Centaur: Host-side SSD Caching for Storage Performance Control

Ricardo Koller
IBM T.J. Watson Research Center

Ali José Mashtizadeh
Stanford University

Raju Rangaswami
Florida International University

***Abstract*—Host-side SSD caches represent a powerful knob for improving *and* controlling storage performance and improve performance isolation. We present Centaur, as a host-side SSD caching solution that uses cache sizing as a control knob to achieve storage performance goals. Centaur implements dynamically partitioned per-VM caches with per-partition local replacement to provide both lower cache miss rate, better performance isolation and performance control for VM workloads. It uses SSD cache sizing as a universal knob for meeting a variety of workload-specific goals including per-VM latency and IOPS reservations, proportional share fairness, and aggregate optimizations such as minimizing the average latency across VMs. We implemented Centaur for the VMware ESX hypervisor. With Centaur, times for simultaneously booting 28 virtual desktops improve by 42% relative to a non-caching system and by 18% relative to a unified caching system. Centaur also implements per-VM shares for latency with less than 5% error when running microbenchmarks, and enforces latency and IOPS reservations on OLTP workloads with less than 10% error.**

I. Introduction

Enterprises constantly consolidate more workloads as virtual machines (VMs) to reduce hardware, power, and maintenance costs. However, doing this well requires VMs to provide the same performance guarantees that physical machines do. Current hypervisors provide per-VM performance isolation guarantees and even knobs to control per-VM shares, limits, and reservations for CPU and memory resources. However, these performance controls and isolation guarantees do not include storage. Storage performance of individual virtualized workloads under contention is difficult to control with precision [13]. There is a need for solutions that enable storage systems to satisfy simple high level goals (e.g., minimizing average VM latency or meeting a specific VM's latency target).

Host-side SSDs are a recent augmentation to disk-based storage stacks that are becoming increasingly mainstream [6], [16], [20], [30], [42]. When used as a cache, the SSD space can be dynamically partitioned across workloads to meet per-workload performance goals. Current cache partitioning techniques optimize aggregate metrics such as effective cache utilization [22] and aggregate cache utility [28]. They provide controls for proportional sharing of the cache space but the utility and limitations of SSD cache partitioning has not been comprehensively explored. In particular, current research on partitioning of SSD caches leaves three questions unanswered:

(i) What motivates host-side SSD cache partitioning? We observed that workloads sharing a host-side cache can cause significant cache *wastage* due to contention effects. Our **first contribution** is to show that such wastage occurs for LRU, Aging-LFU, and ARC [27] replacement algorithms and that this wastage increases as a percentage of the combined workload's working set size. Thus, solutions that can minimize wastage in shared SSD caches are necessary.

(ii) How can we best overcome cache wastage effects via partitioning? Current solutions use conventional cache partitioning techniques that work well for CPU and memory caches [17], [38]. In particular, these techniques rely on assumptions about cache *miss rate curves* that do not hold true for host-side SSD caches. Our **second contribution** involves empirically demonstrating that applying conventional partitioning techniques to SSD caches can result in large errors when estimating cache miss-rate: a 20% increase in cache miss-rate for a set of 9 production storage workloads. Our **third contribution** is an online technique based on dynamic cache partitioning that is free of such assumptions and addresses SSD cache wastage effectively.

(iii) How do we partition caches for per-VM storage latency and throughput control? Previous studies have addressed improving overall cache hit ratio and proportional sharing of the cache. However, they do not address how administrators would map the specified miss-rate based goals into relevant and familiar goals such as per-VM storage latency or throughput reservations. Our **fourth contribution** is demonstrating that host-side SSD caches can be an accurate storage QoS control knob. We develop *latency curves* and *throughput curves* as foundations for building a variety of QoS controls.

We present Centaur, a host-side SSD caching system that is able to meet per-VM performance and isolation goals. It periodically measures latency and miss rate for each VM, uses this information to predict the performance at different cache sizes, and re-partitions the cache using these predictions and the expected goals. We implemented Centaur as a partitioned write-back host-side SSD cache for virtual disks in the VMware ESX hypervisor. Experimental results show that Centaur accelerates the boot time of 28 virtual desktops by 18% when compared with a unified caching solution. Furthermore, for a set of 9 production storage workloads, it reduces cache misses by as much as 20% relative to a unified cache. We also show that our system is also able to balance the average latencies of mixed read/write VM workloads with less than 5% error. Moreover, it can implement latency and IOPS reservations across a mix of VMs running the OLTP benchmark [26]. These fine-grained per-VM performance controls are simply unavailable when using a unified cache or using the existing SSD cache partitioning solutions.

II. Motivation

Host-side SSD cache partitioning offers a powerful control knob for achieving storage performance goals. To make this

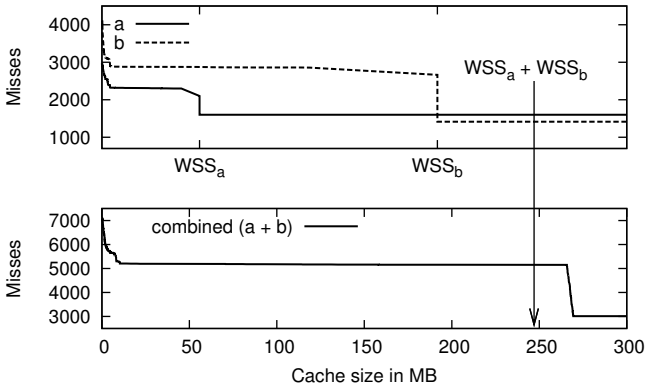

Fig. 1. **Working set sizes and wastage.** Curves a and b are the MRCs of two workloads and $combined$ is the MRC obtained when both a and b run together.

argument, we show that sizing an SSD cache (or a cache partition) can be used to control per-VM latency and throughput. Second, we show that shared SSD caches must be partitioned to avoid cache wastage due to contention effects that are otherwise incurred by unified caches.

A. Cache Size as a QoS Control Knob

To motivate cache size as a QoS knob, we designed a simple experiment to determine if we can control the average IO latency of a workload by varying the size of its host-side SSD cache. We created a workload to access a 10 GB file from a 10K RPM disk using four patterns: random-reads, random-writes, sequential-reads and sequential-writes. The SSD cache implements a write-back, write-allocate, Adaptive Replacement Cache (ARC) [27] algorithm with 4 KB requests for the random workloads and 128 KB requests for the sequential ones. For all cases, the average IO latency decreases monotonically with increase in cache size in the range of at least 2 ms to at most 7 ms in all four cases. This experiment supports the use of host-side SSD cache size for storage QoS control. Later, we discuss how carefully partitioning an SSD cache across VMs can be used effectively for meeting a variety of workload-specific goals. We show that such goals can be fairly broad and include per-VM latency and IOPS reservations, proportional share fairness, and aggregate optimizations such as minimizing the average latency across all VMs.

B. Wastage in Unified Caches

High performance storage caches are expensive and are typically sized to hold the working set of the workload. We define the working set size (WSS) as the size beyond which any increase in cache size does not reduce the miss rate. A common method to estimate a workload's working set size is using a miss rate curve (MRC) as in Figure 1. Given the MRC, a storage cache can be provisioned to match the working set size. This approach works well when a single workload uses the cache exclusively. However, when multiple workloads share a storage cache, it does not.

We replayed the *webresearch* and *webusers* production traces published previously [41] for half an hour on a warmed-up, aging-LFU cache simulator, first individually and then simultaneously. Figure 1 illustrates how the MRCs of two workloads is not adequate for computing the combined cache requirement of two workloads. The caching system uses a unified cache replacement policy at the hypervisor level that is not aware of individual workloads. One would expect the combined workload to have a WSS equal to the sum of the individual WSS. However, the real WSS is 23 MB (almost 10 %) more than the expected 246 MB. We term this additional cache requirement as $wastage$.

To better quantify wastage with host-side SSD caches, we replayed a three-hour segment from eight production workload traces [41]. Individual and time-stamp based combinations of these workloads were replayed on simulators using the LRU, aging-LFU, and ARC [27] replacement policies with cache lines of 4 KB (typical file block size). Figure 2 uncovers a troubling phenomenon—the induced cache wastage (as a fraction of the combined workloads working set) increases as more workloads are made to share the cache. Given the high consolidation ratios in today's virtualized systems (50:1, 100:1), wastage would represent a significant portion of the cache space requirements.

To overcome the cache wastage effects incurred when multiple workloads access a unified cache, Centaur adopts per-workload partitioning of the host-side SSD cache. While previous work implicitly chose to partition a shared host-side SSD cache and applied conventional techniques for partitioning [22], [28], this work provides the foundations for doing so with host-side SSD caches.

III. Centaur Overview

A Centaur-enabled hypervisor manages host-side SSDs as a cache resource to be used within each VM's storage access path. The managed element in Centaur is the set of VMs, and the autonomic manager is a high-level process that monitors the VMs and configures their SSD cache allocations. Individual VMs store their file systems as distinct files managed by the hypervisor. An administrator would assign some or all virtual disks to SSDs for caching and specify performance goals on a per-VM basis. Hypervisors, upon the direction of the autonomic manager, control the storage performance of individual VMs by increasing or decreasing the cache space assigned to each VM to meet administrator goals. Further, unlike previous solutions for CPU caches that continuously repartition the cache [40], SSD cache partitions in Centaur are only periodically resized by the *global scheduler* to adapt to stable changes in the workloads and the storage system [21], [35]. Doing so ensures that stable workload characteristics are reflected post-repartitioning.

We anticipate our solution to be used by administrators towards meeting several types of system goals. The administrator can choose to *(A)* maximize overall performance which translates to minimizing overall cache miss rate or average IO latency across all VM IOs or *(B)* maximize the sum of per-VM average performance which translates to a fairness goal, or *(C)* per-VM performance control which allows specifying latency or IOPS targets on a per-VM basis. A unified cache can support only the first of the three goals above and will incur unwanted cache wastage when doing so.

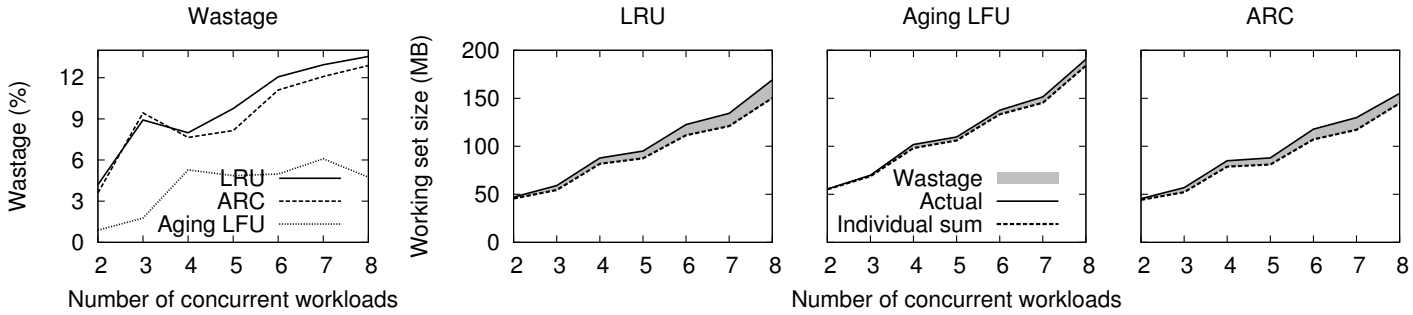

Fig. 2. **How does wastage scale?** The leftmost pot tracks wastage as a percentage of the total cache requirement. It grows as the degree of contention increases. While wastage with Aging-LFU grows slower than with LRU or ARC, its overall cache requirement is higher than either (as shown in the three rightmost plots).

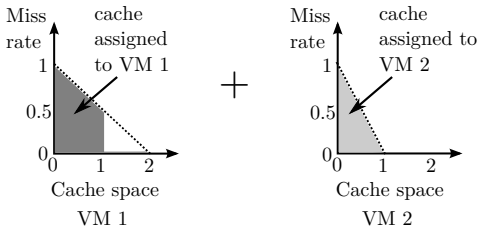

Fig. 3. **Example of cache partitioning.** A cache of size 2 is partitioned across two VMs. VM MRCs are shown with the optimal assignment of cache shown in grey: one cache unit to each VM.

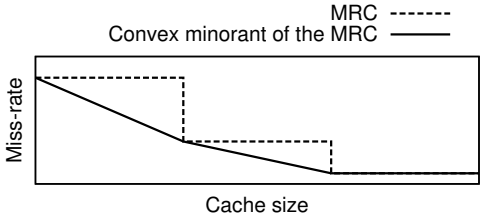

Fig. 4. **Sample MRC and its convex minorant.**

IV. Background on Cache Partitioning

Wastage in shared caches can be addressed with cache partitioning. Most partitioning approaches in the literature are based on *miss rate curves* (MRC), which define the miss rate for a given cache size. We illustrate the use of miss rate curves to find the allocation that optimizes a specific goal—minimizing the sum of average VM miss rates. Assume there are two VMs reading from random locations within a file. VM 1 reads a file of size 2 GB and VM 2 reads a file of size 1 GB. Figure 3 shows the MRCs for both VMs. The miss rate is 1 for a cache size of 0 for both VMs; it is 0 for cache sizes greater than the size of the files read (2 GB for VM 1 and 1 GB for VM 2). For a 2 GB cache, assigning 1 GB to each VM is optimal when minimizing the overall cache miss rate. This partitioning results in a total miss rate of $0.5 = 0.5 + 0$ and is depicted as the shadowed regions in Figure 3.

While this seems straightforward, the number of possible partitions grows exponentially with the number of workloads. Rajkumar *et al.* proved that optimal cache partitioning is NP-hard [32]. Stone *et al.* proposed an approximation algorithm using convex hulls for solving this problem [38]. This approach and its variants have been used for partitioning CPU, main memory, and storage system caches [9], [29], [31], [37], [39], [40], [45].

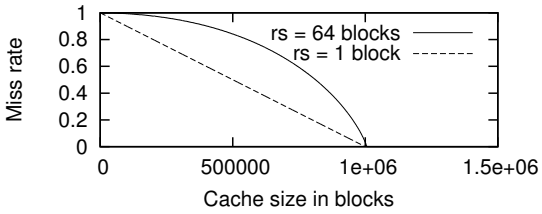

Fig. 5. **MRCs for large requests sizes.** Random reads from a file at two granularities, with request size (rs) = 1 and with rs = 64.

A. The Convex Hull Approach

Current cache partitioning algorithms follow the *convex hull* approach. The convex hull of a set of points is the smallest convex polygon that contains all the points. The convex minorant is the greatest convex curve lying entirely below this polygon (Figure 4). The partitioning algorithm first computes a convex minorant for the MRC of each VM and then applies an iterative, greedy search algorithm on the minorants to find a close-to-optimal partitioning of the cache space as follows:

1) Initialize partition sizes s_k to zero.
2) Calculate the convex minorant $m_i(s_i)$ of the MRC for each VM *i*.
3) Increase by one unit the partition size of the VM k which would benefit the most computed as $m_k(s_k) - m_k(s_k+1)$, the reduction in the miss-rate for the VM.
4) Repeat the previous step until all cache space has been assigned.

B. Problems with the Convex Hull Approach

The above approach fails when used for host-side SSD caches and will fail for other caches with the properties discussed below. Unlike CPU caches, the SSD cache access granularity can be arbitrarily large. For a multi-block cache request, some blocks could be resident in the SSD cache while others are not. Since a block request can complete only when all blocks have been retrieved, we consider a request to the host-side caching layer as a cache hit only when all the blocks accessed are found in the cache. The effect of this requirement is that the probability of a cache hit decreases with the request size. For example, if the cache size is 10 MB, requests of 1 MB are less likely to hit the cache than requests of 4 KB.

We illustrate the problem with the convex hull approach empirically. The dotted line in Figure 5 shows the MRC for a

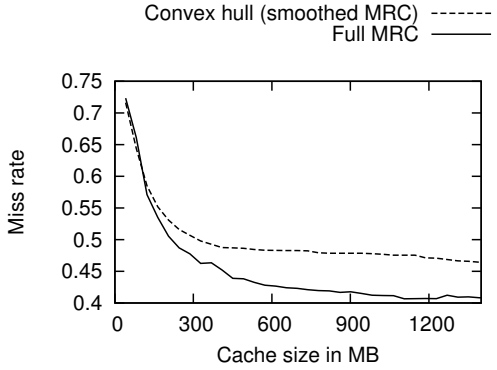

Fig. 6. **Convex hull approach versus using the full MRCs.** Nine production traces were replayed using a partitioned cache of varying size with the convex hull approach and when using full MRCs.

workload that randomly reads from a file of one million blocks using a request size of one block. As expected, the miss rate decreases as a straight line from 1 when there is no cache to a miss rate of zero when the cache has the size of the file. The solid line shows the MRC for the same workload but when requests are of size 64 blocks. The curve becomes concave because as the size of the request increases, the chances of hitting the cache decrease. The problem with the convex hull approach is that both these workloads have the same convex minorant and are therefore treated identically by the allocation algorithm. This can lead to a substantial approximation error. For instance, if the workload issued requests of size 64 and the algorithm assigned it half a million blocks of cache expecting a cache miss-rate of 0.5, in reality, the workload would incur a miss rate of 0.8 instead, an error of 60% in the miss-rate estimation.

To understand the significance of this loss of accuracy in miss-rate estimates, we analyzed an entire day of nine production storage traces [41]. We replayed the traces simultaneously on a cache simulator that partitions the cache space every 5 minutes across all the workloads. Figure 6 shows the total miss rate across all workloads at different caches sizes when partitioned either using the convex hull approach or using full MRCs. The convex hull approach can lead to an increase in total cache miss-rate of as much as 20%.

V. Partitioning for Performance

In this section, we describe how Centaur dynamically partitions a host's SSD cache for improving overall performance and for reducing the sum of the average latencies across all VMs.

A. Partitioning for IO Latency Minimization

Centaur predicts the sum of IO latencies, probabilistically searches the space of potential partitioning outcomes, and chooses the partitioning outcome which results in the smallest sum of IO latencies across VMs. The process (at every epoch) follows the steps listed below:

1) Construct a miss rate curve (MRC) for each VM.
2) Use the MRCs to predict the IO latency for each VM at all possible cache sizes, thereby creating the *latency curve* for each VM.
3) Use a probabilistic search using the latency curves to find the partitioning candidate that produces the smallest sum of IO latencies across all VMs.

Step 1: MRC construction for external caches

Following Denning's locality principle [7], we assume that the MRCs of two consecutive partitioning epochs are likely to be similar and construct MRCs for future use using the IO accesses during the previous partitioning epoch. Centaur's MRC construction technique is an adaptation of Mattson's stack algorithm [25]. Mattson's stack algorithm was originally designed to create MRCs for the LRU cache policy; later it was shown to work for other policies (e.g., LFU [5]). Each time an element is accessed, the *stack distance* (depth in the stack at which the element was found because of a previous reference) for the reference is determined. The algorithm updates a histogram of stack distances upon every access, thereby capturing the hit count at every possible stack distance. The workload's MRC is constructed from the final histogram of stack distances.

We changed Mattson's stack algorithm to account for the unique properties of external caches (which support multi cache-block accesses) and our proposed *batched cache population* mechanism (discussed further in Section VII-C). Our first modification to the Mattson's stack algorithm is at the histogram construction step; we record a hit only if all the blocks forming a single request are found in the cache. If a request has more than one block, we update the histogram only for the block that has the largest stack distance. This ensures that cache hits for multi-block requests are accounted for correctly in the MRC. Specifically, if the cache size is greater than the largest stack distance (for a set of blocks that are accessed in a single request), the request will hit only if the largest of the stack distances is smaller than the cache size.

The second modification accounts for batched cache population. We construct the histogram of stack distances using an LRU stack that is only updated on every batched cache population event. Our goal with this modification is to use this modified histogram and create an MRC that will mimic the miss behavior of a cache whose blocks only get updated during batched population operations. The batched updates are implemented by periodically replacing the LRU stack with a shadow stack that is updated on every access.

Step 2: Construction of latency curves

We construct latency curves for VM workloads using the MRCs computed using our modified Mattson's stack algorithm. IO latency prediction can be done similar to the approach of Soundararajan *et al.* [37]:

$$lat = (1 - mr) * lat_{ssd} + mr * lat_{disk} \quad (1)$$

Where lat is the predicted average IO latency, mr is the miss-rate as obtained from the MRC, and lat_{ssd} and lat_{disk} are the VMs recently measured IO latencies for the SSD cache and disk respectively. However, using previously measured IO latencies can lead to large inaccuracies. A simple scenario where this can happen is when the prediction leads to IOPS saturation on the disk, while the measurements are from an unsaturated state. For example, if the current state lat_{disk} is low, then a prediction of IO latency at higher miss rates would fail because lat_{disk} would not be low anymore. Since latency for a VM is measured while running with other VMs, adding and removing VMs from the hypervisor lead to inaccuracies

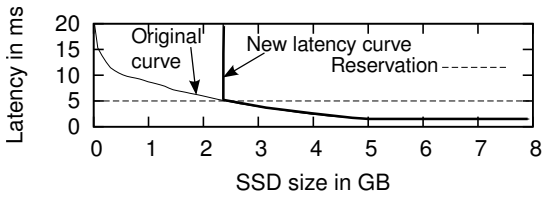

Fig. 7. **Latency reservation.** The latency curve is modified by extrapolating the point at which it hits the reservation to the infinite.

in the latency curve. This and changes in application behavior prompted updating the latency frequently in Centaur.

Our SSD caching solution follows a three step process iteratively: *(i)* plan a partitioning, *(ii)* repartition the cache across the VMs, *(iii)* measure the resulting IO latencies, and then restarts the whole process again. The process is stopped when the deviation between the measured and predicted IO latencies converge to a difference less than a small constant.

Step 3: Probabilistic Search

Once the latency curves have been created, we use a probabilistic search to evaluate potential allocations. These allocations are evaluated by adding up the IO latencies for all VMs at candidate partition sizes. If a candidate partition size of a VM k is s_k and its latency is $L_k(s_k)$, then the search will attempt to minimize $\sum_{k=1}^{N} L_k(s_k)$ such that $\sum_{k=1}^{N} s_k = S$ where S is the total size of the cache. We chose simulated annealing (SA) [19] as the probabilistic search technique to search for the best partitioning outcome. While classic hill climbing only allows changing to a better solution thereby exposing itself to getting trapped in local optimal, SA probabilistically allows changes that may degrade the solution.

B. Partitioning for Other Performance Goals

The above outlined approach for minimizing sum of VM IO latencies can be easily adapted to meet other goals such as maximizing overall IOPS performance. For IOPS maximization, Centaur uses IOPS curves in place of latency curves. It predicts IOPS and generates $1/IOPS$ curves using:

$$IOPS = \frac{IOPS_{disk}}{MR} \tag{2}$$

where MR is miss rate and $IOPS_{disk}$ are the recently measured IOPS of the disk only. Each time the disk has an IO, the SSD has $(1 - MR)/MR$ IOs. Therefore, the total IOs in a second is $IOPS_{disk} + IOPS_{disk} * (1 - MR)/MR$. Minimizing the cumulative $1/IOPS$ curve maximizes overall IOPS across the VMs.

VI. Partitioning for QoS

Centaur extends its approach to performance maximization to meet QoS targets globally as well as on a per-VM basis.

A. QoS: Reservations

The proposed algorithm can be used to implement QoS reservations for both latency and IOPS. For brevity however, we will limit our discussion to latency reservations noting that a similar approach can be used for IOPS reservations. Our approach to reservation modifies the latency curve for each VM with a latency reservation requirement and this change occurs at the latency curves construction step itself and the modified latency curve is used by all later steps of the algorithm. Specifically, for all VMs with latency reservations, we first calculate the cache size at which the target latency reservation is achieved. Next, we create a modified latency curve by extrapolating the latency at that point to $[0, \infty]$. Consequently, cache sizes that are smaller than the one required to meet the target latency reservation all result in infinite latency and thus will not be chosen by the partitioning algorithm. Figure 7 shows the modified latency curve for a VM with latency reservation of 5 ms.

Since QoS reservations are fixed targets (instead of an optimization), we apply a basic form of admission control next. This admission control checks whether the system can satisfy the requested reservations. We check that new reservations are satisfiable based on the current latency predictions (i.e., latencies in the current system state). Notice, however, that this can lead to false negatives or false positives. False positives are solved by having the algorithm go through the iterative process discussed earlier and stopping after a certain number of iterations if there is no convergence. Non-convergence indicates that the reservation target cannot be met given existing reservations and the current system state. While our solution does not address false negatives, we found that false negatives do not occur in practice. Furthermore, false negatives do not compromise the reservations of other workloads.

B. QoS: Proportional Share Fairness

Fairness and performance maximization or reservation are distinct problems and require different approaches. We treat these as different techniques that may not be used together. Again, we restrict our discussion to latency fairness for brevity; IOPS fairness follows a similar approach. For simple fairness with equal shares, we construct latency curves as before and use them to greedily find the partitioning at which latencies are equalized. Starting from an arbitrary partitioning, we minimize the difference of latencies by incrementally resizing the partitions. The key difference in the process are the intermediate steps which attempt instead to minimize is the largest latency difference across all pair of workloads. In each iteration, the algorithm increases the partition size of the VM with the highest latency and decreases the partition size of the VM with the lowest latency. The algorithm stops when the largest latency difference is smaller than a threshold. This algorithm makes the assumption that latency decreases monotonically as cache size increases; if this is not found to be true at any point during its iterative execution, the algorithm stops.

This algorithm can be easily adapted to implement arbitrary shares. For every VM, the shares force an increase or decrease their latency curves artificially. For instance, with shares of 1:2 we retain the latency curve of VM_1 and artificially scale (increase) latencies in VM_2's curve by a factor of two. The new latency curves are then used to achieve simple fairness as described in the previous paragraph.

VII. Implementation

This section discusses how Centaur deals with the host-side flash cache in terms of IO handling, cache replacement,

and data consistency.

A. Cache Implementation

Centaur is a host-side caching layer in the VMware ESX hypervisor that uses the vSCSI filter API available to third party kernel modules. Centaur's kernel module inserts a filter instance per virtual disk, and exposes cache controls to a user-level agent that performs cache partitioning. The implementation supports both a conventional *unified* cache and our *partitioned* cache to compare the performance of these architectures.

With Centaur, as with many virtualization architectures, all IOs initiated by virtual machines are handled by the hypervisor. The virtual disk layer of the ESX hypervisor translates a guest's virtual disk IO into a file access IO to the hypervisor managed file system located in a SAN store. An instance of the filter is attached to a specific virtual disk and redirects all IOs to the local flash on a cache hit or to the SAN on a miss.

Per-VM host-side flash caches, like virtual disks, are stored as individual files in a file system on the flash device. Centaur uses one cache file per VM and implements partitioning by limiting the physical size of these files. The unified and partitioned caches are both write-back caches, that are more effective for mixed workloads that contain both reads and writes [20]. As we shall elaborate later, cache metadata is persistent and journaled to ensure crash consistency for the host-side cache.

B. Cache Partitioning

Centaur's cache partitioning is the key mechanism that eliminates cache wastage by isolating the cache usage of VMs. It is also the knob used for controlling the IO performance experienced by individual VMs when accessing their respective virtual disks.

A *global scheduler* is in charge of partitioning the SSD space among the various cache files and it implements the partitioning algorithms presented in the previous sections. Recent developments in low-overhead MRC construction make the necessary frequent, online MRC construction practically feasible [34], [43], [44]. The global scheduler runs every "partitioning epoch," which is configurable in our implementation. We chose five minutes as the default duration after a pilot examination of its impact for the workloads in our study. The data mover implements partitioning by controlling the size of the per-VM cache files by allocating and freeing blocks in them. Management of individual cache partitions is discussed next.

C. Cache Replacement

Conventional cache replacement policies are designed for CPU and buffer caches. These policies are required to move blocks into the cache directly after a miss, i.e., on-demand, since the cache lies in the data access path. In other words, the cache gets populated as a side-effect of the access itself, and more importantly, with *zero overhead*. We call this behavior of conventional cache replacement mechanisms as *on-demand cache population*. For an external memory cache such as host-side flash, populating the cache incurs additional overhead: potentially evicting an existing dirty element and writing a new element into the cache both incur additional IO latency. While it is certainly important to do so for items that are frequently accessed, it can be counter-productive to do so for every item that misses the cache [35].

An alternate approach to cache population is to perform such operations in batches (e.g., periodically or triggered by cache miss-count): the per-VM caches are (re)populated during "mover epochs" by a data mover process that moves multiple blocks to and from the cache based on the chosen cache replacement policy. *Batched cache population* utilizes storage workload stability properties [4], [11], [41] and is particularly attractive for external caches that are not in the storage access path. With batched cache population, the additional IO operations due to cache population to both the flash cache and the storage system can be deferred and aggregated. Batched cache population can reduce the number of additional writes to populate the cache since spurious cache population with unimportant data can be avoided. Because it enables higher levels of eviction IO parallelism, it can improve eviction performance to both the flash cache. Further, batched cache population reduce the number of evictions from flash since aggregate cache access behavior is considered. On the negative side, batched cache population can potentially incur higher cache misses transitorily, in proportion to the rate of change of the working-set of the workload. We evaluated on-demand and batched (with periodic trigger) cache population techniques in Centaur for the production traces that we analyzed and found that we can improve overall throughput by making such data movement periodically rather than on-demand in the IO path.

D. Crash Consistency

Persistent services require both consistency and persistence of data to recover correctly after a system failure [10]. Ensuring these properties for both data and the internal metadata of the cache is thus critical for a write-back caching solution [20]. Write-back caches can contain dirty data not present in the backing store. Therefore, when restarting a VM we must access its backing store *and* all its cached data on the SSD. Centaur keeps the mapping information for cached data persistently in the SSD cache itself so that it can be queried it when a VM restarts. To minimize the impact on IO performance, Centaur journals the updates to these persistent mappings in memory and then replays the journal before data mover migrations during each mover epoch.

VIII. Evaluation

Our evaluation of Centaur aims to answer the following questions:

Performance maximization (1) How does a partitioned cache perform compared to a unified cache? (2) How do these approaches adapt to the performance properties of the cache?

QoS control (3) How accurate are our proposed proportional share fairness and (4) reservations techniques?

Overheads (5) What are the performance overheads of partitioning and what are the sources of the overheads?

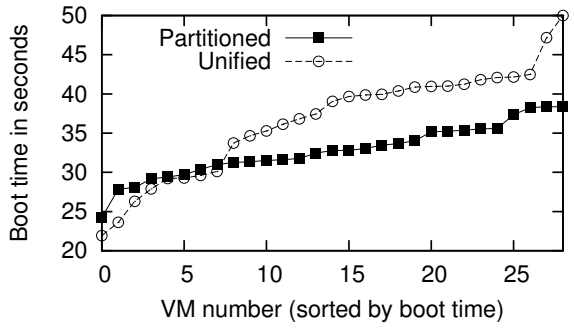

Fig. 8. **28 Virtual Linux desktops booting simultaneously using an 6 GB SSD cache.** A global LFU cache and a partitioned LFU cache were used.

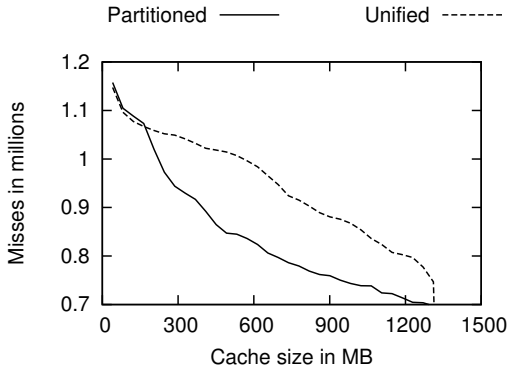

Fig. 9. **Replaying nine production storage traces.**

seq (OIO=8, bs=32k)	Latency (usec)	IOPS
SSD	648	1886
SAN	1548	1689

TABLE I. **Device characteristics.**

A. Experimental Setup

After a sensitivity analysis of the parameters of Centaur with several workloads, we chose the cache block size as 128 KB, the partitioning epoch as 5 minutes, and the mover epoch to be 1 minute long. We used the aging-LFU cache replacement policy [33]. The host was a dual socket machine with AMD Opteron 270 dual core processors, each running at 2 GHz and 4 GB memory. We used two kinds of VMs in our experiments: a Linux Ubuntu VM with a 30 GB virtual disk, one virtual CPU, and 1 GB of memory; and a Windows 7 VM with 40 GB of virtual disk, one vCPU, and 2 GB of memory. We used an 120 GB Intel X25-M SSD as the host-side cache, but limited its available capacity in proportion to the total working-set size of the workloads sharing the cache in a given experiment. The backing store was an EMC Clarion CX4 Model 120. The data mover and the hypervisor were configured to have an issue queue length of 32 outstanding IOs (OIOs) each.

We evaluated Centaur using a mix of microbenchmarks, macro-benchmarks, and production storage traces. The microbenchmarks used Iometer [3] on Windows and fio [2] (with the *libaio* engine) on Linux for workload generation. All experiments used 32 OIOs unless stated otherwise. The macro-benchmark used was Filebench [26] OLTP. Finally, we replayed several production storage traces [41].

B. Performance Maximization

1) Unified vs. Partitioned SSD Caches: In our first experiment, we evaluated the solution's effectiveness when maximizing average IO throughput. Both variants (partitioned and unified caching) use the same implementation except that the unified cache uses no partitions. The partitioned cache was configured to maximize overall IOPS; the unified cache does so by design. We fixed the total cache size to 6 GB for both variants.

The first experiment involved booting 28 Ubuntu 10.04 desktop systems simultaneously and measuring the *rebooting time* as reported by Bootchart [1]. We booted all the systems simultaneously, twice, to allow warming up of the cache and partitioning. Figure 8 shows the rebooting time of all 28 machines for global and partitioned caches. The x axis is sorted by reboot time. Average reboot time was 32 seconds with a partitioned cache and 39 seconds with a unified cache. Compared to the 55 second rebooting time of the 28 VMs without an SSD cache, the unified cache offered a 29% improvement while the partitioned cache offered a 42% improvement. With partitioning, cache wastage is eliminated, and the VMs incur fewer SSD cache misses, experience greater IOPS, and ultimately improved reboot times.

Using a partitioned cache improves fairness across the VMs, because we nearly equally allocated cache space to each VM. This results in the partitioned cache reducing the variance in boot time relative to the unified case. We found that with unified cache some VM's boot up took twice as long as others, because they received one third the cache space in comparison.

In the next experiment, we evaluate the additional cache misses due to cache wastage using the production storage traces described earlier. We replayed nine production workload traces simultaneously on a unified and a partitioned aging-LFU cache. Trace replay duration was 1 day for each trace, and we repeated the experiment for different cache sizes. Figure 9 shows that for sizes larger than 300 MB up to 1.2 GB, the partitioned cache incurred at least 15% fewer cache misses than the unified cache. These reduced misses translated to a reduction of 1 GB of data transferred from the SAN store, which directly impacts performance of the VMs.

2) Adaptation to Specific Storage Characteristics: Some SSD specific performance oddities are slow writes, and sequential read performance that is comparable to that of the SAN store. Conventional caches would minimize the number of IOs to the SAN store, even if these would have been serviced efficiently by the SAN store. The latency and IOPS curves based techniques use observed performance data from individual tiers (SSD cache or SAN store). Thus, our latency and IOPS curves adapt to how individual VMs will benefit from the SSD cache versus the SAN store for a given target performance metric and allocating cache space for the VMs accordingly.

The next experiment illustrates the need to adapt to workloads respond on different devices. We show this by co-locating a sequential read workload reading a 5 GB file with a random workload reading a 10 GB file. We configured the sequential read workload to generate a similar number of IOPS when using the SAN store or the SSD. The workload only generated 11% more IOPS on the SSD, but experiences

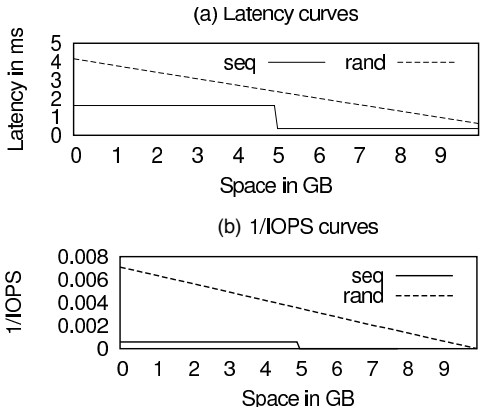

Fig. 10. **Latency and IOPS curves.**

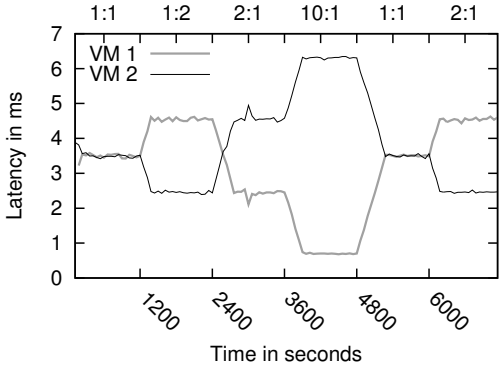

Fig. 11. **Adaptiveness to shares.** How the SSD space for each VM adapts to changes in shares. The workload is Iometer with 50% random, 50% reads.

2.4X greater latency when using the SAN store. Table I lists actual performance for this workload when using the SAN store or the SSD. Figure 10 (a) shows the latency curves for both workloads. To minimize the average latency the best partitioning is to either: (i) allocate half of the cache to each VM, or (ii) allocate the entire cache to the random workload.

However, for maximizing IOPS, allocating half of the cache to each is not a good distribution because the sequential workload gets similar IOPS from the SSD and from the SAN store. Figure 10 (b) shows the cost curves and how the sequential workload benefits only minimally from the SSD cache. Thus, the best distribution to minimize $1/IOPS$ (i.e., maximize IOPS) is to allocate the entire cache space to the random workload. Our solution learns these properties about the workloads dynamically and allocates SSD cache space only to workloads that benefit from it.

C. Storage QoS

1) Proportional Share Fairness: To evaluate how our solution implements proportional share fairness, we dynamically varied the desired (target) latency proportions for two VMs running an instance of Iometer with mixed read-write and random accesses. Each workload accesses a file of 1 GB with 4 OIOs; the initial cache partition size is set to 500 MB per VM. Iterations of the fairness algorithm were configured to occur every 5 seconds and cache size increments and decrements were in units of 10 MB.

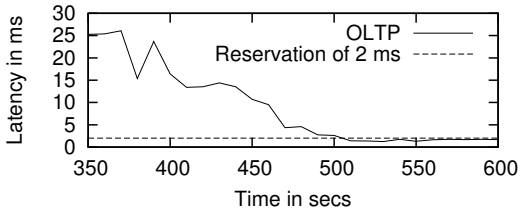

Fig. 12. **Latency reservation.**

Figure 11 shows the outcome of configuring latency shares for VMs. The requested changes in the desired latency shares are listed at the top of the plot. When latency shares are modified, within 2–3 minutes, our solution adapts per-VM latency to the specified shares with less than 5% of error. The changes, however, are not immediate because the cache was configured to only move 10 MB of data from/to the cache every 5 seconds; if we assume that latency is proportional to cache size, then the slope of the latency adaptation would be proportional to the frequency of the iterations.

2) IOPS and Latency Reservations: To evaluate the effectiveness of IOPS and latency reservations, we co-located 3 workloads on the same host and set latency reservations for only one of them, 350 seconds into the experiment. For the workloads, we used Filebench OLTP in its default configuration and a footprint of 8 GB per workload to stress an SSD cache limited to a total of 12 GB across the three workloads. Filebench OLTP performs 8 KB random IOs, with 75% of reads and writes and keeps 16 OIOs at all time.

Figure 12 shows the effectiveness of our solution. In Centaur, reservations prescribe bounds on average latencies and not maximum latencies. Again, reservations are not met immediately since the system was bound by the speed of the data mover. The target workload needed 90% of its footprint cached to meet its latency reservation target which in turn caused 7GB to be moved from the SAN store to the SSD, thus requiring 150 seconds. We performed a similar experiment for IOPS reservation and found that the system was able to meet reservations with less than 10% of error. Additional analysis of the data revealed that the interference introduced by the data mover with respect to meeting the reservation target was much higher for an IOPS reservation than with latency reservation ($\sim$10X more).

D. Overheads

Next, we evaluate the overhead of the cache partitioning solution by comparing storage performance across 3 caching configurations: (1) our partitioned caching system (full sys), (2) when the workload accesses the backing store directly without any caching (van disk), and (3) a variant of our system where we do not maintain persistent block mappings for the SSD cache (modif sys).

We used a single Iometer workload within a Windows VM for this experiment. The workload is a 50% mix of reads and writes on 1 GB with a cache of 100 MB. Figure 13 shows warm cache IOPS, and average and maximum latency for each of the three configurations. The first observation is that in the average case, both caching systems perform better than the system without caching. The caching variants both provide approximately the same average latency and IOPS indicating

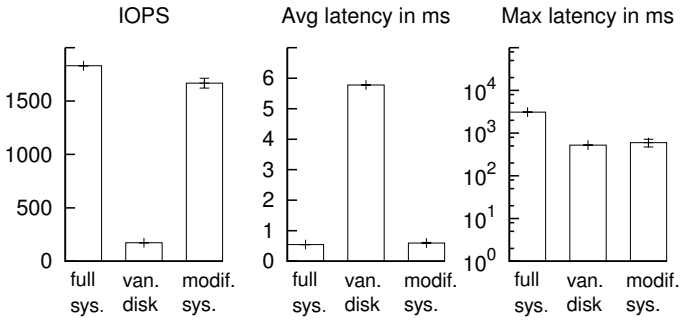

Fig. 13. **Full system (full sys) performance compared to a non-caching solution (van disk) and a simplified cache that does not maintain persistent mappings for cache entries (modif sys).** 4 KB aligned random reads/writes on 1 GB using 100 MB of cache.

that average case overhead is acceptable. However, the maximum latency of the full system with persistent mappings is almost one order of magnitude higher than the one without persistent mappings. We noticed that these spikes in latency occur every mover epoch, and specifically, every time the journal is replayed. The scheduling and execution of mover epochs were not optimized in our system and we expect that optimizing these carefully would help reduce the observed latency spikes. Another interesting observation is that IOPS for the full system are higher than the non-persistent cache variant; the journal replaying was merging and sequentializing writes much better, an unplanned benefit!

IX. Related work

Managing cache contention across multiple workloads is a well-studied problem in the research literature starting with seminal work on memory contention done by Denning [8] to the very recent works on SSD cache partitioning [22], [28], [36]. Centaur distinguishes itself from this body of work in its novel and robust approach to online cache partitioning that is free of assumptions contained within established cache partitioning techniques and that do not apply to external memory flash caches, and by building practically usable QoS control knobs for storage administrators.

A. Partitioning for Performance

Rajkumar *et al.* prove that optimal cache partitioning is NP-hard [32]. Stone *et al.* [38] analytically showed that for minimizing overall miss rate and assuming convex MRC curves, optimal partitioning can be achieved by choosing allocation sizes for which derivatives of miss rates are the same across all workloads. Thiebaut *et al.* [40] then developed a practical implementation of Stone's algorithm by approximating MRC derivatives rather than constructing full MRCs. Suh *et al.* [39] proposed an approach to obtain the convex minorant of MRCs and a greedy partitioning CPU caches.

Arguing that a unified LRU main memory cache cannot avoid cache pollution, several researchers proposed partitioning buffer caches across processes [18], [45], [37] or storage classes [9]. Zhou *et al.* [45] argued that most MRCs for main memory workloads are convex and used a greedy MRC-based approach to partition main memory for minimizing aggregate miss ratio. Soundararajan *et al.* [37] address partitioning of multiple levels of caches including memory and storage server caches to minimize average latency using latency curves that are similar to the ones we use in this work for host-side SSD caches. Goyal *et al.* [9] also address the problem of dynamically partitioning storage server caches to address workload changes, again assuming that MRCs are convex.

Our approach has the following distinguishing characteristics from all of the related work. Previous work on modeling cache wastage has been focused on LRU-based CPU caches [41]. We generalize observations on wastage to other policies such as Aging-LFU and ARC. Further, we present empirical evidence to quantify the rather serious implications of wastage.

Much of the current SSD caching systems do not explicitly address inter-workload cache contention and simply employ unified cache replacement policies [6], [16], [20], [30]. S-Cave implements cache partitioning by estimating cache demand and allocation of cache space to the individual workloads sharing the cache, optimizing the effective utility of a partition for a specific workload [22]. vCacheShare also implements cache partitioning, optimizing *cache utility* but by favorably allocating space to workloads that are better aligned with their cache management mechanisms [28]. While vCacheShare relies on an existing cache partitioning technique, S-Cave relies on a periodic heuristic that reallocates space based on hit-rate feedback. We empirically demonstrate that assumptions made by conventional cache partitioning approaches do not apply to out-of-core caches and that using existing cache partitioning approaches could lead to large errors in partitioning. Centaur is grounded in the established theory of MRCs but free from such assumptions. Further, Centaur is also the first to propose and validate the use of *throughput curves* with a similar capability as *latency curves*.

B. Storage QoS

Storage QoS has been addressed from several perspectives ranging from partitioning memory to IO throttling to storage migration. Host-side SSD cache sizing represents a powerful control knob for storage QoS that is complementary to these solutions. It provides much larger space for storage caching than main memory ballooning. However, it is less responsive than ballooning. In contrast to the IO throttling [12], IO scheduling controls [15], and storage migration [13], [14], [24], [23], that are all ultimately limited by storage performance, SSD caching provides a mechanism that allows greater separation of VM IO performance from storage performance.

Recent work by Sehgal *et al.* on using host-side SSD dynamic cache sizing as an IO latency control knob [36]. They employ a control loop of observed latencies to deliver specified latency guarantees probabilistically. In this paper, we go well beyond meeting specified latency targets and support a variety of optimization objectives such as minimizing IO latencies across groups of workloads, IO throughput control, and proportional share IO throughput fairness.

X. Conclusion

Host-side SSDs provide new opportunities for improving and managing storage performance. In this paper, we demonstrated that host-side SSDs demand new techniques for performance management and propose a system, Centaur, that

addresses both performance maximization and storage QoS goals. We showed that managing the SSD as a unified cache for VMs can lead to *wastage* of cache space with the ARC, LRU and aging-LFU policies. Further, we demonstrated that conventional partitioning techniques, that have worked well for CPU, main memory, and storage buffer caches, do not work well for host-side SSD caches. We then proposed a new cache partitioning approach that employs online MRC construction combined with periodic partitioning and data movement to efficiently allocate cache space to individual workloads. We utilized this solution to build a new, richer set of storage QoS controls for latency and IOPS. An evaluation of Centaur, implemented for VMware ESX, demonstrated that it works well in practice and can be used for meeting both performance maximization and storage QoS goals.

XI. Acknowledgments

We would like to thank Murali Vilayannur, Diplreet Bindra, and Haripriya Rajagopal who inspired this work and helped with the initial phases of the project. This work is supported in part by NSF awards CNS-1018262 and CNS-1448747, and a NetApp Faculty Fellowship.

References

[1] Bootchart. http://www.bootchart.org.

[2] Fio. http://linux.die.net/man/1/fio.

[3] Iometer. http://www.iometer.org.

[4] Bhadkamkar, M., Guerra, J., che, L. U., Burnett, S., Liptak, J., mi, R. R., and Hristidis, V. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *Proc. of USENIX FAST* (2009).

[5] Bilardi, G., Ekanadham, K., and Pattnaik, P. Efficient Stack Distance Computation for Priority Replacement Policies. In *ACM International Conference on Computing Frontiers* (2011).

[6] Byan, S., Lentini, J., Madan, A., Pabon, L., Condict, M., Kimmel, J., Kleiman, S., Small, C., and Storer, M. Mercury: Host-side Flash Caching for the Data Center. In *Proc. of IEEE MSST* (2012).

[7] Denning, P. *The Locality Principle, Communication Networks And Computer Systems (Communications and Signal Processing)*. Imperial College Press, London, UK, 2006.

[8] Denning, P. J. Working Sets Past and Present. *IEEE Trans. Softw. Eng. 6* (1980), 64–84.

[9] Goyal, P., Jadav, D., Modha, D. S., and Tewari, R. CacheCOW: QoS for Storage System Caches. In *Proc. of IWQoS* (2003).

[10] Guerra, J., Marmol, L., Campello, D., Crespo, C., Rangaswami, R., and Wei, J. Software Persistent Memory. In *Proc. of the USENIX ATC* (2012).

[11] Guerra, J., Pucha, H., Glider, J., Belluomini, W., and Rangaswami, R. Cost Effective Storage using Extent Based Dynamic Tiering. In *Proc. of USENIX FAST* (2011).

[12] Gulati, A., Ahmad, I., and Waldspurger, C. A. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proc. of FAST* (2009).

[13] Gulati, A., Ahmad, I., and Waldspurger, C. A. PESTO: Online Storage Performance Management in Virtualized Datacenters. In *Proc. of ACM SOCC* (2011).

[14] Gulati, A., Kumar, C., Ahmad, I., and Kumar, K. BASIL: Automated IO Load Balancing Across Storage Devices. In *Proc. of USENIX FAST* (2010).

[15] Gulati, A., Merchant, A., and Varman, P. J. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proc. of USENIX OSDI* (2010).

[16] Holland, D. A., Angelino, E., Wald, G., and Seltzer, M. I. Flash Caching on the Storage Client. In *Proc. of USENIX ATC* (2013).

[17] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Section 11.1 INTERNAL CACHES, TLBS, AND BUFFERS*. 2009.

[18] Kim, J. M., Choi, J., Kim, J., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References. In *Proc. of USENIX OSDI* (2010).

[19] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by Simulated Annealing. *Science* (1983).

[20] Koller, R., Marmol, L., Rangaswami, R., Sundararaman, S., Talagala, N., and Zhao, M. Write Policies for Host-side Flash Caches. In *Proc. of USENIX FAST* (2013).

[21] Koller, R., Verma, A., and Rangaswami, R. Generalized ERSS Tree Model: Revisiting Working Sets. *Performance Evaluation 67*, 11 (2010), 1139–1154.

[22] Luo, T., Ma, S., Lee, R., Zhang, X., Liu, D., and Zhou, L. S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance. In *Proc. of PACT* (2013).

[23] Mashtizadeh, A., Cai, M., Tarasuk-Levin, G., Koller, R., and Garfinkel, T. XvMotion: Unified Virtual Machine Migration over Long Distance. In *Proc. of USENIX ATC* (2014).

[24] Mashtizadeh, A., Celebi, E., Garfinkel, T., and Cai, M. The Design and Evolution of Live Storage Migration in VMware ESX. In *Proc. of USENIX ATC* (2011).

[25] Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L. Evaluation Techniques for Storage Hierarchies. *IBM Syst. J.* (1970).

[26] McDougall, R. Filebench: Application Level File System Benchmark.

[27] Megiddo, N., and Modha, D. S. Outperforming LRU with an Adaptive Replacement Cache Algorithm. *Computer* (2004).

[28] Meng, F., Zhou, L., Ma, X., Uttamchandani, S., and Liu, D. vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment. In *Proc. of USENIX ATC* (2014).

[29] Prabhakar, R., Srikantaiah, S., Patrick, C., and Kandemir, M. Dynamic Storage Cache Allocation in Multi-server Architectures. In *Proc. of SC* (2009).

[30] Qin, D., Brown, A. D., and Goel, A. Reliable Writeback for Client-side Flash Caches. In *Proc. of USENIX ATC* (2014).

[31] Qureshi, M. K., and Patt, Y. N. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proc. of MIRCO* (2006).

[32] Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D. Practical Solutions for QoS-based Resource Allocation Problems. In *In Proc. of RTSS* (1998).

[33] Robinson, J. T., and Devarakonda, M. V. Data Cache Management Using Frequency-based Replacement. Proc. of ACM SIGMETRICS.

[34] Saemundsson, T., Bjornsson, H., Chockler, G., and Vigfusson, Y. Dynamic Performance Profiling of Cloud Caches. In *Proc. of ACM SOCC* (2014).

[35] Santana, R., Lyons, S., Koller, R., Rangaswami, R., and Liu, J. To ARC or not to ARC. In *Proc. of USENIX HotStorage* (2015).

[36] Sehgal, P., Voruganti, K., and Sundaram, R. SLO-aware hybrid store. In *Proc. of IEEE MSST* (2012).

[37] Soundararajan, G., Lupei, D., Ghanbari, S., Popescu, A. D., Chen, J., and Amza, C. Dynamic Resource Allocation for Database Servers Running on Virtual Storage. In *Proc. of FAST* (2009).

[38] Stone, H. S., Turek, J., and Wolf, J. L. Optimal Partitioning of Cache Memory. *IEEE Trans. Comput. 41*, 9 (1992).

[39] Suh, G. E., Rudolph, L., and Devadas, S. Dynamic Partitioning of Shared Cache Memory. *The Journal of Supercomputing 28*, 1 (2004).

[40] Thiebaut, D., Stone, H., and Wolf, J. Improving Disk Cache Hit-Ratios Through Cache Partitioning. *IEEE Transactions on Computers 41*, 6 (1992), 665 –676.

[41] Verma, A., Koller, R., Useche, L., and Rangaswami, R. SRCMap: Energy Proportional Storage Using Dynamic Consolidation. In *Proc. of USENIX FAST* (2010).

[42] VMware, Inc. VMware Virtual SAN. http://www.vmware.com/products/virtual-san/, 2013.

[43] Waldspurger, C. A., Park, N., Garthwaite, A., and Ahmad, I. Efficient MRC Construction with SHARDS. In *Proc. of USENIX FAST* (2015).

[44] Wires, J., Ingram, S., Drudi, Z., Harvey, N. J. A., and Warfield, A. Characterizing Storage Workloads with Counter Stacks. In *Proc. of USENIX OSDI* (2014).

[45] Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., and Kumar, S. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. *Proc. of ASPLOS* (2004).