ECI-Cache: A High-Endurance and Cost-Efficient I/O Caching Scheme for Virtualized Platforms

Saba Ahmadian
Sharif University of Technology
Tehran, Iran
ahmadian@ce.sharif.edu

Onur Mutlu
ETH Zürich
Switzerland
omutlu@ethz.ch

Hossein Asadi*
Sharif University of Technology
Tehran, Iran
asadi@sharif.edu

ABSTRACT

In recent years, high interest in using *Virtual Machines* (VMs) in data centers and cloud computing has significantly increased the demand for high-performance data storage systems. A straightforward approach to provide a high performance storage system is using *Solid-State Drives* (SSDs). Inclusion of SSDs in storage systems, however, imposes significantly higher cost compared to *Hard Disk Drives* (HDDs). Recent studies suggest using SSDs as a caching layer for HDD-based storage subsystems in virtualized platforms. Such studies neglect to address the endurance and cost of SSDs, which can significantly affect the efficiency of I/O caching. Moreover, previous studies *only* configure the cache size to provide the required performance level for each VM, while neglecting other important parameters such as *write policy* and *request type*, which can adversely affect both performance-per-cost and endurance.

In this paper, we present a *high-Endurance and Cost-efficient I/O Caching* (ECI-Cache) scheme for virtualized platforms, which can significantly improve both the *performance-per-cost* and *endurance* of storage subsystems as opposed to previously proposed I/O caching schemes. Unlike traditional I/O caching schemes which allocate cache size *only* based on *reuse distance* of accesses, we propose a new metric, *Useful Reuse Distance* (URD), which considers the request *type* in reuse distance calculation, resulting in improved performance-per-cost and endurance of the SSD cache. By online characterization of workloads and using URD, ECI-Cache partitions the SSD cache across VMs and is able to dynamically adjust the cache size and write policy for each VM. To evaluate the proposed scheme, we have implemented ECI-Cache in an open source hypervisor, QEMU (version 2.8.0), on a server running the CentOS 7 operating system (kernel version 3.10.0-327). Experimental results show that our proposed scheme improves the performance-per-cost and endurance of the SSD cache by 30% and 65% compared to the state-of-the-art *dynamic* cache partitioning scheme, respectively.

ACM Reference Format:
Saba Ahmadian, Onur Mutlu, and Hossein Asadi. 2018. ECI-Cache: A High-Endurance and Cost-Efficient I/O Caching Scheme for Virtualized Platforms. In *Proceedings of 18 (SIGMETRICS)*. ACM, Irvine, California, USA, 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Virtualization is widely used in data centers and cloud computing in order to improve the utilization of high-performance servers [48]. Integrating various *Virtual Machines* (VMs) running with different operating systems on a server provides more flexibility and higher resource utilization while delivering the desired performance for each VM. In addition, virtualization provides system isolation where each VM has access only to its own resources. In a virtualized platform, shown in Fig. 1, the resource allocation of each VM is managed by a hypervisor. By employing various modules such as a VM scheduler and a memory and network manager, the hypervisor orchestrates the sharing of resources between VMs according to their demand, in order to maximize the overall performance provided by the server (and this maximizes performance-per-cost by enabling the use of a smaller number of physical servers than one for each VM) [51, 53].

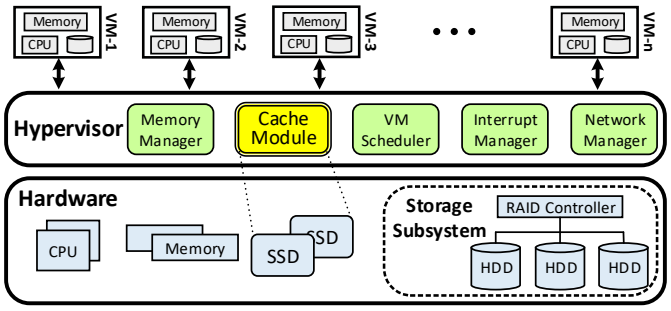


Figure 1: Example state-of-the-art virtualization platform.

With increasing performance requirements of data-intensive applications in data centers, storage subsystems have become performance bottlenecks of computing systems. *Hard Disk Drives* (HDDs), which are used as main media for data storage in storage systems, provide large capacity and low cost, but they suffer from low performance, particularly for random access workloads. The low performance of HDD-based storage systems can be avoided by employing high-performance storage devices such as *Solid-State Drives* (SSDs). Compared to HDDs, SSDs provide higher performance due to their non-mechanical structure used to retrieve and store data. SSDs, however, impose up to 10X higher cost and support only a limited number of reliable writes [10], which makes the replacement of all HDDs by SSDs usually prohibitively expensive [1, 2, 37, 41].

In order to take advantage of the merits of both HDDs and SSDs, several studies from EMC^2, Facebook, FusionIO, Mercury, and VMware [4, 9, 13, 16, 19, 25, 26, 28, 30, 33, 36, 37, 42, 49] employ high-performance SSDs as a caching layer for high-capacity HDDs in storage systems (as shown in Fig. 1). Applying such I/O caching on virtualization platforms requires a proper cache management scheme in order to achieve higher I/O performance. In previous studies, such caches have been used as either a shared global cache [16, 19, 30] or a cache statically partitioned across VMs [9]. The former scheme fails to provide a guaranteed minimum cache space for each VM. This is due to the fact that the

*Corresponding Author

SIGMETRICS, ACM, 2018

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

entire cache is shared between all VMs and each VM can potentially use up an unbounded amount of the entire cache space, affecting the performance of the other VMs. The latter scheme *statically* partitions SSD space between VMs where the partitioning is performed *independently of* the characteristics of the workloads running on VMs. This scheme has two major shortcomings. First, the allocated cache space could be underutilized by the VM if there is low locality of reference in the workload access pattern. Second, since cache space partitioning and allocation are done statically offline, cache allocation for a new VM during runtime is *not* practical using this scheme, which makes the scheme inflexible and the system underutilized. To alleviate the shortcomings of the two aforementioned schemes, *partitioned I/O caching* has been proposed. Variants of this technique dynamically estimate and allocate cache space for each VM by estimating an efficient cache size for each VM [6, 27, 35]. They do so by calculating the reuse distance of the workload, i.e., the maximum distance between two accesses to an identical address [27, 38, 55] . Unfortunately, such I/O caching schemes only focus on estimating cache size for VMs and neglect other key parameters, such as *write policy* (i.e., how write requests are handled by the cache), *request type* (i.e., read or write requests), and their corresponding impact on the workload *reuse distance*, which greatly affects the performance-per-cost and endurance of the SSD cache, as we show in this work.

In this paper, we propose a new *high-Endurance and Cost-efficient I/O caching* (ECI-Cache) scheme which can be used for virtualized platforms in large-scale data centers. ECI-Cache aims to improve both the performance-per-cost and endurance of the SSD cache by dynamically configuring 1) an efficient cache size to maximize the performance of the VMs and 2) an effective write policy that improves the endurance and performance-per-cost of each VM. To this end, we propose a metric called *Useful Reuse Distance* (URD), which minimizes the cache space to allocate for each VM while maintaining the performance of the VM. The main objective of URD is to reduce the allocated cache space for each VM. The reduced cache space is obtained by computing workloads reuse distance based on request type, *without* considering unnecessary write accesses (i.e., writes to a block without any further read access). Employing URD in our proposed I/O caching scheme maximizes the performance-per-cost and also enhances the endurance of the SSD cache by allocating much smaller cache space compared to state-of-the-art cache partitioning schemes. We also propose a detailed analysis of the effect of write policy on the performance and endurance of an SSD cache, clearly demonstrating the negative impact of having the same write policy for VMs with *different* access patterns (as used in previous studies) on the IO performance and SSD endurance. To achieve a sufficiently high hit ratio, ECI-Cache *dynamically* partitions the cache across VMs.

In the proposed scheme, we mainly focus on two approaches: 1) URD based per-VM cache size estimation and 2) per-VM effective write policy assignment, via online monitoring and analysis of IO requests for each VM. In the first approach, we allocate much smaller cache space compared to previous studies for each VM, which results in improved performance-per-cost. In the second approach, we assign an effective write policy for each VM in order to improve the endurance of the I/O cache while minimizing the negative impact on performance. The integration of these two approaches enables ECI-Cache to partition and manage the SSD cache between VMs more effectively than prior mechanisms [6, 27, 35].

We have implemented ECI-Cache on QEMU (version 2.8.0) [39], an open source hypervisor (on the CentOS 7 operating system, kernel version 3.10.0-327). We evaluate our scheme on an HP ProLiant DL380 Generation 5 (G5) server [20] with four 146GB SAS 10K HP HDDs [21] (in RAID-5 configuration), a 128GB Samsung 850 Pro SSD [44], 16GB DDR2 memory, and 8 x 1.6GHz Intel(R) Xeon CPUs. We run more than fifteen workloads from the SNIA MSR traces [47] on VMs. Experimental results show that ECI-Cache 1) improves performance by 17% and performance-per-cost by 30% compared to the state-of-the-art cache partitioning scheme [27], 2) reduces the number of writes committed to the SSD by 65% compared to [27], thereby greatly improving the SSD lifetime.

To our knowledge, we make the following contributions.

- This paper is the first to differentiate the concept of reuse distance based on the *type of each request*. We propose a new metric, *Useful Reuse Distance* (URD), whose goal is to reduce the cost of the SSD cache by allocating a smaller cache size for each VM.
- By conducting extensive workload analyses, we demonstrate the importance of dynamically adjusting the cache *write policy* on a per-VM basis, which no previous I/O caching policy explicitly takes into account. We use these analyses to develop a mechanism that can efficiently adjust both cache size and write policy on a per-VM basis.
- We propose ECI-Cache, which consists of two key novel components: 1) dynamic per-VM cache size estimation, and cache partitioning using the URD metric and 2) per-VM write policy to improve both system performance-per-cost and SSD cache lifetime.
- We implement ECI-Cache in QEMU, an open source hypervisor. Our extensive evaluations of ECI-Cache on a large number of diverse workloads show that ECI-Cache significantly improves performance-per-cost over the best previous dynamic cache partitioning policy and reduces the number of writes to the SSD.

The rest of the paper is organized as follows. Sec. 2 discusses related work. In Sec. 3, we provide an illustrative example and motivation. In Sec. 4, we propose the metric of URD. In Sec. 5, we present our proposed technique. Sec. 6 provides experimental setup and results. Finally, Sec. 7 concludes the paper.

2 RELATED WORK

Previous studies on I/O caching in virtualization platforms investigate 1) the location of the cache or 2) the cache partitioning policy. The former set of works explores caching techniques based on where the I/O cache resides, whereas the latter examines mechanisms for sharing and partitioning of the cache across VMs. Based on the location of the SSD cache, three main alternatives for I/O caching in virtualization platforms have been introduced, as shown in Fig. 2. We next describe possible schemes for I/O caching and discuss their advantages and shortcomings.

2.1 VM-based I/O Caching

In *VM-based I/O caching* (Fig. 2a), each VM in a virtualized platform has full control on a portion of the SSD cache. In this scheme, separate SSD slots are allocated for each VM and the cache management is conducted by VMs. Cache size adjustment, cache partitioning, and sharing techniques cannot be applied in this scheme. In order to employ a *VM-based I/O caching* scheme in a virtualized platform, caching schemes presented in [13, 28, 37, 42] can

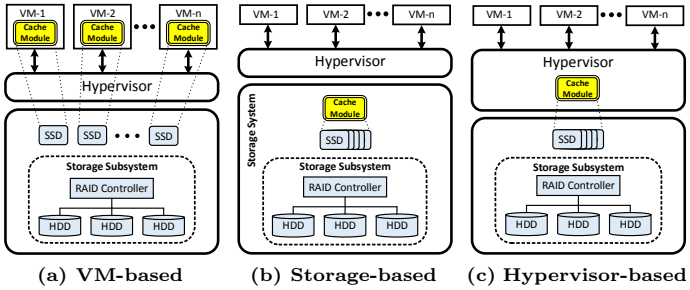


Figure 2: I/O caching using SSDs in virtualized platforms.

be applied on VMs and improve the IO performance of individual VMs, which is likely not efficient in virtualized platforms.

2.2 Storage system-based I/O Caching

In *Storage system-based I/O caching* (Fig. 2b), VMs and the hypervisor have no knowledge of the SSD cache, which prohibits the advantages achieved by cache partitioning schemes. Similar to *VM-based I/O caching*, previous caching techniques, such as [13, 28, 37, 42], can be employed in storage systems, but such techniques cannot be managed in a virtualized platform.

2.3 Hypervisor-based I/O Caching

In *Hypervisor-based I/O caching* (Fig. 2c), cache management is done by the hypervisor. Since the hypervisor has full knowledge about workloads running on VMs, it can perform efficient cache management and cache space partitioning across VMs. This type of I/O caching scheme has been proposed frequently in previous studies [6, 27, 35]. These works mainly focus on cache management, sharing, and partitioning across VMs. From the partitioning perspective, *Hypervisor-based I/O caching* schemes can be divided into two groups: global/static and dynamic cache partitioning. We next describe state-of-the-art *hypervisor-based I/O caching* schemes.

2.3.1 Global Caching and Static Cache Partitioning. Examples of global caching or static cache partitioning schemes include the EMC VFCache [16], NetApp Mercury [9], Fusion-io ioTurbine [19], and vFRM [30]. In global caching, each VM can use up the *entire* SSD cache, thereby potentially adversely affecting the performance of the other VMs [6]. In static cache partitioning, SSD cache space is *equally* partitioned across VMs based on the number of VMs in the platform, without taking into account the data access and reuse patterns of each VM. Static cache partitioning is also unable to allocate cache space for newly-added VMs during online operation.

2.3.2 Dynamic Cache Partitioning. Dynamic cache partitioning schemes alleviate the shortcoming of global and static cache partitioning [3, 5, 6, 27, 35, 54]. These techniques partition the SSD cache across VMs based on the cache space demand of each VM, and they are aware of data access and reuse pattern of VMs. Argon [54] presents a storage server that partitions the memory cache across services based on their access patterns. This scheme allocates minimum cache space for each service to achieve a predefined fraction of hit ratio namely *R-Value*. To estimate the required cache size for each service, Argon employs an online cache simulation mechanism and finds the fraction of accesses that are served by cache (namely I/O absorption ratio). Janus [3] partitions the flash tier between workloads at the filesystem level. Janus maximizes the total read accesses served from the flash tier by allocating the required space for each workload. The required space is estimated based on the ratio of read operations of each workload. S-CAVE [6] is a hypervisor-based online I/O caching scheme that allocates cache space for each VM by dynamically estimating the working set size of workloads running on VMs. To minimize the possibility of data loss, S-CAVE uses the *Write-Through* (WT) policy in cache configuration. vCacheShare [35] is another hypervisor-based I/O caching scheme that dynamically partitions the SSD cache space across VMs. This scheme considers locality and reuse intensity (i.e., burstiness of cache hits) in order to estimate the required cache size for each VM. vCacheShare reduces the number of writes in the SSD cache by using the *Write Around* policy where write operations are *always* directed to the storage subsystem. Such scheme shows improved performance only for read operations while it has no performance improvement for write operations. Centaur [27], another online partitioning scheme for virtualized platforms, aims to maximize the IO performance of each VM as well as meeting QoS targets in the system. It employs *Miss Ratio Curves* (MRCs) to estimate an efficient cache space allocation for each VM. Centaur does *not* consider the negative impact of write operations on SSD cache lifetime in 1) cache size estimation, or 2) write policy assignment. Centaur employs the *Write-Back* (WB) policy to maximize the IO performance without considering the impact of the WB policy on the number of writes into the SSD. CloudCache [5] estimates each VM's cache size by considering *Reuse Working Set Size* (RWSS), which captures temporal locality and also reduces the number of writes into the SSD cache. In addition, this scheme employs a VM migration mechanism to handle the performance demands of VMs in the Cloud.

To summarize, among previous studies, S-CAVE [6], vCacheShare [35], Centaur [27], and CloudCache [5] are the closest to our proposal. However, they only consider cache space partitioning and do not consider adaptive write policies. The cache size estimation scheme presented in S-CAVE, which is based on working set size estimation fails in cache size estimation for workloads with sequential access patterns and has become deprecated, as shown in [35]. vCacheShare and CloudCache perform cache size estimation based on reuse intensity. Such cache allocation schemes are based on assumptions that cannot be applied to the I/O cache in the storage subsystem, as demonstrated in [27]. Reuse intensity based schemes are *only* effective for workloads that are aligned with their size estimation schemes and would not be accurate compared to reuse distance based schemes such as [27]. The state-of-the-art scheme is Centaur, which works based on MRCs and reuse distance analysis. This scheme does *not* consider 1) the impact of request type on reuse distance calculation and 2) the impact of write policy on either endurance or performance.

3 MOTIVATION AND ILLUSTRATIVE EXAMPLE

The main purpose of employing a high-performance cache layer is to reduce the number of accesses to the disk subsystem. Theoretically, an ideal SSD cache layer would provide access latency equal to the SSD device access latency. However, due to limited SSD size and imperfect write policy, the observed average latency of accesses to the SSD cache is much higher than the SSD device latency. For example, we find that the access latency of the caching

technique presented in [28] is 50X higher than the raw access latency of the employed SSD device, as shown in Fig. 3.[1]

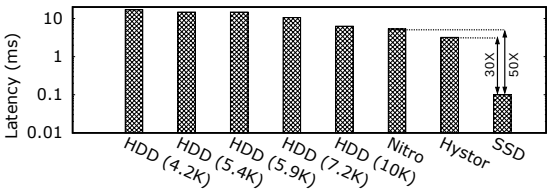


Figure 3: HDD and SSD access latency vs. I/O caching [13, 28].

The major parameters affecting IO cache performance are cache size, write policy, and replacement policy. In this work, we mainly study the effect of cache size and write policy on the performance and endurance of an SSD cache. A commonly-used metric to compare the performance of different cache configurations is cache *hit ratio*. To maximize the cache hit ratio, the cache size should be large enough to house the working set of the running workload. Similar to size, write policy can also affect the performance of the cache particularly for write-intensive workloads. There are three major write policies that are fundamentally different: 1) *Write-Back* (WB), 2) *Write-Through* (WT), and 3) *Read-Only* (RO). WB reduces the number of accesses to the storage subsystem by buffering temporal writes in the SSD cache and writing them back to the storage subsystem only after the buffered dirty blocks are evicted. WB can improve the performance of both read and write operations, but it suffers from low reliability since the SSD cache can become a single point of failure (i.e., buffered writes can get lost before being propagated to the storage subsystem, if the SSD fails). WT policy buffers write operations but also transfers them to the storage subsystem at the same time. This policy improves the performance of *only* read operations but provides a higher level of reliability by assuming that each written block is propagated immediately to the storage system. RO caches only the read operations and sends write operations directly to the storage subsystem without buffering them. Therefore, it is not able to improve the performance of write operations but it keeps them reliable. We conduct experiments to show the impact of cache size and cache write policy on IO performance and SSD endurance. To this end, we perform several experiments on a system with a 200GB HDD and 30GB SSD (our experimental setup is reported in Table 1). We employ EnhanceIO [17] as an open source SSD caching scheme in the experiments. To investigate the impact of write policy on performance and endurance, we run 30 workloads from Filebench on a fixed-size SSD cache with both the WB and RO policies. We omit the results of the WT policy, since WT has the same endurance as WB and provides less performance than WB.

Fig. 4 shows the impact of write policy on both *Bandwidth* (i.e., the amount of data that is transmitted for a workload in one second) and *I/O Per Second (IOPS)* of eight sample workloads (Fig. 4a through Fig. 4h). We make five major observations: 1) the SSD cache has 2.4X performance improvement with the WB policy in the *Fileserver* workload where the RO cache has only 1.6X improvement on the IO performance of this workload (Fig. 4a). 2) WB policy improves the IO performance of *RandomRW* and *Varmail* workloads, both over no caching (Fig. 4b and Fig. 4c). 3) *Webserver* and *Webproxy* workloads achieve good and similar performance with both the WB and RO write policies (Fig. 4d and Fig. 4f). 4) Employing the SSD cache has negative impact on the performance of the *CopyFiles* workload (Fig. 4e). 5) The RO write policy can significantly improve the performance of *Mongo* and *SingleStreamRead* workloads by 20% and 48%, respectively (Fig. 4g and Fig. 4h).

[1]All numbers in Fig. 3 are based on the results reported in [13, 28].

Table 1: Setup of the motivational experiments.

HW/SW	Description
Server	HP Proliant DL380 G5
CPU	8x 1.6GHz Intel(R) Xeon
Memory	16 GB DDR2, Configured Clock Speed: 1600 MHz
HDD	438GB: four SAS 10K HP HDDs in RAID5 (partition size = 200GB)
SSD	128GB Samsung 850 Pro (partition size = 30GB)
OS	Centos 7 (Kernel version: 3.10.0-327)
Filesystem	ext3 (Buffer cache is disabled) [52]

Our main experimental conclusions are as follows:

(1) In workloads such as *Fileserver*, *Varmail*, and *SingleStreamRead*, only a specific cache configuration can improve IO performance. About 45% of the workloads prefer the WB policy and about 33% of workloads prefer the RO policy. Hence, it is necessary to employ a *workload-aware write policy* for the IO cache. Random-access and write-intensive workloads prefer WB while random-access and read-intensive workloads can be satisfied with the RO policy.

(2) In 20% of workloads, e.g., *Webserver* and *Webproxy*, both WB and RO write policies result in similar improvements. Hence, we can employ RO instead of WB in order to reduce the number of writes to the SSD cache. Such workloads that are random and read-intensive do not take advantage of buffering writes in the SSD cache.

(3) In workloads such as *Mongo* and *CopyFiles*, the SSD cache provides little performance improvement. Hence, one can allocate SSD cache space to other workloads that can benefit more from the available cache space. One can prevent the allocation of cache space for the workloads that do *not* benefit from allocated cache, and hence reduce the number of unnecessary writes into the SSD cache.

4 USEFUL REUSE DISTANCE

Traditional Reuse Distance (TRD) schemes [7, 14, 15, 18, 34, 38, 45, 55, 56] work based on the addresses of requests *without* considering the types of the requests. We provide examples demonstrating the benefit of considering the *request type* of the workloads in the calculation of reuse distance. The main objective of this analysis is to present the metric of *Useful Reuse Distance* (URD), which enables assigning smaller amounts of cache space to the VMs while preserving I/O performance.

We examine a sample workload (shown in Fig. 5) and show how reuse distance analysis assigns cache size for the workload in two cases: 1) without considering request type (TRD) and 2) considering request type (URD). In the sample workload given in Fig. 5a, the maximum reuse distance is due to the access of Req_7 to the second sector which was previously (five requests before) accessed by Req_2. Hence, the maximum TRD of the workload is equal to 4, and according to TRD, we should assign cache space equal to 5 blocks in order to maximize the hit ratio of this workload. Fig. 5a also shows the contents of the allocated cache to this workload based on TRD. It can be seen that when we allocate cache space based on TRD, we reserve one block of cache (Block 2) to keep

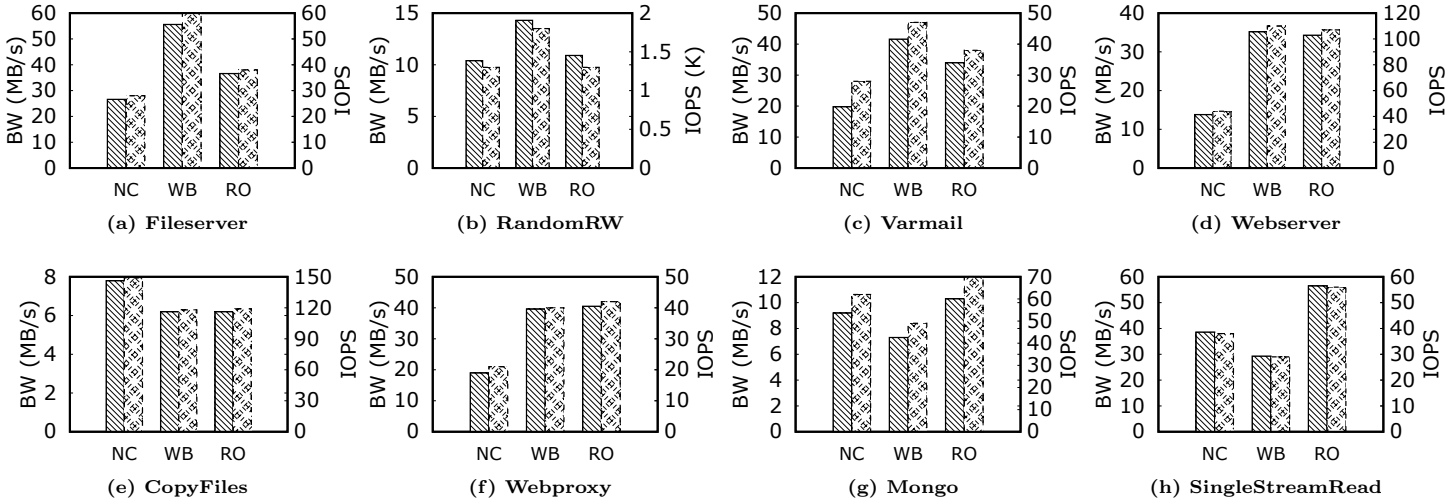


Figure 4: Impact of write policy on the performance of workloads (NC: No Cache, WB: Write Back, RO: Read Only).

data that will be written (modified) by the next request (Req_7) without any read access to the block.

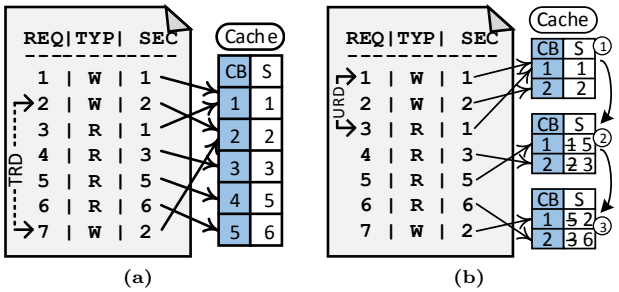


Figure 5: Comparison of cache size allocation (a) without and (b) with considering request type (REQ: Request, TYP: Type, SEC: Sector, W: Write, R: Read, and CB: Cache Block).

Here we classify the sequence of accesses in four groups based on their type (illustrated in Fig. 6): 1) *Read After Read* (RAR), 2) *Write After Read* (WAR), 3) *Read After Write* (RAW), and 4) *Write After Write* (WAW). We show how data blocks of such accesses are stored in both the WB and WT caches (i.e., allocate on write). Fig. 7 shows the operation of the cache for both read and write requests [23]. The operation of cache is defined for two cases: 1) for read accesses, if the data is found in the cache we read data from cache. Otherwise, the data is read from the disk subsystem and is stored in the cache for further access. 2) Write operations are directly written to the cache and may modify the previously written data in the cache. We define cache hit only for read requests.

(1) RAR: In the first read access, a cache miss fetches data from HDD to the cache. The second access reads the data from the cache. In this case, caching the data block of the first access improves the hit ratio at the cost of imposing one write access to the SSD.

(2) WAR: The first read access leads to fetching the data from HDD to the cache. The second access modifies the data in the cache *without* any read access to this block. In this case, caching the data block of the first read access does *not* improve the hit ratio but it comes at the cost of *two writes* into the SSD.

(3) RAW: The first access writes the data to the cache. The second access reads the data from the cache. In this case, caching the data block of the first access increases the hit ratio of the cache at the cost of imposing one write access into the SSD.

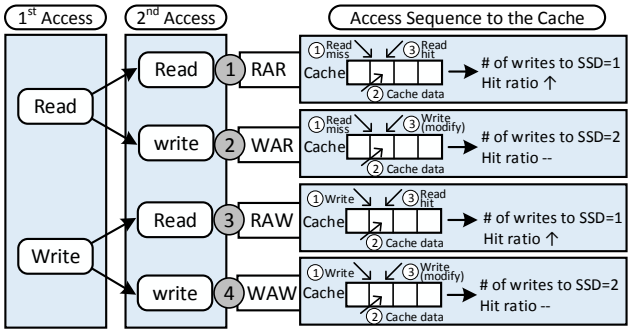


Figure 6: I/O access sequences.

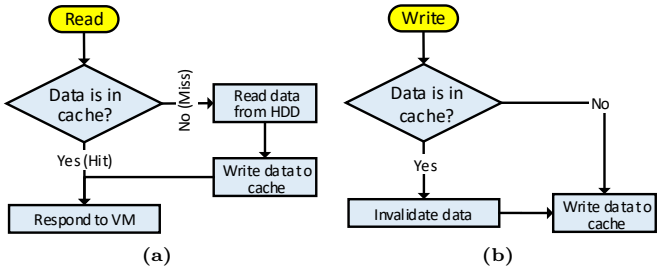


Figure 7: Flowchart of I/O cache access for a (a) Read or (b) Write.

(4) WAW: The first access writes the data to the cache. The second access modifies the data *without* any read access. In this case, caching the data block of the first access does *not* improve the hit ratio but comes at the cost of *two writes* to the cache.

We now show how we can allocate a smaller cache size to the sample workload (shown in Fig. 5b) by distinguishing between the four different types of access patterns we just described. We call this scheme *Useful Reuse Distance* (URD) as it takes into account the *request type* in calculating the reuse distance of the workload. URD only considers accesses to the *referenced* data. It *eliminates* WAW and WAR access patterns from the reuse distance calculation. It considers only the maximum *reuse distance* of RAR and RAW access patterns in reuse distance calculation. The maximum URD of the sample workload (Fig. 5b) is equal to 1, due to the read access of Req_3 to the first sector of disk which was previously (two request before) written by Req_1. In this case, we assign cache size equal to only two blocks. Fig. 5b shows the contents of the allocated cache space based on URD for the sample workload. It can be seen that by employing the concept of URD, we achieve hit ratio similar to the TRD scheme while reducing the allocated cache size.

To summarize, in order to show how URD is able to allocate a smaller cache space compared to TRD and at the same time also achieve a similar hit ratio, we classify the workloads into two groups:

(1) Workloads where RAR and RAW (RA*) accesses are involved in the maximum reuse distance calculation.
(2) Workloads where WAR and WAW (WA*) accesses are involved in the maximum reuse distance calculation.

These workloads are characterized with the following two equations, respectively.[2]

$$\begin{cases} 1: RD(WA*) \leq RD(RA*) \rightarrow TRD \propto RD(RA*), \\ \quad URD \propto RD(RA*) \rightarrow TRD = URD \\ 2: RD(WA*) > RD(RA*) \rightarrow TRD \propto RD(WA*), \\ \quad URD \propto RD(RA*) \rightarrow TRD > URD \end{cases} \quad (1)$$

In the workloads of the first group (Eq. 1: part 1), both TRD and URD work similarly in cache size estimation. On the other hand, in the workloads of the second group (Eq. 1: part 2), the URD of the workload is smaller than TRD and hence URD allocates a smaller cache size compared to TRD. This is because URD considers only the RA* accesses. The maximum reuse distance of RA* requests is smaller than the maximum reuse distance of WA* requests for the workloads in the second group and hence URD provides smaller maximum reuse distance and leads to the allocation of a smaller cache space. In this case, URD achieves a similar hit ratio while allocating a *smaller* cache space compared to TRD.

5 ECI-CACHE ARCHITECTURE

In this section, we describe the architecture of the ECI-Cache. ECI-Cache 1) collects and analyzes the access patterns of the VMs and 2) allocates an efficient and effective cache size and write policy to each VM. Fig. 8 provides an overview of the ECI-Cache architecture in the hypervisor of a virtualization platform. As shown in this figure, ECI-Cache consists of three major components: (1) *Monitor*, (2) *Analyzer*, and (3) *Actuator*. ECI-Cache resides in the path of IO requests coming from VMs to the storage subsystem.

[2]RD: Reuse Distance.

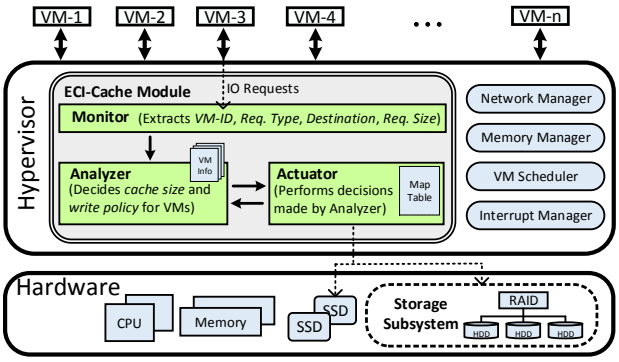
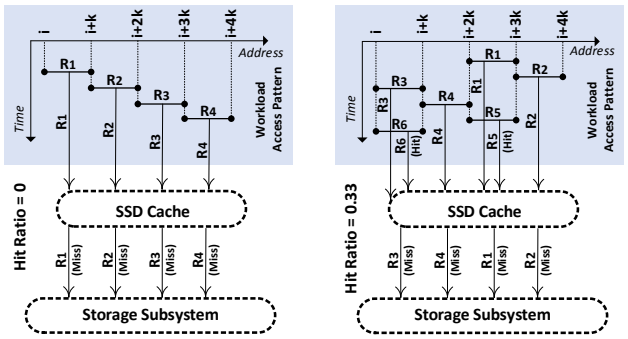


Figure 8: Architecture of ECI-Cache.

Monitor captures and collects information about the IO behavior of each VM. *Analyzer* decides the cache size and write policy by characterizing the IO behavior of the corresponding VM. *Actuator* realizes the decisions made by *Analyzer* by allocating an efficient and effective cache space and write policy for each VM in the SSD cache. We describe each component in more detail:

(1) *Monitor* receives all the IO requests coming from VMs and extracts important information such as *VM Identification Number* (*VM-ID*), *request type*, *destination address*, and *request size* by using *blktrace*, a block layer IO tracing tool [8] that is available in the Linux kernel (version 2.6.16 and upper). *Blktrace* receives event traces from the kernel and records the IO information. We modified the source code of *blktrace* to extract the owner of each request (i.e., the VM that sends the request) at the hypervisor level. Such modification helps us to classify the requests and discover the access patterns of the running workloads in different VMs. The extracted information is passed to *Analyzer*.
(2) *Analyzer* decides 1) the target destination of a given IO request, 2) an efficient cache size for each VM, and 3) the write policy of the I/O cache for each VM, based on 1) the information it receives from *Monitor* and 2) a database it employs, called *VM Info*. *Analyzer* keeps information about each VM, such as cache size, write policy, workload characteristics, and the number of VMs running in the system in the *VM Info* database.
(3) *Actuator* is responsible for realizing the decisions made by *Analyzer*. It allocates the decided cache space for each VM, configures the decided write policy, and also routes the IO requests to the SSD cache or the storage subsystem. *Actuator* keeps logs for blocks stored in either the SSD cache or the storage subsystem in a table (namely *Map Table*). This table is used for responding to future requests.

Typically, in a virtualization platform, there are several VMs running various workloads with different IO behavior. The hypervisor is responsible for partitioning the cache space efficiently between VMs. In workloads with a sequential access pattern, there is little locality of reference and hence buffering data to capture future reuse demonstrates very poor hit ratio (Fig. 9a). In workloads with a random access pattern, the probability of referencing the blocks that are accessed previously is significantly greater than that in workloads with a sequential access pattern and hence buffering data with a random access pattern improves the hit ratio (Fig. 9b). Hence, in our proposed architecture, cache space will be allocated to only VMs with random read or write access patterns. Sec.



(a) Sequential accesses. (b) Random accesses.

Figure 9: Cache hits in sequential and random access patterns.

5.1 describes our proposed algorithm for cache space partitioning across VMs via online characterization of the workloads. Sec 5.2 describes the write policy assignment to each VM.

5.1 Efficient Cache Size Estimation

We propose an ECI-Cache size allocation algorithm, which aims to allocate an efficient cache size for each VM based on the reuse distances of the running workloads. Previously, in Sec. 4, we proposed the metric of URD and provided an example that showed the effect of considering request type on 1) the reuse distance of a workload and 2) the estimated cache size. We also demonstrated that employing URD instead of TRD in cache size estimation preserves the performance of the workload while reducing the allocated cache size. Such a scheme allocates a much smaller cache space to the workloads and hence improves the performance-per-cost of each VM. It also reduces the number of unnecessary writes (due to WAW and WAR operations) into the SSD cache, thereby improving SSD lifetime.

Periodically, ECI-Cache calculates the URD of the running workloads in VMs and then estimates an efficient cache size for each VM. To provide the minimum latency for all VMs, we employ an optimization algorithm that meets the conditions of Eq. 2. In this equation, c_i is the allocated cache space for each VM_i, C is the total SSD cache space, N is the number of running VMs, and $h_i(c_i)$ denotes the achieved hit ratio for the running workload on VM_i when we allocate cache space equal to c_i. In addition, T_{hdd} and T_{ssd} indicate the average read/write access latency to the HDD and the SSD, respectively.

$$\begin{cases} LatencyVM_i = h_i(c_i) \times T_{ssd} + (1 - h_i(c_i)) \times T_{hdd} \\ \text{Objective: } Minimize[\sum_{i=1}^{N} LatencyVM_i] \\ \text{Constraint1: } \sum_{i=1}^{N} c_i \leq C \\ \text{Constraint2: } 0 \leq c_i \leq c_{urd_i} \end{cases} \quad (2)$$

According to the constraints in Eq. 2, ECI-Cache partitions the cache space in a way that aims to *minimize* the aggregate latency of all VMs. Since allocating cache space for workloads with sequential access patterns achieves very poor hit ratio, ECI-Cache allocates cache space for VMs with random access patterns that have a high degree of access locality. ECI-Cache estimates the true amount of cache space for each VM based on URD. The total SSD cache space should be greater than or equal to the sum of the cache sizes of all N VMs. Hence, we have two possible cases for the allocated cache size: (1) the sum of estimated cache sizes for all VMs using URD is *less* than the total SSD cache space or (2) the sum of estimated cache sizes is *greater* than the total SSD cache space. In the first case (i.e., when the SSD cache space is *not limited*), ECI-Cache would allocate the exact estimated cache space for each VM and thus maximize the overall hit ratio. In the second case (i.e., when the SSD cache space is *limited*), since we have shortage of cache space, it is necessary to *recalculate* the cache size for each VM in order to fit each VM into the existing cache space.

Algorithm 1 shows how ECI-Cache estimates and allocates cache space in a virtualized platform with N VMs (Algorithm 4 in the Appendix provides a more detailed version of Algorithm 1). Initially, a previously defined minimum cache size ($c_{i_{min}}$) is allocated to each VM.[3] At specific time intervals (Δt), we separately extract the IO traces of the running workloads on the VMs into text files (line 1 and line 2). The information included in the IO traces for each request are: 1) the destination address, 2) size, 3) type, and 4) VM_{ID} of the request. This information is extracted by the *Monitor* part of the ECI-Cache.

In the next step, we use the collected traces to calculate the URD of the workloads using the *calculateURD* function (line 4). In the *calculateURDbasedSize* function (line 5), based on the calculated reuse distances, we find the required cache space for each VM that maximizes the hit ratio. We check the feasibility of the estimated cache sizes to see if the sum of estimated cache sizes (*csum* which is calculated in line 6) is less than the total SSD cache capacity (line 8). When the condition in line 8 is met and the sum of estimated cache spaces for all VMs is less than or equal to the total SSD cache capacity, we call the cache space allocation "$feasible$"; otherwise (when the condition in line 11 is met) we call the allocation "$infeasible$". In case of in-feasibility (in line 13), we need to recalculate the cache space allocation of each VM such that the SSD cache capacity is not exceeded. To do so, we run the *calculateEffSize* function that employs an optimized minimization algorithm ("$fmincon$") [32] to find the most efficient set of cache space allocations that minimizes the aggregate latency of all VMs under the constraint that the total allocated cache space is less than the SSD cache capacity.[4] The input of the minimization algorithm is (1) the existing SSD cache capacity, (2) the hit ratio function of each VM (which will be described in Algorithm 2) based on allocated cache space (H(c)) that has been extracted by analyzing the reuse distances of the workloads, and (3) the estimated cache sizes by the algorithm which cannot be fit into the existing SSD cache capacity. Finally, in line 15 we allocate efficient cache spaces for each VM.

Algorithm 2 shows the structure of the hit ratio function. $h(c_i)$ provides the hit ratio that can be obtained, if we assign a specific cache space (c_i) to VM_i. This function is extracted from the output of the *calculateURD* function (described in Algorithm 1). In the *calculateURD* function, we extract the ratio of accesses with useful reuse distance N (URD_N). Then, for each c_i ($c_i = URD_N \times cacheBlkSize$), $h_i(c_i)$ is equal to ratio of accesses with useful reuse distance N. The specific cache space c_i is calculated based on the different reuse distances whose hit ratio is the ratio of corresponding reuse distances. In each time interval, in case of infeasibility, we update the hit ratio function of each

[3] In the experiments, $c_{i_{min}}$ is set to $1,000$ blocks.
[4] To this end, we use the "$fmincon$" function from the MATLAB Optimization toolbox [32].

Algorithm 1: ECI-Cache size allocation algorithm.

```
   /* Inputs: Number of VMs: (N), SSD cache size: (C), HDD Delay: (T_HDD),
      SSD Delay: (T_SSD)                                                  */
   /* Output: Efficient cache size for each VM: (c_eff[1..N])             */
1  Sleep for Δt
2  Extract the traces of the workloads running on the VMs including 1)
   destination address, 2) request size, and 3) request type
3  for i = 1 to N do
4  |  URD[i] = calculateURD(VM[i])
5  |  size_urd[i] = calculateURDbasedSize(URD[i])
6  |  csum+ = size_urd[i]
7  end
   /* Check the feasibility of size estimation and minimize overall latency for
      estimated size_urd[1..N]                                            */
8  if csum ≤ C then
9  |  c_eff[1..N] = size_urd[1..N]
10 end
11 else if csum > C then
12 |  Create hit ratio function of VMs (H_i(c)) based on reuse distance of the
   |  workloads.
13 |  c_eff[1..N] = calculateEffSize(size_urd[1..N], C)
14 end
15 allocate(c_eff[1..N], VM[1..N])
   /*
      Functions Declaration:
      calculateURD                                                        */
16 Function calculateURD(VM) is
   |  /* This function calls PARDA [38] which is modified to calculate URD
   |     (reuse distance only for RAR and RAW requests.)                  */
17 |  return URD
18 end
19 /*
      calculateURDbasedSize                                               */
20 Function calculateURDbasedSize(URD) is
21 |  size_urd = URD × cacheBlkSize
22 |  return size_urd
23 end
24 /*
      calculateEffSize                                                    */
25 Function calculateEffSize(size_urd[1..N], C) is
26 |  initialSize = {c_min, ..., c_min}
27 |  lowerBound = {c_min, ..., c_min}
28 |  upperBound = {size_urd[1], ..., size_urd[N]}
29 |  weightVM = {1, ..., 1}
   |  /* Here we use fmincon function from MATLAB Optimization toolbox.    */
30 |  c_eff[1..N] =
   |  fmincon(ObjectiveFunction, initialSize, weightVM, C_tot, {}, {}
   |  , lowerBound, upperBound)
31 |  return c_eff[1..N]
32 end
33 /*
      ObjectiveFunction                                                   */
34 Function ObjectiveFunction() is
35 |  for i = 1 to N do
36 |  |  h[i] = H_i(c[i])
37 |  |  hsum+ = h[i]
38 |  |  csum+ = c[i]
39 |  end
40 |  diff = C − csum
41 |  Obj = diff + (hsum) × T_SSD + (N − hsum) × T_HDD
42 |  return Obj
43 end
```

VM and feed it to the minimization algorithm. The minimization algorithm uses the hit ratio function of running VMs to minimize the sum of latency of all VMs, as calculated using Eq. 2.

5.2 Write Policy Estimation

In order to allocate the most efficient write policy for each VM, ECI-Cache analyzes the access patterns and also the request types of the running workloads on the VMs. We choose between RO and WB policies for each VM. The key idea is to use 1) the RO policy for VMs with write operations without any further read access and 2) the WB policy for VMs with referenced write operations (i.e., write operations with further read access). To minimize the number of unnecessary writes, we assign the RO policy to the caches of VMs with read-intensive access patterns (including RAR and RAW accesses). The RO policy improves the performance of read operations and increases the lifetime of the SSD. In addition, such a scheme is more reliable since it does *not* buffer writes in the cache.

Algorithm 2: The structure of hit ratio function.

```
1  Function H_i(c) is
2  |  if 0 ≤ c < m_1 then
3  |  |  h = h_1
4  |  end
5  |  else if m_1 ≤ c < m_2 then
6  |  |  h = h_2
7  |  end
8  |  ...
9  |  else if m_{k−1} ≤ c < m_k then
10 |  |  h = h_k
11 |  end
12 |  return h
13 end
```

As mentioned previously in Sec. 5, *Analyzer* is responsible for the write policy assignment for each VM and it does so periodically (every Δt). *Analyzer* checks the ratio of WAW and WAR operations (namely, *writeRatio*). If the *writeRatio* of the running workload exceeds a defined threshold, we change the write policy to RO, to avoid storing a large number of written blocks in the cache. This is due to two reasons: 1) such a workload includes a large amount of writes and holding such writes in the SSD cache would likely not have a positive impact on the hit ratio, 2) caching such a large amount of writes has a negative impact on the endurance of the SSD. We select the WB cache policy when the running workload on a VM includes a large fraction of *RAW* accesses. In addition, we assign the RO policy to the caches with a larger fraction of *WAW* and *WAR* accesses.

Algorithm 3 shows how ECI-Cache assigns an efficient write policy for each VM's cache space. Initially, we assign the WB policy for a VM's cache space. Then, periodically, we analyze the behavior of the running workload and re-assess the write policy. In line 3 of Algorithm 3, after a period of time (Δt), we calculate the ratio of WAW and WAR requests (*writeRatio*) for VM_i (line 4). In line 5, we check whether the ratio of WAW and WAR requests is greater than a threshold (namely *wThreshold*) or not. If the ratio of such requests is greater than *wThreshold*, we assign the RO policy for VM_i (in line 6); otherwise the policy remains as WB.

Algorithm 3: ECI-Cache write policy assignment algorithm.

```
   /* Inputs: Number of VMs: (N)                                          */
   /* Output: Efficient cache policy for each VM: (P_i_eff)               */
1  set(P_i_eff, VM_i) = WB /* Initialization                             */
2  for i = 1 to N do
3  |  Sleep for Δt
4  |  writeRatio = (getNumOfWAW(VM_i) + getNumOfWAR(VM_i)) / getNumOfReq(VM_i)
5  |  if writeRatio ≥ wThreshold then
6  |  |  set(P_i_eff, VM_i) = RO
7  |  end
8  end
```

6 EXPERIMENTAL RESULTS

In this section, we provide comprehensive experiments to evaluate the effectiveness of the ECI-Cache.

6.1 Experimental Setup

To evaluate our proposed scheme, we conduct experiments on a real test platform, an HP ProLiant DL380 Generation 5 (G5)

server [20] with four 146GB SAS 10K HP HDDs [21] (in RAID-5 configuration), a 128GB Samsung 850 Pro SSD[5] [44] used as the SSD cache, 16GB DDR2 memory from Hynix Semiconductor [46], and 8 1.6GHz Intel(R) Xeon CPUs [24]. We run the QEMU hypervisor on the *Centos 7* operating system (kernel version 3.10.0-327) and create different VMs running *Ubuntu 15.04* and *Centos 7* operating systems on the hypervisor. The configuration of the device layer is in the default mode where the *request merge* option in the device layer is enabled for a 128-entry device queue size. We have integrated ECI-Cache with QEMU to enable dynamic partitioning of the SSD cache and allocation of an efficient cache space and write policy for each VM.

6.2 Workloads

We use MSR traces from SNIA [47], comprising more than fifteen workloads, as real workload traces in our experiments. We run *Ubuntu 15.04* and *Centos 7* operating systems on the VMs and allocate two virtual CPUs, 1GB memory, and 25GB of hard disk for each VM. The experiments are performed with 16 VMs. Table 2 shows the workloads run on each VM. The SSD cache is shared between VMs. ECI-Cache estimates the most efficient cache size for each VM and partitions the SSD cache space between the VMs based on the running workload's IO pattern and request type. We have also implemented the state-of-the-art IO caching scheme for virtualized platforms, Centaur [27], on our test platform. We run the same experiments with Centaur. Similar to ECI-Cache, Centaur works based on reuse distance but it does *not* consider request type in reuse distance calculation. In addition, it does *not* have any control on the write policy of the SSD cache for each VM.

6.3 Cache Allocation to Multiple VMs

To show how ECI-Cache affects performance, performance-per-cost, and allocated cache space to VMs compared to Centaur, we conduct experiments in two conditions: 1) when the SSD cache capacity is limited, i.e., when the total SSD cache size is less than the sum of the estimated cache spaces for the VMs. In this case, the cache size estimation by ECI-Cache and Centaur may become infeasible and 2) when the SSD cache capacity is unlimited, i.e., the cache has enough space to allocate the required and efficient cache space for each VM.[6] In the experiments, VMs are run concurrently and cache space is partitioned across VMs for both ECI-Cache and Centaur schemes (we perform the same, yet separate and independent experiments for both schemes and then compare the results).

The experiment is performed on 16 running VMs. An initial cache space equal to $10,000$ cache blocks (block size is equal to 8KB) with WB policy is allocated to each VM. The total SSD cache capacity is 3 million blocks. Cache space is calculated in 10-minute time intervals (Δt) for both ECI-Cache and Centaur. We select 10-min time intervals to reduce the time overhead of the URD calculation to less than 5% (this trade-off has been obtained based on the reported URD calculation overheads in Table 3 in Appendix B). Reducing the length of the time interval provides more accurate estimations but it also increases the time overhead of the URD calculation.

Fig. 10a and Fig. 10b show how Centaur and ECI-Cache allocate cache spaces for the VMs in the time interval from $t = 950$ *min* to $t = 1,700$ *min* of the experiment when the SSD cache capacity is limited to 3 million blocks. We make two major observations: 1) Centaur becomes infeasible and reduces the allocated cache spaces of the VMs to fit in the existing SSD capacity and 2) ECI-Cache never becomes infeasible since the sum of estimated cache spaces for the VMs is less than existing SSD capacity. This is because Centaur estimates a *larger* cache space for each VM because it does *not* consider the request type while ECI-Cache estimates much *smaller* cache space because it *does* consider the request type. When the estimated cache space becomes infeasible (in Centaur), Centaur employs an optimization algorithm to find efficient cache sizes which can be fit in the existing SSD cache capacity. To this end, Centaur allocates smaller cache space to each VM and hence achieves a smaller hit ratio than ECI-Cache. In these experiments, in-feasibility does not happen for ECI-Cache. However, when ECI-Cache becomes infeasible, Centaur would be infeasible, too. In such cases, both schemes should apply optimization algorithms to reduce the estimated cache spaces in order to fit in the existing SSD cache.[7] We conclude that in infeasible cases, ECI-Cache provides greater hit ratio and better performance than Centaur.

Fig. 11 shows the details of allocated cache sizes for each individual VM by both Centaur and ECI-Cache. In addition, this figure shows the IO latency of the VMs for both schemes. During the experiment, the running workloads on the VMs finish one by one (because different workloads have different runtimes) and since VMs with the finished workload generates no more IO requests, ECI-Cache excludes those VMs from cache space partitioning. During the course of the experiment, we retrieve the allocated cache space of such VMs. In order to provide more detail, we present the results of each VM separately. The infeasible areas that lead to latency degradation in Centaur are shown by dashed circles around latency lines. We observe that in infeasible states, i.e., when SSD cache capacity is limited, ECI-Cache improves performance and performance-per-cost by 17.08% and 30% compared to Centaur.

6.4 Write Policy Assignment

To show how ECI-Cache assigns an efficient write policy for each VM, we conduct experiments based on Algorithm 3. We characterize incoming requests from different VMs and calculate the ratio of *WAW* and *WAR* operations for the running VMs. Then, we assign an efficient write policy for the VM's cache. In the experiments, we set the value of $wThreshold$ to between 0.2 and 0.9 and achieve different results based on the value of $wThreshold$. Note that we assign the RO policy to a VM cache if the ratio of combined WAW and WAR requests over all requests is greater than or equal to $wThreshold$.

Fig. 12 shows the ratio of different types of the requests in the running workloads. In addition, Fig. 13 shows the number of WAW, WAR, RAR, and RAW accesses for the workloads of the running VMs, which are sampled in 10-minute intervals. Since the first access (either read or write request) to an arbitrary address within a sequence of I/O requests cannot be classified as either WAW, WAR, RAR, and RAW, we denoted the first read and write access

[5] The capacity of selected SSD is *larger* than the sum of efficient cache spaces for the running VMs. Thus, there is no SSD cache capacity shortage in the experiments. As a result, employing an SSD with a larger size would not provide any improvement in the performance of VMs.

[6] The experiments of the second case are provided in Appendix A.

[7] In Section 4, we showed that the estimated cache space based on TRD (Centaur) would be greater than or equal to the estimated cache space by URD (ECI-Cache), and hence Centaur would reduce the cache size of each VM more than ECI-Cache would. In this case, the performance degradation in Centaur would be greater than performance degradation in ECI-Cache.

Table 2: Information of running workloads on the VMs.

VM_{ID}	VM0	VM1	VM2	VM3	VM4	VM5	VM6	VM7	VM8	VM9	VM10	VM11	VM12	VM13	VM14	VM15
Workload	wdev_0	web_1	stg_1	ts_0	hm_1	mds_0	proj_0	prxy_0	rsrch_0	src1_2	prn_1	src2_0	web_0	usr_0	rsrch_2	mds_1
Run Time (min)	1, 140	160	2, 190	1, 800	600	1, 210	4, 220	12, 510	1, 430	1, 900	1, 1230	1, 550	2, 020	2, 230	200	1, 630

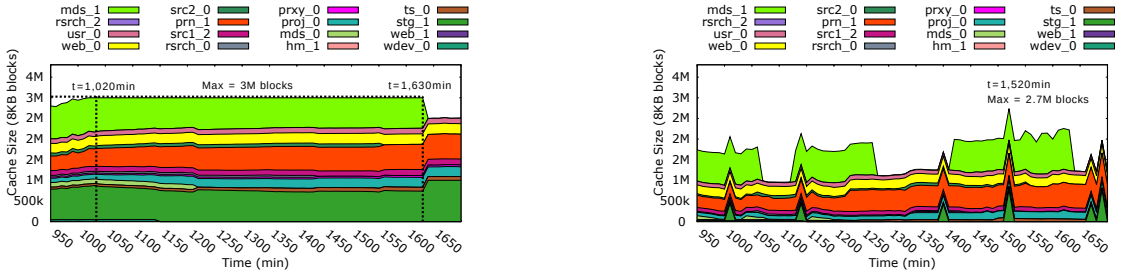


Figure 10: Cache allocation for the VMs in infeasible state by (a) Centaur and (b) ECI-Cache.

to an address as *Cold Read* (CR) and *Cold Write* (CW), respectively. Here we set $wThreshold = 0.5$.

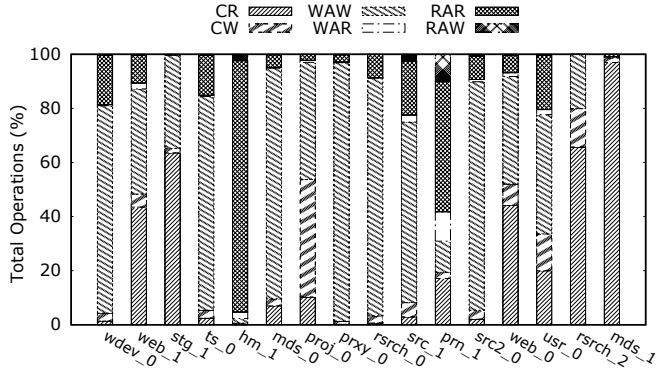


Figure 12: Ratio of different types of requests in the workloads.

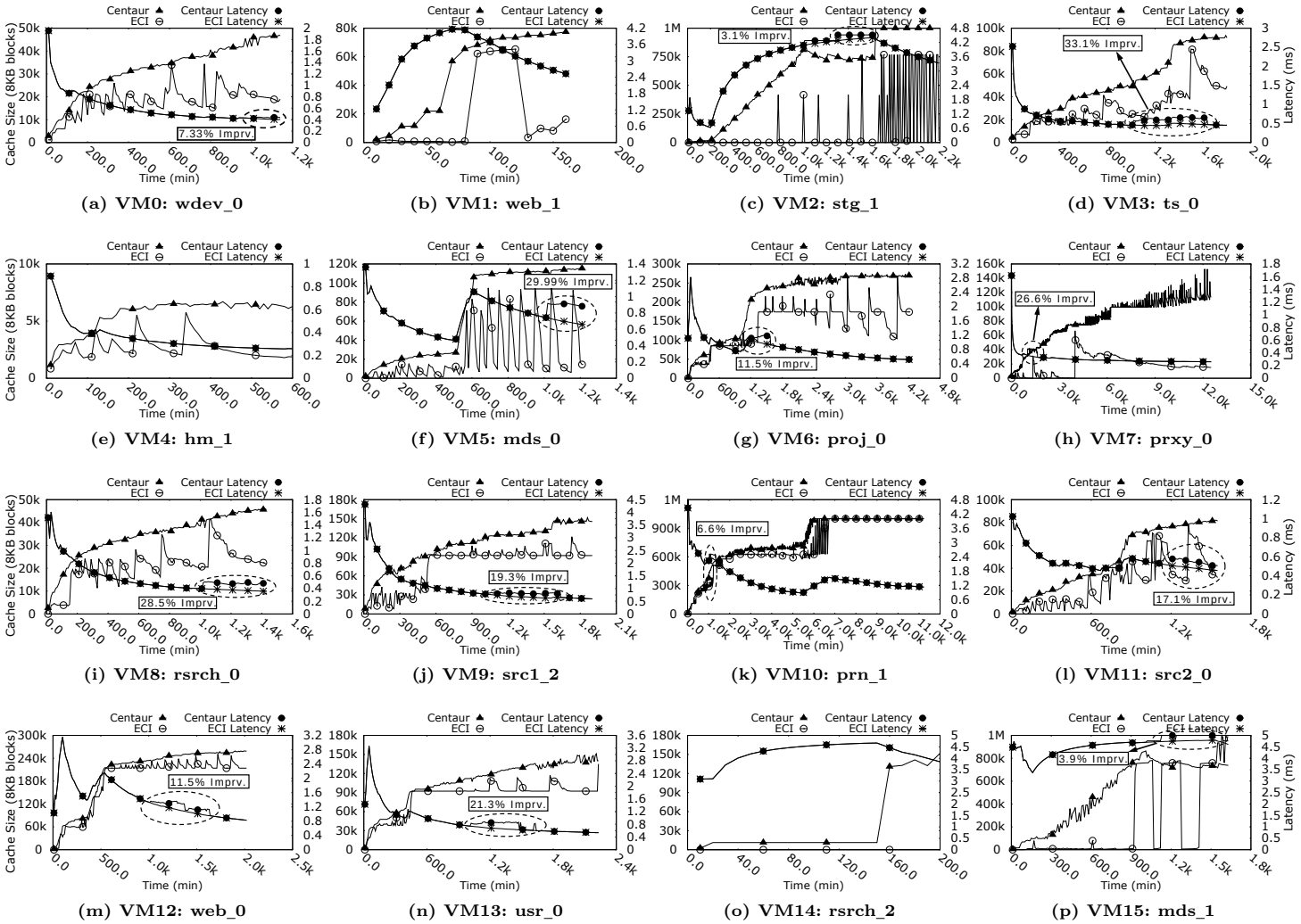
It can be seen that in the first 70-minute interval of web_1 (running on VM1 as shown in Fig. 13b and Fig. 12), neither WAW nor WAR accesses exist and ECI-Cache assigns the WB policy for the allocated cache. Then at $t = 100$ *min*, WAW operations become dominant and the RO policy is assigned to the cache. In VM0, with the running workload of wdev_0, after about 50 minutes, we recognize that 77% of the requests are WAW and the remaining are mostly RAR (shown in Fig. 13a and Fig. 12) and thus at $t = 50$ *min*, the RO policy is assigned to the cache of VM0 by ECI-Cache. As shown in Fig. 13e and Fig. 12, hm_1 running on VM4 consists of mostly RAR operations (more than 92%) without any WAR and WAW accesses and thus the RO policy is assigned to this VM. In time intervals between $t = 0$ to $t = 500$ *min* and $t = 610$ *min* to $t = 1200$ *min*, more than 86% of the requests of proj_0 running on VM6 are CW and WAW operations, and thus ECI-Cache assigns the RO policy to this VM. In the remaining interval ($t = 500$ *min* to $t = 610$ *min*), the WB policy is assigned to this VM. ECI-Cache assigns the RO policy for VMs such as prxy_0 and web_0 that have a large number of WAW and WAR operations. Doing so minimizes the number of unnecessary writes into the cache in these workloads.

6.5 Performance and Performance-Per-Cost Improvement

The results of previous experiments on the proposed test platform indicate a significant performance and performance-per-cost (i.e., performance per allocated cache space) improvement for the running workloads on the VMs, as quantified in Fig. 14. We observed that ECI-Cache is able to estimate a smaller cache space for each VM than Centaur, without any negative impact on performance. In other words, ECI-Cache achieves similar hit ratio while allocating smaller cache space and hence improves performance-per-cost. In addition, ECI-Cache achieves higher performance compared to Centaur in infeasible cases for each VM. Cache size estimation in the proposed scheme is based on the URD metric and thus ECI-Cache allocates much smaller cache space to the VMs compared to Centaur. Centaur estimates cache space based on TRD, which does *not* consider the request type, leading to a higher cache size estimation for each VM.

Fig. 14 shows the achieved performance and performance-per-cost of the VMs when we use the Centaur and ECI-Cache schemes. We observe that allocating cache space based on ECI-Cache for each VM improves the performance and performance-per-cost compared to Centaur in all workloads.

Fig. 15 shows the cumulative latency of the running VMs with Centaur vs. ECI-Cache in *infeasible* states (i.e., when the total SSD cache capacity is limited). In the time interval shown in Fig. 15, ECI-Cache achieves a higher hit ratio than Centaur. Therefore, in *infeasible* states ECI-Cache reduces the latency of the workloads by 17%, on average. We conclude that ECI-Cache improves performance and performance-per-cost for the running VMs by 17.08% and 30%, by intelligently reducing the allocated cache space for each VM.



(a) VM0: wdev_0 (b) VM1: web_1 (c) VM2: stg_1 (d) VM3: ts_0

(e) VM4: hm_1 (f) VM5: mds_0 (g) VM6: proj_0 (h) VM7: prxy_0

(i) VM8: rsrch_0 (j) VM9: src1_2 (k) VM10: prn_1 (l) VM11: src2_0

(m) VM12: web_0 (n) VM13: usr_0 (o) VM14: rsrch_2 (p) VM15: mds_1

Figure 11: Allocated cache space of each VM and corresponding latency in infeasible state with limited SSD cache capacity (ECI-Cache vs. Centaur).

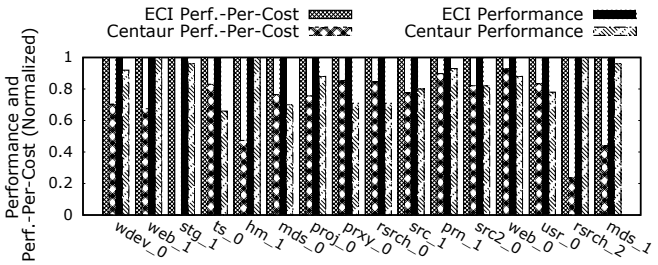


Figure 14: Performance and performance-per-cost achieved by ECI-Cache and Centaur.

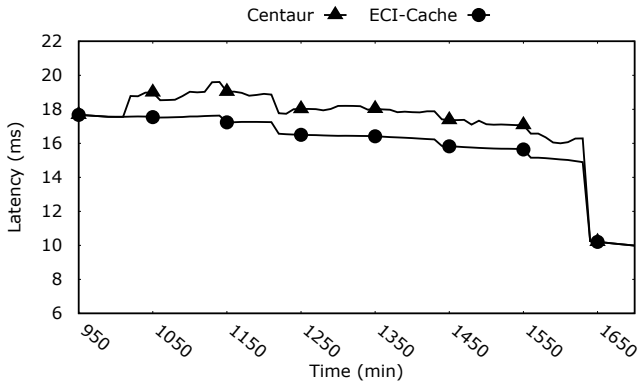
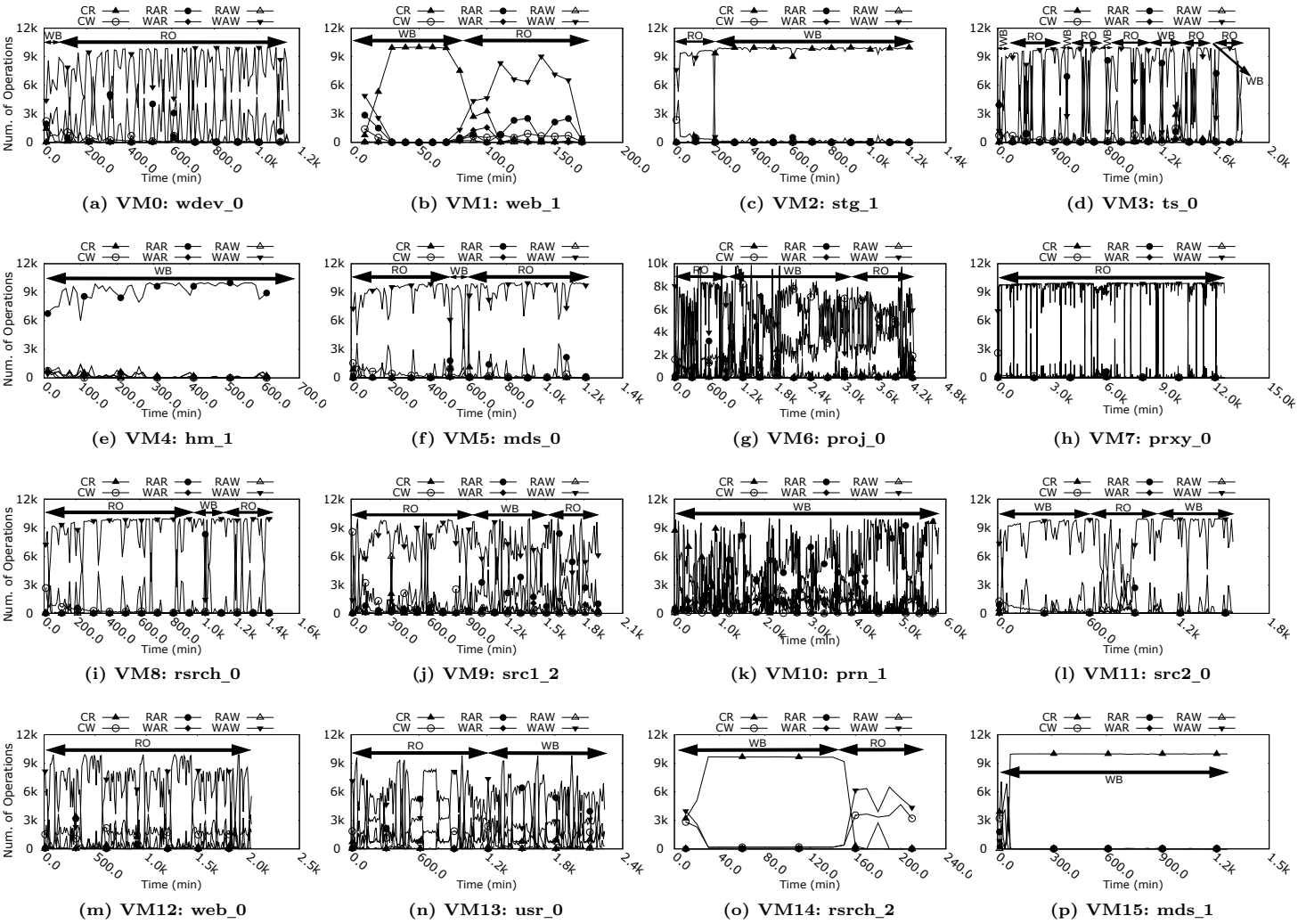


Figure 15: Cumulative latency of VMs with ECI-Cache and Centaur in infeasible states.

6.6 Endurance Improvement

To show the endurance improvement of ECI-Cache, we perform experiments by applying our write policy assignment algorithm and show the impact of the proposed scheme on the number of writes and also the performance of the VMs. Endurance of the



(a) VM0: wdev_0 (b) VM1: web_1 (c) VM2: stg_1 (d) VM3: ts_0

(e) VM4: hm_1 (f) VM5: mds_0 (g) VM6: proj_0 (h) VM7: prxy_0

(i) VM8: rsrch_0 (j) VM9: src1_2 (k) VM10: prn_1 (l) VM11: src2_0

(m) VM12: web_0 (n) VM13: usr_0 (o) VM14: rsrch_2 (p) VM15: mds_1

Figure 13: ECI-Cache write policy assignment to the VMs.

SSD is affected by the number of writes committed to it. Write operations on the SSD impose NAND flash memory writes and increase the P/E-cycle count [10–12, 36, 43, 50]. ECI-Cache has a positive impact on the endurance of the SSD because it *reduces* the number of committed writes to the SSD. While ECI-Cache can effectively manage the committed writes to the SSD, it has no control on the writes initiated by the garbage collection and wear-leveling algorithms used within the SSD. Hence, we report endurance improvement of the SSD cache by using the reduced number of writes as a metric. A smaller number of writes is expected to lead to better endurance. Similar metrics are used in previous system-level studies, such as [22, 29, 31, 40]. Note that the total number of writes for each workload (reported in the experiments) is calculated by Eq. 3 which includes writes from the disk subsystem to the SSD and also the writes from the CPU to the SSD:

$$Total\ Writes = \sum (CR + CW + WAR + WAW) \qquad (3)$$

where CR (Cold Read) is the first read access to an address and CW (Cold Write) is the first write access to an address. Fig. 16 shows the number of writes into the SSD cache and the allocated cache space with Centaur and ECI-Cache. As this figure shows, ECI-Cache assigns a more efficient write policy for each VM as compared to Centaur. We re-conducted experiments of Sec. 6.4 and the results demonstrate that by using the RO policy, we can reduce the number of writes by 44% compared to the caches using the WB policy. When ECI-Cache applies the RO policy on the VMs, only writes due to CRs (Cold Reads) will be written on the cache. Applying the RO policy on the VMs has a negative impact on the hit ratio of RAW operations (by 1.5%). As it

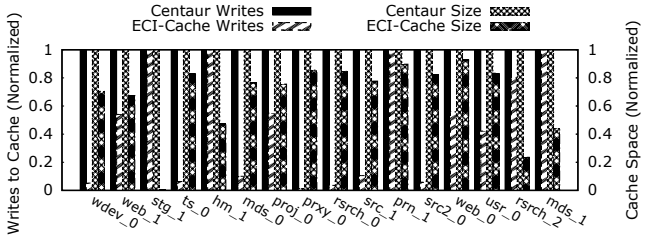


Figure 16: Writes into the SSD cache and allocated cache space to the VMs by Centaur and ECI-Cache.

can be seen in Fig. 16, applying ECI-Cache on the VM running hm_1 has no impact on the number of writes into the cache. This workload mostly consists of RAR and RAW operations where it is necessary to cache referenced data by assigning the WB policy. It is important to note that as presented in Sec. 5.1 (shown in Fig. 14), ECI-Cache allocates about 50% smaller cache space to this VM, which is calculated based on the reuse distance of RAR and RAW operations by using URD. Similarly, for the VMs running stg_1, mds_1, and prn_1, ECI-Cache allocates much less cache space based on the calculated URD for RAR and RAW operations and assigns the WB policy. Hence, ECI-Cache has no impact on the number of writes into the SSD in such VMs. For the remaining VMs running workloads such as mds_0, ECI-Cache reduces the number of writes by 80% and the allocated cache space by 25%. ECI-Cache achieves 18% cache size reduction by 90% reduction in number of writes for ts_0. For proj_0, cache size and the number of writes is reduced by 22% and 46%, respectively. ECI-Cache reduces the number of writes for prxy_0, wdev_0, rsrch_0, src2_0, and src1_2 by 90%, 85%, 87%, 85%, and 88%, respectively. We conclude that ECI-Cache assigns an efficient write policy for each VM and thereby reduces the number of writes into the SSD cache by 65% on average across all VMs.

7 CONCLUSION

In this paper, we presented the ECI-Cache, a new hypervisor-based I/O caching scheme for virtualized platforms. ECI-Cache maximizes the performance-per-cost of the I/O cache by 1) dynamically partitioning it between VMs, 2) allocating a small and efficient cache size to each VM, and 3) assigning a workload characteristic-aware cache write policy to each VM. The proposed scheme also enhances the endurance of the SSD I/O cache by reducing the number of writes performed by each VM to the SSD. ECI-Cache uses a new online partitioning algorithm to estimate an efficient cache size for each VM. To do so, ECI-Cache characterizes the running workloads on the VMs and makes two key decisions for each VM. First, ECI-Cache allocates cache space for each VM based on the *Useful Reuse Distance* (URD) metric, which considers *only* the *Read After Read* (RAR) and *Read After Write* (RAW) operations in reuse distance calculation. This metric reduces the cost of the allocated cache space for each VM by assigning a smaller cache size because it does *not* consider *Write After Read* (WAR) and *Write After Write* (WAW) accesses to a block to be useful for caching purposes. Second, ECI-Cache assigns an efficient write policy to each VM by considering the ratio of WAR and WAW operations of the VM's workload. ECI-Cache assigns the *Write Back* (WB) policy for a VM with a large amount of re-referenced data due to RAW and RAR operations and the *Read Only* (RO) write policy for a VM with a large amount of unreferenced (i.e., not re-referenced) writes due to WAR and WAW accesses. By allocating an efficient cache size and assigning an intelligent cache write policy for each VM, ECI-Cache 1) improves both performance and performance-per-cost (by 17% and 30%, respectively, compared to the state-of-the-art [27]) and 2) enhances the endurance of the SSD by greatly reducing the number of writes (by 65%). We conclude that ECI-Cache is an effective method for managing the SSD cache in virtualized platforms.

ACKNOWLEDGMENTS

We would like to thank the reviewers of SIGMETRICS 2017 and SIGMETRICS 2018 for their valuable comments and constructive suggestions. We especially thank Prof. Ramesh Sitaraman for serving as the shepherd for this paper. We acknowledge the support of Sharif ICT Innovation Center and HPDS Corp.[8]

REFERENCES

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, Moinuddin K Panigrahy, RinaQureshi, and Yale N Patt. 2008. Design Tradeoffs for SSD Performance. In *USENIX Annual Technical Conference (USENIX ATC)*.

[2] Saba Ahmadian, Farhad Taheri, Mehrshad Lotfi, Maryam Karimi, and Hossein Asadi. 2018. Investigating Power Outage Effects on Reliability of Solid-State Drives. In *to appear in Design, Automation Test in Europe Conference Exhibition (DATE)*.

[3] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, François Labelle, Nate Coehlo, Xudong Shi, and Eric Schrock. 2013. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*.

[4] Raja Appuswamy, David C van Moolenbroek, and Andrew S Tanenbaum. 2012. Integrating Flash-Based SSDs into the Storage Stack. In *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*.

[5] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, Dulcardo Zhao, MingArteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. 2016. CloudCache: On-Demand Flash Cache Management for Cloud Computing. In *USENIX Conference on File and Storage Technologies (FAST)*.

[6] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. 2013. S-CAVE: Effective SSD Cahing to Improve Virtual Machine Storage Performance. In *Proceedings of Parallel Architectures and Compilation Techniques (PACT)*.

[7] Erik Berg and Erik Hagersten. 2004. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Performance Analysis of Systems and Software, IEEE International Symposium (ISPASS)*.

[8] Blktrace. 2006. Blktrace: Block Layer IO Tracing Tool. https://linux.die.net/man/8/blktrace. (2006). Retrieved March 2017 from https://linux.die.net/man/8/blktrace

[9] Steve Byan, James Lentini, Anshul Madan, Luis Pabon, Michael Condict, Jeff Kimmel, Steve Kleiman, Christopher Small, and Mark Storer. 2012. Mercury: Host-Side Flash Caching for the Data Center. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*.

[10] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. 2017. Error Characterization, Mitigation, and Recovery in Flash Memory Based Solid-State Drives. *Proceedings of the IEEE* 105 (2017), 1666–1704.

[11] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. 2012. Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. 521–526.

[12] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Crista, Osman S Unsal, and Ken Mai. 2013. Error Analysis and Retention-Aware Error Management for NAND Flash Memory. *Intel Technology Journal* 17, 1 (2013).

[13] Feng Chen, David A Koufaty, and Xiaodong Zhang. 2011. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In *International Conference on Supercomputing*.

[14] Chen Ding and Yutao Zhong. 2001. Reuse Distance Analysis. *University of Rochester, Rochester, NY* (2001).

[15] Chen Ding and Yutao Zhong. 2003. Predicting Whole-Program Locality Through Reuse Distance Analysis. In *ACM SIGPLAN Notices*.

[16] EMC Corporation. 2012. EMC VFCache. http://www.emc2.eu. (2012). Retrieved Feb 2016 from http://www.emc2.eu

[17] EnhanceIO. 2012. EnhanceIO. https://github.com/stec-inc/EnhanceIO. (2012). Retrieved June 2016 from https://github.com/stec-inc/EnhanceIO

[18] Changpeng Fang, Steve Carr, Soner Önder, and Zhenlin Wang. 2004. Reuse-Distance-Based Miss-Rate Prediction on a Per Instruction Basis. In *Proceedings of the workshop on Memory system performance*.

[19] Fusion-IO. 2005. Fusionio ioTurbine. http://web.sandisk.com. (2005). Retrieved Feb 2016 from http://web.sandisk.com

[20] HP-G5. 2010. HP ProLiant DL380 Generation 5. http://h18000.www1.hp.com/products. (2010). Retrieved Dec 2016 from http://h18000.www1.hp.com/products

[21] HP-HDD. 2016. SAS 10K HP HDD. http://h18000.www1.hp.com/products. (2016). Retrieved Dec 2016 from http://h18000.www1.hp.com/products

[22] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. 2016. Improving Flash-Based Disk Cache With Lazy Adaptive Replacement. *ACM Transactions on Storage (TOS)* (2016).

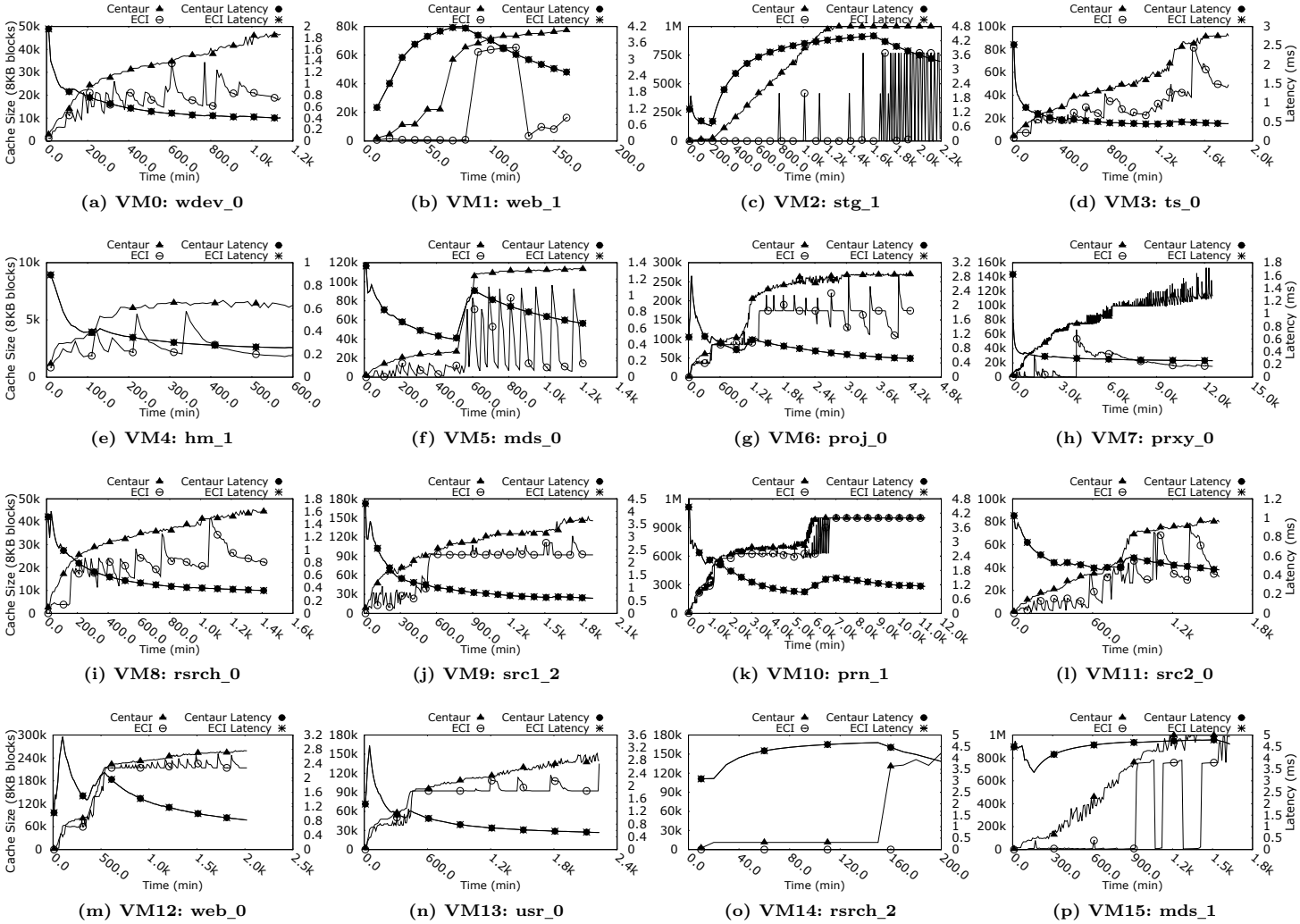
[8] http://hpdss.com/En/

[23] NetApp Inc. 2009. NetApp FlexCache. http://www.netapp.com/us/media/tr-3669.pdf. (2009). Retrieved Jan 2017 from http://www.netapp.com/us/media/tr-3669.pdf
[24] Intel. 2016. Intel(R) Xeon CPU. www.intel.com. (2016). Retrieved Jan 2017 from www.intel.com
[25] Youngjae Kim, Aayush Gupta, Bhuvan Urgaonkar, Piotr Berman, and Anand Sivasubramaniam. 2011. HybridStore: A Cost-Efficient, High-Performance Storage System Combining SSDs and HDDs. In *IEEE Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*.
[26] Yannis Klonatos, Thanos Makatos, Manolis Marazakis, Michail D Flouris, and Angelos Bilas. 2011. Azor: Using Two-Level Block Selection to Improve SSD-Based I/O Caches. In *IEEE International Conference on Networking, Architecture and Storage (NAS)*.
[27] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. 2015. Centaur: Host-Side SSD Caching for Storage Performance Control. In *IEEE International Conference on Autonomic Computing (ICAC)*.
[28] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. 2014. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*.
[29] Yushi Liang, Yunpeng Chai, Ning Bao, Hengyu Chen, and Yaohong Liu. 2016. Elastic Queue: A Universal SSD Lifetime Extension Plug-in for Cache Replacement Algorithms. In *Proceedings of the 9th ACM International on Systems and Storage Conference*.
[30] Deng Liu, Jianzhe Tai, Julia Lo, Ningfang Mi, and Xiaoyun Zhu. 2014. VFRM: Flash Resource Manager in VMware ESX Server. In *Network Operations and Management Symposium (NOMS)*.
[31] Jian Liu, Yunpeng Chai, Xiao Qin, and Yuan Xiao. 2014. PLC-cache: Endurable SSD Cache For Deduplication-Based Primary Storage. In *Mass Storage Systems and Technologies (MSST)*.
[32] MathWorks. 2017. fmincon. https://www.mathworks.com/help. (2017). Retrieved Sep. 2017 from https://www.mathworks.com/help
[33] Jeanna Matthews, Sanjeev Trika, Debra Hensgen, Rick Coulson, and Knut Grimsrud. 2008. Intel® Turbo Memory: Nonvolatile Disk Caches in the Storage Hierarchy of Mainstream Computer Systems. *ACM Transactions on Storage (TOS)* (2008).
[34] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Systems journal)* (1970).
[35] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. 2014. vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*.
[36] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. 2015. A Large-Scale Study of Flash Memory Failures in the Field. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 43. 177–190.
[37] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. 2009. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *ACM European Conference on Computer Systems*. 14. https://doi.org/10.1145/1519065.1519081
[38] Qingpeng Niu, James Dinan, Qingda Lu, and P Sadayappan. 2012. PARDA: A Fast Parallel Reuse Distance Analysis Algorithm. In *Parallel & Distributed Processing Symposium (IPDPS)*.
[39] QEMU. 2016. QEMU: Quick Emulator. http://wiki.qemu.org. (2016). Retrieved March 2016 from http://wiki.qemu.org
[40] Hongchan Roh, Mincheol Shin, Wonmook Jung, and Sanghyun Park. 2017. Advanced Block Nested Loop Join for Extending SSD Lifetime. *IEEE Transactions on Knowledge and Data Engineering* (2017).
[41] Reza Salkhordeh, Hossein Asadi, and Shahriar Ebrahimi. 2015. Operating System Level Data Tiering Using Online Workload Characterization. *The Journal of Supercomputing* (2015).
[42] Reza Salkhordeh, Shahriar Ebrahimi, and Hossein Asadi. 2018. ReCA: an Efficient Reconfigurable Cache Architecture for Storage Systems with Online Workload Characterization. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* PP, PP (2018), 1–1.
[43] Samsung. 2014. Endurance of the SSD for data centers. http://www.samsung.com. (2014). Retrieved July 2017 from http://www.samsung.com
[44] Samsung. 2016. Samsung SSD 850 Pro. http://www.samsung.com. (2016). Retrieved March 2016 from http://www.samsung.com
[45] Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. 2007. Locality Approximation Using Time. In *the Symposium on Principles of Programming Languages (POPL)*.
[46] SKHynix. 2016. Hynix Semiconductor Memory. https://www.skhynix.com/eng/index.jsp. (2016). Retrieved Jan 2017 from https://www.skhynix.com/eng/index.jsp
[47] SNIA. 2016. Microsoft Enterprise Traces, Storage Networking Industry Association IOTTA Repository. http://h18000.www1.hp.com/products. (2016). Retrieved Dec 2016 from http://h18000.www1.hp.com/products
[48] Vijayaraghavan Soundararajan and Jennifer M Anderson. 2010. The Impact of Management Operations on the Virtualized Datacenter. In *ACM International Symposium on Computer Architecture*.
[49] Mohan Srinivasan, Paul Saab, and V Tkachenko. 1991. FlashCache. https://github.com/facebookarchive/flashcache. (1991). Retrieved Jan 2017 from https://github.com/facebookarchive/flashcache
[50] Mojtaba Tarihi, Hossein Asadi, Alireza Haghdoost, Mohammad Arjomand, and Hamid Sarbazi-Azad. 2016. A Hybrid Non-Volatile Cache Design for Solid-State Drives Using Comprehensive I/O Characterization. *IEEE Transactions on Computers* 65, 6 (2016), 1678–1691.
[51] Omesh Tickoo, Ravi Iyer, Ramesh Illikkal, and Don Newell. 2010. Modeling Virtual Machine Performance: Challenges and Approaches. *ACM SIGMETRICS Performance Evaluation Review* (2010).
[52] Stephen Tweedie. 2000. Ext3, Journaling Filesystem. In *Ottawa Linux Symposium*.
[53] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. 2005. Intel Virtualization Technology. *Computer* (2005).
[54] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger. 2007. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the USENIX Conference on File and Storage Technologies (USENIX FAST)*.
[55] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *USENIX Conference on File and Storage Technologies (FAST 15)*.
[56] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program Locality Analysis Using Reuse Distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2009).

A CACHE ALLOCATION IN FEASIBLE STATE

In this section, we show how ECI-Cache and Centaur allocate cache space for the VMs in feasible state. We conduct experiments by applying both schemes in the hypervisor when the SSD cache capacity is unlimited. The experiments are performed on 16 running VMs with an initial cache space equal to 10,000 cache blocks for each VM (block size is equal to 8KB) and using the WB policy for each VM's cache space. Fig. 17 shows the allocated cache space by ECI-Cache and Centaur scheme for the VMs separately. In addition, this figure shows the latency of each VM.

We observed that for write-intensive workloads with a large amount of unreferenced (i.e., not re-referenced) data, such as stg_1 in $VM2$, ECI-Cache allocates significantly smaller cache space (about 14,000 cache blocks) for caching referenced data while Centaur allocates about 1000X larger cache space to that VM. The allocated cache space by ECI-Cache for $VM5$, $VM6$, $VM7$, and $VM12$ in some cases becomes equal to the allocated cache space by Centaur (as shown in Fig. 11f, Fig. 11g, Fig. 11h, and Fig. 11m, respectively). This is because the maximum reuse distance of the workloads is mainly affected by RAR and RAW requests. In Fig. 11f, at $t = 600\ min$, there is a hit ratio drop which is recovered by increasing the allocated cache space by both ECI-Cache and Centaur. It can be seen that the allocated cache space by ECI-Cache is much smaller than the allocated space by Centaur. In addition, there is a hit ratio drop in $VM1$ at the first 50-minute interval where increasing the cache space using the Centaur scheme does not have any positive impact on the hit ratio. This is due to the lack of locality in references of the requests, which mostly include WAR and WAW operations. It can be seen that ECI-Cache does not increase the cache space at this time. At $t = 60\ min$, ECI-Cache increases the allocated cache space, which results in improving the hit ratio. ECI-Cache allocates the minimum cache space for rsrch_2 which is running on $VM14$ while Centaur allocates a much larger cache space (more than 50,000X) than ECI-Cache. This is because this workload mostly consists of WAR and WAW operations with poor locality of reference. Hence, ECI-Cache achieves the same hit ratio by allocating much smaller cache space to this workload. We conclude that in feasible state, both ECI-Cache and



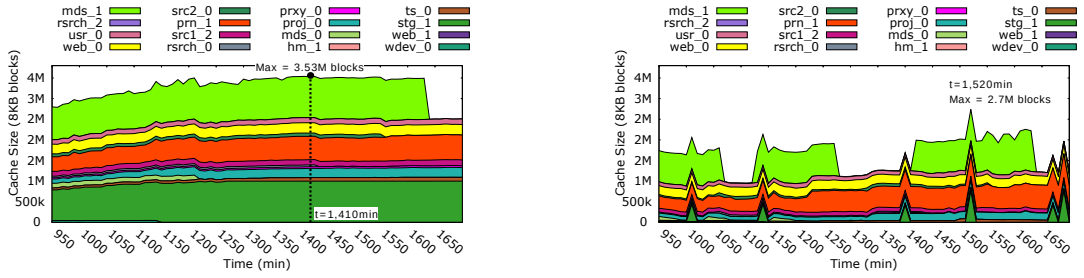
(a) VM0: wdev_0 (b) VM1: web_1 (c) VM2: stg_1 (d) VM3: ts_0

(e) VM4: hm_1 (f) VM5: mds_0 (g) VM6: proj_0 (h) VM7: prxy_0

(i) VM8: rsrch_0 (j) VM9: src1_2 (k) VM10: prn_1 (l) VM11: src2_0

(m) VM12: web_0 (n) VM13: usr_0 (o) VM14: rsrch_2 (p) VM15: mds_1

Figure 17: Allocated cache space of each VM and corresponding latency in feasible state with unlimited SSD cache capacity (ECI-Cache vs. Centaur).



(a) (b)

Figure 18: Cache allocation for the VMs in feasible state by (a) Centaur and (b) ECI-Cache.

Centaur achieve the same performance while ECI-Cache allocates *much smaller* cache space for the VMs compared to Centaur.

Fig. 18a and Fig. 18b show cache space allocation for the VMs in the time interval from $t = 950\ min$ to $t = 1,700\ min$ by Centaur and ECI-Cache, respectively. It can be seen that in all time intervals, ECI-Cache allocates much smaller cache space to the VMs compared to Centaur. We observe that in feasible state (i.e., unlimited SSD cache), ECI-Cache allocates much smaller cache space (29.45%, on average) than Centaur and hence reduces performance-per-cost significantly.

B URD OVERHEAD AND TIME INTERVAL TRADE-OFF

In this section, we report the average delay of calculating URD for the workloads running on the VMs in our experiments. We select the intervals of URD calculation in the experiments in order to limit the time overhead of URD calculation (the time that we

Table 3: Time overhead of URD calculation in the running VMs.

VM_{ID}	VM0	VM1	VM2	VM3	VM4	VM5	VM6	VM7	VM8	VM9	VM10	VM11	VM12	VM13	VM14	VM15
Workload	wdev_0	web_1	stg_1	ts_0	hm_1	mds_0	proj_0	prxy_0	rsrch_0	src1_2	prn_1	src2_0	web_0	usr_0	rsrch_2	mds_1
URD Calculation Time (s)	1.57	0.387	5.531	2.557	0.719	1.777	6.857	15.401	1.919	3.035	22.674	2.187	3.499	3.417	0.376	5.233

should wait until URD is calculated). Table 3 reports the average time it takes to calculate URD for each workload. As reported in the table, the maximum time it takes to calculate URD is about 22.67 seconds, which is reported for the prn_1 workload. In addition, the average time overhead of the URD calculation for all workloads is about 4.82 seconds. We selected 10 min time intervals which is calculated based on maximum URD calculation delay to reduce the time overhead of URD calculation to less than 5% in our experiments.

C WORST-CASE ANALYSIS

In this section, we provide three examples that illustrate corner cases where ECI-Cache fails in cache size estimation, and as a result, does *not* have a positive impact on performance improvement. We find that these cases are uncommon in the workloads we examine.

Case 1. *Sequential-Random workload*: The workload has two intervals: 1) sequential accesses followed by 2) random and repetitive requests (i.e., requests to the previously-accessed addresses). In the first interval, ECI-Cache does not allocate cache space for the workload while the requests in the second interval are random accesses to the previously (not buffered) accesses. Thus, in the first interval, ECI-Cache underestimates the allocated cache size for the workload. Fig. 19a shows an example of such workloads. We elaborate on how ECI-Cache works in each interval for the example workload:

(1) At the end of the first interval (i.e., when ECI-Cache recalculates URD and cache size), the URD of the workload is equal to 0. Hence, no cache space is allocated to this VM.
(2) In the second interval, the workload accesses become random with repetitive addresses where all requests are provided by the HDD and none of them are buffered in the cache (since no cache space is allocated to this VM).
(3) At the end of the second interval, the maximum URD of the workload is equal to three and hence ECI-Cache allocates cache space equal to four blocks for this VM.
(4) In the last interval, the workload issues two accesses to the storage subsystem but neither of them can be supplied by the cache (since the cache has no valid data) and the requests are buffered in the cache without any performance improvement.

We find that allocating cache space in such a manner (i.e., only at the end of the second interval) *cannot* improve the performance of requests of the last interval in this workload. In this case, the Centaur scheme works similar to ECI-Cache.

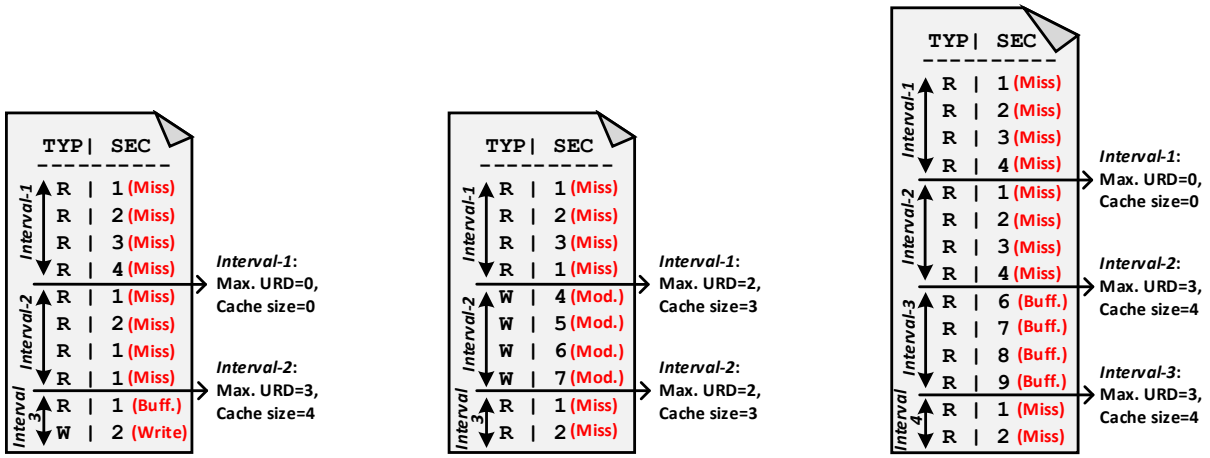
Case 2. *Random-Sequential workload*: In this case, in the first interval, the workload issues random accesses to the storage subsystem and ECI-Cache allocates cache space based on the URD of the requests. In the second interval, the accesses of the workload become sequential *without* any access to previously-buffered requests. ECI-Cache overestimates the allocated cache space for the workload. Fig. 19b shows an example workload for this case. We show how ECI-Cache overestimates the cache space for this example workload:

(1) In the first interval, the workload is random with locality of accesses and at the end of interval. The maximum URD is equal to two. Hence, at the end of the interval, three cache blocks are allocated to this VM.
(2) In the second interval, requests become sequential writes and are buffered in the allocated cache space.
(3) At the end of the second interval, the maximum URD of the workload is equal to two and hence ECI-Cache allocates three cache blocks for this workload.
(4) In the last interval, repetitive requests (that are the same as requests of the first interval) cannot be supplied by the cache since the entire cache space is used up by the accesses of the second interval (sequential writes).

We observe that although enough cache space is allocated for the VM, ECI-Cache cannot improve the performance of the workload compared to the HDD-based system due to the access behavior. In this case, Centaur works similar to ECI-Cache.

Case 3. *Semi-Sequential workload*: Such a workload includes similar sequential accesses in different intervals, which creates a large maximum URD without any locality of reference. In this case, ECI-Cache allocates a large cache space for the VM. Further requests use up the entire cache space without any read hit from allocated cache. Fig. 19c shows an example workload. We elaborate on how ECI-Cache allocates cache size for this example workload:

(1) At the end of the first interval, the maximum URD of the workload is 0 and no cache space is allocated to this VM.
(2) In the second interval, all accesses of the first interval are repeated and are provided by the HDD. Since we have no cache space, none of them are buffered in the cache.
(3) At the end of the second interval, the maximum URD of the workload is equal to three and ECI-Cache allocates four blocks to this VM.
(4) In the third interval, the workload issues read accesses to the storage subsystem and all of them are provided by the HDD (since the allocated cache has no valid data). In this case, these requests are buffered in the cache (all cache blocks are used).
(5) In the last interval, since cache space is used up by accesses of the third interval, the future requests that are identical to requests of the *second* interval miss in the cache and are supplied by the HDD.



(a) Sequential-Random workload: ECI-Cache underestimates cache size

(b) Random-Sequential workload: ECI-Cache overestimates cache size

(c) Semi-Sequential workload: ECI-Cache wastes cache space

Figure 19: Corner cases where ECI-Cache fails in size estimation (TYP: Type, SEC: Sector, W: Write, R: Read, Buff.: Buffer, and Mod.: Modify).

In this example, we observe that ECI-Cache allocates large cache space to the VM and buffers unnecessary data *without* any improvement in performance. We can resolve this problem by changing the length of the intervals.

D CONVEX OPTIMIZATION

In the following, we show why the objective function of the proposed algorithm (Eq. 4) is convex and then show how we solve the function using the MATLAB optimization toolbox. To do so, we first show that the objective function is convex. Then we show that the objective function and its constraints are in canonical form of convex.

$$\begin{cases} LatencyVM_i = h_i(c_i) \times T_{ssd} + (1 - h_i(c_i)) \times T_{hdd} \\ \text{Objective: } Minimize[\sum_{i=1}^{N} LatencyVM_i] \\ \text{Constraint1: } \sum_{i=1}^{N} c_i \leq C \\ \text{Constraint2: } 0 \leq c_i \leq c_{urd_i} \end{cases} \tag{4}$$

$LatencyVM_i$ is a linear function and is convex. The objective function is the sum of $LatencyVM_i$ functions because the sum of convex functions is convex. We express the first constraint of the objective function (constraint 1) as follows ($\mathcal{I}$ is a unity matrix of N by N):

$$\text{Constraint1: } \sum_{i=1}^{N} c_i \leq C \longrightarrow \mathcal{I} \times \begin{bmatrix} c_0 \\ .. \\ .. \\ c_{N-1} \end{bmatrix} \leq C \tag{5}$$

We express the second constraint (constraint 2) as:

$$\begin{gathered} \text{Constraint2: } 0 \leq c_i \leq c_{urd_i} \\ \longrightarrow \mathcal{I} \times \begin{bmatrix} c_0 \\ .. \\ .. \\ c_{N-1} \end{bmatrix} \leq \begin{bmatrix} c_{urd_0} \\ .. \\ .. \\ c_{urd_{N-1}} \end{bmatrix} \\ \text{and} \\ -\mathcal{I} \times \begin{bmatrix} c_0 \\ .. \\ .. \\ c_{N-1} \end{bmatrix} \leq \begin{bmatrix} 0 \\ .. \\ .. \\ 0 \end{bmatrix} \end{gathered} \tag{6}$$

Then, we have:

$$\begin{bmatrix} [1 & ... & 1] \\ \mathcal{I} \\ -\mathcal{I} \end{bmatrix} \times \begin{bmatrix} c_0 \\ .. \\ .. \\ c_{N-1} \end{bmatrix} \preceq \begin{bmatrix} C \\ \begin{bmatrix} c_0 \\ .. \\ .. \\ c_{N-1} \end{bmatrix} \\ \begin{bmatrix} 0 \\ .. \\ .. \\ 0 \end{bmatrix} \end{bmatrix} \tag{7}$$

Eq. 7 is the canonical form of convex and hence the objective function with constraints is convex. We employ MATLAB optimization toolboxes to minimize the objective function.

E DETAILS OF ECI-CACHE SIZE ESTIMATION ALGORITHM

In this section, we present Algorithm 4, which provides the details of Algorithm 1. Table 4 summarizes the key notational elements used in Algorithm 1 and Algorithm 4.

The main objective of Algorithm 4 (similar to Algorithm 1) is to find the most appropriate cache sizes for the running VMs such that the system is able to meet the following conditions: 1) Sum of allocated cache spaces for all VMs is less than or equal to the total SSD cache capacity. 2) Aggregate latency of all VMs

Algorithm 4: ECI-Cache size allocation algorithm (in more details).

```
/* Inputs: Number of VMs: (N), SSD cache size: (C), HDD Delay: (T_HDD), SSD Delay: (T_SSD)                    */
/* Output: Efficient cache size for each VM: (c_eff[1..N])                                                    */
Sleep for Δt
Extract the traces of the workloads running on the VMs including 1) destination address, 2) request size, and 3) request type
/* Here we estimate the efficient cache space for each VM by calculating URD of the running workloads.       */
for i = 1 to N do
    URD[i] = calculateURD(VM[i]) /* Here we find URD for each VM.                                             */
    size_urd[i] = calculateURDbasedSize(URD[i]) /* Here we calculate the estimated cache size for each VM based on its URD. */
    csum+ = size_urd[i] /* We keep the sum of estimated sizes in csum.                                       */
end
/* In the following we check the feasibility of size estimation and minimize overall latency for estimated size_urd[1..N] */
if csum ≤ C then
    /* If this condition is met, our estimation is feasible                                                   */
    c_eff[1..N] = size_urd[1..N] /* We assign the estimated sizes to the efficient sizes.                     */
end
else if csum > C then
    /* If this condition is met, the estimation is infeasible.                                                */
    Update hit ratio function of VMs (H_i(c)) based on updated reuse distance of the workloads. /* The structure of hit ratio function is provided in Algorithm 2. */
    c_eff[1..N] = calculateEffSize(size_urd[1..N], C) /* We call calculateEffSize to find the efficient sizes that fit in total SSD cache space. */
end
allocate(c_eff[1..N], VM[1..N]) /* This function allocates the calculated efficient cache spaces for each VM. */
/*
   Function Declarations:
   calculateURD                                                                                               */
Function calculateURD(VM) is
    /* The purpose of this function is to find the URD of the running workload on VM. This function calls PARDA [38], which is modified to calculate URD (reuse
       distance only for RAR and RAW requests).                                                               */
    return URD
end
/*
   calculateURDbasedSize                                                                                      */
Function calculateURDbasedSize(URD) is
    /* The purpose of this function is to calculate URD based cache size of each VM.                          */
    size_urd = URD × cacheBlkSize
    return size_urd
end
/*
   calculateEffSize                                                                                           */
Function calculateEffSize(size_urd[1..N], C) is
    /* This function is called in infeasible states and aims to minimize the overall latency (sum of VMs latencies). */
    initialSize = {c_min, ..., c_min}/* We set c_min as the initial cache space for each VM.                  */
    lowerBound = {c_min, ..., c_min} /* Here we set the minimum cache space for each VM equal to c_min        */
    upperBound = {size_urd[1], ..., size_urd[N]} /* Here the maximum cache space for each VM is set.          */
    weightVM = {1, ..., 1} /* We assume that the VMs are weighted identically.                              */
    /* All abovementioned variables are the inputs of the fmincon function. We pass ObjectiveFunction to this function. */
    c_eff[1..N] = fmincon(ObjectiveFunction, initialSize, weightVM, C_tot, {}, {}, lowerBound, upperBound)
    return c_eff[1..N]
end
/*
   ObjectiveFunction                                                                                          */
Function ObjectiveFunction() is
    /* This function is called until the condition of Eq. 2 is met by the estimated cache sizes (c[1..N]).   */
    for i = 1 to N do
        h[i] = H_i(c[i]) /* Here we calculate the hit ratio of each VM for c[i] input. This function is updated in Δt intervals. */
        hsum+ = h[i] /* hsum variable keeps the sum of hit ratios of the VMs                                  */
        csum+ = c[i] /* csum variable keeps the sum of cache sizes of the VMs                                 */
    end
    diff = C − csum /* diff variable is used in maximizing the total estimated cache sizes                   */
    Obj = diff + (hsum) × T_SSD + (N − hsum) × T_HDD /* Objective: Maximizing estimated sizes and minimizing sum of VM latencies. */
    return Obj
end
```

is minimum. 3) Allocated cache space for each VM is less than or equal to the estimated cache space by URD. This objective is obtained by the *calculateEffSize* function via the use of the *ObjectiveFunction*. *ObjectiveFunction* minimizes $diff$ which is the difference between sum of allocated cache spaces for the VMs ($csum$) and the SSD cache capacity (C) (i.e., assigns the maximum cache space for each VM). In addition, *ObjectiveFunction* minimizes the aggregate latency of the running VMs.

Table 4: Description of notation used in Algorithm 1 and Algorithm 4.

Variables	
Notation	Description
N	Number of running VMs.
C	SSD cache capacity.
T_{HDD}	HDD delay (i.e., HDD service time).
T_{SSD}	SSD delay (i.e., SSD service time).
$C_{eff}[i]$	Efficient cache size allocated for each VMi.
URD_i	Useful Reuse Distance (URD) for VM_i, which is the output of $calculateURD()$ function.
$Size_{URD}[i]$	Initial efficient cache size suggested by URD for VM_i.
$csum$	Sum of initial efficient cache sizes (i.e., sum of $Size_{URD}[i]$ of all running VMs).
$cacheBlkSize$	Size of cache blocks (equal to 8KB in our experiments).
$initialSize$	Array of initial cache sizes allocated to each VM.
$lowerBound$	Array of minimum cache sizes that can be assigned to the VMs.
$upperBound$	Array of efficient cache sizes suggested by URD. The efficient cache sizes by ECI-Cache for each VM is less than or equal to the suggested cache sizes by URD.
$weightVM$	Weight of VMs.
$h[i]$	Hit ratio of VM_i.
$hsum$	Sum of hit ratio of VMs.
$diff$	Difference between total SSD cache capacity and estimated cache sizes by ECI-Cache.
Obj	The objective variable.
Functions	
$calculateURD$	Input: VM
	Output: Useful Reuse Distance (URD) of each VM.
$calculateURDbasedSize$	Input: URD
	Output: efficient cache size suggested by URD ($size_{URD}$)
$calculateEffSize$	Input: $size_{URD}[1..N]$, C // This function is called only in infeasible states where the existing SSD cache size is less than required cache sizes by VMs.
	Output: efficient cache size for each VM ($c_{eff}[i]$)
$objectiveFunction$	Input: — // This function is used within the $calculateEffSize$ function.