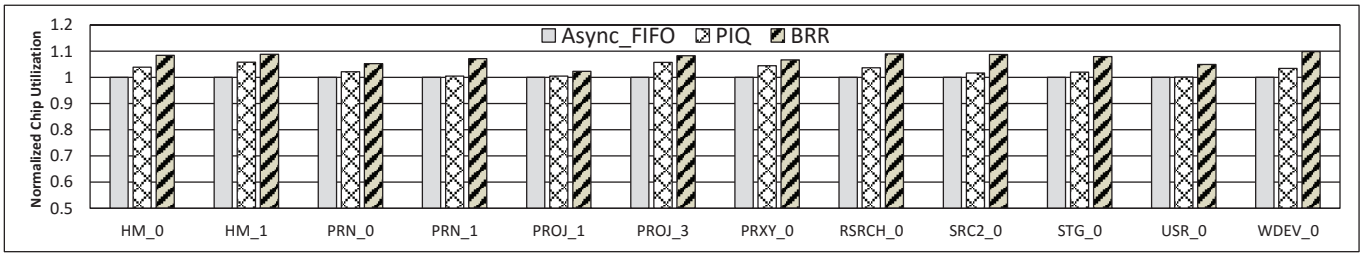

Fig. 12. Normalized chip utilization of Async_FIFO, PIQ, and BRR

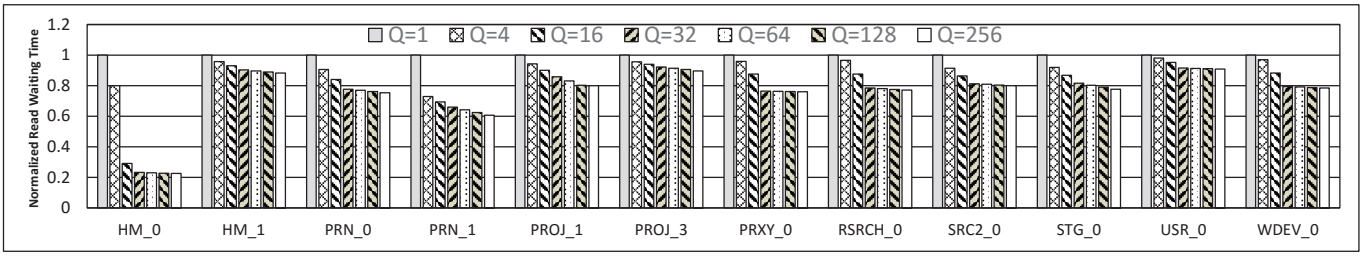

Fig. 13. Normalized read waiting time of BRR with various queue sizes

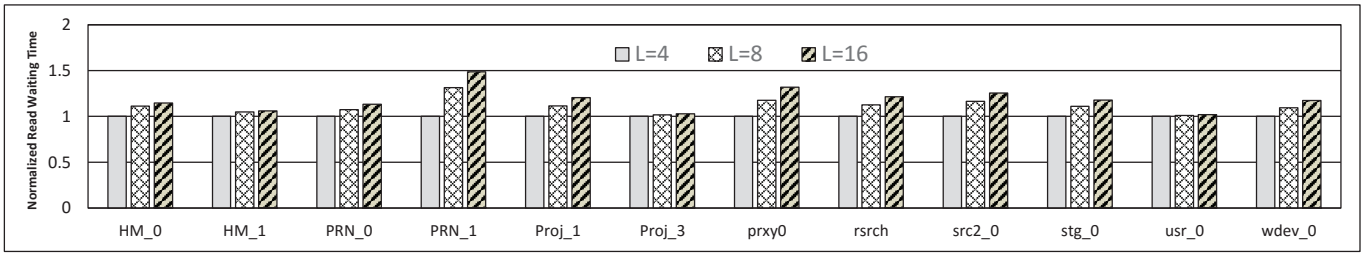

Fig. 14. Normalized read waiting time of BRR with various lengths of parity group

not be too small or too large, e.g, as extremal, when it is 2, the code would be a type of replica and when it is 64, the SSD only tolerates single chip failure.

V. Conclusions

In this paper, we address the I/O contention problem in SSDs with chip-level RAID-5. We first present a method to design the balanced redirected read scheme which can improve the I/O efficiency, and then propose a new RAID-5 layout to further enhance the performance of redirected read. For practical deployment, we implement our scheme with different system settings, such as different queue sizes, different lengths of parity group, atop a widely-used trace driven simulator. Our extensive experiments demonstrate that our scheme reduces the waiting time of read requests significantly and increases chip utilization over the existing schemes.

Acknowledgements

This work was supported by National Nature Science Foundation of China under Grant No. 61379038 and No. 61303048 and Huawei Innovation Research Program under Grant No. HIRPO20140301.

References

[1] SNIA. IOTTA repository. http://iotta.snia. org/.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX ATC*, pages 57–70, 2008.

[3] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 181–192. ACM, 2009.

[4] Y. Du, Y. Zhang, N. Xiao, and F. Liu. CD-RAIS: Constrained Dynamic Striping in Redundant Array of Independent SSDs. In *Cluster Computing, IEEE International Conference on*, pages 212–220. IEEE, 2014.

[5] G. Ganger, B. Worthington, and Y. Patt. The DiskSim Simulation Environment (v4. 0), 2009.

[6] C. Gao, L. Shi, M. Zhao, C. J. Xue, K. Wu, and E. H. Sha. Exploiting Parallelism in I/O Scheduling for Access Conflict Minimization in Flash-based Solid State Drives. In *Mass Storage Systems and Technologies (MSST), 30th Symposium on*, pages 1–11. IEEE, 2014.

[7] L. M. Grupp, J. D. Davis, and S. Swanson. The Bleak Future of NAND Flash MSemory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 2–2. USENIX Association, 2012.

[8] S. S. Hahn, S. Lee, and J. Kim. SOS: Software-based Out-of-order Scheduling for High-performance NAND Flash-based SSDs. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–5. IEEE, 2013.

[9] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang. Efficient Identification of Hot Data for Flash Memory Storage Systems. *ACM Transactions on Storage (TOS)*, 2(1):22–40, 2006.

[10] S. Im and D. Shin. Flash-aware RAID Techniques for Dependable and High-performance Flash Memory SSD. *Computers, IEEE Transactions on*, 60(1):80–92, 2011.

[11] M. Jung, W. Choi, J. Shalf, and M. T. Kandemir. Triple-A: A Non-SSD based Autonomic All-flash Array for High Performance Storage Systems. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 441–454. ACM, 2014.

[12] M. Jung and M. T. Kandemir. Sprinkler: Maximizing Resource Utilization in Many-chip Solid State Disks. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 524–535. IEEE, 2014.

[13] M. Jung, E. H. Wilson III, and M. Kandemir. Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 404–415. IEEE Computer Society, 2012.

[14] J. Kim, J. Lee, J. Choi, D. Lee, and S. H. Noh. Improving SSD Reliability with RAID via Elastic Striping and Anywhere Parity. In *Dependable Systems and Networks (DSN), 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[15] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk Schedulers for Solid State Drivers. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 295–304. ACM, 2009.

[16] S. Lee, B. Lee, K. Koh, and H. Bahn. A Lifespan-aware Reliability Scheme for RAID-based Flash Storage. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 374–379. ACM, 2011.

[17] Y. Lee, S. Jung, and Y. H. Song. FRA: A Flash-aware Redundancy Array of Flash Storage Devices. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 163–172. ACM, 2009.

[18] P. Li, F. Wu, Y. Zhou, C. Xie, and J. Yu. AOS: Adaptive Out-of-order Scheduling for Write-caused Interference Reduction in Solid State Disks. In *IMECS*, volume 1, 2015.

[19] A. Miranda and T. Cortes. CRAID: Online RAID Upgrades Using Dynamic Hot Data Reorganization. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 133–146, 2014.

[20] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-Defined Flash for Web-scale Internet Storage Systems. In *ACM SIGPLAN Notices*, volume 49, pages 471–484. ACM, 2014.

[21] D. A. Patterson, G. Gibson, and R. H. Katz. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, volume 17. ACM, 1988.