Enabling Cost-Effective Flash based Caching with an Array of Commodity SSDs

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

Eunjae Lee
University of Seoul
kckjn97@uos.ac.kr

Jongmoo Choi
Dankook University
choijm@dankook.ac.kr

Donghee Lee
University of Seoul
dhl EXPRESS@uos.ac.kr

Choulseung Hyun
University of Seoul
cshyun@uos.ac.kr

Sam H. Noh
UNIST
http://next.unist.ac.kr

ABSTRACT
SSD based cache solutions are being widely utilized to improve performance in network storage systems. With a goal of providing a cost-effective, high performing SSD cache solution, we propose a new caching solution called SRC (SSD RAID as a Cache) for an array of commodity SSDs. In designing SRC, we borrow both the well-known RAID technique and the log-structured approach and adopt them into the cache layer. In so doing, we explore a wide variety of design choices such as flush issue frequency, write units, forming stripes without parity, and garbage collection through copying rather than destaging that become possible as we make use of RAID and a log-structured approach at the cache level. Using an implementation in Linux under the Device Mapper framework, we quantitatively present and analyze results of the design space options that we considered in our design. Our experiments using realistic workload traces show that SRC performs at least 2 times better in terms of throughput than existing open source solutions. We also consider cost-effectiveness of SRC with a variety of SSD products. In particular, we compare SRC configured with MLC and TLC SATA SSDs and a single high-end NVMe SSD. We find that SRC configured as RAID-5 with low-cost MLC and TLC SATA SSDs generally outperforms SRC configured with a single high-end SSD in terms of both performance and lifetime per dollars spent.

Categories and Subject Descriptors
D.4.2 [Storage Management]: Storage hierarchies; Secondary storage

General Terms
Design, Performance, Experimentation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Middleware’15, December 07-11, 2015, Vancouver, BC, Canada
©2015 ACM. ISBN 978-1-4503-3618-5/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2814576.2814814.

Keywords
Flash Based Caching, SSD Array, Log-Structured Caching Layout, Cost-effective Storage

1. INTRODUCTION
NAND flash memory based Solid State Drives (SSDs) are being used as data cache to improve performance of backend storage such as DAS, NAS and SAN in computer systems [6, 18, 27, 29, 40, 43]. SSD cache solutions are expected to attain SSD-like performance at HDD-like price. For example, the Seagate Momentus XT integrates raw flash memory chips into hard disks for desktop environments to boost boot-up time and application loading [47]. For enterprise configurations, storage vendors are providing customers with high performance solutions based on PCI-e interface SSDs [15].

Besides the performance and price aspects, one important trait of the SSD cache is non-volatility with which data can survive upon power failures or host side crashes. This trait also appears to allow one to employ the write-back policy as a high performance SSD cache solution. However, in reality, if the SSD cache is the only entity with up-to-date data and the SSD fails, then data will be lost. To avoid such risky scenarios, employing RAID techniques to SSD cache has been considered. In practice, EnhanceIO [11] and HGST ServerCache [17] use RAID-1, while LSI ElasticCache [36] and HP SmartCache [19] use RAID-4 or -5.

In this paper, we propose a new caching solution called SRC (SSD RAID as a Cache) for an array of commodity SSDs with the goal of providing a high performing, yet cost-effective solution. In designing such a system, we borrow from both the well-known RAID technique and the log-structured approach [42, 44]. RAID systems have traditionally been deployed for primary storage providing benefits such as high bandwidth and reliability, while the log-structured approach has been known to perform well for flash based storage [28, 34]. We adopt these two well-known technologies into the SSD cache layer by making various design and optimization choices. In so doing, we take into account a technological trend of modern SSDs, in particular, the growth in the erase group size, which can informally be defined as the write unit at which performance is optimized and sustained in modern SSDs [22, 26, 34, 50]. We find that this size is becoming larger, and due to this trend, adoption into an array of SSDs is not obvious.

Using our implementation in the Linux kernel 3.11.7 under...
the Device Mapper (DM) framework, we explore a number of design choices in SRC through a variety of experiments. For example, we consider combining garbage collection and destaging based on the hotness of the data and the current utilization of the system. We also consider distinguishing clean and dirty data when forming a stripe in RAID as clean data in a cache may be lost without compromising reliability. SRC is compared with Bcache5 and Flashcache5, that is, Bcache and Flashcache with experimental settings where the RAID-5 configured underlying SSD cache layer is used. Results show that SRC is at least 2 times better than these solutions in terms of performance. We also compare SRC configured with MLC and TLC SATA SSDs and that configured with a single high-end NVMe SSD (without parity). We find that SRC configured as RAID-5 with low-cost TLC SATA SSDs is generally superior in terms of performance per dollar, while that configured with MLC SATA SSDs is superior in terms of lifetime per dollar spent.

The remainder of this paper is organized as follows. In Section 2, we first review the internal workings of contemporary flash based SSDs. Then, we review existing work related to this study. In Section 3, we specifically look into SSD cache solutions that are available as open source. We look at the limitations of these work and issues that could influence the performance of SSD based caching solutions. In Section 4, based on the findings of the previous section, we present the design of the SRC scheme. In Section 5, we describe the implementation of SRC and present experimental results looking at the various parameter choices that are made in SRC. Comparative evaluations with RAID-5 accommodated versions of Bcache and Flashcache and an evaluation of the cost-effectiveness issue are also given in Section 5. Finally, we conclude the paper with a summary and conclusions in Section 6.

2. BACKGROUND AND MOTIVATION

In this section, we first describe the internal workings of modern flash memory based SSDs. Then, we review some of the work that is related to our study, specifically, SSD based caching and RAID.

2.1 SSD Architecture

Today’s commercial SSDs use NAND flash memory as their storage medium. Flash memory is composed of memory cells where bits are stored. The cells are organize in page units in which basic operations such as read and program are performed. A set of pages then constitutes a block, which is the unit in which the erase basic operation is performed. The number of pages in a block typically ranges in between 32 to 512 pages. The blocks are organized in planes. Then, two or more planes comprise a die, which typically forms a flash package. Some packages have more than one die, and depending on the number of dies per package, these are termed single die package (SDP), double die package (DDP) and quadruple die package (QDP) [24, 33].

For high performance and to support large capacity, state-of-the-art SSDs integrate multiple flash chips. The chips along with SRAM, DRAM, and an internal processor are interconnected through a bus. The SRAM stores data for the processor and the DRAM buffers the flash memory data. Multiple read/write requests stored in DRAM can be served simultaneously along multiple buses and chips when possible. Generally, the size of the DRAM buffer is determined based on the internal parallelism that is to be supported. Firmware called the Flash Translation Layer (FTL) controls the workings among the various components of the SSD. To optimize performance SSD designers go through considerable effort to design the FTL firmware so that it can make full use of all the available hardware resources and features such as the chips, buses, and buffers through interleaving and parallelism [20, 38]. SSDs are connected to hosts through an interface such as SATA or PCI-e like any other disk.

In an effort to increase density, manufacturers have made efforts to shrink the silicon feature size and store more bits per flash memory cell. These efforts have been realized in 1xnm processing, MLC and TLC flash memory, and 3D NAND technologies. However, higher density has resulted in increased negative traits in terms of performance, cell endurance, data retention time, and energy consumption [7, 8, 9, 21, 31]. Furthermore, these negative characteristics are manifested even further with increased program/erase cycles [1, 16]. At the most basic level, flash controllers employ ECC engines to help remedy these issues. The FTL firmware detects the blocks with faulty pages that exceed the number of correctable bits designated by ECC and adopts a bad block management algorithm. Various other advanced techniques are used to alleviate data retention errors and other reliability issues [7, 8, 21, 31, 41].

2.2 Related Work

In this section, we review three aspects that are related to our work. The first is about reducing the effect of garbage collection in SSDs. Several studies have addressed I/O alignment to minimize or eliminate garbage collection (GC) in SSDs. Lee et al. present an LFS-like file system called F2FS for flash storage. This file system sequentializes random writes to minimize internal SSD GC operations [28]. Previously, work by Hyun et al. and Kim et al. have also considered changes to the Linux file system and scheduler to align writes with flash erase blocks [22, 26].

Next, we review studies on SSD based caching. Tang et al. propose RIPQ (Restricted Insertion Priority Queue), a novel SSD caching framework to mimic advanced cache algorithms (e.g. sLRU and GDSF) [50]. Specifically, they first reemphasize the fact that the erase group size has increased to up to 512MB to achieve sustained performance and improve lifetime for cutting edge SSDs. For SSD caching, the size of the replacement unit must be enlarged, leading to hit ratio reduction. Consequently, RIPQ implements priority queues on SSDs and operates on top of priority queues to increase the hit ratio and minimize write amplification of the SSD cache. Also, they gather clean data in the memory buffer and store the data and the metadata together in SSDs for cached data recovery. However, RIPQ does not support the write-back policy as the focus is mainly on read intensive data such as photos.

Saxena et al. suggest a cache architecture, namely FlashTier, where the cache layer is embedded into the FTL of the SSD [45]. To communicate with the operating system, they provide well-defined interfaces to SSDs to exploit their characteristics and to reduce cache block management costs. With such design, FlashTier can incorporate cache replacement with garbage collection in the FTL and also provide a mechanism called silent eviction that allows for simple invalidation of cold data instead of GC. Added benefits such
as improved memory efficiency by offloading the mapping table to the FTL and avoiding re-filling of hot data from slow disks upon crash through logging of metadata of cached data in the OOB (out-of-band) area are provided. Interestingly, the ideas presented through simulations were also realized through OpenSSD based real prototype implementations [46].

Another recent study was presented by Li et al. [30]. Here, Li et al. propose a new cache architecture called Nitro for primary storage systems that combines both compression and de-duplication techniques to further increase caching capacity, leading to increased hit ratio. They report that both deduplication and compression improves hit ratio by up to 19% and 9%, respectively, and hence, deduplication is more effective than compression because primary workloads involve many identical contents. Also, they suggest a new replacement unit, WEU (Write Evict Unit), that aligns with the SSD block size to improve their lifetime. Specifically, Nitro checks whether the incoming file extent is the same as the content that is stored in SSD cache. Then, the extent is compressed and packed into a WEU along with the metadata. When Nitro runs out of space, it makes new WEUs through eviction to primary storage to avoid data migration from/to SSDs. These ideas are validated with commodity SSDs and with simulation based customized SSDs. The results show that performance and lifetime benefits can be achieved by a combination of capacity and flash optimizations.

The work that we present in this paper is similar to these studies in that we also exploit the use of large replacement sizes to fully take advantage of SSDs’ sustained performance. However, our work is unique in that we explore a cost-effective caching solution by making use of an array of commodity SSDs that does not require elaborate new interfaces.

Another line of work on SSD based caching has been on the use of SSD caching in 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 [6]. To overcome consistency limitations of the write-back caching scheme, Koller et al. provide consistent write-back policies [27]. Holland et al. study various configurations and policies for host-side caches [18]. Unlike the study by Koller et al., they insist that the write-through policy is more reasonable than the write-back policy.

Qin et al. propose write-back based caching policies that integrate the general cache flush command to ensure data integrity on primary storage [43]. The workings of the policies are based on the client side SSD failure models. For example, when the flash device is assumed to be unrecoverable, the policy flushes all dirty data from the SSD to primary storage upon an application flush command. Our work is different from the study by Qin et al. in that SRC bundles cached data together with metadata and parity in large segment units and then distributes the segment among all SSDs to protect them from disruptive SSD failures. This allows SRC to avoid flushing the data to primary storage.

The final aspect related to our work is RAID. RAID has been introduced to provide reliable storage [42]. RAID-5, in particular, is a cost-effective and reliable storage system that has been widely deployed. RAID-5, though effective for large sequential writes, suffers from the small write problem. That is, when a small write request arrives, the modified data and its related parity must be updated through either the read-modify-write or reconstruct-write mechanism. These operations incur extra read and write operations leading to significant performance degradation.

To overcome the small write problem, many techniques have been introduced. Stodolsky et al. develop a parity logging scheme that sequentially logs the updated parity into a log region [49]. The up-to-date parity are moved to their original locations when the log region runs out of space. Wilkes et al. propose AutoRAID, a hybrid approach where frequently updated data are stored in RAID-1, while infrequently updated data are stored in RAID-5 so that parity overhead of RAID-5 is minimized [51]. Our study is similar to these studies in that a log-structure is employed to reduce parity overhead. However, our study differs in that our primary use of RAID is as a reliable cache and not as primary storage. By exploiting the fact that RAID is being used as a cache and the characteristics of the SSDs, instead of HDDs, that comprise RAID, we are able to come up with various mechanisms to enhance the effect of the cache.

Finally, this paper is an extension of our previous work [40] where we presented our basic idea with experiments conducted via DiskSim-based simulations. In this paper, we present new observations and enhancements that were made while implementing our scheme with real SSDs. Naturally, we also report new experimental results that were obtained from the Device Mapper based prototype that we implemented in Linux.

3. STUDY OF EXISTING SOLUTIONS

In this section, we discuss in detail two existing open source SSD cache solutions, namely, Bcache and Flashcache. We look at the limitations of these work and issues that could influence the performance of SSD based caching solutions.

3.1 Existing SSD Cache Solutions

In this section, we present observations made from exper-
when the dirty data ratio exceeds cache is lost. While Bcache destages dirty data immediately close to write-back, though more data may be lost if the respectively, by default.) Higher ratios reflect policies more values are set to 10% and 20% for Bcache and Flashcache, respectively, that controls the dirty data ratio, that is, the ratio of dirty data that may reside in the cache. (These

Figure 1: Performance comparison of Bcache and Flashcache on various RAID levels

Filshcache is designed to optimize read performance by using a set-associative policy to map data in SSDs. Specifically, it divides caching space into multiple sets, which in turn consists of 4KB blocks. (Set size ranges from 128KB to 4MB with a default size of 2MB.) Filshcache has separate partitions to keep metadata and cached data separately. To maintain the consistency of modified data in the SSD cache, it writes the metadata of the dirty data to the SSD cache, while, for clean data, it keeps the metadata only in memory. As a result, loading clean data to Flashcache from the underlying primary storage is accomplished with a clean data write to cache and an in-memory update. Flashcache always ignores flush commands from the upper layer and immediately returns acknowledgements in order to improve performance. However, this approach is vulnerable to file system inconsistency.

Bcache partitions caching space into multiple buckets, whose size is 4MB by default. The bucket size can range from 4KB to 16MB. To optimize random writes, Bcache collects small write requests in memory and writes them to a bucket sequentially. Also, to optimize metadata management, Bcache employs a B+tree-like index mechanism. When a write miss occurs in the cache, Bcache first writes dirty data to the cache, and then logs metadata into the journal area with a flush command. Later, metadata in the journal area are copied to their locations in the metadata area. By virtue of this journaling mechanism, we expect Bcache to outperform Flashcache, particularly for random write workloads. Like Flashcache, Bcache keeps metadata only in memory for clean data to minimize management cost. As a result, cached clean data disappears upon power or SSD failure.

To improve performance for random writes on SSDs, both Bcache and Flashcache employ a write-back-like policy, but with a parameter, termed writeback_percent and dirty_thresh_pct, respectively, that controls the dirty data ratio, that is, the ratio of dirty data that may reside in the cache. (These values are set to 10% and 20% for Bcache and Flashcache, respectively, by default.) Higher ratios reflect policies more close to write-back, though more data may be lost if the cache is lost. While Bcache destages dirty data immediately when the dirty data ratio exceeds writeback_percent, Flash-

cache is more tolerant to missing the threshold. Hence, for busy systems using Flashcache, the dirty data ratio may actually exceed dirty_thresh_pct. However, write-through mode is still recommended by default due to the possibility of dirty data loss upon SSD failure.

To compare the write-back and write-through policies, we perform a set of experiments under the setting presented in Table 1. This setting will be used throughout the experiments undertaken for this paper with one SSD or four SSDs being employed depending on the experiments of interest. For these experiments, we make use of the FIO (Flexible I/O) benchmark that measures 4KB I/O performance with Uniform Random distribution [12]. The Uniform Random distribution generates random write requests that evenly access all blocks. Settings for request size, iodepth, and the number of threads is 4KB, 32, and 4, respectively, and the workload is generated for 10 minutes.

Table 2 shows the bandwidth results, where WT and WB command is issued after every 512KB, while for the random workload a flush command is issued after every 32 4KB requests. We see from the results in Table 3 that with the flush command, performance deteriorates considerably.

In summary, the results here tell us 1) that the write-back policy is beneficial in terms of performance and 2) that the flush command used with write-through is the key source of
performance bottleneck in caching systems using commodity SSDs. Hence, the flash command should be used sparingly and judiciously.

### 3.2 Exploiting RAID for SSD Caching

As just seen, using the write-back policy is beneficial. However, previous studies have shown that SSDs are unreliable as flash memory wears out [16]. Data retention issues, read disturbs, and other SSD vulnerabilities may also be reason enough to just dismiss the write-back policy for SSD based caches [7, 8, 21, 31].

To resolve this issue, Flashcache suggests a mirroring volume (e.g., RAID-1) containing two SSDs [14]. Bcache also claims that it will provide enhanced data reliability through mirroring of dirty data and metadata in the future [5]. In this paper, we first take these approaches and also explore other combinations of RAID levels (e.g., RAID-4 and -5). With RAID, faults can be tolerated with redundancy and maintainability improves through on-the-fly replacement of failed (or about-to-fail) drives. RAID also provides performance benefits by utilizing disk parallelism. Hence, RAID is a natural option to be explored.

To see the effect of RAID on existing SSD cache solutions, we set the experiment setting so that Bcache and Flashcache individually makes use of the RAID configured underlying SSD cache layer. The base experiment setup is the same as the single SSD cache case (Table 1), but this time making use of four 128GB SSDs connected to the host system through SATA 3.0 ports. All configurations use the write-back policy for caching. Specifically, RAID-0 stripes data among four SSDs without data redundancy such that the cache capacity is 4 x 128GB. We regard this option as the superior configuration even though data is not protected from failures. RAID-1 mirrors data and thus, cache capacity is 2 x 128GB, which is half the total capacity. For RAID-4 and -5 configurations, three quarters of the SSDs are for data while one quarter is for parity and thus, cache capacity is 3 x 128GB. In RAID-4, one SSD is dedicated for storing parity while parities are evenly distributed in RAID-5. For the workload, we use the FIO benchmarks.

Figure 1 shows the bandwidth (MB/s) results. It shows that the RAID-0 configuration performs the best compared to other configurations due to no redundancy. RAID-1 configuration shows roughly halved performance because half the SSDs are utilized as mirrored space. For RAID-0 and RAID-1 configurations, Bcache is worse than Flashcache due to the flash command. With parity-based configurations (e.g., RAID-4 and -5), Flashcache shows degraded performance due to parity updates. In contrast, Bcache shows better performance compared to Flashcache as it employs the log-structured approach.

In summary, the results here show that RAID can enhance data reliability, but hurt performance. Also, the log-structured approach provides a nice opportunity to eliminate read-modify-write operations. Finally, we once again find that the flash command is harmful to performance.

### 3.3 Erase Group Size and Cost

The effect of garbage collection (GC) on SSD performance is considerable. Hence, studies have been made to minimize the cost of GC. Previous studies have addressed this issue by generating write requests to an SSD that are sufficiently large such that flash blocks are allocated together and erased together [22, 26, 34, 50]. The particular size choice is referred to as the erase group size. Recent studies have shown that the erase group size is becoming large.

We also perform a set of experiments to extract the erase group size from the commodity SSD used in our prototype. This is done for OPS (Over-Provisioned Space) sizes ranging from 0% to 50% for each experiment set [39]. Note that increasing the OPS size has the effect of reducing the usable storage space of the SSD.

Figure 2 shows the results. The y-axis in this figure is the throughput (MB/s), while the x-axis is the write request size. As the size of each write request reaches 256MB, the bandwidth of the SSD reaches roughly 400MB/s independent of the OPS size. This tells us that the erase group size is 256MB, which we use throughout our experiments later.

Finally, we discuss the issue of cost. One of our goals of this project is to provide a cost-effective caching solution. For this, we gather the price and performance values of commodity SSDs of a product line from two particular manufacturers that we summarize in Table 4. The performance numbers are from specifications obtained from the manufacturers’ website, while the price is that of the cheapest available on the web that we could find. Here, we find that the price of each SSD is proportional to its capacity but the performance is nearly the same within the same product line. We also find that the key deciding factor for price is the host interface. The cost of a PCI-e product is substantially higher than that of the SATA product. This is because the host interface is the performance limiting factor in modern SSDs.

Flash cache solutions are already available as end products. However, in reality, their price is expensive because they use proprietary techniques. Our goal is to provide a cost-effective cache solution balancing performance and price by making use of off-the-shelf commodity SSDs. The performance/cost findings presented above tell us that aggregating multiple low-cost, low-capacity SSDs may be better than purchasing a single high-end, high-cost product so long as the aggregate performance of the former can match

---

Table 5: Comparison of SSD Caching Solutions

<table>
<thead>
<tr>
<th>Name</th>
<th>SSD Type</th>
<th>Erase Group Alignment</th>
<th>Write-Back Support</th>
<th>flash Optimization</th>
<th>Clean Data Persistency</th>
<th>RAID Support</th>
<th>Open Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlashTier [45]</td>
<td>Custom</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Nitro [30]</td>
<td>Custom</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Reliable Write-Back [43]</td>
<td>Commodity</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>RIPQ [50]</td>
<td>Commodity</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>FlashCache [13]</td>
<td>Commodity</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Bcache [4]</td>
<td>Commodity</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SCC</td>
<td>Commodity</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
that of the latter [42]. This also provides added support for the use of the RAID approach that we undertake.

Finally, we summarize the various flash solutions that are found in the literature and their properties in comparison to our SRC scheme in Table 5.

4. SSD RAID AS A CACHE

In this section, we present our host-side SSD cache solution, which we call SRC (SSD RAID as a Cache). Our design goals are to provide an SSD cache solution using an array of commodity SSDs that 1) is cost-effective, 2) employs the write-back policy for high performance, and 3) is optimized for contemporary SSDs with a large erase group size.

Specifically, for cost-effectiveness, SRC comprises an array of low-cost commodity SSDs that use the general block interface such as SATA as the caching device. Also for cost-effectiveness, SRC maintains a DRAM-only buffer to pack clean as well as dirty data for segment writes, even though many storage vendors employ an NVRAM-based buffer atop SSDs to keep dirty data safe. The issue regarding safety of dirty data is discussed below.

For high performance, we employ the write-back policy. When using write-back for cache, one must consider the tradeoff between durability and performance. The flush command could be used to enhance durability, but as discussed previously, this will result in performance degradation. Unlike Flashcache, which ignores the flush command and Bcache, which issues the command for every metadata write, we take an approach similar to the one taken by Qin et al. [43].

As previously discussed, determining the erase group size properly has considerable effect on the performance and lifetime of SSDs. In SRC, to align the write size with the large erase group size of modern SSDs, we borrow the log-structured scheme that stores data blocks along with metadata and parity blocks all at once in SSDs when constructing stripes [32, 35, 51]. This has a positive effect of not only removing all read-modify-write operations that are detrimental to RAID performance, but also enhancing the lifetime of SSDs [2, 38]. To avoid inconsistency with data blocks, we calculate the checksum of the data and store it in metadata blocks. The metadata block is itself also checksummed. We describe this in more detail in the next section.

Figure 3 shows the architecture and data layout of SRC. SRC provides a caching solution on top of primary storage connected to a network interface (e.g., iSCSI protocol). In the following subsections, we discuss in detail the workings of SRC.

4.1 Segment Group: A Log Based Approach

Segment Group Layout: Cache space in SRC is determined by the number of SSDs $M$ that comprise the array of SSDs and the size $S$ of each SSD as shown in Figure 3(a). (From here on, our discussion will be made in the context of our implementation, that is, $M = 4$ and $S = 128$GB.) In SRC, this total space is divided into $N$ Segment Groups (SG) as shown in Figure 3(b)(1). The SG size is determined by the erase group size of the underlying SSDs, which is 256MB in our case. Then, SG is 1GB as we have 4 SSDs. Each SG is also divided into smaller units called a segment, across the SSDs. A segment is composed of 512KB units from all 4 SSDs. (The 512KB is an implementation choice made as it is the largest unit in which data can be transferred to the storage device.) Thus, a segment size is 512KB×4, that is, 2MB for our case.

Writes in SRC are made to SGs. For maximum performance that makes full use of the erase group size, the write unit should be 1GB. However, as this may be too large in a realistic setting, SRC writes in segment units, that is, in 2MB units. Metadata and parity information is included in a segment write as shown in Figures 3(b)(2) and (3). We refer to the SG where writes are made to be an active SG, and there is only one active SG at any time.

In our implementation, the very first SG is used to hold the superblock that records important information such as the magic number, create time, device size, and so on. The superblock is only 4KB in size and never modified, hence, we fill the rest of the space with dummy data and set this SG to be read-only. When SG runs out, SRC generates empty SGs by performing GC as in LFS [44]. We will discuss how SRC performs this operation in more detail later in Section 4.2.

Segment Buffer: SRC keeps an in-memory structure called the segment buffer, whose size is the same as the segment size. SRC maintains two separate segment buffers, one
for clean data and one for dirty data. (Note Figure 3(a).) Upon write requests, SRC collects data in the segment buffer for dirty data. When this segment buffer fills up, the entire buffer is written to an unused segment in the active SG. This write may be done with or without a flush command. We discuss this matter in detail later. Then, write requests from the upper layer are acknowledged when the Host-side Cache returns its acknowledgement to SRC.

In case of read misses, SRC first fetches the requested data from primary storage to the temporary staging buffer, at which time an acknowledgement is sent to the upper layer. The data in the staging buffer is later moved to the segment buffer for clean data, and later, when this segment buffer is full, it performs a full segment write to an unused segment.

**Partial Segment:** Timeouts are used to ensure durability of dirty data. Timeouts occur when a write does not happen for $T_{\text{WAIT}}$ time, which is set to 20 microseconds in this study, since the last write incurring a segment write. However, the segment buffer for dirty data may not be full upon a timeout. This is similar to the situation that arises in LFS [44]. Note that the issue of partial writes do not occur for the clean data segment buffer as clean data brought into the buffer have copies in primary storage and can be lost without consequences.

**Flush Command Control:** Previously, we explained that SRC durability is enforced by making use of the flush command issued by the upper layer. To further ensure data and metadata consistency, SRC provides means of issuing flush commands as two particular points; specifically, at every segment write point or at every SG write point, that is, when the current active SG is now full and is about to move on to a new active SG. We discuss the implication of these choices with experimental results later in Section 5.

**Metadata management:** In each segment, SRC stores the metadata along with the data and parity as depicted in Figures 3(b)(2) and (3). The metadata is an extension of the LFS summary structure [28, 44]. Specifically, to verify the validity of the summary information, SRC maintains the checksum, signature, and version information of the metadata blocks. Also, for every data block, the LBA and the checksum information is kept in the metadata blocks.

To enable fast recovery and drive scaling, we distribute the metadata along with the data and parity as depicted in Figures 3(b)(2) and (3). The metadata is an extension of the LFS summary structure [28, 44]. Specifically, to verify the validity of the summary information, SRC maintains the checksum, signature, and version information of the metadata blocks. Also, for every data block, the LBA and the checksum information is kept in the metadata blocks.

To enable fast recovery and drive scaling, we distribute the metadata across all SSDs. Specifically, SRC maintains dedicated metadata blocks to indicate the corresponding data blocks stored in each SSD. Also, metadata blocks are stored at both the beginning and the end of a segment. We denote the former as $M_S$ and the latter as $M_E$.

SRC also maintains an in-memory mapping table to quickly translate the logical addresses to physical addresses. The memory overhead for this, in our implementation, is a 16 byte entry per 4KB, which is roughly 0.3% of the storage capacity. Specifically, for a 512GB SSD, the requirement would be roughly 1.5GB. We believe this is acceptable for today’s servers; the system used for our experiments has 32GB RAM.

**Failure Handling:** Let us now describe the recovery process of the on-SSD metadata blocks. Upon a power failure, SRC scans the metadata blocks to restore consistency. If the generation numbers of $M_S$ and $M_E$ match, SRC considers the whole segment as being consistent. Otherwise, it searches the partial metadata blocks or discards this segment and returns it as a free segment.

Besides drive failure handling, SRC also takes into account silent data corruption [3]. To ensure data integrity, SRC compares the original and calculated checksums when reading data from SSDs. If a checksum mismatch is found, SRC recovers the data through parity calculations or by re-fetching them from primary storage depending on the state of the data.

### 4.2 Free Space Reclamation

In a typical cache, when space runs out, a reclaiming process is invoked, generally referred to as garbage collection (GC). Generally, this involves evicting (or destaging) selected victim blocks from the cache to primary storage. This process is called S2D (SSD to Disk) GC in this paper. Another way to GC for systems that take the log-structured approach for storage management is to migrate valid data as is done in LFS [44]. In particular, SRC, which is also log-structure based, can reclaim Segment Groups (SGs) by migrating data. As this movement of data would be among SSDs, we will call such means of GC as S2S (SSD to SSD) GC [44].

At first glance, since we are employing these policies at the cache level, S2D GC seems more effective than S2S GC because S2S GC needs to copy the valid data in the to-be-cleaned SGs to an empty SG, while S2D GC simply removes clean data or writes dirty data to primary storage, which must eventually happen anyway. Also, S2D may possibly improve SSD performance as the utilization of the SSDs is lowered. However, S2D does not take into account the hotness of the data in cache. If the data to be removed or destaged are hot, keeping them in cache in spite of the GC cost may be beneficial. This is the main benefit that could be brought in by S2S GC.

Specifically, SRC provides two GC policies. One is the pure S2D GC. The other is a variant of S2S GC that we refer to as Sol-GC. Sol-GC applies S2S or S2D depending on the cache utilization and the hotness of the victim blocks. Sol-GC monitors cache utilization and, if utilization exceeds $U_{\text{MAX}}$, it triggers S2D GC to reclaim space in the cache; S2D GC is the only means used when utilization is lower than the threshold. However, if cache utilization is below $U_{\text{MAX}}$, it employs S2S GC in a selective manner. Specifically, for dirty data and hot clean data, SRC performs S2S GC. However, clean cold data is simply discarded as moving it may be for naught. Hotness of data is determined by a per-page based bitmap stored in RAM.

SRC also provides two victim SG selection policies, namely, FIFO and Greedy. Specifically, the FIFO policy simply selects the victim SG in the order in which SGs are used. This is a simple policy that does well on SSDs as it always makes sequential writes. The Greedy policy, on the other hand, chooses the least utilized SG among the SGs to minimize GC cost. We will compare the performance of these policies in Section 5.

### 4.3 Clean Data Redundancy

Though parity is an essential part of RAID, another option becomes possible when RAID is used as a cache. Specifically, as clean data in cache is a copy of data in primary storage, redundancy for clean data within the cache is not required. Therefore, another design choice that can be made is to omit the parity for clean data in cache. With this choice, without compromising robustness, comes two advantages; one is reduced parity write overhead and the other is
5. SRC IMPLEMENTATION AND RESULTS

In this section, we first present the experimental platform that we use to implement and experiment with SRC. Then, we present experimental results regarding the design space issues that we discussed in the previous section. We also compare SRC with a RAID-5 accommodated version of Bcache and Flashcache and consider the cost-effectiveness issue of SRC.

5.1 Experiment Platform

To implement SRC, we modify Akiras’s DM-Writeboost [10] as a block-level caching solution performing under the Device Mapper (DM) framework distributed with the Linux kernel 3.11.7. Specifically, DM-Writeboost is designed for write caching using a single device and hence, thousands of lines of code have been modified to implement our SRC scheme. We plan to stabilize SRC and make it available as open source.

In our current SRC implementation, the Segment Group size is set to 1GB as we have 4 SSDs each with a 256MB erase group size in a RAID configuration. The segment size is set to 2MB such that a Segment Group is divided into 512 segments. For fair and consistent measurement, we initialize the SSDs with the TRIM command before we conduct the experiments. To quickly reach steady state where GC is triggered in the SSDs, we sequentially fill the SSDs with dummy data until only the OPS size remains [48]. For our experiments that make use of 128GB SSDs, our OPS size is set to 2GB such that 126GB are sequentially filled with dummy data. Of these 126GBs, we utilize only 18GB as our cache space to expedite our experiments.

To evaluate long-term performance, we make use of several block-level traces from the Microsoft Production Server (MPS) and Microsoft Cambridge Server (MCS) [25, 37] that are summarized in Table 6. The Exchange trace from MPS is a random I/O workload obtained from the Microsoft employee e-mail server. This trace is composed of 9 volumes of which we use the traces of volumes 5 and 9 denoted exch5 and exch9, respectively. The MSN trace, also from MPS, is extracted from 4 RAID-10 volumes on an MSN storage back-end file store. We use the traces of volumes 0, 1, 4, and 5 denoted msn0, msn1, msn4, and msn5, respectively. All other traces other than these are from MCS. The MCS traces are those collected from 36 volumes in 16 servers of a small scale data center. Each volume is based on RAID-1 for boot volume or RAID-5 for data volume and are a collection of a one week period. We chose some of the traces among these to be included in our experiments. Specifically, the concatenated name and number such as prxy0 refer to the server (proxy server) the trace was collected from and the volume or RAID-5 for data volume and are a collection of a one week period. We chose some of the traces among these to be included in our experiments. Specifically, the concatenated name and number such as prxy0 refer to the server (proxy server) the trace was collected from and the volume number (0 for this example) of the trace.

As shown in Table 6, the traces are organized into three groups, namely, Write, Mixed, and Read, groups of which the names represent their main characteristics. The specific traces from MPS and MCS were chosen so that the working set of each of the trace groups would be similar in size; specifically, roughly 50GB. Finally, since the workloads are traces and our SRC is a real working system in Linux, we specifically, roughly 50GB. Finally, since the workloads are traces and our SRC is a real working system in Linux, we conduct the experiments. To quickly reach steady state where GC is triggered in the SSDs, we sequentially fill the SSDs with dummy data until only the OPS size remains [48]. For our experiments that make use of 128GB SSDs, our OPS size is set to 2GB such that 126GB are sequentially filled with dummy data. Of these 126GBs, we utilize only 18GB as our cache space to expedite our experiments.

In our current SRC implementation, the Segment Group size is set to 1GB as we have 4 SSDs each with a 256MB erase group size in a RAID configuration. The segment size is set to 2MB such that a Segment Group is divided into 512 segments. For fair and consistent measurement, we initialize the SSDs with the TRIM command before we conduct the experiments. To quickly reach steady state where GC is triggered in the SSDs, we sequentially fill the SSDs with dummy data until only the OPS size remains [48]. For our experiments that make use of 128GB SSDs, our OPS size is set to 2GB such that 126GB are sequentially filled with dummy data. Of these 126GBs, we utilize only 18GB as our cache space to expedite our experiments.

Figure 4: Impact of Erase Group Size, where $U_{MAX}$ is 90%.

Table 6: Characteristics of Trace Sets

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erase Group Size (MB)</td>
<td>2, ..., 64, 128, 256, 512</td>
</tr>
<tr>
<td>Free Space Management</td>
<td>S2D, Sel-GC</td>
</tr>
<tr>
<td>$U_{MAX}$ for Sel-GC</td>
<td>10% to 95%</td>
</tr>
<tr>
<td>Victim Selection Policy</td>
<td>FIFO, Greedy</td>
</tr>
<tr>
<td>Clean Data Redundancy</td>
<td>PC, NPC</td>
</tr>
<tr>
<td>RAID Level</td>
<td>0, 4, 5</td>
</tr>
<tr>
<td>Flush Command Control</td>
<td>Per Segment, Per SG</td>
</tr>
</tbody>
</table>

write caching using a single device and hence, thousands of lines of code have been modified to implement our SRC scheme. We plan to stabilize SRC and make it available as open source.

In our current SRC implementation, the Segment Group size is set to 1GB as we have 4 SSDs each with a 256MB erase group size in a RAID configuration. The segment size is set to 2MB such that a Segment Group is divided into 512 segments. For fair and consistent measurement, we initialize the SSDs with the TRIM command before we conduct the experiments. To quickly reach steady state where GC is triggered in the SSDs, we sequentially fill the SSDs with dummy data until only the OPS size remains [48]. For our experiments that make use of 128GB SSDs, our OPS size is set to 2GB such that 126GB are sequentially filled with dummy data. Of these 126GBs, we utilize only 18GB as our cache space to expedite our experiments.

Table 6: Characteristics of Trace Sets

<table>
<thead>
<tr>
<th>Set</th>
<th>Trace</th>
<th>Req Size (KB)</th>
<th>Size (GB)</th>
<th>Read Ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>prxy0</td>
<td>7.07</td>
<td>84.44</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>exch9</td>
<td>21.06</td>
<td>110.46</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>mds0</td>
<td>9.59</td>
<td>11.08</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>stg0</td>
<td>11.95</td>
<td>23.16</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>msn0</td>
<td>21.75</td>
<td>31.28</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>msn1</td>
<td>17.84</td>
<td>30.80</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>src12</td>
<td>29.25</td>
<td>53.23</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>src20</td>
<td>7.39</td>
<td>11.28</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>src22</td>
<td>56.33</td>
<td>62.12</td>
<td>36</td>
</tr>
<tr>
<td>Mixed</td>
<td>rsscb0</td>
<td>9.07</td>
<td>12.41</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>exch5</td>
<td>18.02</td>
<td>85.62</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>bs0</td>
<td>8.88</td>
<td>53.54</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>fim0</td>
<td>6.80</td>
<td>34.91</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>web0</td>
<td>10.20</td>
<td>25.60</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>prn0</td>
<td>12.53</td>
<td>66.79</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>msn6</td>
<td>21.73</td>
<td>31.28</td>
<td>6</td>
</tr>
<tr>
<td>Read</td>
<td>tso</td>
<td>9.28</td>
<td>15.95</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>uro</td>
<td>22.81</td>
<td>48.694</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>prxy0</td>
<td>9.75</td>
<td>20.87</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>src12</td>
<td>59.31</td>
<td>37.40</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>msn5</td>
<td>10.94</td>
<td>124</td>
<td>75</td>
</tr>
</tbody>
</table>

Increased cache capacity.

In SRC, the administrator can select one of two modes, namely Parity for Clean (PC) and No-Parity for Clean (NPC). (See Figure 3(b)(3).) In PC mode, SRC writes parities for clean data, while for NPC mode no parities are kept. Hence, in NPC mode, clean data will disappear when an SSD fails, resulting in degraded read performance until hot clean data is reloaded to the SSD cache from primary storage. In contrast, in PC mode, caching service is not disrupted by SSD failure as clean data can be recovered on the fly with the parity information. We will compare and discuss the quantitative performance implications of these modes in Section 5.
Based on the workload traces to run the experiments. Each group of traces are run separately, but with all traces within each group running simultaneously and each trace being replayed by four threads. Unless otherwise stated, all results reported for the experiments are those accumulated for 10 minutes of execution. Hence, note that each experiment system will be at a different stage of execution when the experiments are terminated and the numbers reported.

The key evaluation metrics that we observe are the throughput (MB/s), which is used to represent the overall performance of the caching system, and I/O amplification, which is used to evaluate how much I/O requests are amplified at the cache layer. This is calculated by dividing the observed I/Os at the cache layer by the actual I/Os requested. We also consider the cost-effectiveness of the various solutions.

### 5.2 Exploration of Design Space

We now consider the effect of the design space parameters for SRC that is summarized in Table 7.

#### Erase Group Size

To evaluate the impact of the erase group size on performance, we vary the size from 2MB to 1024MB while other parameters are set to their default values. Figure 4 shows that performance improves as the erase group size increases. Among the large sizes, performance differs slightly depending on the type of workload. Overall, the 256MB size that we chose based on the observation made in Section 2.2 for SRC looks appropriate. In contrast, I/O amplification is minimized when the size is 2MB as a small size unit is more likely to be fully utilized.

#### Free Space Management

Let us turn our attention to free space management. There are two policy choices that need to be made with free space management. That is, the policy for GC and the policy for choosing the victim for GC. For GC, we compare the S2D and Sel-GC policies, whose results are obtained with $U_{MAX}$ set to 90% and shown in Table 8.

![Figure 5: Impact of $U_{MAX}$ on Sel-GC, where the Erase Group Size is 256MB.](image)

<table>
<thead>
<tr>
<th>GC Scheme</th>
<th>S2D</th>
<th>SEL-GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Victim Policy</td>
<td>FIFO</td>
<td>Greedy</td>
</tr>
<tr>
<td>Write</td>
<td>301(1.26)</td>
<td>312(1.28)</td>
</tr>
<tr>
<td>Mixed</td>
<td>491(1.34)</td>
<td>466(1.34)</td>
</tr>
<tr>
<td>Read</td>
<td>480(1.15)</td>
<td>596(1.14)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trace</th>
<th>Parity for Clean (PC mode)</th>
<th>No-Parity for Clean (NPC mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>431.13 (1.80)</td>
<td>507.89 (1.65)</td>
</tr>
<tr>
<td>Mixed</td>
<td>520.95 (1.56)</td>
<td>547.36 (1.53)</td>
</tr>
<tr>
<td>Read</td>
<td>669.67 (1.26)</td>
<td>725.95 (1.21)</td>
</tr>
</tbody>
</table>

Table 8: Free Space Management Performance (MB/s) (Numbers in parentheses refers to I/O amplification)

Table 9: PC and NPC Mode Performance (MB/s) (Numbers in parentheses refers to I/O amplification)

Table 10: RAID Level Performance (MB/s) (Numbers in parentheses refers to I/O amplification)

Table 8. The results show that Sel-GC considerably outperforms S2D for both the FIFO and Greedy victim selection policies. This shows that conserving hot data via S2S copying during GC has a substantial positive effect on performance. In contrast, S2D shows smaller I/O amplification as it is making less copies among the SSDs.

As for victim selection, Table 8 shows that the FIFO and Greedy policies give-and-take based on the workload and GC scheme used. Overall, the FIFO policy slightly outperforms the Greedy policy for the Write and Mixed workloads. The reason for this is that when the workload is write intensive the FIFO policy has a natural effect of pushing cold blocks to front of the FIFO queue as hot rewritten blocks will be added to the back end because of the log-structured nature of SRC. In contrast, we see that the Greedy policy shows better performance than the FIFO policy by up to 1.24 times for the Read workload because the Greedy policy identifies the least utilized SGs and evicts them through GC.

$U_{MAX}$ Threshold: We now consider the effect of $U_{MAX}$ on Sel-GC. Recall that Sel-GC makes use of S2S when utilization is below $U_{MAX}$ and S2D when above. Figure 5 shows that performance of SRC increases with the increase of $U_{MAX}$ peaking at 90%, then dropping off as it is increased even more. This shows that efforts to keep hot data in the SSDs is worth doing, but being too aggressive can be harmful. However, again, see that I/O amplification also increases with $U_{MAX}$.

Clean Data Redundancy: We now consider the effect of not keeping parity for clean data, that is, the use of NPC mode. Table 9 presents measurement results that show NPC mode outperforming PC mode for the three workloads. In particular, NPC mode performs much better than the PC mode for the Write workload improving performance by around 18%. The reason NPC mode is more effective with write intensive workloads is that the small space that is saved has an amplifying effect as data is continuously written compared to read intensive workloads where data once written sits still.

RAID Protection Level: We now consider the different RAID protection levels, specifically, RAID-0, -4, and -5. Table 10 shows the effects of the RAID levels for the three workload types with other parameters set to default. As expected, RAID-0, which has no redundancy, shows the best performance as it fully utilizes all SSDs. We see that...
Bcache and Flashcache are originally single SSD solutions. Bcache and Flashcache deployed as RAID-5. (We do not maintain parity for protection against SSD failure, as we make use of a single high-cost, high-end NVMe SSD, representing the rightmost bar in Figure 6, with the RAID-5 configurations. Note also that a single NVMe SSD based SRC is always better than the RAID-5 configurations (Figure 6(a)). However, except for the Mixed workload making use of company A products, TLC SSD based configurations are better in terms of performance per dollar as shown in Figure 6(c), which shows the throughput (in MB/s) attained per dollar used.

Similarly, Figures 6(b) and (d) show the lifetime effect of the SSDs. The lifetime numbers shown in Figure 6(b) represent the expected days to live and are calculated using the lifetime estimation equation [23] based on endurance values in Table 12 and assuming that 512GB of write data issued by the workloads are processed per day. The numbers given in Figure 6(d) are obtained from Figure 6(b) by taking into account the cost of the SSD. For example, for A-MLC(SATA) with Write workload, value 2140 from Figure 6(b) is divided by $418, the cost, resulting in 5.12. For lifetime, we observe that SRC making use of MLC SSDs is always better than TLC based SRC.

We now compare the performance and endurance of SRC configured with a single high-cost, high-end NVMe SSD, represented by the rightmost bar in Figure 6, with the RAID-5 configurations. As we make use of a single NVMe SSD and do not maintain parity for protection against SSD failure, the overall performance of using the NVMe SSD is slightly better than the RAID-5 configurations (Figure 6(a)). However, we see that the RAID-5 configurations are generally comparable, giving and taking based on the workload and type of SSD, in terms of cost-effectiveness as exemplified in Figure 6(c). In terms of lifetime, Figures 6(b) and (d) show that the RAID-5 configurations are superior compared to the NVMe SSD configuration. Note also that a single NVMe SSD, though superior in performance, is a fail-stop solution, making it vulnerable to failures, whereas on-the-fly replacement of failed SSDs is possible in RAID-5 configurations.

### 5.4 SRC vs Existing Solutions

In this section, we compare the performance of SRC with Bcache and Flashcache deployed as RAID-5. (We do not consider RAID-4 as the results are similar.) Recall that Bcache and Flashcache are originally single SSD solutions.
As with the settings in Section 3, the experimental setting is set so that Bcache and Flashcache individually make use of the RAID-5 configured underlying SSD cache layer. We will denote Bcache and Flashcache with such settings as Bcache5 and Flashcache5, respectively. We also evaluate two versions of SRC; one with default settings, denoted SRC, and the other also with default settings except that the GC policy is S2D instead of Sel-GC. For Bcache5 and Flashcache5, we set the RAID chunk size to 4KB, which is optimal for 4K random write workloads. Also, the cache set size in Flashcache5 and the bucket size in Bcache5 are equally set to 2MB, while writeback_percent in Bcache5 and dirty_threshold_pct in Flashcache5 are both set to 90%.

Figure 7(a) compares the performance of the four schemes. We see that SRC shows much better performance than the other schemes for all the workload groups. Specifically, SRC outperforms Bcache5 by 2.83, 2.92, and 3.09 times for the Write, Mixed, and Read groups, respectively, and Flashcache5 by 2.50, 2.75, and 2.34 times for the Write, Mixed, and Read groups, respectively. Though direct comparison with Bcache and Flashcache may not be adequate, the results here show that the features introduced in SRC results in substantial performance gains.

Of the four schemes, we find that Bcache5 performs the worst. The major reason behind the low performance of Bcache5 is its frequent issue of the flush command. Note that in contrast to Figure 1, we observe that Flashcache is doing better than Bcache. The main reason for this is that unlike the experiments of Figure 1 where all requests are of 4KB size, with real workload traces I/O requests are much larger. Hence, the negative effect of small writes is reduced considerably.

Comparing SRC and SRC-S2D, we see that SRC does better. However, SRC incurs higher I/O amplification as shown in Figure 7(b). This is because Sel-GC copies hot data among the SSDs during GC. As a consequence, we see that the hit ratio for Sel-GC is higher than S2D as shown in Figure 7(c).

6. CONCLUSION

RAID systems have traditionally been deployed for primary storage providing benefits such as high bandwidth and reliability, while NAND flash memory based SSDs are now being employed as data caches for various storage systems. In this paper, we looked into the workings of existing caching solutions and observed the technical trends of modern SSDs, in particular, the growth in the erase group size. Based on these observations, we designed and implemented SRC (SSD RAID as a Cache), an SSD-based RAID system to be used as a write-back cache for primary storage, that aligns data in erase group size boundaries.

Using our implementation in the Linux kernel 3.11.7 under the Device Mapper (DM) framework, we explored a number of design choices in SRC through a variety of experiments. We found that combining garbage collection and destaging based on the hotness of the data and the current utilization of the system resulted in enhanced performance. Also, distinguishing clean and dirty data when forming a stripe in RAID resulted in better management of the cache without compromising reliability. SRC was compared with Bcache5 and Flashcache5, that is, Bcache and Flashcache with experimental settings where the RAID-5 configured underlying SSD cache layer is used. Results showed that SRC is substantially better than these solutions in terms of performance.

As future work, we are concentrating on adding features to SRC to ease management of SSDs for fast recovery and scalability, which is another goal of SRC. Our initial results are promising, and we expect to provide a stable means to expand or contract the number of SSDs in RAID-5 in a smooth and seamless manner while providing sustained performance. Other more efficient policies are also being considered. For example, with Sel-GC, currently we do not distinguish hot clean data and dirty data when using S2S. Separating these two types of data is being considered. Other victim SG selection policies are being considered as well. We also plan to implement advanced flash-based caching schemes (e.g., [30], [43], [45], and [50]) and compare them with SRC. Quantitative measurements of SRC based on state-of-the-art flash devices such as NVMe SSDs are also being planned. Finally, as mentioned previously, we plan to stabilize SRC and soon make it available as open source.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their detailed comments that helped to improve the presentation of the paper. We also thank Heejin Park for providing useful comments. This research was supported in part by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 2012R1A2A2A01045733) and by Seoul Creative Human Development Program (CAC15106). Sam H. Noh is the corresponding author of this paper.
8. REFERENCES


