S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance

Tian Luo[1] Siyuan Ma[1] Rubao Lee[1] Xiaodong Zhang[1] Deng Liu[2] Li Zhou[3$]

[1]The Ohio State University
Columbus, OH
{luot, masi, liru, zhang}@cse.ohio-state.edu

[2]VMware Inc.
Palo Alto, CA
liud@vmware.com

[3]Facebook Inc.
Menlo Park, CA
lzhou@facebook.com

***Abstract*—A unique challenge for SSD storage caching management in a virtual machine (VM) environment is to accomplish the dual objectives: maximizing utilization of shared SSD cache devices and ensuring performance isolation among VMs. In this paper, we present our design and implementation of S-CAVE, a hypervisor-based SSD caching facility, which effectively manages a storage cache in a Multi-VM environment by collecting and exploiting runtime information from both VMs and storage devices. Due to a hypervisor's unique position between VMs and hardware resources, S-CAVE does not require any modification to guest OSes, user applications, or the underlying storage system. A critical issue to address in S-CAVE is how to allocate limited and shared SSD cache space among multiple VMs to achieve the dual goals. This is accomplished in two steps. First, we propose an effective metric to determine the demand for SSD cache space of each VM. Next, by incorporating this cache demand information into a dynamic control mechanism, S-CAVE is able to efficiently provide a fair share of cache space to each VM while achieving the goal of best utilizing the shared SSD cache device. In accordance with the constraints of all the functionalities of a hypervisor, S-CAVE incurs minimum overhead in both memory space and computing time. We have implemented S-CAVE in vSphere ESX, a widely used commercial hypervisor from VMWare. Our extensive experiments have shown its strong effectiveness for various data-intensive applications.**

***Index Terms*—Performance, storage, I/O, SSD, cache, virtual machine**

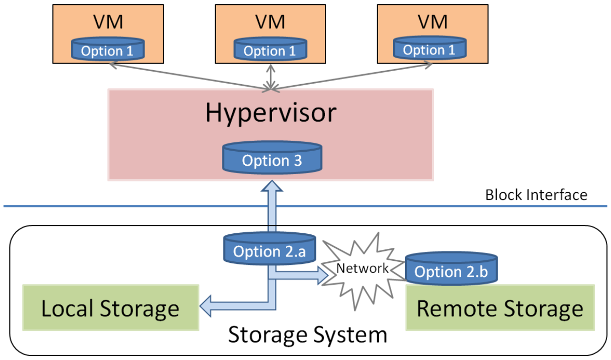
I. Introduction

In the era of big data, cloud computing and booming of data centers, virtual machine (VM) systems have become a basic supporting infrastructure. In a typical VM environment, multiple virtual machines with different operating systems and applications can simultaneously execute on the same host [1]. There are two goals in the design and management of a VM system: (1) providing system isolation so that each VM is limited to its own virtual resources, and (2) delivering high performance for each individual VM and high resource utilization for the overall system.

Bridging the speed gap between main memory and storage devices, e.g. hard disk drives (HDDs), in a VM environment is critical to the productivity of the entire system and performance of its client VMs. With the rapid advancement of flash-based solid-state drives (SSDs), SSD-based storage caching has been widely studied in conventional systems [2]–[4]. However, it would be impractical to directly borrow existing solutions in VM systems, because a SSD caching design in VM systems must be consistent with the two above mentioned goals. To achieve the goal of high resource utilization, each VM should be granted an appropriate amount of SSD cache space that should be adaptive to runtime dynamics. To achieve the goal of performance isolation, the effective SSD caching space of each VM should not be interfered by other VMs running on the same system.

According to the hierarchy of a VM system (shown in Figure 1), there are three options for SSD caching management, namely, Option 1: VM-based SSD caching; Option 2: storage system-level SSD caching; and Option 3: hypervisor-based SSD caching. Based on our thorough analysis of these options (in Section II), we argue that hypervisor-based SSD caching (Option 3) is the best choice. Therefore in this paper, we present S-CAVE, a hypervisor-based SSD caching facility, which effectively manages a shared SSD cache device for multiple VMs. We have implemented and tested S-CAVE in a commercial hypervisor from VMware.

Fig. 1: Design Options of SSD Caching in a VM environment



A. *Outline of Our Solution: S-CAVE*

We take the following steps to design S-CAVE so that it well fits in a VM system.

First, for each running VM, we use a light-weight module to monitor its I/O activities and manage its cache space in a transparent way, which means we do not make any modification to guest OSes or user applications.

Next, we propose a metric called "rECS" (simplified for *ratio of effective cache space*) to identify the cache space demand of each VM at runtime. It is a critical reference for cache space allocation. In essence, rECS is the ratio of the

[$] Li Zhou worked in this project while he was working for VMware Inc.

cache space that is being effectively used by a VM to the total cache space that has been allocated to this VM. According to our analysis and quantitative experiments, rECS can identify VM's cache space demand at a high accuracy.

Furthermore, a VM environment is highly dynamic, where multiple VMs with different and changing cache demands are running on the same host. To respond runtime dynamics and guarantee effective and fair sharing of cache space, we design a dynamic control mechanism to periodically cross-compare the cache demand of each VM, and adjust space allocation accordingly: VMs with an increased demand will be granted more cache space, while VMs with an decreased demand will be deprived of a portion of its already allocated cache space. In order to prevent incorrect adjustments, historic decisions will be taken into account as a feedback when a new decision is being made.

B. *Technical Challenges to Build S-CAVE*

We have mainly addressed three technical challenges in the design and implementation of S-CAVE.

- **Effective cache space allocation.** Our system design must be able to effectively allocate cache space among multiple VMs in order to achieve the two goals: (1) guarantee performance isolation among VMs; (2) achieve high performance for each individual VM and maximize an overall resource utilization,
- **Highly responsive to runtime dynamics.** Unlike conventional systems, a VM environment has multiple VMs sharing the same set of hardware resources with significantly high dynamics. Therefore, our design must be adaptive to the fast changing demands of VMs: space allocation decisions should be made dynamically and executed efficiently.
- **Low system overhead.** As a hypervisor-based caching solution, S-CAVE resides on the host and share the same set of hardware resources with VMs. Thus, it is critical to minimize its cost (e.g. memory footprint and computing time).

C. *Contributions*

This paper makes the following contributions. (1) We have identified and studied several unique management challenges of SSD caching in a VM environment in contrast to a conventional system. (2) We have proposed and verified a metric called rECS to identify the SSD cache space demand of each VM. (3) We have designed S-CAVE, a hypervisor-managed SSD cache, to support a multi-VM environment. We have implemented S-CAVE in VMWare vSphere ESX [5], a widely used commercial hypervisor. Experiment results have shown strong effectiveness of our design.

The rest of the paper is organized as follows. Section II studies the design choices of SSD caching in a VM environment. Section III presents the architecture of S-CAVE. Section IV introduces metric rECS. Section V describes the mechanism to allocate cache space among multiple VMs. Section VI presents our evaluation results. Section VII discusses related work. Section VIII concludes this paper.

II. Why is Hypervisor-based Caching Most Effective?

Figure 1 shows the hierarchy of a typical VM system, where we present three options to design and implement an SSD cache device. The first two are adoptions of existing approaches, and S-CAVE belongs to the last option.

A. *Option 1: VM-based SSD Caching*

Since a VM can request a virtual SSD (a fixed part of a physical SSD) in the same way as it can request a virtual HDD, it could directly use a virtual SSD as a caching device (Option 1 in Figure 1). This option has two advantages: (1) A VM with such a virtual SSD cache is expected to gain the best performance per given SSD capacity, because it has the best knowledge to make placement decisions and has a full control of its virtual devices. (2) It guarantees system isolation among VMs since I/O activities of one VM will not affect the cache size or cached data of another VM.

However, this option has two significant disadvantages. (1) Guest OSes and/or user applications must be modified to manage the cache device. It incurs burdens on users, and is hardly possible for legacy systems. (2) The size of a virtual SSD cache cannot be dynamically adjusted to respond the changing demands of a VM, thus unable to maximize utilization of storage resources. Although for some other resources, like the main memory, they could be dynamically and transparently allocated among VMs, while each running VM has the illusion that its total resource amount is constant to pre-configured parameters, it is impractical to accomplish the same effect for storage devices. For example, in order to maintain the illusion of constant capacity for a virtual SSD when we want to shrink its capacity at runtime, data stored on that virtual SSD has to be moved to another storage device. This has to be done in the background without affecting normal VM activities. However, data movement between two storage devices would incur a prohibitively high cost. Therefore, this option is essentially a static-partitioning approach.

B. *Option 2: Storage System-level SSD Caching*

In this option, SSD caching is managed as a part of the storage system. Since a storage system can be either locally attached or remotely connected, the SSD cache could also be either local on the host side (Option 2.a in Figure 1), or remote in a storage server (Option 2.b in Figure 1). In both cases, the SSD cache is managed within the storage sub-system. This option has two advantages. (1) It is an easy-to-use plug-in solution. (2) Improvement within the storage system is of general purpose and can be achieved without an involvement of VMs or the hypervisor.

However, this option has several drawbacks due to its isolated design approach. (1) The overall VM system performance can be sub-optimal. Because the block interface between a storage system and the virtualization software stack is primitive without the ability to deliver rich semantic information [4], a local optimization in the storage sub-system may not be translated into the overall VM system performance. (2) Since the management between the VM system and the storage system is disconnected, performance isolation cannot be guaranteed because of I/O interferences among different VMs may occur.

C. Option 3: Hypervisor-based SSD Caching

In this option, SSD caching is directly managed by the hypervisor, which is the management center layered between VMs and hardware resources (Option 3 in Figure 1). This is the option to be focused in this paper, which can address all the limitations in Options 1 and 2, but retain their advantages. We propose a hypervisor-based SSD caching solution for the following benefits. (1) A hypervisor can manage SSD cache for VMs in a transparent way. This addresses the concern of modifying guest OSes or applications. (2) Since most VM activities, particularly all I/O requests, will go through the hypervisor, this gives it a unique advantage to collect critical information for effective SSD cache management. (3) With a full access privilege to hardware resources, a hypervisor could directly enforce space allocation decisions to maximize resource utilization in an efficient way.

However, these benefits cannot be achieved without a proper management scheme. For example, a hypervisor can also manage SSD cache space in a naive way, such as using a general-purpose cache management algorithm. While this approach is easy for an implementation, our study will show that it cannot achieve the above mentioned benefits. In the following section, we will present how S-CAVE manages a shared SSD cache in VM systems to gain all the benefits.

III. Architecture of S-CAVE

Figure 2a gives a high-level illustration to the architecture of S-CAVE. For each VM, S-CAVE launches a module, called Cache Monitor, to manage its allocated cache space and keep the cache transparent to the VM. In order to effectively allocate the shared SSD cache space among multiple VMs, we use a central control, called Cache Space Allocator, to collect and analyze cache usage information from each cache monitor and make cache allocation decisions accordingly.

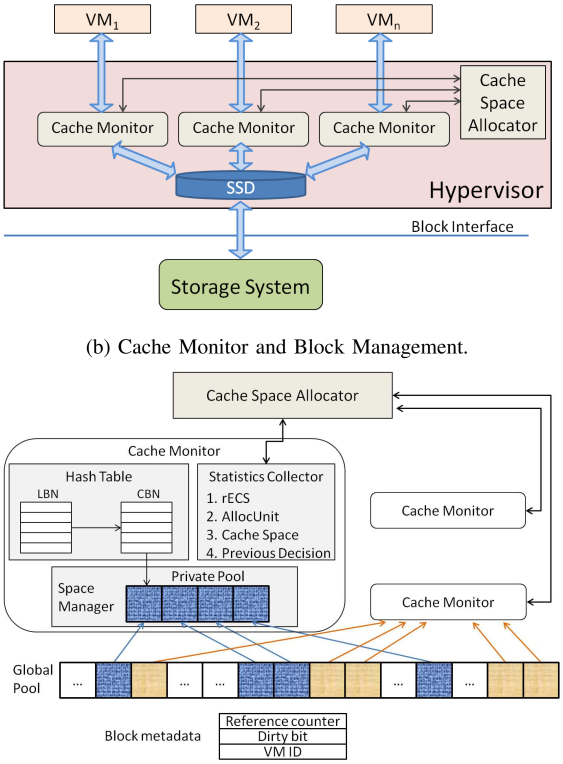
Figure 2b gives a more detailed view about the design of cache monitor and block management. The cache is managed in the granularity of fixed-size blocks[1]. We define "global pool" as a set of all blocks from the cache device, and "private pool" as a set of blocks allocated to one VM. Each block is associated with three metadata fields: reference counter, dirty bit and VM ID. The reference counter records the number of hits on a block, and is reset to zero when the block is reloaded with new data. The dirty bit is set by writes. The VM ID marks the ownership of a block. All blocks in a private pool have the same VM ID, and a free block has a "null" VM ID.

In the following, we will first describe the design of the cache monitor, and then the cache space allocator.

For every VM, we use a cache monitor to manage its allocated cache space. This module consists of 3 components (Figure 2b): one hash table, one space manager, and one statistics collector. (1) The hash table is used to look up blocks in the cache device. Every cached block corresponds to one hash entry, which is in the form of $\{LBN, CBN\}$. LBN, or logical block number, is the original address of the block. CBN, or cached block number, is the block's corresponding position in the cache device. (2) The space manager manages blocks in the VM's private pool. Because the size of each private pool could be dynamically changed, in additional to normal operations such as cache insertion and eviction, the space manager has more responsibilities: when more cache space is granted, it needs to expand the private pool to its new size by reclaiming free blocks from the global pool. And if cache space is reduced, it needs to timely return the required amount of space back into the global pool by freeing a certain number of cached blocks. (3) The statistics collector observes the I/O activities and usage status of its private pool, and updates metrics including rECS, AllocUnit, as well as other bookkeeping information. These metrics and information items play important roles in cache space allocation, and will be elaborated in later sections.

Fig. 2: Design of S-CAVE.

(a) An Illustration of the S-CAVE Architecture.



(b) Cache Monitor and Block Management.

The Cache Space Allocator plays a central control role, and is responsible for making cache allocation decisions among multiple VMs, based on information from the statistics collector of each VM. (The process of decision making will be described in more details in Section V.) In order to adapt to runtime dynamics, decisions are made periodically at fixed time intervals. Each interval is called a time window.

The key for S-CAVE to be effective is to allocate the appropriate amount of cache space to each VM. This challenge can be addressed in two steps.

1) Identifying the cache space demand of each VM (Section IV).
2) Making cache space allocation decisions by considering the demands of all VMs (Section V).

[1] While the basic block size could be different, such as 512B or 1MB, we use 4KB in this paper.

IV. Cache Demand of One VM

In order for the cache space allocator to make right decisions for each VM, it is required that each cache monitor provide accurate information about the SSD cache space demand of its monitored VM.

Previous works on shared space paritioning have studied how to identify the space demand of each application. For example, some research at CPU cache level has proposed to monitor the number of hits/misses of each cache unit and use the numbers as a basis to compute the space demand [6]. Some other research at main memory level has proposed to use the change of hit ratio in a time window as a metric to predict the space demand [7], [8].

However, the above hit ratio-related techniques cannot be applied to a shared storage cache. First, due to the filtering effect of higher-level caches, e.g. CPU cache and memory cache, a storage cache observes a much weaker locality. In addition, combined with the factor of a lower cost for storage medium, a storage cache normally has a much larger capacity, which causes a further diminishing benefit in and sensitivity to hit ratios [9]. Therefore, we must find a suitable metric to identify each VM's space demand for a shared storage cache device.

A. rECS: Ratio of Effective Cache Space

For the above reasons, we propose a metric called $rECS$ (*ratio of Effective Cache Space*). It is a ratio of the size of cache space that is being effectively used to the total size of cache space that has been allocated.

To be specific, assume for a VM, the total allocated cache space contains N blocks, and within a time window, m unique cached blocks have been accessed (cache hits), then its $rECS$ value during this time window is $m/N \times 100\%$.

Compared with previous approaches, rECS is a metric that combines the benefits of both ends: low overhead and high accuracy. In the following, we will first describe how we can obtain rECS values in an efficient way (Section IV-B). And then , we will study why rECS could accurately describe a VM's demand for SSD cache space (Section IV-C).

B. Runtime Estimation of rECS

Estimating the value of rECS at runtime is related to the choice of cache replacement algorithms, most of which are LRU and its variants. These algorithms can be classified into two categories: linked-list based and array based. An faithful implementation of LRU or its improved versions such as LIRS [10] requires a linked list. Other variants like CLOCK [11], [12] are array based.

We have chosen CLOCK to be the cache replacement algorithm in our system due to its low lock contention and low space overhead, which are important factors in a VM environment[2]. Specifically, every space manager maintains its own clock to manage its private pool.

To compute rECS values at runtime, we set up two counters for each VM: $N^{all}_{VM_i}$ and $N^{e}_{VM_i}$, where $N^{all}_{VM_i}$ is the total number of cache blocks allocated to VM_i, and $N^{e}_{VM_i}$ is used to count the number of unique cache blocks being effectively used by VM_i. At the beginning of a time window, both counters are set to 0, and a scanner will be started to scan the metadata of every block in the global pool for its reference counter. Since the global pool contains blocks for different VMs, when encountering a block for VM_i, only its corresponding counter ($N^{all}_{VM_i}$) should be incremented by one. If the reference counter is greater than 0, then $N^{e}_{VM_i}$ will be incremented by one as well.

In order to obtain accurate rECS values, we need to finish scanning all reference counters within a time window. Therefore, we are faced with a tradeoff between accuracy and efficiency. A small time window enables responsive cache allocation adjustments, but making it impractical to complete a full scan on time. A large time window allows the scan to finish on time, but making cache allocation less responsive to runtime dynamics.

In order to estimate rECS values efficiently while keeping a high accuray, we adopt a sampling technique. Every time window is split into multiple small sampling periods, and interleaved by idle periods to minimize computation overhead. Within each sampling period, scanning starts from a random block, and stops when the sampling period times out. The final $rECS$ value for a time window is the average of values from all sampling periods. To further minimize the loss of accuracy, we count in previous $rECS$ values with a smoothing parameter α, so that $rECS = rECS_{old} \times \alpha + rECS_{sampled} \times (1 - \alpha)$. With sampling, we could obtain rECS values efficiently and accurately, thus enabling us to choose a small time window fore more responsive space allocation adjustments[3].

C. Insights into rECS

In this section, we present the insights into rECS, and analyze the benefits of using rECS instead of a hit-ratio-based metric for cache space allocation, especially when the two metrics give different suggestions.

First, we define the following notations:

- ΔE: change of rECS in the past time window.
- ΔH: change of hit ratio in the past time window.

For a running VM, its ΔE and ΔH may change in different directions, resulting in four different combinations. In Figure 3, we illustrate these combinations with four quadrants. In two of these quadrants, ΔE and ΔH give the same cache allocation suggestions. In the other two quadrants, they give different suggestions.

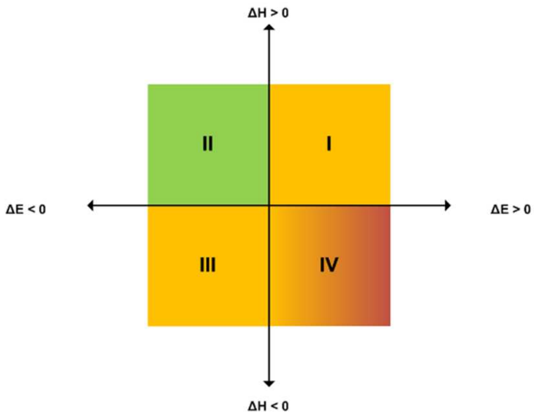
Quadrant I and III: In this case, ΔE and ΔH change in the same direction, so they would give the same cache allocation suggestions.

Quadrant II: In this case, the hit ratio of a VM is reported to have increased while its rECS is reported to have dropped in the latest time window. Combining the two measurements, we would characterize the changing access pattern in the following

[2] A comparison of these algorithms is out of the scope of this paper.

[3] In our system, we have set the length of each time window to be 1 second, consisting of 5 sampling periods, each of which is 0.1 second, interleaved by idle periods. Parameter α is set to be 80%.

Fig. 3: Four Combinations of ΔE and ΔH



way: *the increased data accesses hit on a decreased size of data set.* Thus, the decision made by rECS, which is to decrease the cache size of this VM, could effectively respond this access pattern.

Quadrant IV: In this case, the hit ratio of a VM is reported to have dropped while its rECS is reported to have increased. The combined measurement results characterize another unique access pattern: *the decreased data accesses hit on an increased size of data set.* This reflects an access locality change shifting from accesses on a concentrated region to a larger region with less frequent accesses on average. For example, the VM is loading a new data set that starts to exhibit locality in a new execution phase. Thus, the decision made by rECS, which is to to increase the cache size of this VM, could effectively cope with this access pattern.

As hit ratio changes become less sensitive in weak-locality accessing layers, such as an SSD cache for storage I/O requests, the rECS metric shows its unique strength by effectively identifying the cache space demand of each VM. Our experiment results (Section VI) well support the decisions made by rECS on various access patterns including the patterns characterized by Quadrants II and IV.

V. Cache Space Allocation Among VMs

While rECS can identify the cache demand of a single VM, an effective cache space allocation solution needs to consider and balance the demands of all VMs and hardware resources. In this section, we will first propose our cache space allocation scheme (Algorithm 1), and then present details including space management, write policy and finally discuss related overhead.

A. Cache Space Allocation

Algorithm 1 describes the process of how an allocation decision is made. In the beginning (Line 2 - 5), it checks the number of running VMs. If only one VM is running, this VM will be granted all the cache space. Then (Line 7 - 10), it checks if there is still enough free space (5% threshold, more details in Section V-B). It is not necessary to enforce allocation control until free space becomes limited. Then (Line 12 - 16), the $rECS$ value (E) of each VM is calculated and based on which, we get a $Score$ for each VM. Next (Line 18 - 21), we should find the VM with the highest $score$ (VM_{max}) to increase its cache space, and record the decision as “increase”. The amount to increase is determined by parameter $AllocUnit(VM_{max})$. In the following (Line 23 - 32), we find the VM with the lowest score (VM_{min}). If this VM has already suffered from a large number[4] of decisions that continuously reduce its cache space, we need to test the decision by giving back some space, and record the decision as “increase”. Otherwise, we can reduce its cache space and record the decision as “decrease”, and increment the counter $DCounter$ by one. The last code segment (Line 34 - 39) makes sure that we do not over-allocate.

Algorithm 1 Cache Space Allocation

```
1:  //Assuming totally N running VMs
2:  if $N == 1$ then
3:      $Cache(VM_0) = \infty$
4:      exit
5:  end if
6:
7:  if Total free space $> 5\%$ then
8:      For each VM: $Cache(VM_i) = \infty$
9:      exit
10: end if
11:
12: for each $VM_i$ $(1 \leq i \leq N)$ do
13:     $E(VM_i) = E_{old}(VM_i) * a + E_{sampled}(VM_i) * (1 - a)$
14:     $Score(VM_i) = E(VM_i) - E_{old}(VM_i)$
15:     $E_{old}(VM_i) = E(VM_i)$
16: end for
17:
18: $Score(VM_{max}) = MAX\{Score(VM_i) | 1 \leq i \leq N\}$
19: $Cache(VM_{max}) += AllocUnit(VM_{max})$
20: $Decision(VM_{max}) = “increase''$
21: $DCounter(VM_{max}) = 0$
22:
23: $Score(VM_{min}) = MIN\{Score(VM_i) | 1 \leq i \leq N\}$
24: if $DCounter(VM_{min}) > m$ then
25:     $Cache(VM_{min}) += AllocUnit(VM_{min}) * 2$
26:     $Decision(VM_{min}) = “increase''$
27:     $DCounter(VM_{min}) = 0$
28: else
29:     $Cache(VM_{min}) -= AllocUnit(VM_{min})$
30:     $Decision(VM_{min}) = “decrease''$
31:     $DCounter(VM_{min}) += 1$
32: end if
33:
34: $Deficit = \sum_1^N Cache(VM_i) - TotalCacheSize \times 95\%$
35: if $Deficit > 0$ then
36:     for each $VM_i$ $(1 \leq i \leq N)$ do
37:         $Cache(VM_i) -= Deficit/N$
38:     end for
39: end if
```

Allocation Unit: Each time the algorithm adjusts the cache space for a VM, the amount of change is determined by parameter $AllocUnit(VM_i)$, which is the average number of blocks VM_i could access from hard drives within a time window, and obtained by counting the average number of

[4] In our system, we use a threshold m, which is set to 5.

accesses in recent time windows. In other words, $AllocUnit$ is essentially the recent throughput of missed I/Os for a VM. The purpose of this parameter is to ensure that new data to be accessed by VM_{max} in the next time window can be accommodated with enough new space, and in the meantime, VM_{min} is deprived of a proper amount of cache space.

B. Cache Space Management

Having studied how to allocate cache space among multiple VMs, we are now in a position to discuss how to manage cache space and how to enforce an allocation decision for each VM. As shown in Figure 2, the cache device is managed in the granularity of blocks. If the cache space of a VM is decreased, its space manager will start to forcefully evict the coldest blocks. Evicted blocks will be marked "free" by setting their VM_ID to null. If the cache space of a VM is increased, then upon receiving new data to cache, its space manager will start to check the global pool for free blocks, whose VM IDs are "null". Once a free block is found, it will claim it by setting the VM_ID metadata field to its VM ID and save new data into this block.

It is critical for VMs with an increased SSD cache space to find free blocks timely. Therefore, we reserve a small portion of blocks as free space. This can be achieved with a global cleaner that scans the global pool, and reclaims blocks whose reference counter is 0, regardless of its owner VM. Therefore, free blocks are scattered in the global pool.

The global cleaner works according to two watermarks: a high watermark, W_{high}, and a low watermark, W_{low}. Once the amount of free blocks drops below W_{low}, the global cleaner is launched to reclaim data blocks, and stop working when the amount of free blocks reaches W_{high}. In our experiments, W_{high} is set at 5%, and W_{low} is set at 1%.

C. Write Policy

There are generally 3 policies to treat writes in a cache device: write-back, write-through or write-invalidate. While write-back and write-invalidate may gain some performance benefits, because they reduce the number of writes to slower HDDs, it is most important to ensure data consistency, particularly in case of a device failure or power failure. Therefore, we use write-through as the writeh policy in S-CAVE. To be specific, if a block accessed by a write request exists in the cache, the cached version and the original block on an HDD will both be updated simultaneously. If a block accessed by a write request does not exist in the cache, the cache device will be bypassed by this request. Solutions from mainstream storage vendors, such as NetApp, have also chosn the write-through [13] policy. A comparative study of these write policies would be out of the scope of this paper.

D. Overhead of S-CAVE

The overhead of S-CAVE mainly comes from two parts: memory footprint and some processing time.

1) Memory Footprint: One source of memory footprint is the hash tables, one for each VM, as illustrated in Figure 2b. The total size of all hash tables is proportional to the SSD cache capacity, because only a cached block could have one corresponding hash entry. The key of each hash entry, LBN, is the original storage offset. We store it with 32 bits. In the granularity of 4KB, it can address a storage device of 128 TB, which is reasonably big. For CBN, we use the same length.

Another source of memory overhead is from the metadata for each block. As shown in Figure 2b, every block has three metadata fields: reference counter (4 bits)[5], dirty bit (1 bit), and VM_ID (4 bits). Therefore, the total memory footprint for each 4KB cache block is 73 bits, with 64 bits for an hash entry, and 9 bits for block metadata. Thus the total space overhead is 0.23%. Other data structures, such as those in the statistics collector (Figure 2b), incur a negligible and constant space overhead for each VM.

2) Computation Overhead: One source of computation overhead is from the process to estimate rECS values at runtime. This has been minimized with sampling.

Another source of computation overhead is resulted from hash table lookups. For each I/O request, a hash table lookup will be invoked to determine if the requested block is already cached. This operation incurs some latency.

This latency can be decomposed into two parts: (1) locking: each bucket in a hash table has a lock to protect its linked-list of hash entries. (2) searching: within a bucket, we need to execute linear search of hash entries. If an entry with the desired key (LBN) is found, then it is returned and search ends. Otherwise, we need to check until the end of the list to conclude that a block is not cached.

In our design, locking overhead is minimized by partitioning each hash table into multiple sub-hash tables, and by increasing the number of buckets in each sub-hash table, so there would be less conflicts to the same bucket. For searching-related cost, we organize the linked list of each bucket with LRU, and move the most recently used block to the head of its list, in order to minimize the search cost for a cache hit. To further reduce the cost for a cache miss, we are considering extra data structures, such as a bloom filter, like in [14] or [15].

VI. Performance Evaluation

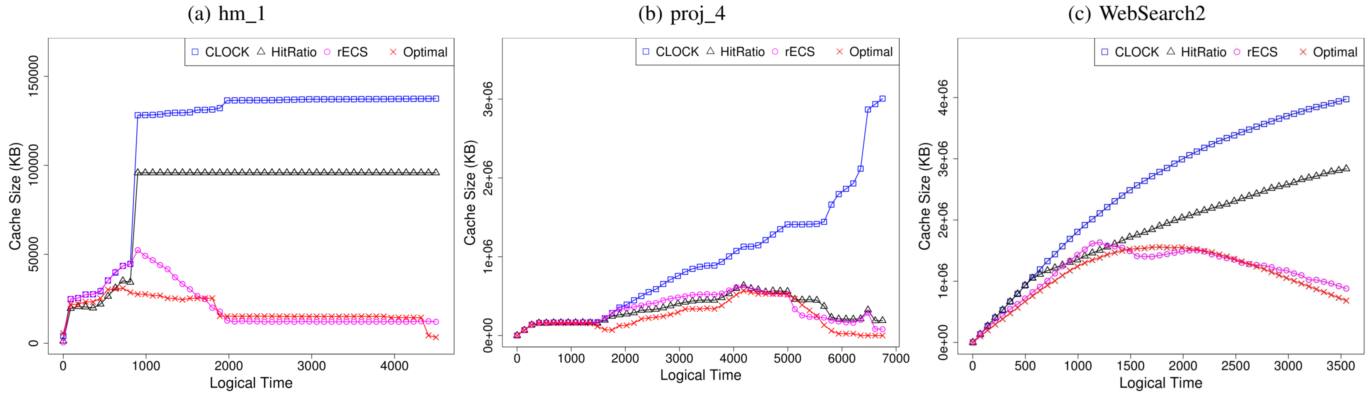
In this section, we will first demonstrate the effectiveness of metric rECS to identify the cache space demand of a single VM, and then present results of S-CAVE when managing a multi-VM environment.

A. Experimental setup

We have implemented S-CAVE in vSphere ESX [5], a widely used commercial hypervisor from VMware. In our experiments, this enhanced hypervisor was deployed in a HP Proliant 360 G5 server, which has a quad core Intel®Xeon®X5570 processor, 12 GB of main memory, two Seagate 1TB hard with a speed of 15.7K rpm, In additions, we have used one Intel®910 SSD as the caching device. Key performance parameters of this SSD are shown in Table I.

[5] We do not need an accurate counter for cache hits. 4 bits are sufficient to differentiate hot blocks from cold blocks.

Fig. 4: Effectiveness of Metric rECS. X-axis: Logical Time, at the unit of time window. For this experiment, each time window contains 128 requests. Y-axis: Allocated Cache Size.



(a) hm_1

(b) proj_4

(c) WebSearch2

TABLE I: Key performance numbers of Intel SSD 910

Sequential Read/Write	Random Read/Write
2000/1000 MBps	180K/75K IOPS

B. Workloads

We have selected 7 workloads from three sources: two real-world trace repositories and one benchmark: (1) SNIA IOTTA Repository [16], from which we have selected 4 MSR Cambridge Traces: hm_1, proj_3, proj_4, mds_1. (2) UMass Trace Repository [17], from which we have selected WebSearch_2, trace from a search engine. (3) TPC-H [18], which is a benchmark for warehouse-style workloads which have intensive I/O activities. We have configured this benchmark at the scale factor of 10, and collected traces from two longest running queries, tpch_q9, tpch_q21. Due to our choice of "write through" as the write policy (Section V-C), each write request is always sent to both the SSD cache and the underlying HDD. thus its latency is determined by the slower device. In this sense, the performance of a write-intensive workload is irrelevant to SSD caching. Therefore, the chosen workloads are read intensive. We have conducted all the following experiments with trace replay.

C. Effectiveness of Metric rECS

In this section, we demonstrate that the metric rECS can effectively identify the cache demand of a VM, so that based on rECS, we could achieve nearly optimal cache space allocation.

For this purpose, we use one VM to run a single workload each time. The VM is started with a small cache size, which will then be dynamically adjusted at runtime. This setup is different from a multi-VM scenario in that cache allocation decisions are not made by cross-comparing the demands of different VMs. The decision solely depends on the VM's own behavior: cache space is increased if its rECS value increases, and reduced if its rECS value decreases. During the process, we still use historic decisions as a feedback to prevent possible incorrect decisions. Another distinction of this setup is that a time window is determined by the number of requests (128 requests in this experiment), instead of wall clock time. So that we could better compare the results of different metrics.

In this experiment, we have chosen 3 workloads, hm_1, proj_4 and WebSearch2, one from each of the above mentioned sources. These workloads have repetitive access patterns, so we can choose a representative segment of each trace to replay. On the one hand, a simpler trace enables us to better understand the effects of rECS. On the other hand, we need the total SSD capacity to be more than the data set size of each workload, in order to avoid the issues of capacity miss.

Results are shown in Figure 4. For the purpose of comparison, we present 4 results for each workload: CLOCK, Optimal, HitRatio and rECS. **CLOCK** means the cache is managed in a straightforward way: directly by the CLOCK algorithm, which does not try to identify a VM's runtime demand. So the allocated cache space would be proportional to the total number of unique blocks that have been accessed. The **Optimal** curve is generated by offline analysis. It shows the minimum necessary cache size at each moment. We estimate the optimal size at time t by counting the number of blocks contained in the intersection: (unique blocks accessed before time t) $\bigcap$ (unique blocks to be accessed after time t). The **HitRatio** curve is obtained with a similar dynamic cache adjustment mechanism as described above. The only difference is that adjustment is driven by the change of hit ratio, instead of rECS. The **rECS** curve is the result achieved by using rECS as a metric to drive the allocation of cache space.

We can see from these figures that CLOCK incurs the highest amount of under-utilized cache space. HitRatio achieves better space efficiency by reducing the amount of under-utilized cache space. Especially in Figure 4b, its curve is close to the optimal one. However, as shown in Figures 4a and 4c, HitRatio still causes a large amount of cache under-utilization. In contrast, with rECS, we could achieve nearly optimal space efficiency: **the allocated cache space is close to the Optimal curve, and results are consistent for all workloads.**

Therefore we can conclude that rECS is a metric that can accurately identify a VM's cache demand at runtime.

D. Experiments With Multiple VMs

In this section, we present our evaluation results of running multiple VMs on one shared SSD cache device. We will demonstrate the benefits of S-CAVE over managing the cache in a straightforward way.

1) Locality of Workloads: Before running workloads concurrently, we must first understand locality of each workload.

As in Table II, column "HDD" shows the execution time of running a workload in a VM that resides directly on an HDD, without a caching device. Column "Cache" shows the performance obtained with a fixed-size SSD cache, which is managed by the CLOCK algorithm. For cache-resulted performance, we only present those obtained with a sufficiently small cache space. For example, workload "tpch_q9" finishes in $5,857$ seconds with 4GB of cache space. The performance of this workload will not improve if given more than 4GB of cache space, but will be worse if the cache space is smaller.

TABLE II: Standalone excution time (seconds)

	HDD	Cache	Speedup
WebSearch2	19521	3022 (8GB)	6.46x
tpch_q9	25234	5857 (4GB)	4.31x
tpch_q21	20578	6503 (8GB)	3.16x
proj_4	10359	8185 (32GB)	1.27x
mds_1	5837	5686 (32GB)	1.03x

According to this table, WebSearch2 significantly benefits from the cache device, so it has strong locality. Workloads tpch_q9, tpch_q21 feature median locality. Workloads proj_4 and mds_1 are of weak locality.

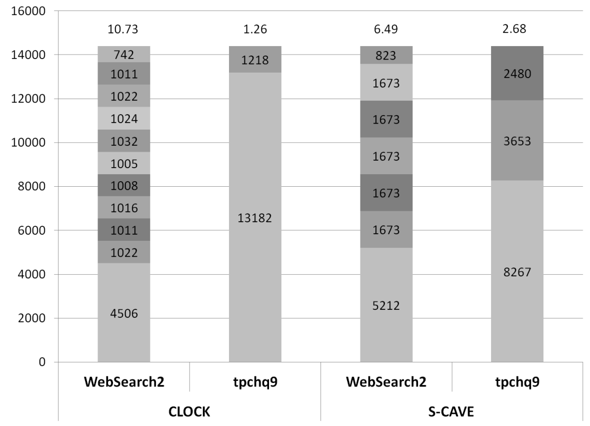
2) Experiment Setup for Co-running VMs: In every experiment, we choose two different workloads to separately execute on two VMs. In order to avoid interferences in the storage hierarchy, each VM resides on a dedicated HDD. But they share one SSD cache. Each workload will be executed repeatedly until timeout. The total experiment time is chosen so that the longest workload can finish at least one iteration. The last iteration of each workload is partially finished, but we know exactly how much has been finished in the last iteration because of trace replay. We compare the results of S-CAVE with the results when there is no S-CAVE management, in which the cache is directly managed by the CLOCK algorithm.

3) Co-running Workloads with Strong Localities: First, we will study the performance of running two strong-locality workloads. According to the performance numbers in Table II, we choose "WebSearch2" and "tpch_q9". The shared SSD cache space is configured to be 8GB in total, so that no workload can get enough cache space to achieve the same performance in Table II.

Results are shown in Figure 5. With CLOCK, tpch_q9 has finished 1.26 iterations, and WebSearch2 has finished 10.73 iterations. In S-CAVE, tpch_q9 has finished 3.68 iterations and WebSearch2 has finished 6.49 iterations. The first iteration of each workload is slow because of cold start. And the last iteration of each trace is partially finished due to timeout.

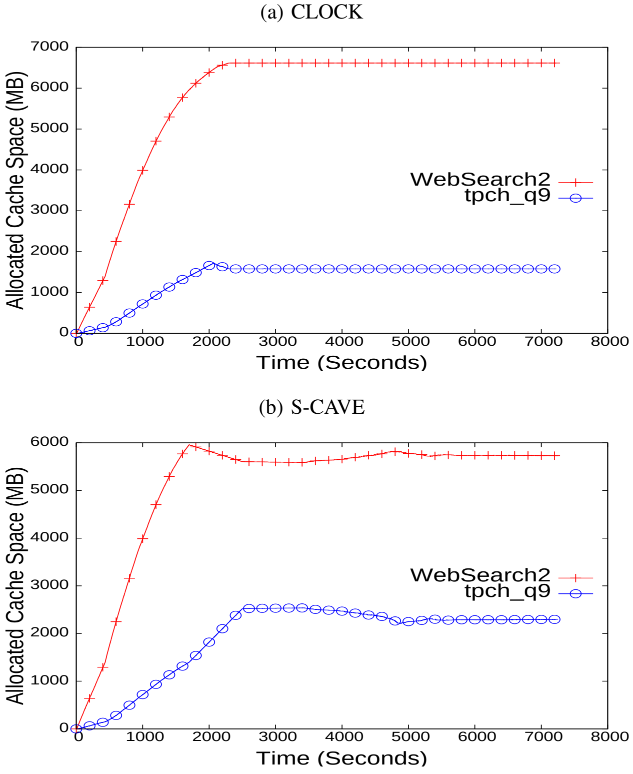
With a limited SSD cache space, it is expected to observe slow-down in the first iteration. However, by comparing with the "Cache" performance in Table II, CLOCK has unfairly treated tpch_q9, whose first iteration had a slow-down of 2.25 (from $5,857$ seconds to $13,182$ seconds), while WebSearch2 only has a slow-down of 1.49 (from $3,022$ seconds to $4,506$ seconds).

Fig. 5: Performance of Co-run (tpch_q9 and WebSearch2)



On the contrary, S-CAVE has achieved performance isolation by allocating cache space according to the relative cache demand of each VM. By comparing with Table II again, first iteration of tpch_q9 suffered a slow-down of 1.41 (from $5,857$ seconds to $8,267$ seconds), while WebSearch2 experienced a slow-down of 1.72 (from $3,022$ seconds to $5,212$ seconds).

Fig. 6: Runtime Cache Allocation.

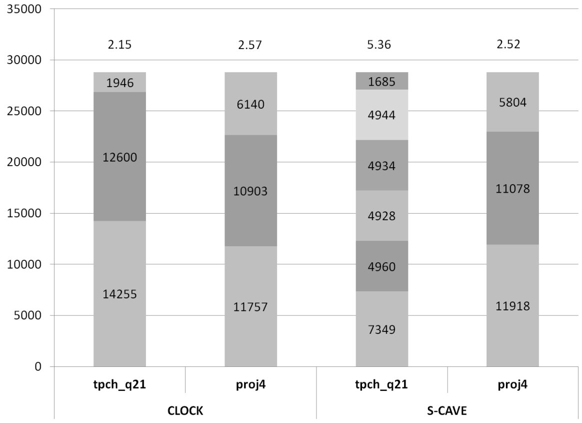

(a) CLOCK

(b) S-CAVE

This can be better explained with Figure 6, which illustrates the runtime cache allocation for the first $8,000$ seconds of the experiment, after which cache allocation becomes stable. With CLOCK, the VM running WebSearch2 quickly takes up the majority of cache space, and tpch_q9 could not get its fair share during the whole process. In contrast, within S-CAVE while the VM running WebSearch2 is still granted relatively more cache space, the VM running tpch_q9 also gets a fair amount of cache space to make efficient progress.

4) Co-running Workloads with Mixed Locality: In this experiment, we study the performance of running two workloads with totally different localities. According to Table II, we choose "tpch_q21" (strong locality) and "proj_4" (weak locality), and the SSD cache space is configured to be 32GB.

Fig. 7: Performance of Co-run (proj_4 and tpch_q21)



Results in Figure 7 show that in a cache device without proper management, a weak locality workload will pollute cache and significantly deteriorate the performance of a strong-locality workload. This observation is consistent with existing studies, such as [19]. By comparing with Table II, CLOCK has caused 2.19 times slow-down for the first iteration of tpch_q21 (from $6,503$ seconds to $14,255$ seconds), while proj_4 is not much affected due to its low locality.

In comparison, S-CAVE has successfully achieved performance isolation and delivers much better performance. The first iteration of tpch_q21 is $7,349$ seconds, $1.94X$ faster than the result obtained with CLOCK.

The benefits of S-CAVE can also be seen from throughput. In CLOCK, proj_4 has finished 2.57 iterations and tpch_q21 has finished 2.15 iterations. In S-CAVE, proj_4 has finished 2.52 iterations, and tpch_q21 has finished 5.36 iterations, a significant improvement.

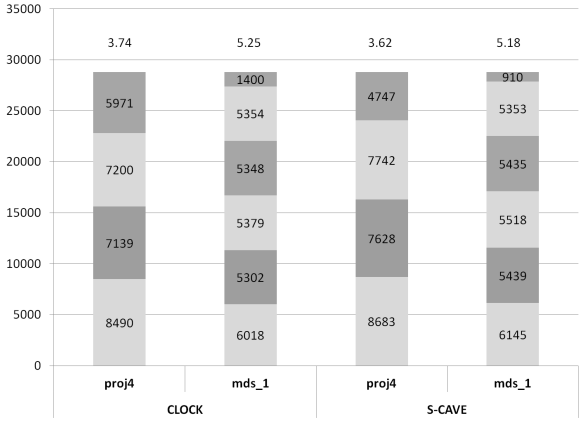
TABLE III: IO statistics in Co-run (proj_4 and tpch_q21)

	CLOCK	S-CAVE
I/Os for tpch_q21	6,854,523	17,088,484
I/Os for proj_4	7,020,532	6,852,710
(total)	13,875,055	23,941,194
IOPS	482	831

The improved throughput by S-CAVE can be better explained by Table III, which shows the IOPS (average number of I/Os per second) for each workload. As we can see from this table while proj_4 is not cache-sensitive, but tpch_q21 has significantly benefited from the effective cache management of S-CAVE.

5) Co-running Workloads with Weak Locality: Finally, we will study the performance of running two weak-locality workloads. According to Table II, we choose mds_1 and proj_4. The SSD cache has been configured with a total space of 32GB.

Fig. 8: Performance of Co-run (mds_1 and proj_4)



As can be seen from the results shown in Figure 8, weak-locality workloads cannot benefit from a cache device. Therefore, it is important to minimize overhead in this case. S-CAVE may constantly try to adjust cache space allocation (thus causing some overhead), but due to the characteristics of these workloads, any allocation decision will not improve performance. CLOCK may result in less overhead for such overheads because it does not make any efforts to identify each VM's demands.

However, we can see from Figure 8 that S-CAVE incurs a negligible overhead as well: the throughput of S-CAVE is only slightly lower than that of CLOCK. In CLOCK, proj4 has finished 3.74 iterations and mds_1 has finished 5.25 iterations. In S-CAVE, proj_4 has finished 3.62 iterations (3.2% lower than that in CLOCK), and mds_1 has finished 5.18 iterations (only 1.3% lower than that in CLOCK). This slight performance loss is due to runtime cache space adjustments that could not yield any effect, because of the workloads' inherent characteristics.

TABLE IV: IO statistics in Co-run (mds_1 and proj_4)

	CLOCK	S-CAVE
IOs for proj_4	8838598	8625157
IOs for mds_1	12282914	12076883
(total)	21121512	20702040
IOPS	734	719

The low overhead of S-CAVE can also be explained with Table IV. The slight loss in IOPS when the SSD cache is managed by S-CAVE, is consistent with the performance results shown in Figure 8, thus reaffirming our conclusion that the overhead caused by the efforts of S-CAVE to monitor and adapt to VM activites is negligible.

E. Summary of Experiment Results

With experiment results, we can summarize the effectiveness of S-CAVE as follows. (1) Metric rECS is able to accurately capture the cache demand of a VM at runtime. (2) S-CAVE can effectively allocate SSD cache space to accomplish the following three goals: (a) performance isolation among VMs with strong or different cache demands; (b) high SSD cache utilization by minimizing inter-VM interferences; and (c) minimum overhead when managing VMs with no locality.

VII. Other Related Work

Related work can be classified into the following three categories.

SSD caching in non-VM environments: Recent development of SSD caching has been advanced from traditional replacement algorithm study [10], to incorporating SSD device specific characteristics and semantic information from software (OSes and applications) [2]–[4]. Their goal is to identify "valuable" blocks that should be placed on the cache. However, these solutions are designed in the context of conventional systems, where the SSD cache may not be shared by multiple entities, thus cannot be applied to a virtual system environment, in which the SSD cache is shared by multiple VMs.

Non-caching approaches to exploiting SSDs: Some other work has studied how to best utilize SSD by approaches other than caching. For example, Hystor [20] and its related product Fusion Drive [21] are hybrid solutions where the SSD device is not a transparent cache for slower HDDs: each block in the SSD does not have a duplicate in one of the HDDs. Intelligent algorithms and OS information are utilized to identify and migrate data blocks between different devices. However, the SSD device in these approaches is not shared by multiple entities either.

Shared resource management in a VM environment: In addition to SSD cache, a VM environment has other shared resources, such as CPU, memory, disk, network, and etc. The most related work focuses on the management of main memory [7], [8], [22]. A series of approaches have been proposed to "guess" how a guest OS is using its memory and have adopted metrics like hit ratio to guide resource allocation [7], [8], [22]. In comparison, our approach attempts to fully exploit the advantages of a hypervisor to monitor VMs' I/O activities and directly control storage devices.

SSD caching in a VM environment has also been mentioned in [13]. However, their approach is a traditional cache device that is managed by a general-purpose cache replacement algorithm. In contrast, S-CAVE is specifically designed to manage an SSD cache device that is shared by multiple concurrent VMs.

VIII. Conclusion

In this paper, we have analyzed three design options for SSD cache management in a VM environment, and have made a strong case for a hypervisor-based design of S-CAVE. In S-CAVE, we use metric rECS to identify the cache demand of each VM, and for effective cache management among multiple VMs, we have integrated rECS into a dynamic control mechanism. By cross-comparing each VM's cache demand and previous allocation decisions, S-CAVE dynamically and effectively adjusts cache space allocation. We have implemented S-CAVE in vSphere ESX [5], a widely used commercial hypervisor from VMware. Experiment results have proven the effectiveness of metric rECS in monitoring each VM. In a multi-VM environment, S-CAVE could achieve high performance for each individual VM and for the overall system as well, through effective cache space allocation and direct control of hardware resources.

Acknowledgement

We thank the anonymous reviewers for their constructive comments. The work was supported in part by the National Science Foundation under grants CCF-0913050 and CNS-1162165.

References

[1] A. Gulati, G. Shanmuganathan, X. Zhang, and P. Varman, "Demand Based Hierarchical QoS Using Storage Resource Pools," in *USENIX ATC*, 2012.

[2] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "SSD Bufferpool Extensions for Database Systems," *PVLDB*, vol. 3, no. 2, pp. 1435–1446, 2010.

[3] T. Luo, R. Lee, M. P. Mesnier, F. Chen, and X. Zhang, "hStorage-DB: Heterogeneity-aware Data Management to Exploit the Full Capability of Hybrid Storage Systems," *PVLDB*, vol. 5, no. 10, pp. 1076–1087, 2012.

[4] M. P. Mesnier, F. Chen, J. B. Akers, and T. Luo, "Differentiated Storage Services," in *SOSP*, 2011, pp. 57–70.

[5] http://www.vmware.com/products/vsphere/esxi-and-esx/index.html.

[6] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *MICRO*, 2006, pp. 423–432.

[7] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: monitoring the buffer cache in a virtual machine environment," in *ASPLOS*, 2006, pp. 14–24.

[8] P. Lu and K. Shen, "Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache," in *USENIX Annual Technical Conference*, 2007, pp. 29–43.

[9] A. J. Smith, "Disk Cache-Miss Ratio Analysis and Design Considerations," *ACM Trans. Comput. Syst.*, vol. 3, no. 3, pp. 161–203, 1985.

[10] S. Jiang and X. Zhang, "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *SIGMETRICS*, 2002, pp. 31–42.

[11] S. Jiang, F. Chen, and X. Zhang, "Clock-pro: An effective improvement of the clock replacement," in *USENIX Annual Technical Conference, General Track*, 2005, pp. 323–336.

[12] A. S. Tanenbaum, *Modern operating systems (2. ed.)*.

[13] S. Byan, J. Lentini, A. Madan, and L. Pabon, "Mercury: Host-side flash caching for the data center," in *MSST*, 2012, pp. 1–12.

[14] H. Song, S. Dharmapurikar, J. S. Turner, and J. W. Lockwood, "Fast hash table lookup using extended bloom filter: an aid to network processing," in *SIGCOMM*, 2005, pp. 181–192.

[15] Q. Li and J. Garcia-Luna-Aceves, "Opportunistic Routing Using Prefix Ordering and Self-Reported Social Groups," in *ICNC*, 2013, pp. 28–34.

[16] "SNIA IOTTA Repository," http://iotta.snia.org/.

[17] "UMass Trace Repository," http://traces.cs.umass.edu/index.php/Storage/Storage.

[18] "TPC Benchmark H," http://www.tpc.org/tpch/.

[19] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang, "MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases," *PVLDB*, vol. 2, no. 1, pp. 373–384, 2009.

[20] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems," in *ICS*, 2011, pp. 22–32.

[21] "Fusion Drive," http://en.wikipedia.org/wiki/Fusion_Drive.

[22] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," in *OSDI*, 2002.