vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment

Fei Meng, *North Carolina State University;* Li Zhou, *Facebook;* Xiaosong Ma,
North Carolina State University and Qatar Computing Research Institute;
Sandeep Uttamchandani, *VMware Inc.;* Deng Liu, *Twitter*

**This paper is included in the Proceedings of USENIX ATC '14:
2014 USENIX Annual Technical Conference.**

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment

Fei Meng†, Li Zhou¶,∗Xiaosong Ma†§, Sandeep Uttamchandani‡ and Deng Liu◇∗

†North Carolina State University, fmeng@ncsu.edu
¶Facebook Inc., lzhou@fb.com   §Qatar Computing Research Institute, xma@qf.org.qa
‡VMware Inc., suttamchandani@vmware.com   ◇Twitter Inc., dengl@twitter.com

Abstract

Server Flash Cache (SFC) is increasingly adopted in virtualization environments for IO acceleration. Deciding the optimal SFC allocation among VMs or VM disks is a major pain-point, dominantly handled manually by administrators. In this paper, we present *vCacheShare*, a dynamic, workload-aware, policy-driven framework for continuous and automated optimization of SFC space partitioning. Its decision-making is based on multiple IO access characteristics. In particular, vCacheShare adopts a cache utility model that captures both longer-term locality behavior and transient locality spikes.

This paper validates the growing applicability of analytical programming techniques to solve real-time resource management problems, traditionally addressed using heuristics. We designed vCacheShare to coordinate with typical VM mobility events and implemented it within the widely used ESXi hypervisor. We performed extensive evaluation using 13 representative enterprise IO workloads, one IO benchmark, and two end-to-end deployment test cases targeting Virtual Desktop Infrastructure (VDI) and data warehousing scenarios respectively. Our results verified the advantage of vCacheShare over implicit management schemes such as global LRU, and confirmed its self-adaptive capability.

1 Introduction

Solid State Disks (SSDs) are being increasingly used in virtualized environments as an SFC (Server Flash Cache) to accelerate I/O operations of guest Virtual Machines (VMs). However, growing CPU bandwidth and memory capacities are enabling higher VM-to-server consolidation ratios, making SFC management a nightmare. The onus of proportional allocation of SFC space among VMs is handled manually by administrators today, based on heuristics and oftentimes simply guesswork. Besides, allocations should not be one-time activities, but continuously optimized for changing characteristics of workloads, device service times, and config-

uration events related to VM mobility. In order to effectively leverage flash for its $/IOPS advantages and to address existing performance bottlenecks within the infrastructure, it is critical to extend the hypervisor to automate and continuously optimize the space management of SFC, similar to other hardware resources.

The primary use-case of SFC is to speed up applications running inside VMs. Scenarios with performance bottleneck of the backend storage arrays benefit the most, delaying capital investments in provisioning of new hardware. At the hypervisor level, SFC space optimization translates to reducing the I/O latency of VMs, which are prioritized based on administrator input. The optimization needs to take into account multiple dimensions:

- VM priority: VMs have different importance depending on applications running inside.
- Locality of reference: A VM running low-locality workload(s) does not benefit as much from caching and should therefore receive smaller allocation.
- I/O access characteristics: A VM running write-heavy workload(s) may receive smaller benefit and incur higher cost with flash caching, due to SSD devices' asymmetric read-write performance and write durability concerns.
- Backend storage device service times: A VM Disk on faster (or less busy) backend storage benefits less from SFC allocation.
- Configuration events: Hypervisors are optimized to continuously monitor and optimize the placement of resources from a virtualized pool of hardware. Guest VMs can be migrated to other servers without application down time, with ready functionality available such as VMware vMotion.

Current commercial solutions [1, 2, 3, 4, 5, 6] require administrators to carve static space allocations at the time of enabling SFC for VMs. Meanwhile, memory virtualization techniques within hypervisors are not directly applicable to SFC space management. The reasons are two-fold. First, memory resources are explicitly requested by users and the hypervisor has to satisfy

---

∗With VMware during this work

such requests with reasonable latency, while cache resources are transparent to applications and allocated by the hypervisor for performance enhancement. Second, typical VMs expect their data to be memory-resident in most cases, and heavy swap activities are seen as an indication of intensive memory contention and may subsequently trigger VM migration. In contrast, flash cache is designed as an intermediate level of storage, in order to accelerate IO rather than to accommodate all VMs' combined working set. At the same time, existing CPU cache partitioning techniques [7, 8, 9] will not be able to exploit options available to managing flash-based SSD cache spaces. The latter problem is significantly different in aspects such as response time, space availability for optimization, access patterns, and performance constraints. In general, affordable in-memory book keeping and the much slower storage access enable a much larger solution space and more sophisticated algorithms. Unique flash storage issues such as write endurance also bring additional complexities.

Another alternative is to retrofit IO QoS management techniques [10, 11] for SFC management. IO QoS aims at providing fairness and prioritization among requests serviced under bandwidth contention. This problem is different from SFC management, as the definition of resource contention is much less straight-forward for caching — the criteria here is not "actively used cache space" by the VM, but rather the performance gain through caching based on its IO access locality behavior.

In this paper, we present *vCacheShare*, a dynamic, workload-aware, policy-driven framework for automated allocation of flash cache space on a per-VM or per-VM-Disk basis. vCacheShare combines the dimensions of data locality, IO operation characteristics, and device service time into a self-evolving *cache utility model*. It approaches flash cache allocation/partitioning as a constrained optimization problem with the objective of maximizing the cumulative cache utility, weighted by administrator-specified VM priorities.

Contributions This paper addresses the pain-point of optimal space partitioning of SFC in a virtualization environment. Our goal is to develop a mathematical optimization solution for SFC management. With the trend of CPU threads per socket doubling every 12-18 months [12], we think the time has come for resource management techniques to adopt such strategies for dynamic multi-dimensional optimization.

We consider the major contributions of this work as follows: **(1)** We addressed a unique challenge in managing time-varying IO behavior common to many VM workloads, namely, the conflict in long-term behavior analysis required for accurate cache hit ratio estimation and fast response required for handling transient locality bursts. In response, we propose a novel cache utility modeling approach that takes into consideration both long-term reuse patterns and short-term reuse intensity levels. **(2)** We designed vCacheShare, a cost-effective, adaptive framework that scales in today's high-VM-density environments, based on the proposed techniques. **(3)** We implemented our prototype within a widely used commercial hypervisor, VMware ESXi 5.0, and performed extensive evaluation using a combination of 10+ real-world traces, one IO benchmark, and two workload deployment tests with up to 100 VM instances.

2 A Bird's Eye-view of vCacheShare

2.1 Target Environment

Typically in a datacenter using virtualization, multiple hosts share the same SAN/NAS storage to utilize the rich set of functionality provided by the hypervisor. The hosts sharing storage form a *cluster* within the datacenter. In such an environment, a file or block storage is visible to the *Virtual Machine (VM)* as a *Virtual Machine Disk (VMDK)*. One VM could have multiple VMDKs just as a physical machine may have several physical disks. The hypervisor (sometimes referred to as *server*) internally tracks VMs and VMDKs using Universally Unique Identifiers (UUIDs). The UUIDs for VMs and VMDKs are unique cluster-wide. In our target environment, both VMs and VMDKs can be moved non-disruptively across physical resources — commonly referred to as vMotion and Storage vMotion respectively.

vCacheShare adopts the *write around* cache policy, i.e., writes bypass the SFC and directly go to the back-end disks. There are several reasons to focus on read intensive workloads. First, there are multiple layers of memory cache above SFC, which are shown to have reduced the read-after-write percentage [13] and consequently weakened the benefit of write caching. Second, write around prolongs the SFC lifetime by reducing writes and prioritizing data to be re-read. Finally, it simplifies design and implementation by relaxing consistency requirement. In fact, several commercial SFC solutions [2, 4] started from write around or write through, including VMware's recent vFRC (Flash Read Cache).

2.2 vCacheShare Overview

vCacheShare decides the SFC space allocation sizes on per-VM or per-VMDK basis, based on runtime analysis of the VMs/VMDKs' IO access characteristics, along with hardware/administrative settings.

Figure 1 depicts vCacheShare's location in our target environment, as well as its internal architecture. The vCacheShare decision making workflow requires the coordination among its major components: (1) *monitor*, which intercepts IO requests and logs information about them, (2) *analyzer*, which periodically processes trace
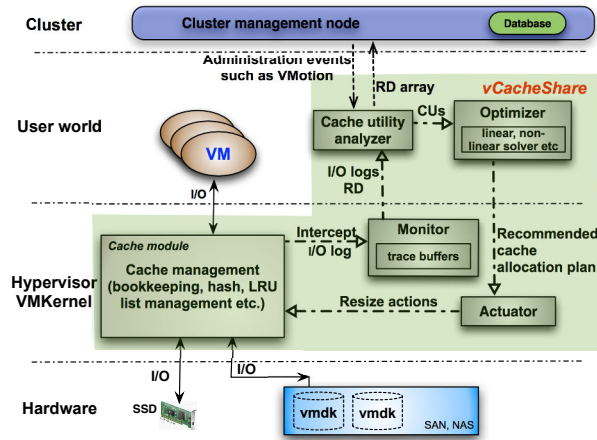
Figure 1: vCacheShare architecture

data logged by the monitor and extracts per-VMDK access characteristics, (3) *optimizer*, which, based on the analyzer-extracted workload characteristics and a *cache utility model*, dynamically optimizes the cache partitioning plan, and (4) *actuator*, which executes the optimized plan. The analyzer and optimizer are implemented as user-level agents, while monitor and actuator are implemented within the VMkernel.

In addition to runtime workload monitoring input, vCacheShare also takes from administrators two key configuration parameters: (1) a per-VMDK IO priority level and (2) per-VMDK cache allocation lower and upper bounds. The lower bound is also referred to as *reserved cache space*. Not all VMs/VMDKs need flash cache enabled: for example, some VMDKs may be provisioned out of an all-flash array.

In the rest of the paper, we base our discussion on per-VMDK cache space allocation, though the techniques and observations also apply to per-VM allocation.

2.3 Event-Driven Analysis and Re-Optimization

The vCacheShare design is well integrated with the hypervisor's VM management. In particular, its workflow has to be coordinated with common VM state transitions and migration operations (for both VMs and storage devices). Table 1 lists vCacheShare's actions triggered by such common VM management activities. Though the events are hypervisor specific, we believe that the vCacheShare workflow and event-handling methodology can be applied to other hypervisors as well.

3 Monitoring

The vCacheShare monitoring module sits within the hypervisor kernel, on the IO path between the VMs and the SFC. It intercepts, records, and analyzes all IO accesses from the VMs on each physical host. For each VMDK, it performs periodic online trace processing to extract the reuse pattern needed by the vCacheShare cache utility model (Section 4).

vCacheShare's trace data collection is performed on a per-VMDK basis. For each cache-enabled VMDK, vCacheShare sets up a circular log buffer in kernel memory for storing trace entries. Their content can be committed to SSD storage regularly and asynchronously to reduce memory usage. The in-memory and on-SSD circular buffer sizes, as per-VMDK configurable parameters, limit the maximum memory/SSD space consumed by vCacheShare IO trace collection. In our experiments, each VMDK has a 4MB on-SSD trace buffer, plus two 64KB in-memory ones for double buffering.

Each trace entry is 15 bytes long, containing 8 fields. Among them, the VMK_UUID and the VMDK_UUID fields identifies the IO request issuer (VM) and the destination device (VMDK), respectively. The timestamp field logs the request time. isRead records the IO operation type, while LBA and len define the initial logical block address and IO request size (in terms of blocks). latency records the total service time to complete the IO. For each entry, most of the data collection happens before the IO request hits the flash caching module, except for latency, which is logged upon IO completion. Finally, isCached tags IOs serviced from the flash cache rather than from the backend storage. As measured with our implementation, such trace collection activities add only 4-5 nanoseconds to the overall microsecond- or millisecond-level IO service time.

Note that the above vCacheShare monitoring and trace collection is in addition to the hypervisor's built-in IO profiling. Such profiling gather high-level statistics through approaches such as moving window averages, regarding IOPS, latency, read-write ratio, request sizes, etc., on a per VM or VMDK basis. The IO statistics are averaged over a user configurable monitoring window (default at 20 seconds). Some of these data items, e.g., read percentage and average latency, are used in vCacheShare's cache utility model calculation.

4 Dynamic Cache Utility Analysis

vCacheShare performs its cache space allocation optimization based on *cache utility (CU)* [7] on a per-VMDK basis. CU reflects the effectiveness of allocating SFC space, i.e., the relative performance gain brought by additional SFC space allocation.

Cache utility depends on several factors. Intuitively, a VMDK generates better cache utility if it displays good reference locality or is built on backend storage devices with higher read latency (either due to slower device speed or saturation). In addition, vCacheShare favors read-heavy workloads, both due to the current write around caching design and well-known performance plus endurance problems with flash writes. Such limitations

VM Events	Actions by vCacheShare Framework
VM Power-Off, Suspend, vMotion Source	Trigger optimization to re-allocate; free IO trace buffers for all associated VMDKs
VM Bootstrap, Power-On, Resume	Make initial allocation based on reserved cache space and priority settings; start trace collection
vMotion Destination	Trigger optimization to re-allocate based on IO characteristics migrated with the VMDKs involved in vMotion
Storage vMotion (runtime backend device change)	Suspend analysis/optimization till completion; evict device service latency history; trigger re-optimization upon vMotion completion
VM Fast Suspend, vMotion Stun	Reserve cached data; lock cache space allocation to involved VMs by subtracting allocated size from total available cache size

Table 1: Actions by vCacheShare upon VM Events

make read-intensive access patterns more flash-cache-friendly even with write caching turned on in the future. Finally, vCacheShare takes into account an attribute much less often factored in for cache partitioning: the intensity of re-accesses. This metric considers both locality and access speed, allowing our model to favor workloads that generate more reuse of cached data in unit time.

Device access speed and read-write ratio can be measured rather easily. Hence our discussion focuses on estimating access locality and reuse intensity.

4.1 Measuring Locality

Many prior studies have examined cache hit ratio prediction/estimation based on workload access characteristics [14, 15]. vCacheShare adopts a low-overhead online estimation approach that examines access reuse distance, similar to that used in CPU cache partitioning [14]. However, vCacheShare explores reuse distance for storage cache partitioning, where more sophisticated follow-up algorithms could be explored due to both more resource availability and slower data access rates.

For each identified re-access, where the same block is revisited, the *reuse distance* is measured as the number of distinct blocks accessed between the two consecutive uses of that block [14]. vCacheShare constantly monitors the *distribution of reuse distances*, to measure the temporal locality in accesses to each VMDK. Considering that our current read-only cache design, only read accesses are counted in vCacheShare's reuse distance calculation. Such per-VMDK overall temporal locality for reads is expressed in a CDF (cumulative distance function) of reuse distances.

To dynamically build and update the reuse distance distribution, vCacheShare first extracts relevant data from its IO traces using a two-phase $O(n)$ process, where n is the number of trace entries. This Map-Reduce like process can easily be parallelized to use multiple cores available, if necessary.

vCacheShare maintains one hash table per VMDK. In the "Map" phase, IO traces are scanned into the appropriate hash table. We pre-process IO logs here to address large block IOs and un-aligned 4K accesses. A log entry request for greater than 4KB is divided into multiple 4KB entries, with LBA adjusted for each. This is inline with our target flash cache implementation, where cache book-keeping is done at the 4KB granularity. Based on our analysis of real-world workloads from the SNIA MSR traces [16], only 5-10% of accesses are unaligned, which are discarded in our processing. For each read log entry, a pointer to the entry is inserted into the per-VMDK hash table, using its LBA as the key.

In the "Reduce" phase, vCacheShare scans the values of the per-VMDK hash tables, generating for each a *Reuse Distance Array* (*RD Array*). This is an array of lists that for each LBA, stores its reuse distances between each pair of adjacent accesses. The reuse distance CDF can then be easily constructed using histograms.

As mentioned in Section 3, vCacheShare maintains per-VMDK cyclic trace buffers. The per-VMDK hash tables are updated according to a time stamp marking the oldest valid log entry in the trace buffer. Older entries are discarded from the hash tables, during either insertion or periodic garbage collection sessions. This way, vCacheShare automatically assesses access patterns in a sliding-window manner, with a *sampling window size* equal to the size of the cyclic trace buffer.

4.2 Locality-based Hit Ratio Estimation

With the reuse distance CDF calculated as described above, we can produce a rough cache hit ratio estimation for each VMDK at a given cache size. Assuming an LRU or LRU-like cache replacement algorithm, the hit ratio can be computed by dividing the total number of cache hits (re-accesses whose reuse distance falls under the cache size) by the total number of accesses.

The effectiveness of such estimation, however, depends heavily on the aforementioned sampling window size. Intuitively, if the sampling window is too small, vCacheShare will not be able to fully capture a workload's temporal locality. On the other hand, large sampling windows will produce slower response to workload behavior changes, as new access patterns will be "dampened" by a large amount of older entries. In addition, larger sampling windows require more space and time overhead for trace collection and analysis.

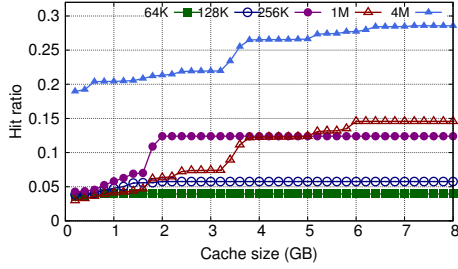Figure 2 demonstrates the impact of sampling win-

Figure 2: Impact of sampling window size on hit ratio estimation

dow size on hit ratio estimation, through a representative workload, "web", member of the SNIA MSR traces [16]. It shows that in general, larger sampling window sizes produce higher cache hit ratio estimates, and reach hit ratio saturation points later (with exceptions likely due to well known caching anomalies). On the other hand, a reasonably large sampling window size (e.g., 1MB) seems to produce very similar trend in hit ratio growth to a window size a few times larger (4MB). Other read-intensive traces in the MSR collection show similar results. We choose to leave the sampling window size as a tunable parameter. Note that a larger flash device offers more cache space and can benefit from larger sampling windows. This goes well with our design of having SSD-resident cyclic trace buffers, as the same device can accommodate higher space overhead as well.
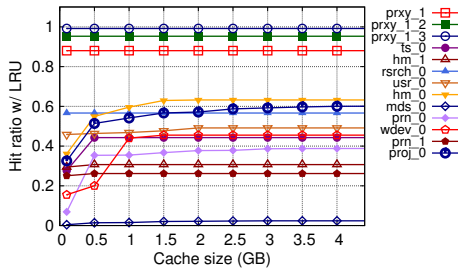

Figure 3: Hit ratio as a function of cache size

To further verify the need of cache partitioning, Figure 3 shows the cache hit ratios of a selected set of representative MSR traces under different cache sizes. Each trace is replayed for the first one million IO accesses. We can clearly see that (1) different workloads from the same enterprise environment have diverse locality properties, and (2) among workloads that possess certain locality, there are large variances in their working set sizes.

4.3 Measuring Reuse Intensity

Trace based temporal locality analysis has been shown to provide quite accurate hit ratio estimation [14, 15]. However, for cache partitioning between concurrent accesses to multiple VMDKs, such analysis fails to consider the *relative speed* of cache accesses. In particular, it does

not capture bursts in data reuse. vCacheShare needs to identify and absorb *locality spikes*, caused by common activities such as boot storms in VDI environments [17]. Fast response is especially challenging with a larger sampling window, due to its dampening effect.

For this, we introduce another factor in modeling CU: *reuse intensity*, which measures the **burstiness of cache hits**. This metric captures the fast changing aspect of CU, to bias cache allocation toward VMDKs undergoing locality spikes. More accurately, for $VMDK_i$, its reuse intensity RI_i is defined as

$$RI_i = \frac{S_{total}}{t_w \times S_{unique}} \quad (1)$$

Here, for a given monitoring time window size t_w, S_{total} and S_{unique} describe the total read access volume and the total size of unique blocks read (i.e., access footprint). E.g., with t_w at 5 minutes, within which 1G blocks are read from $VMDK_i$ accessing 1000 unique blocks, the resulting RI_i will be $\frac{1G}{1000 \times 300s}$. This metric effectively captures the locality-adjusted per-VMDK read throughput: an influx of accesses to new blocks brings similar growth to total access volume and footprint, therefore lowering RI; an influx of re-accesses, on the other hand, increases the former but not the latter, inflating RI.

t_w is another tunable vCacheShare parameter, preferably with relatively small values for better responsiveness. It asserts little computation overhead, and can be maintained as a per-VMDK moving average. In situations where detailed trace collection/analysis is impractical or overly expensive, RI can contribute independently to the vCacheShare CU model as a cheap and less accurate alternative, since it does reflect temporal locality. However, t_w should ideally be decided adaptively based on the rate of locality change measured in real time. We are currently performing follow-up study on devising such a dynamic algorithm. In this paper, we empirically observed that t_w values between 2 and 60 seconds are suitable for the workloads tested. Our experiments used 60 seconds unless otherwise noted.
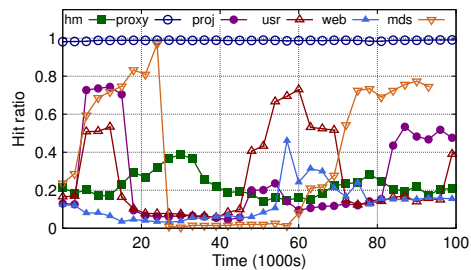

Figure 4: Temporal pattern in cache hit rate; the hit rate is calculated as a moving average

To verify the existence of locality spikes, we examine the temporal locality shifts in the MSR workloads.

Figure 4 plots the hit rate changes for six sample traces along the execution timeline. The hit ratio in y axis is calculated with a 60-second moving window. The results illustrate the time-varying nature of reference locality, as well as the existence of transient locality spikes in multiple traces (such as *mds*, *usr*, and *proj*).

4.4 Summary: Cache Utility Model

Putting everything together, vCacheShare's CU Model for $VMDK_i$ is generated as a function of its estimated cache hit rate $H_i(c)$ (where c is the target cache partition size), reuse intensity RI_i, average target device latency l_i, and read ratio RR_i (fraction of total accesses that are reads): If we expand the variables to show the input of the model, we end up with the following equation:

$$CU_i = l_i \times RR_i \times (H_i(c) + \alpha \hat{RI}_i) \qquad (2)$$

Here $H_i(c)$ generates the estimated hit ratio for $VMDK_i$ at cache partition size c, based on its locality observed in the previous sampling window. $\hat{RI}_i$ gives the reuse intensity observed for the same VMDK, *normalized* to the highest RI across all VMDKs in the previous intensity monitoring window. Therefore, both $H_i(c)$ and RI_i have values between 0 and 1. α is included as an additional tuning knob to adjust the relative weight between longer-term, more accurate temporal locality, and short-term locality spikes. Though set to 1 in our evaluation, system administrators can change α to favor persistently cache-friendly workloads or fast-changing, bursty accesses.

5 Optimization

vCacheShare approaches SFC space allocation as a constrained optimization problem. Its optimizer explores different permutations of per-VMDK space allocation values, calculates cache utility as the objective function, and returns the permutation with the global maximum in the search space. More specifically, vCacheShare adopts the following objective function:

$$\sum_{i=1}^{n} priority_i \times CU_i,$$

where n is the number of VMDKs, $priorite_i$ is the user- or administrator-defined priority level (e.g., based on QoS requirement) of $VMDK_i$, and CU_i is the cache utility of $VMDK_i$ defined earlier.

The vCacheShare optimizer will search for the global optimum in the form of a recommended cache allocation plan: $< c_1, c_2, \ldots, c_n >$, which satisfies the following constraints:

$$c_1 + c_2 + \ldots + c_n = C$$

$$c_{min} \leq c_i \leq c_{max}$$

where C is the total available server-side flash cache size, and c_{min}/c_{max} is the administrator-specified per-VMDK cache space lower/upper bound.

This optimization problem is NP-Hard, with much existing research on heuristics such as simulated annealing [18] and hill-climbing [19]. These techniques approximate the optimal solutions via linear, non-linear, or piecewise linear algorithms, among others. We consider such constrained optimization a mature field and beyond the scope of this work. Our prototype implementation uses an open-source simulated annealing tool [18], while vCacheShare is designed to be able to utilize alternative optimization algorithm plug-ins.

6 Execution

The execution module actuates changes in per-VMDK SFC allocation. It also controls the bootstrapping process when vCacheShare is enabled for the first time on a server or when new VMs are added.

For bootstrapping, vCacheShare allocates the administrator-specific per-VMDK cache space lower bound (also referred as *reserved* size) for each enabled VMDK. The rest of the available cache size is then divided among the VMDKs, proportional to their priorities. When a VMDK is first added to a running vCacheShare instance, as its CU estimate is not yet available, it again receives the reserved allocation, by reclaiming space proportionally from existing allocations according to VMDK priorities.

Once bootstrapped, vCacheShare manages SFC as per-VMDK linked lists of blocks, each with a *current_size* and a *target_size*. Upon invocation, the actuator only changes the *target_size* for each list, based on input from the optimization module. The actual sizes then automatically adapt gradually, with VMDKs gaining allocation grabbing SFC blocks from those losing.

With this approach, the speed of cache allocation change automatically reflects the activity level of the VMDKs gaining allocation, again favoring workloads with locality spikes. Such incremental and gradual adjustment also avoids thrashing, where the system oscillates rapidly between two states. Lastly, such lazy eviction maximizes the use of cached data for those VMDKs losing cache allocation.

7 Experimental Evaluation

7.1 Prototype Implementation

We implemented a vCacheShare prototype in the widely used VMware ESXi 5.0 hypervisor.

The trace analysis, CU computation, and optimization modules are implemented as agents in the user world, with ~2800 lines of C++ code. The modeling and optimization results are persisted via service APIs for the monitoring database currently available on a cluster wide management node.

The rest of vCacheShare is implemented in the kernel, with ~2500 lines of C code. First, to enable runtime cache allocation optimization, an SFC framework was implemented within the hypervisor. All the read IOs below 64KB are intercepted to check if the requested data is cached. The cache itself is managed with LRU, with 4KB blocks. The flash device is used exclusively for caching (including vCacheShare management usage) and cannot be directly accessed by the guest VMs or the hypervisor filesystem layer.

7.2 Test Setup

Unless otherwise noted, our experiments used an HP Proliant DL385 G7 Server, with two AMD Opteron 6128 processors, 16GiB memory, and Intel 400GB PCI-E SSD 910. Local SATA disk drives and an EMC Clariion Storage Array are used to stored VMDKs. The tests used 2-100 VMs, running Windows Server 2008 or Ubuntu Linux 11.04, each assigned a single vCPU, 1GB memory, and a 8GB VMDK.

We used the MSR Cambridge traces from SNIA [16]. The traces represent a variety of workloads: user home directories (*usr*), project directories (*proj*), print server (*prn*), hardware monitoring(*hm*), research projects (*rsrch*), web proxy (*prxy*), source control (*src*), web staging(*stg*), terminal server (*ts*), web SQL server (*web*), media server (*mds*), and test web server (*wdev*). They represent IO accesses at the storage disk tier and have accounted for buffer cache as well as application caching effects. This aligns well with vCacheShare's target placement within the hypervisor kernel.

7.3 Result 1: Proof-of-concept Verification

Justifying Cache Partitioning First of all, we demonstrate that explicit, workload-aware cache partitioning is necessary by showing the inadequacy of implicit strategies such as global LRU [7, 20, 21]. In this experiment, we replay two MSR workload traces with a VMware in-house cache simulator with both LRU and vCacheShare replacement algorithms. VM1 runs *src*1_0, which performs a simple data scan, while VM2 plays *prxy*1, with much higher temporal locality. This simulation worked as a proof-of-concept assessment before we set out to implement our vCacheShare prototype, due to the complexity of coding in a real hypervisor.

Figure 5 shows the comparison between using a globally shared LRU cache (GLRU) and vCacheShare (vCS) from a representative segment of the VMs' execution. It plots the VM1, VM2, as well as the overall cache allocation (shown as percentage of the overall cache space occupied) and hit ratio (cumulative from time 0).

This test clearly illustrates the advantage of vCacheShare. With global LRU, the zero-locality VM1 actually grabs *more* cache space than VM2 does, with a hit

(a) Cache utilization
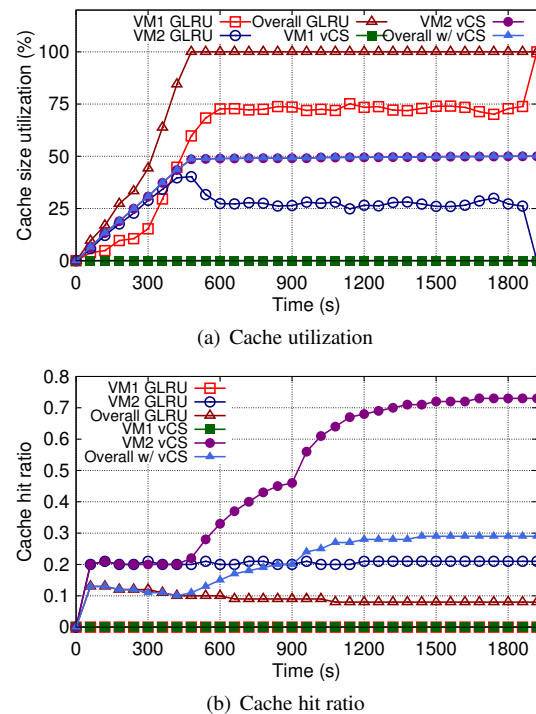

(b) Cache hit ratio

Figure 5: Cache partitioning effectiveness: vCacheShare vs. global LRU

ratio of near zero all the time. Though LRU automatically favors blocks revisited, the fast scanning behavior of VM1 still prevents VM2 from getting enough cache space. With vCacheShare, instead, VM2 gets sufficient cache space to store its entire working set, allowing the cumulative cache hit ratio to gradually increase as the impact of initial cold misses weakens. VM1, on the other hand, is correctly recognized as of little locality and consequently has hardly any cache space allocation. This avoids the space cost of keeping VM1's blocks in the cache brought in by cold misses, only to be evicted later as in the case of using global LRU. Note that vCacheShare is able to reduce the total cache space usage (50% vs. 100% with global LRU), thereby leaving more space for other VMs, while delivering 21% higher overall hit ratio at the end of the execution segment.

Hit Ratio Estimation Accuracy We then assess vCacheShare's capability of predicting a workload's cache hit ratio based on its online temporal locality monitoring. Figure 6 depicts the results in terms of *hit ratio estimation error*, which measures the absolute difference between the predicted and actual cache hit ratios, which evolves for each sampling window along the execution timeline (*x* axis). We replayed week-long SNIA real workload traces in our evaluation. As shown, vCacheShare is able to yield accurate hit ratio estimation in the vast majority of cases, with most data points aggregat-
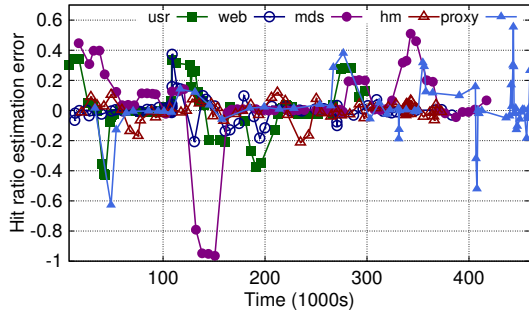
Figure 6: vCacheShare hit ratio estimation accuracy

ing within the [-0.1, 0.1] error range. Meanwhile, there are evident spikes in estimation error, due to fast workload behavior changes. With each spike, vCacheShare is able to correct its large error, though the converging speed depends on the sampling window size as discussed previously. In our analysis of traces collected from a production data center environment, the mean value for the deviation was 0.22 with a standard deviation of 0.4. Fortunately, our reuse intensity metric helps offset the lack of responsiveness to locality changes caused by larger sampling windows.

Necessity of Reuse Intensity (RI) Now we demonstrate that only hit ratio prediction is not enough when the workloads exhibit bursty or intensive accesses within a short time period.
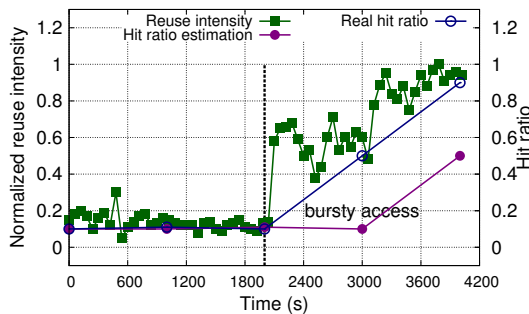

Figure 7: Reuse intensity of MSR trace *web*

Figure 7 shows the RI change (normalized to the largest RI value throughout the execution) of the aforementioned MSR *web* trace. Note the hit ratio prediction is performed on a longer time interval, e.g. several minutes or longer. In contrast, RI is generated more frequently. Figure 7 highlights this by two representative segments of execution concatenated, with small and large hit ratio estimate error respectively. Similar contrast exists for all the traces though. Clearly from the figure, when the hit ratio estimation is close to zero, the RI is close to zero too. However, if there is bursty access between two sampling windows, RI will capture it much more promptly and adapt the cache utility accordingly.

7.4 Result 2: Adaptivity Evaluation

Next, we evaluate vCacheShare's capability of adapting to different policy settings as well as storage backend performance settings. In these experiments, we have 4 VMs running Iometer [22] as a workload generator. In the baseline run, Iometer is set to have 100% random read operations. All the VMs are assigned the same IO priority of 1000 and run the same IO workload, accessing per-VM VMDKs with identical configurations. The expected vCacheShare decision is therefore uniform partitioning across all VMs/VMDKs. The total server flash cache size controlled by vCacheShare is 8GB.
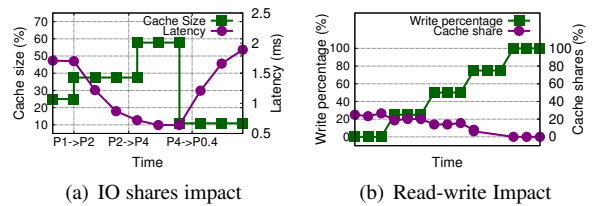

(a) IO shares impact

(b) Read-write Impact

Figure 8: Adaptivity

Varied IO Priorities & Varied Read-Write Ratio: Figure 8(a) shows the effect of varying the IO priority setting on vCacheShare's cache partitioning and resulted IO latency. We change the IO priority of one of the VMDKs along the time line (x axis) to a series of settings: P1=1000, P2=2000, P4=4000, and P0.4=400. Following every priority value change, vCacheShare promptly adjusts this VMDK's cache allocation, while the cache partitioning among the other VMDKs remains uniform. The corresponding latency change follows suite, though it takes longer to stabilize as this VMDK's actual cache footprint expands/shrinks. In the second experiment, the read-write ratio for one of the VMDKs was varied using Iometer. With vCacheShare's bias toward read accesses and read locality, this VMDK sees steady reduction in its cache allocation shares, as shown in Figure 8(b).
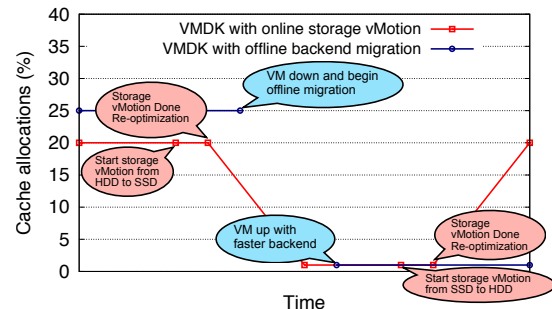

Figure 9: Cache allocation during migration events

Varied Backend Device Latency & Event Handling: In the backend device latency sensitivity test, one VMDK is migrated offline to an flash-backed device. Figure 9 shows the cache allocation for this backend migration scenario. vCacheShare detects a significantly

lower service latency from the "victim" VMDK (eliminating the benefit of flash caching) and consequently redistributes nearly all of its cache allocation shares to the other VMDKs.

Figure 9 also shows the behavior of vCacheShare during an online storage vMotion event that moves one VMDK non-disruptively between storage devices. During this transfer, a transient phase where data is sequentially read from the source disk could skew the locality analysis. vCacheShare's event handling design allows it to suspend its analysis till the completion of the vMotion operation. Since vMotion is from disk-backed device to the flash-backed device, the lower IO service time causes a reduction in the cache allocation, which is reversed when the same VMDK is later moved back to HDD.

7.5 Result 3: End-to-end Evaluation

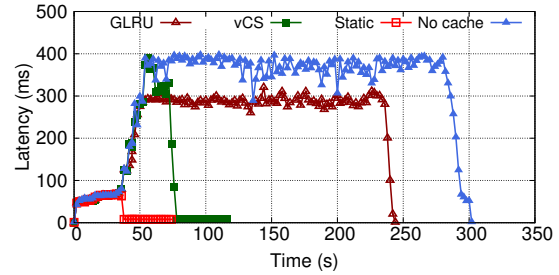We then evaluate the overall performance of vCacheShare using different workloads.

Iometer Test Case: Here we deploy dynamic Iometer benchmark workloads in two VMs (with the same IO priority) sharing SFC. RI is calcuated here with t_w set to 30 seconds. Iometer profiles are initially identical for both VMs but adjusted at run time, producing varied IO working set and intensity. More specifically, we manipulated access locality by (1) shrinking VM1's footprint around time point 480s to increase data reuse level and (2) at 1140s letting VM2 replay its past 300 seconds' access traces, to create an artificial reuse burst.

Figure 10(a) shows the cache allocation changes (except for static allocation, which produces fixed 50% − 50% split). It can be seen that vCacheShare made considerable adjustment to space allocation in the two aforementioned phase of locality change, while global LRU made only minor adaptation during VM2's replay and basically ignored VM1's footprint change.
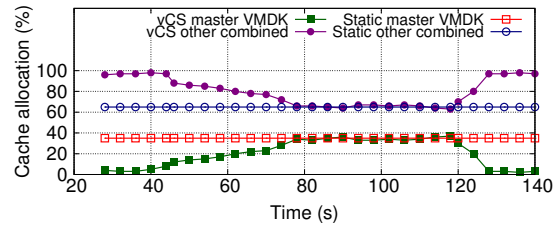
As a result, vCacheShare brings better overall performance, with Figure 10(b) showing the average access latency averaged over 30s time windows. Compared to GLRU and static allocation strategies, vCacheShare decreased the overall average access latency during the execution segment by 58.1% and 67.4%, respectively.

Virtual Desktop Infrastructure Test Case: VDI has become an important trend in the enterprise. It represents a challenging storage workload, with low average IOPS but transient 10-20X load spikes caused by boot storms or app load storms [23], when all the virtual instances are performing the same task, accessing the base OS image from the *Master VMDK*.
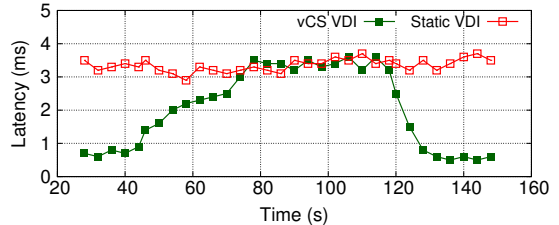
In this experiment, we provision 100 VDI instances based on 4 master images (WinXP, Win2k8 64bit, Win2k8 32bit and Ubuntu 11.04 64bit). Each master VMDK is 8GB, running an in-house VDI workload generator. The total SFC is 80GB and the static cache size

(a) VDI master VMDK I/O latency during boot storm

(b) Cache allocation with VDI boot storm

(c) Iometer VM latency

Figure 11: VDI cache allocation and latency

applied for each master VMDK is the entire virtual disk size. SFC is managed with vCacheShare, static and global LRU (GLRU) allocation policies. Along with VDI VMs, one additional read intensive Iometer VM (IO access footprint as 80GB) is sharing the same SFC. The RI calculation uses a t_w of 2 seconds. Figure 11 shows the average (among all master VMDKs) latency improvement for master VMDK, the cache allocation as well as Iometer VM's latency during the boot storm process. The performance of a no-SFC configuration ("No cache") is also shown. The boot storm starts at 2 seconds for all experiments and finishes at 300 seconds without caching, 116 seconds for vCacheShare, 74 seconds for static allocation, and 244 seconds for GLRU. The master VMDK has to be brought in from disk-based backend, creating severe queuing delay before the cache is warmed up. This process takes around 38 seconds (till the 40s time point).

The boot storm lasts over 240 seconds for no cache and GLRU. However, in the vCacheShare setup the load spike is captured quickly by its reuse intensity (*RI*) monitoring. After the master VMDK is cached, the boot storm completes using only 114 seconds, 37.5% and 46.2% of time consumed by no cache and GLRU respectively.

Static allocation starts caching once the boot storm
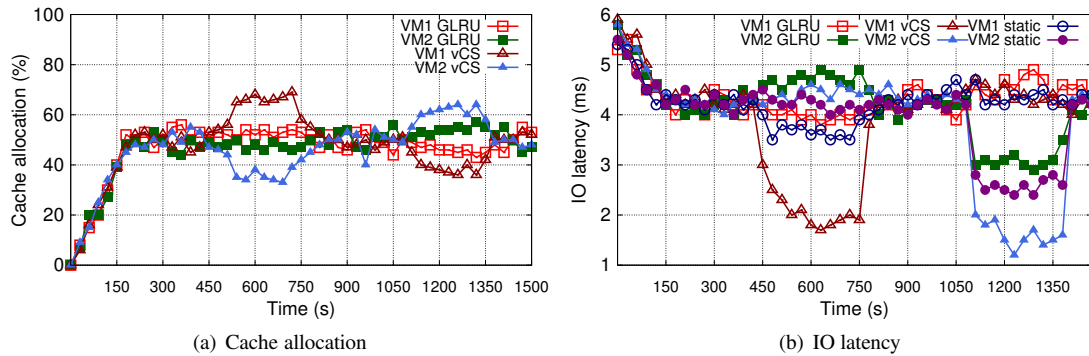
(a) Cache allocation   (b) IO latency

Figure 10: Iometer test results

starts, so it serves boot storm out of SFC the fastest, thus beat vCacheShare during this process. However, it can not reclaim the cache space allocated to master VMDK after the boot storm is over. When there are multiple versions of master images, more and more SFC space needs to be allocated for such one-time use. Worse, static management itself is not able to adaptively adjust cache size for different workloads. From Figure 11(c), although static SFC management shortens the boot storm, it significantly degrades the performance of other VMs before and after boot storm. In our test case, Iometer performs 81% better with vCacheShare compared to static allocation in terms of average latency.

(a) G-LRU cache policy

(b) vCacheShare policy

Figure 12: Cache allocation for TPC-H queries

Data Warehousing Test Case: In this test, we setup our test case using TPC-H [24] (using a scale factor of 3), with Postgres DB configured with default values on

an Ubuntu 11.04 VM. A scale factor of 3 is used for the TPC-H database, with Postgres configured with default values. RI here is calculated with t_w set to 10 seconds. TPC-H contains a collection of 22 queries, demonstrating varied data locality patterns. In our experiments we set up three identical VMs, each running the same three OLAP queries (similar to standard TPC-VMS [25] settings), but in different order: VM1 runs queries 20, 2, 21, VM2 runs 2, 21, 20, and VM3 runs query 21, 20, 2. Among them, query 20 has highest locality, 2 has little, and 21 sits in between. Here we omit static partitioning results as its performance is inferior to GLRU.

Results in Figure 12 reveal three major observations. First, vCacheShare makes very different partitioning decisions compared to GLRU, assisted by cache utility modeling to bias toward workloads that can benefit more from caching. E.g., the VM running query 2 (with the lowest locality) actually receives *more* allocation than other VMs when using GLRU, due to its higher IO access rate. vCacheShare correctly detects its lack of data reuse and reduces its allocation to the default reserve level. Second, when there is a workload change (switch between queries), vCacheShare is able to converge to a new relative allocation level faster than GLRU, with its active RI modeling plus hit ratio prediction. Finally, as a result, vCacheShare is able to improve the overall performance by 15.6%, finishing all queries in 430 seconds compared to 510 seconds with GLRU.
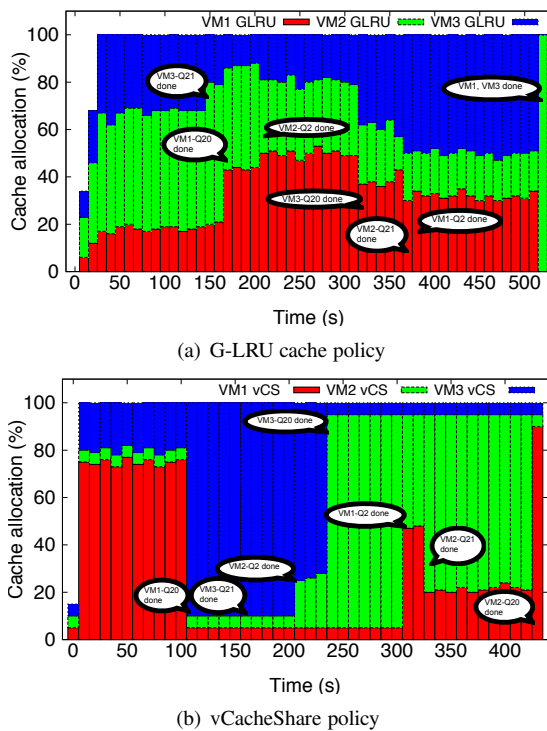
8 Related Work

SFC solutions are rapidly gaining adoption, with systems such as EMC VFCache [1], NetApp Mercury [2], as well as schemes developed at Fusion-io [4] and Facebook [26]. To our best knowledge, existing commercial solutions support only static, administrator-defined SFC partitioning or global sharing policies.

Related academic studies have demonstrated that global LRU is not efficient compared to partitioned shared cache for both CPU [7, 20] and disk [21]. Argon [27] partitions the memory cache between different services, providing isolation between the hit rate of

each service. The difference with our work is that Argon optimizes the cache hit rate for individual services, while vCacheShare optimizes the overall cache utilization (for aggregate I/O performance). Additionally, unlike vCacheShare, Argon requires administrator involvement each time there are changes in the workload pattern. Janus [28] performs flash allocation optimization for tiered flash-disk storage. The major difference here is that vCacheShare targets SFC systems in block granularity while Janus optimizes file based tiered storage.

The most closely related recent work on flash cache partitioning is S-CAVE [29]. Its optimization is based on runtime working set identification, while vCacheShare explores a different dimension by monitoring *changes* in locality, especially transient bursts in data reuse.

Memory virtualization [30] facilitates transparent sharing of memory among multiple VMs. Recent interest in using flash as extended memory [31, 32] has focused on resolving the access semantics and extending the interfaces beyond block IO, rather than the dynamic space management issues addressed in this paper. Similarly, techniques proposed for CPU cache partitioning [7, 8, 9, 20, 33] target problem settings significantly different from IO caching, which is much more resource (both compute and storage) constrained.

vCacheShare leverages several well-studied concepts in analytical modeling, optimization, and execution. E.g., reuse distance analysis has been used in memory access patterns for decades [14, 34], including cache management policies such as LRU-K [35] and LRFU [36]. Recently, Xiang et al. theoretically proves that hit ratio can be constructed from reuse distance [37], while vCacheShare demonstrates it in practice. Distance-based analysis of temporal and spatial localities has been characterized for file system caching [35, 36]. Low level disk access patterns have been analyzed for uses such as file system or disk layout optimizations [38, 39]. vCacheShare complements existing work by contributing a new approach in locality monitoring based system self-configuration, which may potentially be applied beyond SFC partitioning.

9   Conclusion

In this paper, we presented the motivation, design, and evaluation of vCacheShare, a dynamic, self-adaptive framework for automated server flash cache space allocation in virtualization environments. Through our implementation and experimentation, we confirmed that long observation windows are desirable for accurate cache hit ratio estimation, but may cause slow response to locality spikes and bring higher trace collection/processing overhead. This can be compensated by employing simultaneously short-term locality metrics. Meanwhile, the relationship between observation window size and cache hit ratio estimation accuracy requires further study.

We have also demonstrated that continuous IO access monitoring and analysis is affordable for the purpose of cache space management, even with today's high-VM-density environments.

Acknowledgement

References

[1] EMC VFCache. http://www.emc.com/about/news/press/2012/20120206-01.htm.

[2] Steve Byan, James Lentini, Anshul Madan, and Luis Pabon. Mercury: Host-side Flash Caching for the Data Center. In *MSST*, 2012.

[3] SANRAD VXL Software - Accelerate Virtual Machines with Host Based Flash. http://www.sanrad.com/VXL/4/1/8.

[4] Turbo Boost Virtualization. http://www.fusionio.com/products/ioturbine/.

[5] FlashSoft Reduces I/O Latency for Increased Server and Storage Performance. http://www.sandisk.com/products/flashsoft/.

[6] Proximal Data AutoCache. http://proximaldata.com/.

[7] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO'06*.

[8] J. Chang and G.S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS'07*.

[9] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *PACT'04*.

[10] Ajay Gulati, Chethan Kumar, and Irfan Ahmad. Modeling Workloads and Devices for IO Load Balancing in Virtualized Environments. *SIGMETRICS'09*.

[11] S. Uttamchandani, L. Yin, G. Alvarez, J. Palmer, and G. Agha. Chameleon: A Self-evolving, Fully-adaptive Resource Arbitrator for Storage Systems. *USENIX ATC'05*.

[12] IDC Worldwide Server Virtualization Tracker. http://www.idc.com/tracker/showproductinfo.jsp?prod_id=39.

[13] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *USENIX ATC'02*, 2001.

[14] C. Ding and Y. Zhong. Predicting Whole-program Locality through Reuse Distance Analysis. In *ACM SIGPLAN Notices 2003*.

[15] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.

[16] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical Power Management for Enterprise Storage. *Trans. Storage*, November 2008.

[17] Fred Schimscheimer. Server and Storage Sizaing Guide for Windows 7. Technical report, VMware. Inc, 2011.

[18] Simulated Annealing Information. http://www.taygeta.com/annealing/simanneal.html.

[19] Hill Climbing. http://en.wikipedia.org/wiki/Hill_climbing.

[20] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Comput.*, 41(9):1054–1068, September 1992.

[21] Dominique Thiébaut, Harold S. Stone, and Joel L. Wolf. Improving Disk Cache Hit-Ratios Through Cache Partitioning. *IEEE Trans. Comput.*, 41(6):665–676, June 1992.

[22] Iometer: an I/O subsystem measurement and characterization tool for single and clustered systems. http://www.iometer.org/.

[23] Vijayaraghavan Soundararajan and Jennifer M. Anderson. The impact of Management Operations on the Virtualized Datacenter. In *ISCA '10*.

[24] TPCH-H. http://www.tpc.org/tpch/.

[25] TPC-VMS. http://www.tpc.org/tpcvms/tpc-vms-2013-1.0.pdf.

[26] Releasing Flashcache. http://www.facebook.com/note.php?note_id=388112370932.

[27] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *FAST '07*.

[28] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, Francois Labelle, Nathan Coehlo, Xudong Shi, and Eric Schrock. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In *USENIX ATC*, 2013.

[29] Tian Luo, Siyuan Ma, Rubao Lee, Xiaodong Zhang, Deng Liu, and Li Zhou. S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance. In *PACT'13*.

[30] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI'02*.

[31] Mohit Saxena and Michael M. Swift. FlashVM: Virtual Memory Management on Flash. In *USENIX ATC'10*.

[32] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *USENIX NSDI'11*.

[33] Nathan Beckmann and Daniel Sanchez. Jigsaw: Scalable Software-defined Caches. In *PACT'13*.

[34] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[35] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *SIGMOD '93*.

[36] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.*, 50(12), December 2001.

[37] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: A Higher Order Theory of Locality. In *ASPLOS '13*.

[38] Sumit Narayan and John A. Chandy. Trace Based Analysis of File System Effects on Disk I/O. In *SPECTS'04*.

[39] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *USENIX ATC'07*.