Improving Performance and Lifetime of the SSD RAID-based Host Cache through a Log-Structured Approach

Yongseok Oh
University of Seoul
ysoh@uos.ac.kr

Jongmoo Choi
Dankook University
choijm@dankook.ac.kr

Donghee Lee
University of Seoul
dhlee@uos.ac.kr

Sam H. Noh
Hongik University
http://next.hongik.ac.kr

ABSTRACT

This paper proposes a cost-effective and reliable SSD host cache solution that we call SRC (SSD RAID Cache). Cost-effectiveness is brought about by using multiple low-cost SSDs and reliability is enhanced through RAID-based data redundancy. RAID, however, is managed in a log-structured manner on multiple SSDs effectively eliminating the detrimental read-modify-write operations found in conventional RAID-5. Within the proposed framework, we also propose to eliminate parity blocks for stripes that are composed of clean blocks as the original data resides in primary storage. We also propose the use of destaging, instead of garbage collection, to make space in the cache when the SSD cache is full. We show that the proposed techniques have significant implications on the performance of the cache and lifetime of the SSDs that comprise the cache. Finally, we study various ways in which stripes can be formed based on data and parity block allocation policies. Our experimental results using different realistic I/O workloads show using the SRC scheme is on average 59% better than the conventional SSD cache scheme supporting RAID-5. In case of lifetime, our results show that SRC reduces the erase count of the SSD drives by an average of 47% compared to the RAID-5 scheme.

Categories and Subject Descriptors
B.3.2 [Memory Structure]: Design Styles—Mass storage; D.4.2 [Operating Systems]: Storage Management—Secondary storage, Storage hierarchies

General Terms
Design, Performance, Reliability

Keywords
SSD RAID Cache, Log-Structured Approach, Destage, Replacement, Parity, Performance, Lifetime, Cost-effective

1. INTRODUCTION

Solid State Drives (SSDs) are becoming popular in computer systems due to their many positive traits. Accordingly, they are now being considered as high-performance data caches to improve networked storages such as NAS and SAN based storage [1, 2]. Many storage vendors have been developing such solutions to improve I/O performance in cloud computing systems [3, 4, 5]. The main goal of SSD based cache solutions is serving SSD-like performance with HDD-like cost-effectiveness.

For cost-effectiveness, manufacturers are using MLC (Multi-Level Cell) flash memory chips that store two bits per cell in their SSDs. Recently, TLC flash memory chips that store three bits per cell are being considered as SSD components due to their low cost [6]. However, performance and endurance of TLC flash memory chips is considerably limited [7]. For example, erasing a block and writing data in TLC chips are both much slower than SLC or MLC chips.

The erase count of TLC blocks is expected to be only around 1,000 times, which is at least a magnitude smaller than that of MLC and SLC chips. Furthermore, bit error rates of MLC and TLC flash chips are higher than that of SLC chips [7, 8]. These negative aspects may result in disastrous data loss, especially if the SSD cache solution uses the write-back policy and new data is being stored in the SSD. Furthermore, current SSD cache architectures may not be robust against drive failures and data corruption. Therefore, additional data protection mechanisms may need to be deployed to safeguard data residing in the SSD cache.

One such data protection mechanism may be to use conventional RAID techniques in managing multiple low price SSDs. Though this approach is a viable one, it still suffers from performance and reliability limitations of its own. First, in RAID-5, the overhead of parity updates through either read-modify-write or reconstruct-write is inevitable when we directly use the RAID-5 technique as an SSD cache solution. This parity overhead may lower the performance of an SSD cache solution. In addition to this problem, RAID-5 quickly shortens the lifetime of the SSDs as parity manipulation incurs more writes. With more writes, reliability of the SSD cache will suffer and wear-out will be aggravated as well [8, 9].

This paper proposes a cost-effective SSD cache solution that we call SRC (SSD RAID Cache), which provides cost-effectiveness by using multiple low-cost SSDs and reliability by retaining data redundancy through RAID. Specifically, SRC dynamically stores the updated data with parity data in a log-structured manner on multiple SSDs. This has the
effect of eliminating the read-modify-write operation, which drastically reduces the parity manipulation overhead. In addition, within the proposed framework, we propose to eliminate parity blocks for stripes that are composed of clean blocks. This is possible as the SSD comprised RAID is a cache of the primary storage. Hence, if the data blocks are clean, this means that the original copy resides in primary storage and hence, the cache version may be lost without compromising reliability. We also propose the use of destaging, instead of garbage collection, when the SSD cache is full. We show that the performance and lifetime implications of these techniques are significant. Finally, we study various data and parity block allocation policies in forming stripes within our proposed framework. We discuss the pros and cons of the various approaches for the various workloads that we consider.

Our experimental results using different realistic I/O workloads show that using the SRC scheme is on average 59% better, with a maximum of 68% and minimum of 55%, than the conventional SSD cache scheme supporting RAID-5. In case of lifetime, our results show that SRC reduces the erase count of the SSD drives that comprise the cache by an average of 47%, with a maximum of 87% and minimum of 27%, compared to the RAID-5 scheme.

The remainder of this paper is organized as follows. Section 2 summarizes related work. In Section 3, we describe the design of the SRC scheme, while in Section 4 we describe how we implemented SRC. Section 5 discusses the evaluation environment and gives a detailed discussion of the results of the evaluation. Finally, we conclude the paper with a summary and conclusion in Section 6.

2. RELATED WORK

In this section, we review existing work related to this study. First, we discuss schemes related to SSD caches. These recent studies have focused on how to improve performance and to enhance the lifetime of SSDs. Then, we review studies that propose cost-effective storage solutions that employ multiple low price SSDs or HDDs.

2.1 SSD Based Cache

NAND flash based SSDs have been used as a data cache for several purposes. Kgil et al. propose a novel flash cache management scheme that identifies and allocates read and write requests separately into read and write regions so that the garbage collection cost and the performance of flash memory cache is optimized [10]. As MLC has strength in price while SLC in performance and endurance, Hong et al. utilize SLC programming mode of flexible MLC flash memory chips to harvest the advantages of both SLC and MLC [11]. In addition to these work, Seagate tries to develop and sell a real hybrid HDD product that integrates raw flash memory into an HDD [12].

There have been studies that use commodity SSDs as a block-device cache for hybrid storage systems. Chen et al. propose a hybrid storage solution, namely Hysort, that combines low-cost HDDs and high-speed SSDs [13]. Pritchett and Thottethodi propose a highly-selective caching mechanism to capture data that are to be likely re-accessed in SSD cache [14]. Oh et al. introduce an optimal partitioning scheme that dynamically splits the cache spaces into read, write and over-provisioned spaces according to workload patterns [15]. Saxena et al. suggest a unified logical address space within the OS and SSD cache to reduce cache block management cost [16]. These studies attempt to minimize the high cost of data caching operations in flash memory storage as writes are much more expensive than reads. Also SSDs have been used for networked storage systems. Byan et al. introduce a host-side cache approach where a write-through SSD cache is deployed in the enterprise Hypervisor environment [1]. To overcome consistency limitations of the write-back caching scheme, Koller et al. provide consistent write-back policies [2]. These studies have only focused on improving the performance of lifetime of SSD cache where they deploy a single SSD device. However, our study is quite different in that data corruption and SSD drive failures are considered.

2.2 Cost-effective Storage

RAID is a cost-effective and reliable storage system [17]. Unfortunately, it suffers from the small write problem when the parity protection mechanism is employed. That is, when a write request arrives in RAID, the modified data and its related parity must be updated together for data recoverability. In contrast to large sequential writes, the parity can be updated through either read-modify-write or reconstruct-write techniques. These operations incurs additional read and write operations, leading to significant performance degradation.

To overcome the small write problem, many techniques have been introduced [18][19]. Wilkes et al. suggest a hybrid approach where frequently updated data are stored in RAID-1, while infrequently updated data are stored in RAID-5 [20] so that parity overhead of RAID-5 is minimized. Our study is similar to this work in that a log-structured approach is employed to reduce parity overhead. However, our scheme does not incur any GC operations that copies valid blocks to other free stripes in RAID. Also, our scheme is targeted for reliable cache systems, not for primary storage systems.

SSD based RAID solutions have focused on improving reliability and performance with low price SSDs. Balakrishnan et al. develop a skewed parity distribution scheme to reduce data loss probability [8]. Kim et al. propose a globally coordinated garbage collection (GC) scheme that synchronously triggers all GC processes of SSDs depending on requests so that performance is optimized [21]. Moon et al. study reliability of SSD based RAID and provides a model to calculate the lifetime of RAID [9].

3. DESIGN OF SRC

In this section, we describe the design of the SRC scheme, first concentrating on the key features of SRC. Then, we look into the detailed layout of the blocks in the SRC scheme and the various allocation policies in forming a stripe in SRC.

3.1 Key Features of SRC

SRC is essentially a RAID management based caching scheme with the underlying cache composed of low-cost SSDs. The goal of SRC is to provide a cost effective, reliable host-side cache for mass storage systems.

The overall architecture of the storage system employing SRC is shown in Figure 1. The storage system consists of two layers, namely, the SRC Layer and the Primary Storage Layer. The Primary Storage Layer has an HDD based system and provides large storage service to the upper layer.
We note that this layer is not limited to specific storage systems. The SRC Layer consists of numerous SSD drives. We discuss the various software components of SRC later.

Before discussing the novel features that are introduced with SRC, we first discuss the various policies involved with the cache aspect of SRC. First, the cache space is logically partitioned into the read space and the write space. The read caching space keeps the clean data copied from the lower level storage and the write caching space keeps the dirty data updated from the upper layer. Note that the read caching space may also incur writes if the SRC Layer replaces invalid data blocks. SRC data blocks are brought up from the Primary Storage Layer.

Second, we assume that the write cache uses the write-back policy instead of the write-through policy. The write-through policy does not take advantage of the features of SSDs such as high random performance because it needs to wait for acknowledgements for write requests from both the Primary Storage Layer and the SRC Layer. With SSDs, which have non-volatility characteristics, the write-back policy is more efficient than the write-through policy without any loss in consistency.

Finally, there is the replacement issue. In SRC, the cache is managed in stripe units. Hence, when space is in need, a stripe has to be “evicted”. The stripe to be evicted is determined using a clock-based policy. If the stripe selected to be evicted is a read stripe, that is, a stripe in the read caching space, this stripe is simply thrown away as the original copy of this cached data resides in the Primary Storage Layer. However, if the selected stripe is a stripe that is in the write caching space, this stripe must be destaged to storage. We will discuss this aspect in more detail later.

Let us now discuss the novel features of SRC that we propose. SRC has three key features. The first is that SRC employs the log-structured scheme that stores data blocks sequentially in SSDs [22]. Employing the log-structured scheme removes all read-modify-write operations that are detrimental to RAID performance. We do not claim to be the first to make use of the log-structured scheme for RAID [20, 23, 24]. Nevertheless, we claim that the use of the log-structured based RAID scheme in the cache allows for other features as explained below.

The second feature is that SRC does not maintain parity blocks for stripes in read caching space. Note that read requests that miss in SRC generate data writes to SSD cache as data must be brought into the SSD cache from Primary Storage Layer. However, even though SRC is RAID based, data brought into the SSD cache by read requests need not have parity blocks as their original is in the lower layer. If one of the SSDs ever fault, the original data can always be found. Omitting parity for stripes in read caching space improves performance without compromising the robustness against SSD failures.

The final feature involves replacement. Typically, when an SSD runs out of space, garbage collection (GC) is performed to make space. GC collects valid data scattered over various blocks into an empty block and then sets the now empty blocks as free blocks. GC operations are known to be costly and to hurt the performance and lifetime of SSDs.

As SRC is composed of SSDs, a similar situation can arise. With SRC, however, we take a different approach. Instead of GC, SRC actively applies destaging to make room in SSD cache. Rather than copying valid data in stripes to other stripes as would be done with GC, the destaging scheme simply removes clean data in the SSD cache or writes back dirty data to HDDs and reclaims their space. Judicious selection of data to be destaged is crucial for performance. To select data to be destaged, we use a per-stripe clock bitmap, which has one bit per stripe. Similarly to the well-known clock algorithm [25], if a read is requested to data in a stripe, its corresponding bit, which is initially 0, is set. For destaging, the bits in the bitmap are examined in clock-wise direction starting from the last examined bit. If it finds a 0 bit, it destages data in the corresponding stripe; otherwise it sets the bit to 0 and continues to examine the other bits. We use this per-stripe clock bitmap because it is simple and, more importantly, scalable, which is a crucial requirement for huge SSD caches.

3.2 Internal Organization of SRC

We now explain the layout of SRC on the multiple SSDs. Figure 2 shows the internal stripe layout of SRC. The caching space provided by the numerous SSD drives is split into \( N \) stripes similar to RAIDs. These stripes are managed in a log-structured manner. Each stripe comprises of SSDs that forms a RAID system, and in each SSD several blocks\(^1\) can also be a part of the same stripe. In addition to storing

\(^1\)Note here that the term block refers to the logical block as seen by the upper layer, for example, at the file system. It is not referring to the flash memory block unit.
cached data, the SRC scheme maintains parity data to recover data against drive failures or data corruption. In our example diagram, we have 4 blocks for each SSD in the same stripe for a total of 16 blocks. data blocks D0∼D11 and parity blocks P0∼P3, forming a single stripe. In the case a drive fails, SRC enters degraded mode and data caching is disabled until a new drive is added.

Given the internal organization of the stripes in SRC, the allocation of various blocks can affect the performance of the cache. In particular, how the read and write blocks will be distributed among the stripes has to be determined, which we refer to as the allocation issue.

For allocation, we consider the data and parity blocks separately. For the parity blocks, we consider two types of parity distributions, namely ‘Fixed’ and ‘Rotated’. The Fixed approach stores parity blocks in a dedicated SSD like RAID-4, while the Rotated approach distributes parity blocks in a rotated manner like RAID-5.

For data blocks, we again separate them into read and write blocks. Then, there are two ways to allocate them among stripes. One is to simply ignore the distinction and allocate read and write blocks together in a stripe. We refer to this type of allocation ‘Mixed’ striping. The other method is to distinguish the read and write blocks so that each stripe will consist of only read or write blocks. We refer to this type of allocation ‘Separated’ striping. In this case, as noted previously, read stripes do not hold parity blocks.

Combining the two parity distribution methods and the two read-write block allocation methods, we have a total of 4 different allocation strategies, namely, Mixed-Fixed, Mixed-Rotated, Separated-Fixed, and Separated-Rotated, which are depicted in Figure 3. We will compare and discuss the quantitative performance implications of these approaches in Section 5.

4. IMPLEMENTATION

As depicted in Figure 1, the software components of SRC, as shown in the SRC Layer are the **Sequential I/O Detector, Page Replacer, Mapping Manager, and Parity Writer**. The Sequential Detector identifies consecutive I/O requests and makes them bypass the SRC Layer to avoid cache pollution.

---

**Table 1: Configuration of storage system**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDD</td>
<td>Model</td>
<td>Maxtor Atlas 10K</td>
</tr>
<tr>
<td>No. of Disks</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>No. of SSDs</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>No. of Packages</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Cleaning Policy</td>
<td>Greedy</td>
<td></td>
</tr>
<tr>
<td>Reserve Space</td>
<td>10%</td>
<td></td>
</tr>
<tr>
<td>GC Threshold</td>
<td>9%</td>
<td></td>
</tr>
<tr>
<td>Blocks per Chips</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>Pages per Block</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Write Time</td>
<td>350us</td>
<td></td>
</tr>
<tr>
<td>Read Time</td>
<td>35ms</td>
<td></td>
</tr>
<tr>
<td>Erase Time</td>
<td>1.5ms</td>
<td></td>
</tr>
<tr>
<td>Chip Xfer Time</td>
<td>25ms</td>
<td></td>
</tr>
<tr>
<td>Copyback Time</td>
<td>On</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2: Characteristics of I/O workload traces**

<table>
<thead>
<tr>
<th>Workload</th>
<th>Avg. Req. Size (KB)</th>
<th>Request Amount (GB)</th>
<th>Read Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read</td>
<td>Write</td>
<td>Read</td>
</tr>
<tr>
<td>Financial</td>
<td>5.73</td>
<td>7.2</td>
<td>6.76</td>
</tr>
<tr>
<td>Exchange</td>
<td>9.81</td>
<td>12.56</td>
<td>37.43</td>
</tr>
<tr>
<td>MSR</td>
<td>47.18</td>
<td>21.01</td>
<td>1.88</td>
</tr>
<tr>
<td>Web</td>
<td>15.14</td>
<td>8.60</td>
<td>15.24</td>
</tr>
</tbody>
</table>

The Page Replacer evicts the selected victim stripe. Specifically, the dirty blocks among the victim stripe are destaged to the Primary Storage Layer, while the clean blocks are simply considered removed from the SRC layer.

The Mapping Manager maintains a logical to physical address mapping table to quickly lookup the location of the requested data in the storage system. In order to detect data corruption, the Mapping Manager also keeps checksum information of each data block along with the mapping information. In particular, the validity of the data is checked using the checksum value. If valid, that is, the requested data is not corrupted it is served to the upper layer. Otherwise, SRC attempts to recover the original data through the RAID reconstruction mechanism.

The Parity Writer is the component that calculates and stores the parity block to the specific SSD drive. When a full stripe write occurs, the Parity Writer simply calculates and stores the parity block to the specific SSD drive. When a full stripe write occurs, the Parity Writer simply calculates and stores the parity block to the specific SSD drive. When a full stripe write occurs, the Parity Writer simply calculates and stores the parity block to the specific SSD drive. When a full stripe write occurs, the Parity Writer simply calculates and stores the parity block to the specific SSD drive.

---

5. PERFORMANCE EVALUATION

To evaluate SRC, we implement the SRC scheme in a hybrid storage simulator based on the CMU DiskSim [28] and MSR SSD Extension [29]. We also set up typical cache solutions that use RAID-0 and RAID-5 approaches to represent real working environments. In all the schemes, four SSDs are used as the data cache and five HDDs are used as primary storage. Also, unless otherwise mentioned, the stripe size is fixed to 64KB for all the experimental results presented.
The specific parameters of the HDDs and SSDs are given in Table 1.

For the workload, we use realistic workload traces that have been used in various other studies [10, 15, 14, 30]. The first is the Financial trace that is a random write intensive I/O workload obtained from an OLTP application running at a financial institution [26]. The Exchange trace is a random I/O workload obtained from the Microsoft employee e-mail server [27]. This trace is composed of 9 volumes of which we select and use the trace of volume 2. The MSN trace is extracted from 4 RAID-10 volumes on an MSN storage back-end file store [27]. We choose and use the trace of volume 0. The Web trace is a random read intensive I/O workload obtained from a web search engine [26]. The Web trace is unique in that 99% of the requests are reads while only 1% are writes. The characteristics of the four traces are summarized in Table 2.

5.1 SRC vs Conventional RAID

We first compare the performance of SRC with conventional RAID-0 and RAID-5 based schemes, which are meant to represent typical host storage settings in real environments. For the SRC scheme, we use the results of the Separated-Fixed policy. (We discuss the performance of the other policies in the following section.)

Figure 4 shows the response times of SRC and RAID-5 relative to the performance of RAID-0. As expected, RAID-0 shows the best performance compared to other schemes except for the Web workload. For the Web workload, SRC performs even better than RAID-0.

The reason RAID-0 performs so well is that this scheme has no parity overhead as exemplified in Figure 5. This figure shows the amount of I/O performed by the various components for each scheme for the Exchange workload. (Results for the other workloads are not presented for brevity.) As shown in the figure, RAID-0 only reads and writes data, and there is no activity for parity, whereas for RAID-5 considerable data is moved for parity management. In contrast, we see that for SRC, though there is activity for parity, it is consolidated into a single SSD and its effect on overall performance is minimal. Overall, Figure 4 shows that our SRC scheme outperforms RAID-5 considerably; by 55% for the Financial workload, 68% for the Exchange workload, 56% for the MSN workload, and 59% for the Web workload.

Though RAID-0, in general, performs the best, it does not provide data protection against drive failures and data corruption. As a result, we argue that RAID-0 is an inappropriate solution for an SSD cache that uses the write-back policy.

Finally, Figure 6 shows the erase count of SSDs that comprise the cache layer for the Exchange workload. From these, we can see that the expected lifetime of the SSDs using the SRC scheme is double that of RAID-5. Our conclusion is that the SRC scheme brings about considerable performance and lifetime benefits by reducing the parity update overhead.

5.2 Comparison of Allocation Policies

In this section, we discuss the performance of the four allocation policies, namely, Mixed-Fixed (read-write Mixed striping with Fixed parity SSD), Mixed-Rotated (read-write Mixed striping with Rotated parity SSDs), Separated-Fixed (read-write Separated striping with Fixed parity SSD), and Separated-Rotated (read-write Separated striping with Rotated parity SSDs) described in the previous section.
policies. In the figure, the $x$-axis denotes workloads and the $y$-axis denotes the mean response time of I/O requests normalized to that of the Mixed-Fixed policy.

Figure 7 shows that, for workloads with a mix of read and write requests, fixing the parity to a single SSD is a clear winner as Mixed-Fixed and Separated-Fixed show better response times than both their rotated counterparts. Also, with the parity SSD fixed, the results in Figure 7 tell us that the Mixed policy is actually better than the Separated policy. However, for the read intense Web workload, we see that Separated-Fixed and Separated-Rotated perform considerably better than the other policies. This tells us that, for these types of read intense workloads, the data block allocation policy is more influential than the parity placement policy.

Effect of Fixed Parity Policy

In the following, we discuss the reasoning behind the superior performance of fixing the parity to a single SSD. We use the results of only the Financial workload as the observations of the other traces are similar.

Figure 8 shows the response times of the various allocations policies. In the figure, the $x$-axis represents workloads and the $y$-axis denotes the mean response time of I/O requests normalized to that of the Mixed-Fixed policy.

Figure 7 shows that, for workloads with a mix of read and write requests, fixing the parity to a single SSD is a clear winner as Mixed-Fixed and Separated-Fixed show better response times than both their rotated counterparts. Also, with the parity SSD fixed, the results in Figure 7 tell us that the Mixed policy is actually better than the Separated policy. However, for the read intense Web workload, we see that Separated-Fixed and Separated-Rotated perform considerably better than the other policies. This tells us that, for these types of read intense workloads, the data block allocation policy is more influential than the parity placement policy.

Effect of Fixed Parity Policy

In the following, we discuss the reasoning behind the superior performance of fixing the parity to a single SSD. We use the results of only the Financial workload as the observations of the other traces are similar.

Figure 8 shows the response times of the various allocations policies. In the figure, the $x$-axis represents workloads and the $y$-axis denotes the mean response time of I/O requests normalized to that of the Mixed-Fixed policy.

Figure 7 shows that, for workloads with a mix of read and write requests, fixing the parity to a single SSD is a clear winner as Mixed-Fixed and Separated-Fixed show better response times than both their rotated counterparts. Also, with the parity SSD fixed, the results in Figure 7 tell us that the Mixed policy is actually better than the Separated policy. However, for the read intense Web workload, we see that Separated-Fixed and Separated-Rotated perform considerably better than the other policies. This tells us that, for these types of read intense workloads, the data block allocation policy is more influential than the parity placement policy.

Effect of Fixed Parity Policy

In the following, we discuss the reasoning behind the superior performance of fixing the parity to a single SSD. We use the results of only the Financial workload as the observations of the other traces are similar.

Figure 8 shows the response times of the various allocations policies. In the figure, the $x$-axis represents workloads and the $y$-axis denotes the mean response time of I/O requests normalized to that of the Mixed-Fixed policy.

Figure 7 shows that, for workloads with a mix of read and write requests, fixing the parity to a single SSD is a clear winner as Mixed-Fixed and Separated-Fixed show better response times than both their rotated counterparts. Also, with the parity SSD fixed, the results in Figure 7 tell us that the Mixed policy is actually better than the Separated policy. However, for the read intense Web workload, we see that Separated-Fixed and Separated-Rotated perform considerably better than the other policies. This tells us that, for these types of read intense workloads, the data block allocation policy is more influential than the parity placement policy.

Effect of Separated Policy

Let us now turn our attention to the Separated policy. In Figure 7, we showed that the Separated-Fixed and Separated-Rotated policies are approximately 20% better in response time than the other policies for the Web workload, which is 99% reads and 1% writes. To demonstrate the impact of the Separated policy, we measure the parity overhead and erase counts of each SSD for the Web workload and depict their results in Figure 10. Recall from Figure 3 of Section 3.2 that the Separated policy allows SRC to forgo the writing of

Note that cleaning time here is that consumed internal to the SSD, not the cleaning time for the log-structured approach. SRC does not incur cleaning for the log-structured approach as it uses destaging for dirty blocks.
parity blocks for read stripes and, in the place of the parity blocks, data blocks are written.

In Figure 10(a), we observe that the Separated-Fixed policy has no parity overhead, while Mixed-Fixed has roughly 3GB of additional parity writes. Furthermore, as shown in Figure 10(b), the average erase count of Separated-Fixed is reduced to half that of Mixed-Fixed. We conclude that for read dominant workloads, eliminating parity information for read stripes can have a positive effect on performance and lifetime.

**Effect of Stripe Size**

Similar to RAID, in SRC, finding the appropriate stripe size is an important issue that can influence performance. To investigate this issue, we measure the average response times as we vary the stripe size from 16KB to 1MB.

Again, in the following, we present the results for only the Financial and Web workloads using Figure 11. Let us first discuss the Financial workload shown in Figure 11(a). We see that the response time degrades slightly as the stripe size increases until it reaches 128KB. This can be explained by Figure 11(b), which shows that parity overhead increases as the stripe size increases. This is because parity updates occur more frequently as more and more small write requests cannot be fully filled in a single stripe.

We see, however, that the average response time suddenly drops as the stripe size increases beyond 128KB. This can be explained by the improvement seen in the HDD head positioning time depicted in Figure 11(c). The reason behind this is that with large stripes it is more likely to destage in larger sizes resulting in reduced seek time and rotational delay in HDDs.

For the Web workload, we only see a slight rise in the average response time as the stripe size is increased. We see that there is no significant parity overhead involved and that positioning time of HDD is not affected as writes to HDDs are minimal. However, we do see that with larger stripes we see a slight decrease in the hit ratio as cache management occurs in larger grains. This has a negative impact on the performance for the Web workload.

**6. CONCLUSION**

NAND flash memory based SSDs are now being employed as data caches for various storage systems. Since SSDs are still considerably more expensive than HDDs, cost-effectiveness and reliability are important issues when designing SSD cache solutions. To this end, we investigated the management of an SSD cache that employs the RAID technique over the SSDs to enhance reliability.

The scheme that we proposed, which we call SRC (SSD RAID Cache), is based on the log-structured write approach that results in eliminating read-modify-write operations that hinder the performance of conventional RAID-5 systems.

Within this framework, we proposed various techniques to enhance the performance of the SSD RAID cache and the lifetime of SSDs that comprise the SSD RAID. We also presented and analyzed various allocation policies possible for the SRC scheme.

Through experiments, we showed that our proposed scheme outperforms the typical RAID-5 approach and performs comparatively to the RAID-0 approach. We also showed that the lifetime of SSDs may be prolonged considerably as the erase count of SSDs that comprise the SSD cache is reduced. Specifically, our experimental results using various realistic I/O workloads showed that using the SRC scheme is on average 59% better, with a maximum of 68% and minimum of 55%, than the conventional SSD cache scheme supporting RAID-5. In case of lifetime, our results showed that SRC reduces the erase count of the SSD drives by an average of 47%, with a maximum of 87% and minimum of 27%, compared to the RAID-5 scheme.

**Acknowledgments**

This research was supported in part by Seoul Creative Human Development Program funded by Seoul Metropolitan Government(No. HM120006), by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 20121A2A01045733), and by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2010-0025282).
REFERENCES


