ReCA: an Efficient Reconfigurable Cache Architecture for Storage Systems with Online Workload Characterization

Reza Salkhordeh, Shahriar Ebrahimi, and Hossein Asadi, *Senior Member, IEEE*

Abstract—In recent years, *Solid-State Drives* (SSDs) have gained tremendous attention in computing and storage systems due to significant performance improvement over *Hard Disk Drives* (HDDs). The cost per capacity of SSDs, however, prevents them from entirely replacing HDDs in such systems. One approach to effectively take advantage of SSDs is to use them as a caching layer to store performance critical data blocks in order to reduce the number of accesses to HDD-based disk subsystem. Due to characteristics of Flash-based SSDs such as limited write endurance and long latency on write operations, employing caching algorithms at the *Operating System* (OS) level necessitates to take such characteristics into consideration. Previous OS-level caching techniques are optimized towards only one type of application, which affects both generality and applicability. In addition, they are not adaptive when the workload pattern changes over time. This paper presents an efficient *Reconfigurable Cache Architecture* (ReCA) for storage systems using a comprehensive workload characterization to find an optimal cache configuration for I/O intensive applications. For this purpose, we first investigate various types of I/O workloads and classify them into five major classes. Based on this characterization, an optimal cache configuration is presented for each class of workloads. Then, using the main features of each class, we continuously monitor the characteristics of an application during system runtime and the cache organization is reconfigured if the application changes from one class to another class of workloads. The cache reconfiguration is done online and workload classes can be extended to emerging I/O workloads in order to maintain its efficiency with the characteristics of I/O requests. Experimental results obtained by implementing ReCA in a 4U rackmount server with SATA 6Gb/s disk interfaces running Linux 3.17.0 show that the proposed architecture improves performance and lifetime up to 24% and 33%, respectively.

Index Terms—Solid-State Drives, Data Storage Systems, Performance, Endurance, I/O Caching.

1 Introduction

With the ever-increasing demand for computational power, applications have become orders of magnitude more performance consuming compared with their descendants. To meet the performance demands, the computational power of computer systems has been increased by orders of magnitude in the recent decades. Storage systems, on the other hand, had a marginal performance improvement in the same timespan. *Hard Disk Drives* (HDDs) as the conventional devices employed in storage systems, have mechanical components which puts a tight upper limit on their maximum performance. This has made storage systems as a major performance bottleneck in computer systems.

To alleviate the performance gap between storage systems and the other parts of computer systems, many approaches have been employed at the *Operating System* (OS) level in recent years. Optimizing the ordering of I/O requests [1], placing relevant data pages close to each other [2], and merging small requests [3] are some of the basic approaches proposed at the OS-level aimed to enhance performance of storage systems. Such optimizations are mostly implemented at the OS block I/O layer since it routes all I/O requests issued from both applications and filesystems to the disk subsystem and has necessary means to optimize the storage system (shown in Fig. 1). In addition to optimizing the requests in the block I/O layer, employing a high performance device as a caching layer for HDDs has been practiced in recent years. As an example, the free space in the main memory can be used as a cache for storage devices in order to reduce the average response time. Due to the volatile characteristic of the main memory, this technique cannot be reliably used in storage systems, where data loss due to power outage result in significant customer dissatisfaction.

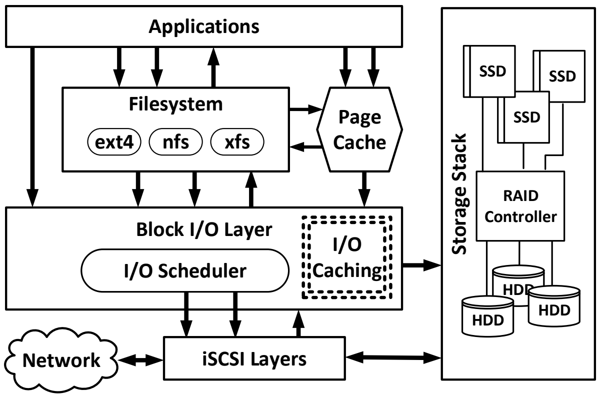


Fig. 1: Linux I/O stack diagram

To cope with both limited performance of HDDs and data loss concern in the main memory, recent studies have suggested using *Solid-State Drives* (SSDs) as a caching layer for HDDs [4], [5] (a.k.a., I/O caching shown in Fig. 1). Enterprise SSDs have 8.5x higher cost compared to enterprise HDDs and the price gap between them will not be closed in the upcoming years as

• *Reza Salkhordeh, Shahriar Ebrahimi, and Hossein Asadi are with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran.*
E-mail: salkhordeh,shebrahimi@ce.sharif.edu and asadi@sharif.edu

the price trend suggests [6], [7] (shown in Fig. 2). Therefore, SSDs should be used alongside with HDDs in order to design a cost-efficient, high-performance storage system. To effectively use SSDs as the OS cache layer, modifications in the conventional caching policies such as *Least Recently Used* (LRU) and Clock have been suggested by the previous studies [8], [9], [10]. Another suggested approach in the previous work is prioritizing various request types such as filesystem metadata, read, and random requests by using either absolute or relative priority [11], [12], [13], [14]. The absolute priority employed in previous studies is inefficient as observed by a recent study [15]. On the other hand, while the relative priority has more flexibility, the complexity of the request types and the storage devices (in particular SSDs) makes this approach inefficient in many cases. In addition, the coefficients for the relative priority cannot be adjusted dynamically in case a workload pattern changes during runtime. Another group of previous studies attempts to improve the LRU-based algorithms used as the eviction policy in the caching solutions [8], [9], [10], [16]. Unlike caching in the memory subsystem, improving the hit ratio in the storage system will not necessarily improve the average response time. This can be justified by the fact that a caching algorithm with a lower hit ratio that mostly redirect sequential requests to HDDs will likely provide higher performance than a caching algorithm with higher hit ratio that mostly redirect random requests to HDDs. This is due to HDDs demonstrate at least two orders of magnitude higher performance in sequential requests compared to random requests.

This paper proposes a *Reconfigurable Cache Architecture* for storage systems, called *ReCA*, which aims to improve system performance by categorizing I/O intensive applications and their characteristics with respect to their performance on various cache configurations. Based on a comprehensive characterization on the performance of various request types for both HDDs and SSDs, ReCA classifies I/O intensive applications into five major categories: a) random consumers, b) sequential producer/consumers, c) random producer/consumers, d) archival consumers, and e) large file generators. The proposed characterization study leads us to design an optimal cache configuration for each category of workloads to maximize system performance. Based on the observed characteristics of the running workload, one of the five categories which suits best the workload will be dynamically adjusted. Emerging workload types can be easily added to ReCA by modifying a data file without need for changing the architecture and/or re-compiling the code. This can also be done online during runtime. ReCA is also compatible to run multiple applications simultaneously by characterizing each application separately. ReCA will be reconfigured as soon as the running workload pattern transforms into another workload category. To the best of our knowledge, none of the previous studies have suggested a reconfigurable cache architecture for storage systems taking into account the workload patterns to improve I/O performance.

We have implemented ReCA on an open source caching platform, called *EnhanceIO* [17], with addition of more than 500 lines of code to the OS block I/O layer. The experiments have been done using a 4U server with SATA 6 Gb/s HDDs and SSDs to remove the interface bottlenecks. Linux kernel 3.17.0 is used throughout the experiments as the OS kernel of the testbed. The workloads used in the experiments consists of more than 75 traces from FileBench [18], FIO [19], Postmark [20], HammerDB [21], and many other publicly available storage traces. Experimental results show that ReCA can improve performance up to 24%

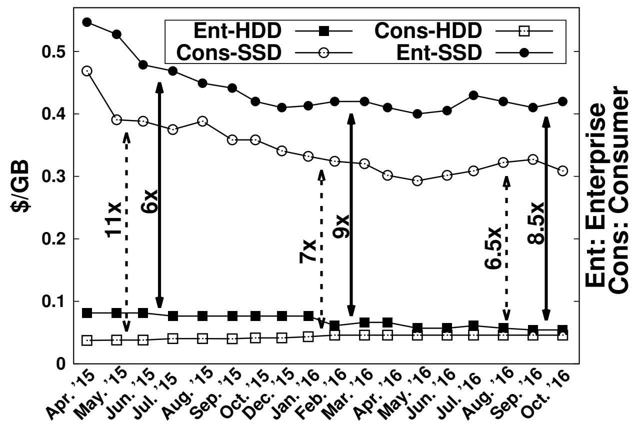


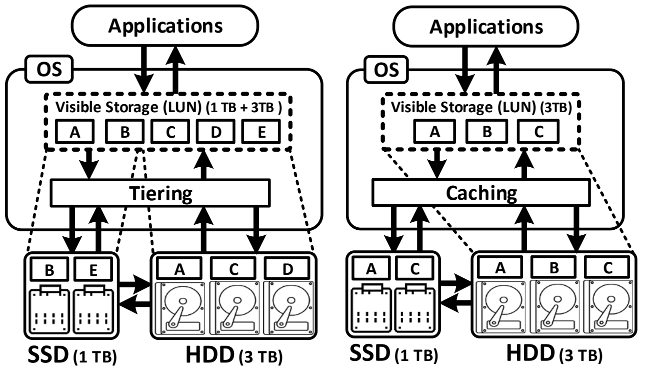
Fig. 2: SDD and HDD Price Gap Trend

(16%, on average) for various workloads compared to previous studies while removing the need for mirrored-cache configuration for SSDs in many applications and improving the lifetime of SSDs up to 33%.

The main contributions of this work are as follows.

- By examining performance of various storage devices under several request types, we demonstrate that assumptions made by previous studies on SSD performance such as superior read performance of SSDs compared to its write performance [22], [23] may not hold in all workloads or request types.
- A comprehensive workload characterization over 75 traces has been conducted in this paper to analyze I/O behavior of diverse applications with respect to various SSD caching policies and architectures which has not been done in previous studies.
- We conducted a detailed analysis investigating the effect of filesystem metadata on performance of storage systems which demonstrates that *(a)* filesystem metadata requests do not have the same request type (as opposed to the assumptions made in previous studies) and *(b)* the impact of filesystem metadata requests significantly differs across workload categories. Such important observations are fed into the proposed caching architecture to further improve performance of storage systems.
- Based on the proposed workload characterization, a *Reconfigurable Cache Architecture* (ReCA) has been proposed which aims to dynamically adapt to the changes in the I/O behavior and reconfigures itself toward optimal cache policy and configuration for the currently running workload. Emerging workload categories can be added to ReCA in runtime to be able to optimize itself toward new workloads. ReCA is able to reconfigure *cache line size*, *write policy*, and *eviction policy* online without any prior knowledge on the pattern of the running workload. This is achieved by an online monitoring system designed into ReCA that can adapt to the workload changes.
- While ReCA mainly aims to improve performance by optimizing cache architecture, it also removes the need for redundant SSDs in four out of five workload categories by employing *write-through* policy. This policy selection results in reducing the cost of caching (by removing redundant SSDs) and improving reliability by not keeping dirty data pages in the I/O cache. In addition, the lifetime of SSDs is improved (up to 33%) in two workload categories by redirecting all write requests to the disk subsystem.

The rest of this paper is organized as follows. Section 2 discusses the related work. In Section 3, an analysis on the workload



(a) Tiering (b) Caching

Fig. 3: Various Storage Architectures using SSDs

characterization will be presented. The proposed architecture is introduced in Section 4. The overall workflow of the proposed architecture is detailed in Section 5. The experimental setup and results are reported in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

SSDs can be integrated into the storage stack as a caching layer or as a high performance tier above HDDs in a tiering technique as depicted in Fig. 3. Both of these techniques have been examined in the previous studies and many algorithms have been proposed using either tiering or caching techniques. Tiering attempts to split data pages between storage devices called *tiers* in order to reach the required performance, power, and cost efficiency [15], [24], [25] (shown in Fig. 3a). In tiering, a data page will only reside on one of the storage devices, while in caching techniques (Fig. 3b), a copy of data block will be moved to the cache. Hence, the space efficiency of tiering is higher than caching, as illustrated in Fig. 3. Tiering solutions mainly focus on migrating data pages between tiers and are mostly suitable for steady workloads without any sudden changes in workload pattern [26].

Caching solutions, on the other hand, try to adapt themselves with the sudden changes in the workload pattern and perform more efficient in workloads with higher locality [26]. In addition, they are suitable for computer systems running many applications simultaneously or running virtualization platform [27]. LRU is used as the baseline algorithm for most caching solutions and many improvements for this algorithm have been suggested in previous studies. A technique based on *set sampling* and *set dueling* has been suggested in [28] to choose between two eviction policies for a group of cache sets based on hit ratio. Despite this technique is the best metric for performance evaluation of CPU caches, it is not suitable for SSD caches due to their characteristics and endurance limitations. ARC [8] maintains two LRU queues for capturing both recency and frequency of accesses to data pages to identify the most effective data pages for caching. mARC [10] is an extension of ARC which tries to identify unstable workloads and prevents caching data pages which might result in cache pollution while allowing performance critical data pages to be cached at the block I/O layer. One of the drawbacks of ARC is neglecting the cost of placing data pages in cache for each missed access. This limitation is addressed in LARC [9] by preventing cache pollution and keeping hot data pages in cache for a longer time. LARC, however, reacts very slowly to changes in the workload pattern and misses many opportunities for caching performance critical data pages.

Another group of the previous studies have focused on prioritizing one type over other request types in order to exploit SSDs or workload characteristics. AutoRAID [29] employs a redundant array (similar to RAID-1) as a tier for hot data pages while storing cold data pages in RAID-5 array. Hot data pages are identified dynamically which enables AutoRAID to reconfigure itself in case of workload changes. HDD-based caching has also been suggested in [30], [31] by dedicating a small portion of HDDs to hot data pages to reduce the disk head movement. Although AutoRAID and a few of follow-up studies have reconfiguration ability, they mainly are HDD-based which is not applicable to emerging storage devices and applications (unlike SSD-based architectures such as ReCA). A simple read/write counter for prioritizing requests has been suggested in [32]. Assigning absolute priority to metadata requests over regular requests has been suggested in [11], [12], [33] which is demonstrated to be inefficient in many cases [15]. In addition, this approach requires modifications in several layers of OS which reduces the generality of this approach and is restricted to the examined filesystem since they use different algorithms and, moreover, modifications cannot be easily applied to the other filesystems. SSDs have a relatively high performance on random requests, therefore, previous studies suggested assigning higher priority to random requests over sequential requests. One technique that employs this prioritization was proposed in [12] where the priority for caching a data page is proportional to the inverse of its size. A three-level table has been employed in order to calculate the overall priority of each data page and based on the priorities, candidate data pages will be selected for caching in SSDs. A similar approach has been employed in [34] for improving performance in the virtualization environments. RPAC [16] is another approach that considers locality of accesses to HDDs. By prioritizing data pages near each other and caching these data pages in SSDs, the locality of accesses in HDDs increases which will improve its performance. HybridStore [35] proposes a planner which optimizes the total cost of a storage subsystem based on its performance behavior and SSD lifetime using *Integer Linear Programming* (ILP). In addition, a dynamic controller has been proposed that models performance behavior of SSDs in runtime. Unlike HybridStore, ReCA employs a more detailed workload characterization over real hardware and aims to optimize itself based on SSD characteristics. The dynamic controller of HybridStore, however, can be employed in ReCA to further increase its ability to increase SSD lifetime. Moirai [36] is a hypervisor caching architecture able to provide various caching *Quality-of-Service* (QoS) options such as minimum bandwidth per *Virtual Machine* (VM) and employing different cache policies for VMs. The dynamic approach of ReCA can be exported to hypervisor-based caching architectures similar to Moirai [36] to better optimize caching for VMs.

LeavO cache [37] tries to improve reliability of SSD caching by keeping both old and new data pages in SSD. A more space-efficient architecture has been proposed in [38] to reduce the number of writes to SSD cache which improves its lifetime by storing modifications compared to old data pages. Employing a log-structured approach can also improve SSD lifetime as suggested in [39], [40]. Such architectures can be used alongside ReCA to further improve its endurance and/or reliability.

Modifying the interface between OS and the disk subsystem to further improve the performance is examined in [5]. Such

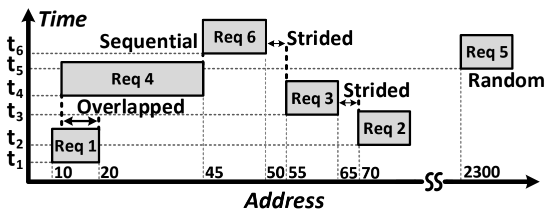


Fig. 4: Arbitrary sequence of requests over time

techniques, however, require hardware and firmware modifications which limits both their applicability and generality. Altering the storage stack to feed semantic information of applications to the caching layer has been examined in [13] to improve the performance of databases. Such technique also suffers from poor generality. SSD caching can also be employed in the client-server solutions [41], [42], [43]. Reducing the number of writes in SSDs and improving their lifetime have been studied in many works [44], [45]. Clustered tiering consists of HDD, SSD, *Network Attached Storage* (NAS), and other storage devices has been suggested by several previous studies [46], [47], [48]. The main aim of these studies, however, differs from the goal of this paper which is mainly improving the performance of the storage subsystem.

Due to the importance of the request classification, several classification studies have also been proposed in previous work [49], [50], [51]. The studies presented in [50], [51] are focused to investigate and characterize I/O requests using the semantic information available at the filesystem layer which vastly differs from the target information in our study at the block I/O layer. Another characterization study was proposed in [49], which aims to classify I/O requests at the storage device level (a single SSD) to address long write latency, high write energy, and limited endurance of *Phase-Change Memory* (PCM) cache designed for SSDs, by employing a small DRAM alongside PCM, while maintaining the same performance level of previous architectures. The discussion of such studies targeting internal architecture of storage devices is beyond the scope of this paper.

3 Workload Characterization

The main goal of any SSD caching architecture is to identify the performance critical data pages and predict the workload pattern. The actual response time of requests in HDDs and SSDs significantly depends on the previous requests due to the HDD head position and internal buffers/caches in both HDD and SSD which prevents us from using average response time reported in device datasheets. Therefore, before proposing an effective caching architecture, conducting a study on the workload patterns and methods for identifying performance critical data pages is crucial. Here, we present a comprehensive study on identified parameters affecting the I/O performance which will be used in the proposed reconfigurable cache architecture in order to improve performance compared to the previous studies.

Since HDDs and SSDs have distinct characteristics and internal structures, they exhibit differing behavior over various request types. On the other hand, analyzing the performance of the storage devices, which is crucial for choosing the right type of data pages for caching, highly depends on the request types. Therefore, before analyzing the performance of storage devices, a classification on the request types is required. This classification (Sec. 3.1) enables us to benchmark storage devices from various perspectives. The complexity of the storage devices, specially SSDs, increases the importance of this benchmarking. In addition, commercial

TABLE 1: Testbed configuration

Device	Model	Sequential Read/Write*	Random Read/Write*
CPU	Xeon E5620	-	-
Memory	32 GB DDR3	10 GB/s	
HDD	Western Digital Red Pro HDD	120 MB/s	80/150 IOPS
SSD	Samsung SSD 850 PRO	480/440 MB/s	20k/30k IOPS

*Actual measured performance in our experiments

SSDs employ many optimizations and buffering techniques in order to improve the performance and/or lifetime. Moreover, the complex structure of SSDs cannot be predicted easily without such experiments. Request interleaving is another important factor affecting the performance of the storage subsystem which will be discussed in detail in Sec. 3.2. After careful analysis of the request types, workloads will be categorized and based on the proposed categorization, optimal cache configuration for each workload category will be presented in Sec. 3.3. Apart from the request types, there are many semantic information available at the OS level which can help us identify the request pattern for each data page. Requests issued for accessing filesystem metadata are one of the most important data requests as they have significant effect on the filesystem performance [11], [12], [15]. As such, a complete analysis has been done in this work regarding filesystem metadata and how a caching architecture can benefit from such analysis (Sec. 3.4).

3.1 Request Classifications

In the proposed classification, I/O requests are classified into four major classes which are: a) *sequential*, b) *random*, c) *strided*, and d) *overlapped*. Since the response time of a request is dependant to the previous requests, each request will be evaluated and classified based on the previous requests. Fig. 4 depicts an arbitrary sequence of requests over time. Upon arrival of a request, it will be compared to the previous requests in order to identify its type. The number of the past requests used for comparison depends on the general architecture of the OS and employed devices. Here, we assume size of 64 requests for request history queue which is close to the queues employed in the disk and the I/O scheduler in OS. In Fig. 4, request 6 is classified as *sequential* since its starting point is exactly after the ending point of one of the requests issued earlier. The starting point of a *strided* request has a small gap from the ending point of one of the previous requests similar to request 2 compared to request 3 in Fig. 4. The performance of the strided requests is lower than the sequential requests since in rotational disks, disk head has to move a few sectors to start responding to the second request. In addition, I/O scheduler in OS tries to merge sequential requests before sending them to disk and strided requests cannot be merged together because of the existing small gap between them. If there is no request adjacent to a request, similar to request 5, it will be considered as *random* which will have the worst performance on HDDs compared to other request types. Request 4 is an *overlapped* request since it shares a data page with one of the previous requests.

In order to obtain the performance characteristics of SSDs and HDDs, we have benchmarked two disk drives with the request classes using *IOMeter* benchmarking tool [52]. The specification of the drives used for this experiment is presented in Table 1. Since our goal is to measure the overall disk performance, all disk optimizations such as read-ahead or disk cache have been enabled. Random requests are 4KB requests where their addresses are randomly distributed over the address space. The probability distribution for choosing the request addresses was set to uniform distribution to measure the pure random performance of disks. The

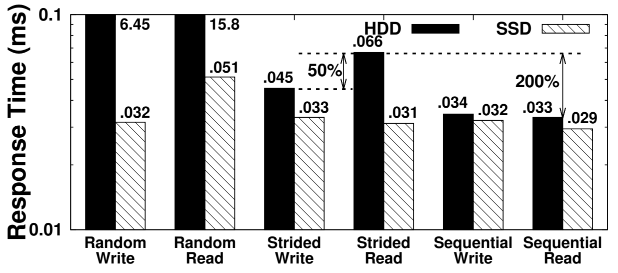


Fig. 5: HDD and SSD latencies for various request types

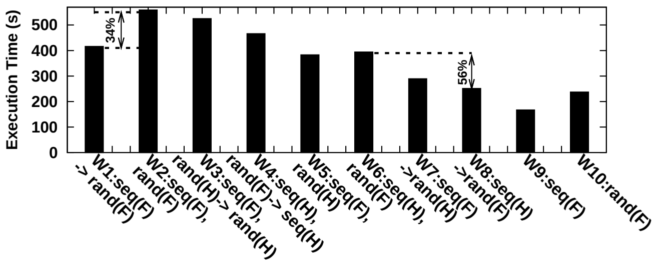


Fig. 6: Execution time of synthetic workloads on HDD

sequential requests have 4KB size and the address of each request is set to the ending address of the last request. Strided requests have the same size as sequential requests and each request is issued to the ending address of the last request, plus an offset which is set to 8KB.

Fig. 5 shows the average response time of the benchmarked SSDs and HDDs for the mentioned request classes. Read and write requests are separated since HDDs and SSDs demonstrate different performance behavior. In addition, the embedded cache in disks can cache all write requests (when running in write-back mode) but it can only respond to read requests from cache when the data page is present in the disk cache. Contrary to the assumptions made in the previous work, HDDs do not have symmetric performance in all requests and their performance in random requests is asymmetric. HDDs can serve random write requests with twofold performance compared to random read requests due to the internal caching mechanism. HDDs have symmetric performance in sequential requests and the difference between read and write performance in strided requests is about 50%. The performance gap between strided and sequential requests is about 30% in write requests and 200% in read requests. The reason for such performance improvement in sequential read requests is the read-ahead functionality in the disk controller. This experiment shows that the assumptions made in previous work that HDDs have symmetric performance is partially valid but there are some cases that the performance difference between read and write requests is greater than 2x.

We have also examined the assumptions made in the previous work considering that the request type does not affect the performance of SSDs and their performance in read requests is higher than in write requests. Fig. 5 shows that although these assumptions are valid in most cases, there are numerous workloads in which these assumptions do not hold for SSDs. For example, SSDs have higher performance in random write requests compared to random read requests, similar to HDDs. The difference between these two, however, is less than their difference in HDDs which is about 60%. Apart from the mentioned difference, the performance of SSDs in read and write requests is similar to each other and the performance difference is less than 10% in strided and sequential requests. Therefore, strided requests should have higher priority for caching than sequential requests. Although random write requests have less response time than random read requests in SSDs, the performance gain of caching random read requests is higher than random write requests due to the performance difference of read and write requests in HDDs and SSDs.

3.2 Request Interleaving

Classifying requests is only a part of the workload characterization process and there are other important factors which should be considered in order to accurately predict the effect of each request on the performance. The order of the requests and the interleaving of requests from different classes significantly affect the performance. In order to show the effect of interleaved requests on performance, we have benchmarked HDDs with ten synthetic workloads consists of a set of random and sequential requests. Each workload has two stages and the second stage will start as soon as all requests from the first stage are completed. Each stage contains only either sequential, random, or both random and sequential requests. Since the execution time of the workloads are similar to each other on SSDs, we only analyze the performance of HDDs for the sake of brevity.

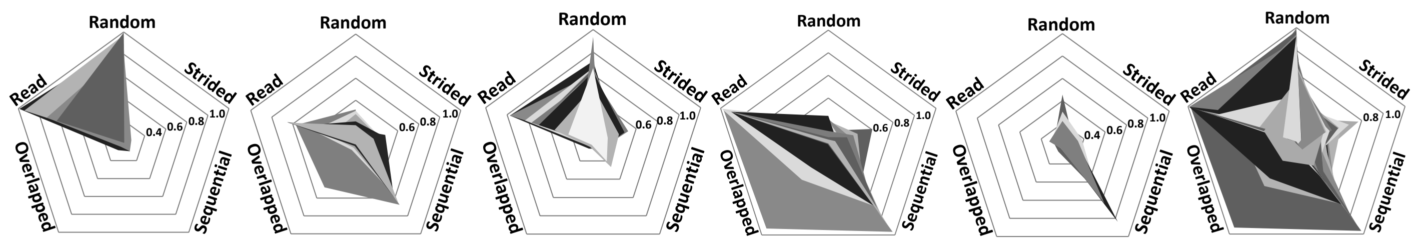
Fig. 6 shows the execution time of various synthetic workloads for HDD. The "->" sign shows the transition from the first to second stage and "," denotes that the workloads are running simultaneously. "F" and "H" letters inside the parenthesis denote executing 20,000 and 10,000 requests, respectively. Workloads W1 to W4 have the same number of requests, W5 and W7 have 50% less random requests, and W6 and W8 have 50% less sequential requests compared to the other workloads. W2 has 34% higher execution time compared to W1 although they have the same number of requests since the requests of W2 are interleaved with each other which results in more disk head movements. The effect of interleaving of requests is related to the type of the interleaved requests and how much requests are interleaved, e.g., W3 and W4 have different execution time. Another interesting observation in Fig. 6 is that although W5 and W6 have different number of request types, their performance is similar since their requests are interleaved and the disk observes almost similar pattern for both. The difference between the execution time of the interleaved requests depends on the ratio of the requests. W1 and W2 have 34% execution time difference while W6 and W8 have 56% difference in their execution time.

3.3 Workload Categories

In order to categorize I/O workloads [20], [21], [53], [54], [55], [56], we have explored many traces from various sources in addition to the traces obtained by running benchmark suits. Over 75 traces have been analyzed and based on this analysis, five categories for workloads have been proposed which are a) random consumers, b) sequential producer/consumers, c) random producer/consumers, d) archival consumers, and e) large file generators. In the remainder of this section, these categories will be detailed.

3.3.1 *Random Consumers:*

Random consumers are read-dominant workloads that access the underlying storage device with random requests such as database management systems. These workloads exhibit very low performance in HDD-based disk subsystem. In addition, the write-back cache of HDDs which is able to cache random write requests and send them to the rotational magnetic part of the disk at a later



(a) Rand. Cons. (b) Seq. Prod./ Cons. (c) Rand. Prod./ Cons. (d) Archival Cons. (e) Large File Gen. (f) Overall

Fig. 7: Workload characterization based on the request types

time, is unable to perform well for *random consumers* since most requests are read. Fig. 7a shows the characterization of majority of analyzed workloads categorized as *random consumers*. The cache configuration for this category should have a small cache line size (same as application block size, e.g., 4-8KB) since almost all requests are random with a small size and, hence, using large cache line size will result in occupying the cache with undesirable data. In addition, the access frequency is the most suitable metric for identifying performance-critical data pages since all request have the same size and request type. Similarity of the requests means that there is no need for filtering a specified request type in order to cache or directly send them to the disk.

3.3.2 Sequential Producer/Consumers:

Most organizations have a sharing system based on network filesystems that users can access their files and share them with the others. File types in this scenario are mostly documents that have a medium file size. Since the underlying system is based on a filesystem, upon editing a file, the whole file contents will be uploaded to the server. This will result in sequential accesses to the server which contains both read and write requests. The optimal cache configuration for this category should consider the recency of accesses since users tend to access the newly created files more than old files. The cache line size should preferably be set to larger values (128KB) since requests are mostly sequential and the eviction policy should be similar to LRU algorithm in order to catch the recency of accesses. The degree of file sharing determines the number of overlapped requests in these workloads. Fig. 7b depicts the characterization for this category which shows that the percentage of the overlapped requests varies inside this category. The interleaving of sequential requests from accessing various files will result in a semi-random access pattern observed at the disk level since different files reside on different parts of the disk. Therefore, identifying the I/O stream for each file and either caching all requests or ignoring them will help the cache to send more sequential requests to the disk and improve its performance.

3.3.3 Random Producer/Consumers:

Fig. 7c shows the characterization of the workloads in this category. Although most requests in this category are random (similar to the *random consumers* category), issuing many write requests by the applications in this category results in vast difference between two categories. The performance gain on random read requests is almost twofold of random write requests as reported in Fig. 5, which motivates us to favour random read requests more than random write requests for caching. Our analysis shows that most mail servers and a few of *On-Line Transaction Processing* (OLTP) servers belong to this category. Therefore, in addition to the access frequency which is beneficial in identifying performance critical random requests, using recency is also helpful since for example, users access their new e-mails more often. The cache configuration for this category consists of a small cache line size, a priority based eviction policy which assigns higher priority to read requests, and accumulates the priority based on access frequency. This will enable us to capture recency, frequency, and read priority simultaneously.

3.3.4 Archival Consumers:

File Transfer Protocol (FTP) and media servers have large number of concurrent users that access files for read purpose. The size of the files in this category is quite large and almost all requests are sequential. Fig. 7d depicts the percentage of various request types for *archival consumers* category. Since many concurrent users access the same set of files, the percentage of the overlapped requests is higher in this category compared to the *sequential producer/consumers* category. The concurrent users that access the same file may read different parts of the file at the same time which results in numerous strided requests. The cache line size for this category can be set to a large value (128KB) in order to reduce the overhead of moving data pages to/from the cache. The proposed eviction policy for this category is set to consider both frequency and recency of accesses to data pages. The priority of each data page will be calculated based on its position in the LRU queue and the number of accesses to that data page. Detecting sequential request streams can help to either cache or bypass cache for all requests of a stream. This technique can reduce the randomness of the requests in both SSD and HDD which will improve performance.

3.3.5 Large File Generators:

Surveillance systems that store videos and monitoring systems have a few processes that write large files to the disk. Almost all requests in the workloads in this category are writes and each process issues sequential requests. Fig. 7e depicts the characterization of the workloads in this category. Although all processes issue sequential requests, there exists many random requests in the examined workloads. Our analysis shows that interleaving of the requests from various processes is the root cause of many sequential requests to appear as random in this category. The proposed cache configuration for this category should have a large cache line size and should be configured as write-back mode. The eviction policy for this category is similar to *archival consumers* with a slight modification. Upon evicting a data page, ReCA searches through the cache and all data pages with the physical address near the evicted data page will be evicted as well. Since sequential requests have higher performance in HDDs, evicting data pages with physical address near each other will reduce the performance cost of cache flushing and ensures that the sequential requests from a process are committed at the same time to the HDD together and the disk observes less number of random requests.

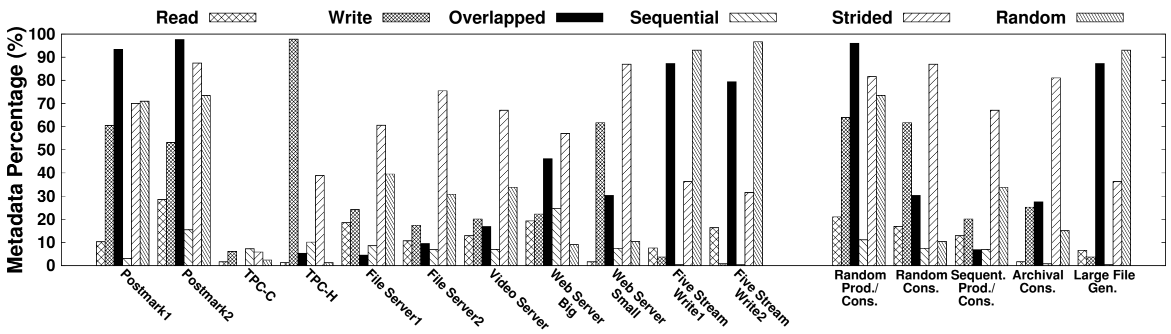


Fig. 8: Percentage of filesystem metadata requests in various request types of workloads

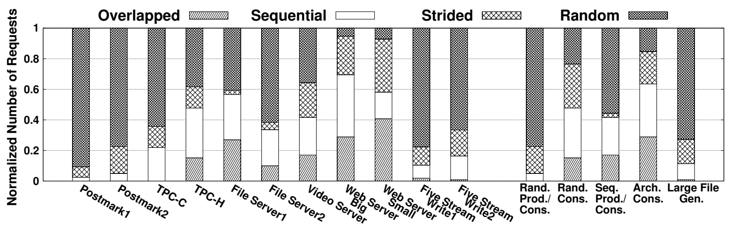


Fig. 9: Filesystem metadata request type breakdown

3.3.6 *Workload Generality:*

Fig. 7f shows the characterization for all analyzed workloads that cover majority of possible scenarios. The only gap in the analyzed workloads is the lack of workloads with large number of strided requests. This is due to the fact that applications are very unlikely to access the underlying disk with a strided pattern. Database management systems, however, might access disk with strided pattern in order to read their indices. The captured behaviour of database management systems do not show this pattern since they try to optimize the requests issued to HDDs and prevent occurrence of such pattern. ReCA is designed in such a way that new workload categories and their optimal cache configuration can be added even in runtime without any architecture modifications. Therefore, if new applications with unusual behaviour emerged, they can be added to ReCA to maintain its workload generality.

3.4 Filesystem Metadata

Filesystem issues many requests for managing its internal data structures and storing additional information for each file. These additional data blocks are called filesystem metadata blocks. Since each metadata block contains information about numerous number of user data blocks and in addition prior accessing a user data block, its corresponding metadata block should have been accessed, the metadata requests have higher impact on the overall performance than individual user requests. The importance of filesystem metadata from performance perspective has been analyzed and demonstrated in previous studies [11], [12], [15]. These studies, however, neglect further analysis of filesystem metadata, its various types, and the characterization of filesystem metadata in different workload types. For example, in reading sequential data blocks from a file, only one metadata request (to read an indirect block) is required to find physical location of 1024 data blocks. Reading random data blocks from a file, however, requires reading one indirect block per data block since requests are scattered across file logical addresses.

Fig. 9 depicts the characterization of the filesystem metadata for many representative workloads and the average of all workloads for each workload category. The overlapped requests have been considered with the other request types in Fig. 9 for simplicity. If a request is flagged as overlapped, its main type is ignored in order to present all request types in one figure. As shown in Fig. 9, different workload categories have different metadata access patterns which is in contrary to the assumptions made in the previous work [12], [15]. Random consumers category, despite of being a random workload, has low percentage of random filesystem metadata requests. Our careful analysis shows that applications in the *random consumers* category have less locality than applications in *random producer/consumers*. Higher locality results in issuing less metadata requests to disks since filesystem tries to cache metadata information and each cached metadata block can serve more requests if the workload demonstrates higher locality. Therefore, in workloads with higher locality, the requested metadata blocks from disk (which are less in number) will be more random. In workloads with less locality, more metadata requests will be issued to disk and since the filesystem tries to place metadata requests close to the actual user data, requests will be less random and instead more sequential or strided. Such important observation will clarify the difference between *random consumers* and *random producer/consumers*. Workload categories have other dissimilarities such as different strided requests percentage as well. The mentioned differences support our claim about the need for treating filesystem metadata requests differently across workload categories.

To further analyze the access pattern of the filesystem metadata requests, the percentage of the filesystem metadata requests for each request type is depicted in Fig. 8. The contribution of the filesystem metadata differs vastly across workload categories which makes using one single rule for prioritizing them not accurate enough. For example, both of the *large file generators* and *archival consumers* categories are sequential workloads, however, the breakdown of the filesystem metadata requests differs vastly across them. One common pattern among all workload categories is that most strided requests are for filesystem metadata which confirms our assumption that applications do not issue many strided requests to disk. Such information enabled us to further optimize the workload characterization process and prioritize filesystem metadata requests based on their impact on the overall performance for each workload category.

It is noteworthy that identifying filesystem metadata at the block I/O layer in the OS is not possible since filesystems do not leverage these information. To this end, previous studies tried to modify the kernel structures in order to pass the mentioned information to the lower layers of I/O stack. This approach is possible in the proposed workload characterization. If modifying OS layers is intractable, previous work will be unlikely to consider the filesystem metadata. The proposed workload characterization, however, can use the average probability that a request is filesys-

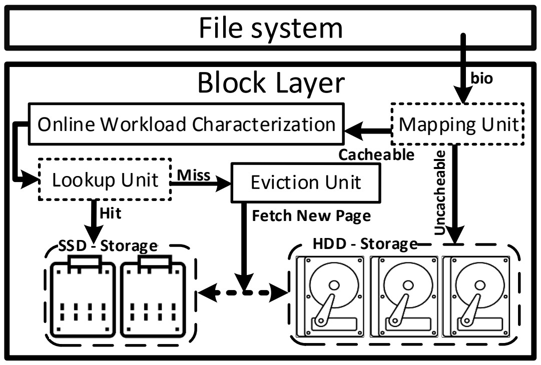


Fig. 10: The proposed storage architecture (ReCA)

tem metadata based on the information provided in Fig. 8. This makes the proposed workload characterization more accurate than the previous studies in identifying filesystem metadata requests.

4 Proposed Architecture

In the proposed architecture, we monitor the running workload and based on the observed request types, the most suitable workload category is selected for the running workload. We then compare the running workload with the predefined workload categories based on the percentage of the request types and the interleaving of the requests. Based on the selected category, the cache configuration will be set to one of the predefined configurations presented earlier in Section 3.3. The cache reconfiguration will be done online and the users will not notice the reconfiguration process, except for the performance improvement upon completing the reconfiguration.

4.1 Overall architecture

Fig. 10 shows the high level architecture of ReCA which consists of four major modules: a) *mapping unit*, b) *workload characterization unit*, c) *lookup unit*, and d) *eviction unit*. Two units with dashed-line boxes are general units which exists in almost all caching architectures and units with solid lines boxes are presented in the proposed architecture. The mapping unit receives requests from upper layers and breaks them into smaller requests with the same size as the cache line size. If the request is tagged as *uncachable*, the mapping unit will send it directly to the disk and if the corresponding data page exists in the cache, it invalidates its block. The lookup unit searches through the cache and if it finds the requested data page, it redirects the request to SSDs and otherwise, it is sent to HDDs. *Online workload characterization unit* monitors the arrived requests and tries to find the best suited workload category and based on these information, it updates the internal data structures and reconfigures the cache. Contrary to the conventional caching algorithms that move a data page to the cache in case of a miss, ReCA either does not move the page to the cache or will move it at a later time which is decided by the eviction unit in the proposed architecture. The eviction unit receives the miss requests and decides whether it should be placed in cache or not. This unit also decides which data pages are no longer needed in the cache and can be evicted in order to free up cache space for new incoming data pages.

Fig. 11 depicts the detailed modules and data structures used in ReCA. The boxes with dotted lines show the data structures, boxes with dashed lines are for general modules, and boxes with solid lines depict the modules designed for the proposed architecture. In order to identify the request type, the queue manager puts the

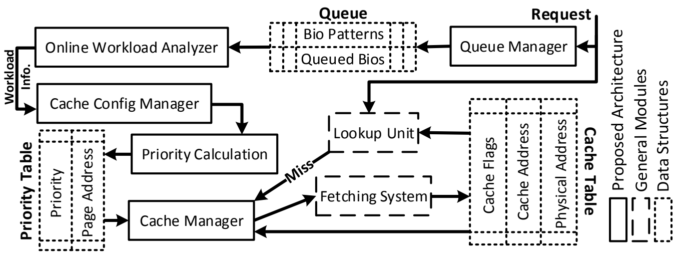


Fig. 11: ReCA: Internal details

requests into a queue with length of 64 to compare them with the previous requests and decide their types. *Lookup* unit looks through the *cache table* in order to find the requested data pages. If a data page is not available in the cache, the *cache manager* will be notified. This module fetches the priority of the requested data page and if it has higher priority than a data page which resides in the cache, it calls the *fetching system* to replace the data page in cache and updates the *cache table*. *Online workload analyzer* monitors the incoming requests and if the pattern of the requests changes, the *cache configuration manager* will be notified to change the *priority table* to reflect the change in the workload. The monitoring function can detect phases within an application. In addition, multi process applications working with the same data will be recognized by the monitoring function. *Priority table* is managed using *B-tree* and can be expanded up to 50% of the total free memory. *Priority calculation* unit also will be notified to change the algorithm for calculating the priority. Since each workload category has a unique algorithm for deciding the performance criticality of each data page, the history table includes five columns for each data page which imposes significant overhead to the system. In order to reduce this overhead, at any given time, only one column of the history table is active and when the workload category changes, the corresponding data structures will be freed.

4.2 Proposed Algorithms

ReCA uses three main procedures to manage the cache internals and three other procedures for characterization of the workloads and reconfiguring the cache. Algorithm 1 shows the procedures for managing the cache, i.e., *lookup_function*, *cache_manager*, and *access_history_manager*. The former procedure outlined in lines 1 through 14, is called for each incoming request. If the request size is greater than the cache line size, it will be divided into several subrequests. The incoming request will be responded after all subrequests have been served. If an error occurs during processing of a subrequest, the request will be replied by an error code (not shown in Algorithm 1). *Lookup function* searches through the cache and tries to serve as many as possible requests from cache. Since this function is the main entry point of ReCA, *Access History Manager* is called from this function (line 3) in order to update the request queue. When a miss occurs, *Cache Manager* will be called in order to decide whether or not the requested data page should be moved to SSD. If the cache has empty space, the page will be moved to SSD for future references. When the cache is fully occupied, the priority of the requested data page will be compared to the priority of the data pages residing in the cache and if the requested data page has higher priority, it will replace the data page with minimum priority. Since the overhead of managing and comparing the priorities becomes significant in case of large cache size, ReCA divides the cache into several sets

Algorithm 1 Caching Algorithm

```
1: procedure LOOKUP FUNCTION
2:     Upon reaching of a new I/O as new_io
3:     Access_History_Manager(new_io)
4:     for each cache_line_size block accessed through new_io as blk do
5:         if blk is in cache_table then
6:             cache_table.Priority[blk.addr] ← history_table.Priority[blk.addr]
7:             issue request to cache
8:         else
9:             cache_blk ← Cache_Manager(blk)
10:            if cache_blk != NULL then
11:                cache_table[cache_blk] ← HDD[blk.addr]
12:            end if
13:        end if
14:    end for
15: procedure CACHE_MANAGER(blk)
16:    for every block in cache_table as blk do
17:        if blk.Flag == INVALID then
18:            Return blk
19:        end if
20:    end for
21:    min_pr_blk ← page cache block with minimum priority
22:    if cache_priority_table[min_pr_blk] < priority_table[blk.addr] then
23:        if cache_table[min_pr_blk].flag == DIRTY then
24:            Flush cache_table[min_pr_blk] to HDD
25:        end if
26:        Return min_pr_blk
27:    else
28:        Return NULL
29:    end if
30: procedure ACCESS_HISTORY_MANAGER(new_io)
31:    Queue.Push(new_io)
32:    if Queue.length > Queue_Length_Threshold then
33:        old_io ← Pop(Queue)
34:        io_type ← Characterization(old_io)
35:        for each cache_line_size block accessed through old_io as blk do
36:            priority_table[blk.addr] += Priority_Calc(io_type)
37:        end for
38:    end if
39:    io_list.append(io_type)
40:    if io_list.length > Workload_Check_Threshold then
41:        Analyze_Workload(io_list)
42:    end if
```

and only compares the priorities within a set. Having a low hit ratio in the cache can result in high number of pending jobs for moving data pages between disk and SSD which will impose significant overhead. In order to limit this overhead, the number of pending jobs is limited and queued jobs will be started asynchronously. *Access History Manager* manages request queue which is used for determining the workload category. Upon evicting a data page from the queue, the priority of the data page will be updated. In lines 39 through 42, *Analyze_Workload* is called periodically after processing a number of requests in order to analyze the current workload characteristics and change the workload type if necessary. The number of processed requests between calls to *Analyze_Workload* determines the reaction time of ReCA to the changes in the workload. Small values trigger reconfiguration more frequently which incur more performance overhead to the system while using large values delays identifying the change in the workload. The value used for *Workload_Check_Threshold* in ReCA is set to 100,000 requests.

The main characterization and reconfiguration processes used in ReCA are shows in Algorithm 2. *Characterization* function analyzes a request based on the other requests in a queue and identifies its type. Characterizing individual requests is the first step of the workload characterization process which is outlined in lines 1 through 21 based on the request classification presented in Section 3.1. The next step after characterizing the requests is prioritizing them based on their performance on storage devices which is done in *Priority_calc*. This function calculates the assigned priority for a request and adds it to the priority of the corresponding data page. The calculated priority consists of overlapped (if request is flagged as overlapped), access type, and read/write priorities. In the implementation, read/write priority is set to be dependant to access type in order to have more flexibility in assigning priorities (not shown in Algorithm 2). The priority values are extracted from predefined priorities of the workload categories which are based on the analysis discussed in Section 3.3. New workload categories with their corresponding priorities and optimal cache configurations can be added to *characteristics table* in runtime to extend the characterization capabilities of ReCA.

The actual reconfiguration is done in *Analyze_workload* function which first decides the current workload type and then triggers the reconfiguration process if necessary. The proposed cache reconfiguration process is online without disturbing application IOs. The cache will not be flushed entirely and only the required data pages are brought to the cache and lower priority data pages are evicted. Workload identification is based on the predefined workload types and the collected data from the current workload. ReCA calculates the *Euclidean* distance between the current workload type and each of the predefined workload types. The workload type with minimum Euclidean distance will be selected as the new current workload. After choosing the workload type, cache data structures will be reconfigured. The reconfiguration process starts by changing the priorities of data pages based on the new workload type. After updating the priorities, many data pages will have lower priorities which results in eviction from cache. The limitation employed in the number of concurrent evictions prevents the performance degradations because of the large number of evictions after updating the workload type. Hence, the performance impact of cache reconfiguration is limited and applications will not observe any performance degradation. ReCA can have different eviction policies based on the workload type which is set in Line 33 of Algorithm 2. Workload types can have different cache line sizes which are optimized towards their characteristics.

In case of running multiple applications simultaneously, ReCA can identify category of each application, separately by tagging the process *Identification Number* (ID) of applications in *(io list)* and calculating the workload type for each process. This technique can also be employed in virtualization environments by characterizing each virtual machine category. The memory overhead of maintaining required data structures is negligible and does not affect the performance of ReCA.

When *current_workload* is updated due to changes in the workload characteristics, the cache line size may need to be updated as well. Since both *cache_table* and *priority_table* are dependant to the cache line size, ReCA reconstructs these tables. In case of updating from a larger cache line size to a smaller cache line size, ReCA extends each entry into many entries and sets the priority of new entries to $\frac{original\_priority}{\#\_of\_new\_entries}$. Therefore, the total value of priorities in the *priority_table* remains the same. Migrating from a small cache line size to a larger cache line size, however, is not trivial since a large cache line size spans across many data pages and if one of the data pages belonging to a cache line does not exist in the cache, the incoming request will not be responded. To fix this issue, ReCA fills the missing data pages asynchronously and stores a bit for each data page in cache lines in order to identify the missing data pages. If a request for a missing data page arrives, ReCA (a) issues a synchronous request to the disk subsystem, (b) retrieves the data page, and then (c) responds the user request. The retrieved data page will be sent to SSD in order to fill the missing data page.

Algorithm 2 Characterization Algorithm

```
procedure CHARACTERIZATION(OLD_IO)
    for each I/O in Queue as temp_io do
        if old_io.size > Seq_threshold or old_io.end == temp_io.begin then
            result.access ← sequential
            Break
        else if (old_io.begin > temp_io.end and old_io.begin - temp_io.end < Strd_threshold) or (old_io.end < temp_io.begin and temp_io.begin - old_io.end < Strd_threshold) then
            result.access ← strided
        end if
        if (old_io.begin > temp_io.begin and old_io.begin < temp_io.end) or (old_io.end > temp_io.begin and old_io.end < temp_io.end) then
            result.isOver ← True
        end if
    end for
    if result.access == null then
        result.access ← random
    end if
    if old_io.type equals write then
        result.R_Wr ← WRITE
    else
        result.R_Wr ← READ
    end if
    Return result
procedure PRIORITY_CALC(io_type)
    result ← 0
    if io_type.isOver then
        result += characteristics_table[current_workload].over_priority
    end if
    result += characteristics_table[current_workload].acc_priority[io_type.access]
    result += characteristics_table[current_workload].r_wr_priority[io_type.io_type.R_Wr]
    Return result
procedure ANALYZE_WORKLOAD(io_list)
    current_workload ← workload type with minimum Euclidean distance to io_list
    Update Priority_Table based on current_workload
    eviction_policy ← characteristics_table[current_workload].eviction_policy
    Reconstruct cache_table for new cache_line_size
```

5 Workflow

In this section, the overall workflow of ReCA is presented. The workflow which enables ReCA to be reconfigurable in the run-time with optimized configuration regarding the current workload consists of a set of offline analysis and a comprehensive characterization in addition to a novel online process for cache optimization. The overall workflow of ReCA consists of offline and online processes is demonstrated in Fig. 12. The goal of offline processes is to investigate the performance of storage devices over real-world applications and cache configurations and then decide the optimal cache configuration for each application type. The online processes try to *(a)* monitor the currently running application, *(b)* identify its type, and *(c)* reconfigure cache architecture to best suit the application performance requirements.

5.1 Offline Processes in ReCA

To determine an optimal cache configuration for various applications, the storage devices are extensively examined using synthetic and real-world applications (*Hardware Analysis* in Fig. 12). A wide range of enterprise applications are analyzed and categorized in the next step to investigate their I/O behaviour (*Workload Analysis* in Fig. 12). Since many of the inner-workload interactions between requests cannot be evaluated without actual experiments, another step is added to the offline processes (*Workloads Characterization* in Fig. 12) in which the actual performance of various applications is tested over real storage devices to determine their actual performance. The output of *Workloads Characterization* determines the actual workload categories used in ReCA along their performance characteristics. In *Cache Opt. Config. Finder* unit, enterprise applications will be executed under various cache architectures and policies in order to determine the optimal cache configuration for the target application type. Such cache architectures and policies are given to the ReCA online section by filling *Workload Config. Table*.

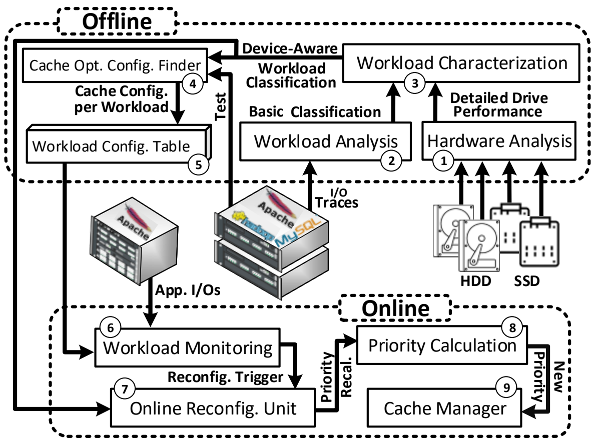


Fig. 12: ReCA workflow

5.2 Online Processes in ReCA

In addition to the traditional modules of a caching architecture, ReCA employs other modules to detect the application type and reconfigure the cache accordingly. Currently running application is monitored in *Workload Monitoring* and upon detection of any change in the workload behaviour (based on predefined workload types in *Workload Config. Table*), the reconfiguration process is triggered. The reconfiguration process takes the current and the target cache configurations into account and plans the reconfiguration with minimal performance overhead and no downtime for application I/Os. The internal priority and history of data pages is also updated to reflect the changes in the application type.

6 Experimental Results

This section presents the experimental setup and results for evaluating ReCA against previous caching architectures. Section 6.1 describes the experimental setup. The evaluations are presented in Section 6.2. Parameter sensitivity is discussed in Section 6.3. Section 6.4 reports ReCA performance under multiple running applications. Performance overhead is presented in Section 6.5. Finally, the performance of running ReCA in tiering mode is discussed in Section 6.6.

6.1 Experimental Setup

In order to accurately evaluate ReCA, all experiments in this paper have been performed in a physical server with Xeon E5620 CPU and 32 GB of main memory. The operating system of the testbed was Ubuntu14.04 running Linux kernel 3.17.0. The specification of the employed HDDs and SSDs is presented in Table 1. The code base for implementation of ReCA[1] is EnhanceIO which is an open source caching solution [17]. In all experiments, cache size is set to 20% of the total unique data pages in the workload, unless explicitly said otherwise. All architectures were given enough cache warm-up time to reach a stable state, roughly 10-20 minutes. ReCA has been compared with LARC algorithm which is a variation of LRU algorithm proposed specifically for SSD caching as opposed to many of the other LRU variations that have general purpose. This makes LARC best suited for comparing with ReCA. In addition to LARC, the proposed architecture is compared to an access frequency algorithm that counts the number of accesses to data pages. This algorithm represents the frequency algorithms while LARC represents the recency algorithms. Note that the

1. Source code of ReCA will be publicly available with the same license as EnhanceIO upon acceptance of the paper.

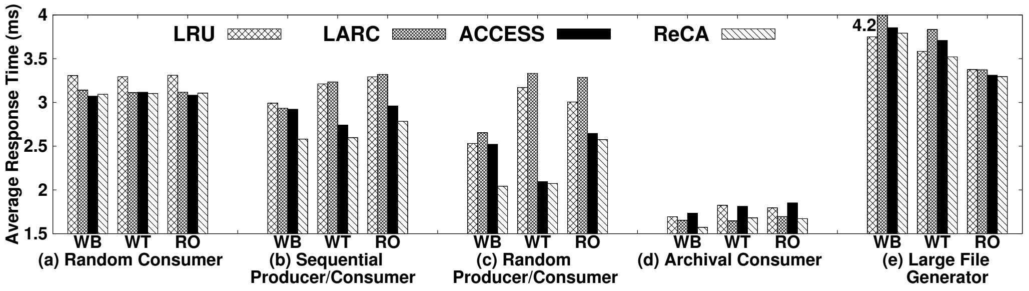


Fig. 13: Average Response Time in various Write Policies

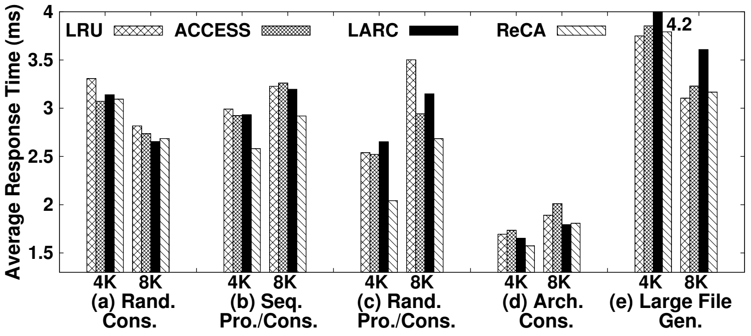


Fig. 14: Average Response Time of Various Cache Architectures for two Cache Line Sizes (4KB, 8KB)

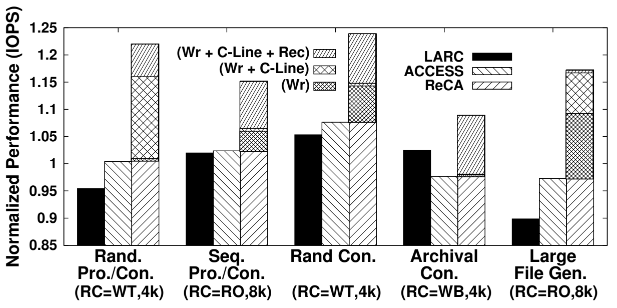


Fig. 15: Performance of Various Cache Architectures

access frequency algorithm counts the number of accesses to all data pages in the target address space which makes its memory overhead significantly higher than ReCA and LARC. To further show the efficiency of each algorithm, a baseline LRU algorithm is also considered in the experiments. Since the performance of SSDs varies as they age, SSDs are trimmed after each experiment using *blkdiscard* utility in Linux operating system. Due to the effect of request scheduler and garbage collection in SSDs, response time of requests might have considerable variations. To reduce the performance variations, experiments had long execution time so that SSD reaches a steady state with small variation in requests response time. By repeating the experiments, we have made sure that the variations have negligible effect on the results.

6.2 Experimental Results

Fig. 13 shows the average response time of caching architectures under various write policies and workload categories. ReCA outperforms LARC algorithm in almost all workload categories and write policies which approves the efficiency of the proposed algorithm for prioritizing data pages. The difference between performance of various write policies is not the same across workload categories. In *Random Consumers* category, the difference is negligible while *Large File Generators* category has significant performance gap between different write policies. Combination of this observation and the fact that write policies have different endurance and reliability costs (read-only extends cache lifetime while write-back shortens SSD lifetime but increases data loss probability), supports our claim that cache write policy should be adaptive.

Cache line size, similar to eviction and write policies, is a key factor in determining performance of a caching architecture. Fig. 14 depicts the effect of various cache line sizes on average response time of examined workloads where the cache line size of ReCA is set to fixed values instead of being reconfigurable to show the effect of using various cache line sizes on the performance. Larger cache line sizes are omitted from Fig. 14 for the sake of brevity. Although using smaller cache line sizes enables cache to more accurately select suitable data pages for caching, memory footprint increases as cache line size decreases. In addition, using larger cache line size improves performance in many workloads (Fig. 14a and Fig. 14e). One of the key observations in Fig. 14a is that despite being a random workload, 8KB cache line size outperforms 4KB cache line size in all caching architectures. Further investigations show that this behaviour is also observed by using 8KB as data block size by several applications in *Random Consumers* category. ReCA has higher performance compared to the other caching architectures in most of the configurations examined in Fig. 14. Reconfiguring cache line size in the runtime enables ReCA to maintain its high performance even in case of a change in workload pattern while other architectures only use a fixed cache line size which will be inefficient in various workloads.

Fig. 15 compares the performance of ReCA with all discussed optimizations along with performance of the other caching architectures, all normalized to the performance of LRU caching policy. The configuration selected for each workload category is also depicted in Fig. 15. As shown in this figure, ReCA outperforms LRU, LARC, and *Access* architectures in all workload categories by employing efficient policies and cache structures. The performance improvement is higher in random workloads (on average) which are the target categories for using caching architectures. ReCA improves performance in *Random Producers /Consumers* and *Large File Generators* category by 22% and 24%, respectively, compared to LARC which confirms the efficiency of ReCA even in sequential workloads. The performance gain achieved by ReCA is broken down to the employed optimizations to demonstrate the effect of each parameter on the overall performance. *Wr*, *C-Line*, and *Rec* denote write policy, cache line size, and reclaiming policy, respectively. Choosing *write-through* and *read-only* policies in ReCA removes the need for using mirrored configuration since no dirty data page exists in the cache. Using *read-only* policy also improves SSD lifetime by reducing the number of writes in SSD up to 33% (not shown in figures). The number of writes is obtained directly from real SSDs

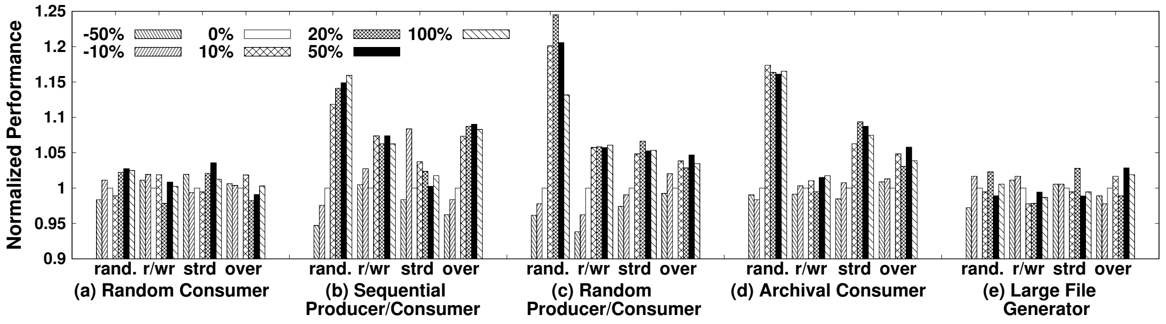


Fig. 16: Normalized Performance of Various Cache Parameter Priorities in ReCA

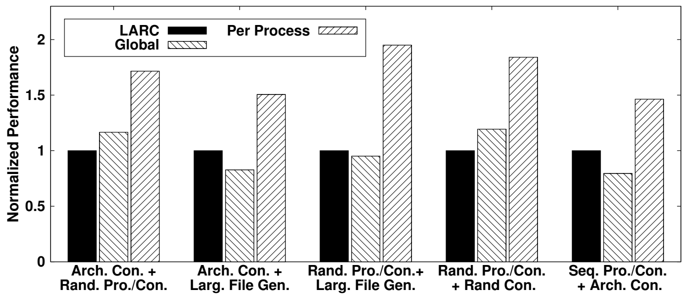


Fig. 17: Performance of *Global* and *per Process* Characterization in ReCA

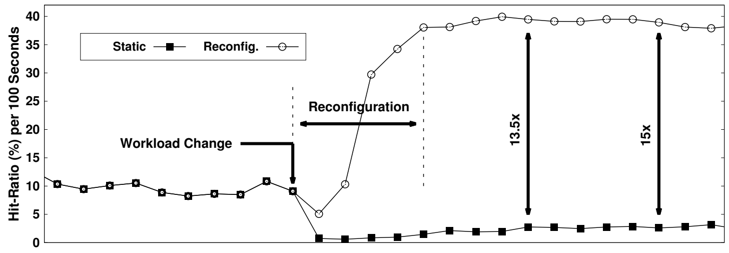


Fig. 18: Hit Ratio of ReCA *with* and *without* Reconfiguration

which shows the actual number of writes in flash chips including the effect of *Flash Translation Layer* (FTL) optimizations and write amplification factor.

FTL in SSDs tries to reduce the number of actual writes in SSD flash chips required to respond to user write requests. The average number of extra writes is called *write amplification factor* [57]. By sending less write requests to SSD, ReCA reduces FTL garbage collection overhead that increases performance. Additionally, FTL will have more clean data blocks to perform optimizations and decrease write amplification factor. Hence, issuing less writes by ReCA will significantly affect the SSD lifetime.

6.3 Parameter Sensitivity

The efficiency of selected parameters for prioritizing requests has been investigated in Fig. 16. The experiments have been conducted by both positive and negative values for parameters to fully explore the sensitivity of choosing right value for each parameter. For each workload category and parameter, a value with highest performance is selected. ReCA is configured with such values which enables it to simultaneously improve performance, lifetime, and cost (removing the need for mirrored SSDs).

6.4 Mixed Workloads

ReCA is able to characterize running applications separately to optimize itself based on application requirements. To show the effectiveness of this technique, applications from different categorizes are run simultaneously *with* and *without* separate characterization. Fig. 17 depicts the normalized performance of ReCA under many combinations of workloads compared to LARC. Configuring cache based on each application requirement can result in more than 2x performance compared to LARC. Additionally, *per process* caching has almost the same performance gain compared to *global* caching which emphasizes on the importance of separate characterization in ReCA. Although *Archival consumers* and *Sequential producers/consumers* are both sequential workloads, separating cache configurations results in 80% performance improvement which demonstrates that the cache configuration can significantly affect the performance in sequential workloads as well as random workloads.

The CPU overhead of *per process* caching is negligible since the CPU overhead is dependent to the number of incoming requests and remains the same in both global and per process characterization. However, there is slightly more CPU overhead in per process characterization due to managing extra data structures which is negligible in our experiments (less than 2% CPU usage). Although in per process caching characterization data structures are stored per process, they contribute to a small percentage of total ReCA memory usage. Most of the consumed memory is used for data page mapping in SSD which remains the same in both global and per process characterization. The actual increase in memory usage of per process characterization is about 5-10%.

6.5 Reconfiguration Overhead

The reconfiguration process consists of *(a)* reconfiguring the data structures and *(b)* eviction of data pages from cache. The first stage is relatively fast and can be done in less than 10 seconds without interfering with user requests. The second stage requires issuing I/O requests to both HDD and SSD. To decrease the impact of reconfiguration process on applications, a limit is put over the number of I/O requests issued for reconfiguration. To evaluate the efficiency of ReCA reconfiguration and its performance overhead, hit ratio of ReCA *with* and *without* reconfiguration upon changing the workload type is examined. In both scenarios, workload *fileserver* from *Sequential producer/consumer* category is running and cache configuration is optimized toward its requests. Workload type is changed into *exchange server* from *Random producer/consumer* category and changes in the hit ratio is observed. Fig. 18 depicts the hit ratio of ReCA *with* and *without* reconfiguration. The performance overhead of reconfiguration process is very small and cache reaches its optimal state in less than five minutes of changing workload type. Although static technique had optimal configuration in the first workload, due to the change in the workload type, it will be completely inefficient in the new workload. This shortcoming exists in all previous studies in I/O caching [9], [10], [11], [12], [33].

6.6 Tiering in ReCA

Tiering can outperform caching when the workload is steady and contains no sudden changes. This is due to the fixed intervals between migrations in tiering (ranges from hours to days), unlike caching which can demote a data page from cache or promote a data page to cache for each incoming requests. Therefore, if one does not expect running applications to change their I/O behavior, tiering is more suitable while if sudden changes are likely to happen such as in cloud environments, ReCA will be more efficient to work in caching mode.

7 Conclusion

In this paper, we first demonstrated that solely considering the request type is not effective in prioritizing requests for caching. Based on this observation, a comprehensive workload characterization is conducted to find an optimal cache configuration for each application type. In addition, an analysis has been done on filesystem metadata to consider such semantic information in the proposed architecture. To utilize workload characterization in caching, an online *Reconfigurable Cache Architecture* was proposed for storage systems that monitors incoming application I/Os and reconfigures cache when detecting a change in the running application phase. In addition, it is revealed that in many applications, mirrored cache requirement can be lifted without any reliability concern while maintaining the performance of application intact. The lifetime of SSDs used for caching also can be extended in many applications by changing the cache policy to *read-only* without significant degradation in performance. The experimental results showed that ReCA improves performance up to 24% (16% on average) and up to 33% lifetime improvement compared to previous studies while removing the need for mirrored SSDs in majority of the workloads.

References

[1] J. Axboe, “Cfq io scheduler,” in *presentation at linux. conf. au, Jan*, 2007.

[2] T. Y. Ts’o and S. Tweedie, “Planned extensions to the Linux EXT2/EXT3 filesystem,” in *Proceedings of the Freenix Track: USENIX Annual Technical Conference (ATC)*, 2002, pp. 235–244.

[3] J. Axboe, “Linux block IO present and future,” in *Ottawa Linux Symp*, 2004, pp. 51–61.

[4] T. Pritchett and M. Thottethodi, “SieveStore: A highly-selective, ensemble-level disk cache for cost-performance,” in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010, pp. 163–174.

[5] M. Saxena and M. M. Swift, “Design and prototype of a solid-state cache,” *Transactions on Storage (TOS)*, vol. 10, no. 3, pp. 1–34, Aug. 2014.

[6] J. C. McCallum, “Disk drive prices (1955-2015),” http://www.jcmit.com/diskprice.htm, 2015, accessed: 2016-08-13.

[7] PCPartPicker, LLC, https://pcpartpicker.com/.

[8] N. Megiddo and D. Modha, “Outperforming LRU with an adaptive replacement cache algorithm,” *Computer*, vol. 37, no. 4, pp. 58–65, April 2004.

[9] S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen, “Improving flash-based disk cache with lazy adaptive replacement,” *ACM Transactions on Storage (TOS)*, vol. 12, no. 2, pp. 8:1–8:24, 2016.

[10] R. Santana, S. Lyons, R. Koller, R. Rangaswami, and J. Liu, “To ARC or Not to ARC,” in *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2015, pp. 14–14.

[11] Y. Klonatos, T. Makatos, M. Marazakis, M. Flouris, and A. Bilas, “Azor: Using two-level block selection to improve SSD-based I/O caches,” in *Proceedings of the 6th IEEE International Conference on Networking, Architecture and Storage (NAS)*, Jul. 2011, pp. 309 –318.

[12] F. Chen, D. A. Koufaty, and X. Zhang, “Hystor: making the best use of solid state drives in high performance storage systems,” in *Proceedings of the 13th International Conference on Supercomputing (ICS)*, 2011, pp. 22–32.

[13] M. Mesnier, F. Chen, T. Luo, and J. B. Akers, “Differentiated storage services,” in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 57–70.

[14] T. Luo, R. Lee, M. Mesnier, F. Chen, and X. Zhang, “hStorage-DB: heterogeneity-aware data management to exploit the full capability of hybrid storage systems,” *VLDB Endowment*, vol. 5, no. 10, pp. 1076–1087, Jun. 2012.

[15] R. Salkhordeh, H. Asadi, and S. Ebrahimi, “Operating system level data tiering using online workload characterization,” *The Journal of Supercomputing*, vol. 71, no. 4, pp. 1534–1562, 2015.

[16] F. Ye, J. Chen, X. Fang, J. Li, and D. Feng, “A regional popularity-aware cache replacement algorithm to improve the performance and lifetime of SSD-based disk cache,” in *IEEE International Conference on Networking, Architecture and Storage (NAS)*, Aug 2015, pp. 45–53.

[17] STEC, “Enhanceio ssd caching software,” https://github.com/stec-inc/EnhanceIO.

[18] A. Wilson, “The new and improved filebench,” in *Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[19] J. Axboe, “Fio-flexible io tester,” http://freecode. com/projects/fio, 2008.

[20] J. Katcher, “Postmark: A new filesystem benchmark,” Network Appliance, Tech. Rep. TR3022, Oct. 1997.

[21] S. Shaw, *HammerDB: the open source oracle load test tool*, 2012, accessed: 2015-08-10.

[22] B. Debnath, S. Subramanya, D. Du, and D. Lilja, “Large block CLOCK (LB-CLOCK): A write caching algorithm for solid state disks,” in *Proceedings of the IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sept 2009, pp. 1–9.

[23] M. Saxena, M. M. Swift, and Y. Zhang, “FlashTier: A lightweight, consistent and durable storage cache,” in *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012, pp. 267–280.

[24] J. Chen, Q. Wei, C. Chen, and L. Wu, “FSMAC: A file system metadata accelerator with non-volatile memory,” in *Proceedings of the 29th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2013, pp. 1–11.

[25] X. Wu and A. Reddy, “Exploiting concurrency to improve latency and throughput in a hybrid storage system,” in *Proceedings of the IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, Aug 2010, pp. 14–23.

[26] R. Appuswamy, D. van Moolenbroek, and A. Tanenbaum, “Integrating flash-based SSDs into the storage stack,” in *Proceedings of the 28th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, Apr. 2012, pp. 1 –12.

[27] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu, “vCacheShare: Automated server flash cache space management in a virtualization environment,” in *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2014, pp. 133–144.

[28] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, “Adaptive insertion policies for high performance caching,” in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007, pp. 381–391.

[29] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, “The HP AutoRAID hierarchical storage system,” *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 108–136, 1996.

[30] A. Miranda and T. Cortes, “CRAID: Online RAID upgrades using dynamic hot data reorganization,” in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014, pp. 133–146.

[31] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, “BORG: Block-reorganization for self-optimizing storage systems,” in *Proccedings of the 7th Conference on File and Storage Technologies (FAST)*, 2009, pp. 183–196.

[32] X. Wu and A. L. N. Reddy, “Managing storage space in a flash and disk hybrid storage system,” in *IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sept 2009, pp. 1–4.

[33] S. Liu, J. Jiang, and G. Yang, “Macss: A metadata-aware combo storage system,” in *Proceedings of the International Conference on Systems and Informatics (ICSAI)*, May 2012, pp. 919 –923.

[34] C.-K. Kang, Y.-J. Cai, C.-H. Wu, and P.-C. Hsiu, “A hybrid storage access framework for high-performance virtual machines,” *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 5s, pp. 157:1–157:24, Oct. 2014.

[35] Y. Kim, A. Gupta, B. Urgaonkar, P. Berman, and A. Sivasubramaniam, “HybridStore: A cost-efficient, high-performance storage system combining SSDs and HDDs,” in *Proceedings of the 19th IEEE International*

Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), Jul. 2011, pp. 227 – 236.

[36] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, "Software-defined caching: Managing caches in multi-tenant data centers," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 174–181.

[37] E. Lee, Y. Oh, and D. Lee, "SSD caching to overcome small write problem of disk-based RAID in enterprise environments," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*, 2015, pp. 2047–2053.

[38] C. Li, D. Feng, Y. Hua, and F. Wang, "Improving RAID performance using an endurable ssd cache," in *45th International Conference on Parallel Processing (ICPP)*, Aug 2016, pp. 396–405.

[39] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random write considered harmful in solid state drives," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012, pp. 12–12.

[40] B. Mao, H. Jiang, S. Wu, L. Tian, D. Feng, J. Chen, and L. Zeng, "HPDA: A hybrid parity-based disk array for enhanced performance and reliability," *ACM Transactions on Storage (TOS)*, vol. 8, no. 1, pp. 4:1–4:20, 2012.

[41] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013, pp. 45–58.

[42] S. Byan, J. Lentini, A. Madan, and L. Pabn, "Mercury: Host-side flash caching for the data center," in *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, April 2012, pp. 1–12.

[43] R. Koller, A. J. Mashtizadeh, and R. Rangaswami, "Centaur: Host-side ssd caching for storage performance control," in *IEEE International Conference on Autonomic Computing (ICAC)*, July 2015, pp. 51–60.

[44] Y. Chai, Z. Du, X. Qin, and D. Bader, "WEC: Improving durability of ssd cache drives by caching write-efficient data," *IEEE Transactions on Computers (TC)*, vol. PP, no. 99, pp. 1–1, 2015.

[45] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, "Cost effective storage using extent based dynamic tiering," in *Proceedings of the 9th USENIX Conference on File And Stroage Technologies (FAST)*, 2011, pp. 20–20.

[46] E. Kakoulli and H. Herodotou, "OctopusFS: A distributed file system with tiered storage management," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2017, pp. 65–78.

[47] K. Oh, A. Chandra, and J. Weissman, "TripS: Automated multi-tiered data placement in a geo-distributed cloud environment," in *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017, pp. 12:1–12:11.

[48] M. Abd-El-Malek, W. V. Courtright, II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie, "Ursa minor: Versatile cluster-based storage," in *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies (FAST)*, 2005, pp. 5–5.

[49] M. Tarihi, H. Asadi, A. Haghdoost, M. Arjomand, and H. Sarbazi-Azad, "A hybrid non-volatile cache design for solid-state drives using comprehensive I/O characterization," *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1678–1691, 2016.

[50] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2008, pp. 119–128.

[51] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proceedings of the 11th USENIX Conference on Annual Technical Conference (ATC)*, 2000, pp. 4–4.

[52] IOMETER, Team, "Iometer: I/o subsystem measurement and characterization tool," *Open source code distribution: http://www. iometer.org*, 1997.

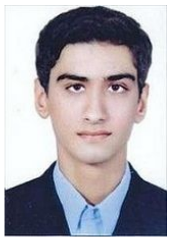
[53] Storage Networking Industry Association, "Microsoft enterprise traces," http://iotta.snia.org/traces/130, accessed: 2015-08-10.

[54] *TPC benchmark C*, 5th ed., Transaction Processing Performance Council (TPC), Feb. 2010, accessed: 2015-08-10.

[55] *TPC benchmark H (Decision Support)*, 2nd ed., Transaction Processing Performance Council (TPC), Sep. 2008, accessed: 2015-08-10.

[56] A. Wilson, "The new and improved filebench," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[57] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of International Systems and Storage Conference (SYSTOR)*, 2009, pp. 10:1–10:9.



Reza Salkhordeh received the B.S. degree in computer engineering from Ferdowsi University of Mashhad in 2011, and M.S. degree in computer engineering from Sharif University of Technology (SUT) in 2013. He is currently a Ph.D. candidate at SUT. His research interests include operating systems, solid-state drives, and data storage systems.

Shahriar Ebrahimi Shahriar Ebrahimi received the B.S. degree in computer engineering from Sharif University of Technology (SUT) in 2015. He is currently a M.S. student in computer architecture engineering in SUT. His research interests include storage systems, computer architecture, reconfigurable systems, and application of machine learning in storage systems.

Hossein Asadi (M'08, SM'14) received the B.S. and M.S. degrees in computer engineering from the Sharif University of Technology (SUT), Tehran, Iran, in 2000 and 2002, respectively, and the Ph.D. degree in electrical and computer engineering from Northeastern University, Boston, MA, USA, in 2007. He was with EMC Corporation, Hopkinton, MA, USA, as a Research Scientist and Senior Hardware Engineer, from 2006 to 2009. From 2002 to 2003, he was a member of the Dependable Systems Laboratory, SUT, where he researched hardware verification techniques. From 2001 to 2002, he was a member of the Sharif Rescue Robots Group. He has been with the Department of Computer Engineering, SUT, since 2009, where he is currently a tenured Associate Professor. He is the Founder and Director of the *Data Storage, Network, and Processing* (DSN) Laboratory and the director of High-Performance Computing Center (HPC), and Information Technology Center (ITC) at SUT. He spent three months in the summer 2015 as a Visiting Professor at the the School of Computer and Communication Sciences at the Ecole Poly-technique Federele de Lausanne (EPFL). He has also co-founded the first startup company in the Middle East, called HPDS, designing and fabricating midrange and high-end data storage systems. He has authored and co-authored more than sixty technical papers in reputed journals and conference proceedings. His current research interests include data storage systems and networks, solid-state drives, operating system support for I/O and memory management, and reconfigurable and dependable computing. Dr. Asadi was a recipient of the Technical Award for the Best Robot Design from the International RoboCup Rescue Competition, organized by AAAI and RoboCup, a recipient of Best Paper Award at the 15th CSI Internation Symposium on Computer Architecture and Digital Systems (CADS), the Distinguished Lecturer Award from SUT in 2010, the Distinguished Researcher Award from SUT in 2016, and the Distinguished Research Institute Award from SUT in 2016. He is also recipient of Extraordinary Ability in Science visa from US Citizenship and Immigration Services in 2008. He has also served as the publication chair of several national and international conferences including CNDS2013, AISP2013, and CSSE2013 during the past three years. Most recently, he has served as a Guest Editor of IEEE Transactions on Computers in 2016, a Program Co-Chair of the 18th International Symposium on Computer Architecture & Digital Systems (CADS2015), and the Program Chair of the 22nd National CSI conference on Computer (CSICC2017).