Improving Flash Resource Utilization at Minimal Management Cost in Virtualized Flash-based Storage Systems

Jianzhe Tai[§], Deng Liu[‡], Zhengyu Yang[§], Xiaoyun Zhu[†], Jack Lo[†] and Ningfang Mi[§]

[§]*Northeastern University,* [†]*VMware Inc.,* [‡]*Twitter Inc.*

Abstract—Effectively leveraging Flash resources has emerged as a highly important problem in enterprise storage systems. One of the popular techniques today is to use Flash as a secondary-level host-side cache in the virtual machine environment. Although this approach delivers IO acceleration for VMs' IO workloads, it might not be able to fully exploit the outstanding performance of Flash and justify the high cost-per-GB of Flash resources. In this paper, we design new VMware Flash Resource Managers (VFRM and GLB-VFRM) under the consideration of both performance and the incurred cost for managing Flash resources. Specifically, VFRM and GLB-VFRM aim to maximize the utilization of Flash resources with minimal CPU, memory and IO cost in managing and operating Flash for a dedicated enterprise workload and multiple heterogeneous enterprise workloads, respectively. Our new Flash resource managers adopt the ideas of thermodynamic heating and cooling to identify data blocks that can benefit the most from being put on Flash and migrate data blocks between Flash and magnetic disks in a lazy and asynchronous mode. Experimental evaluation of the prototype shows that both VFRM and GLB-VFRM achieve better cost-effectiveness than traditional caching solutions, i.e., obtaining IO hit ratios even slightly better than some of the conventional algorithms as Flash size increases yet costing orders of magnitude less IO bandwidth.

Index Terms—Flash resource management, IO access pattern, Flash utilization, IO hit ratio, Virtualized storage systems

1 Introduction

With rapid developing of cloud computing, virtualized storage techniques become highly demanded for providing more capacity, and high performance, reliability and availability. NAND-based Flash memory is being widely deployed as a per-virtual disk, second-level cache in such a virtualized storage system to improve the IO performance and reduce the power consumption. Such a cache is managed using caching policies such as LRU or its variants, aiming to maintain the most likely-to-be accessed data for future reuse [1], [2]. While straightforward, these approaches have disadvantages in the following two aspects: the first aspect is cost- and performance-effectiveness. Since the cache is statically pre-allocated to each virtual disk, and the caching algorithm computes the cache admission and eviction independent of the IO activities of other virtual machines, it is difficult for the hypervisor to cost-effectively partition and allocate Flash resources among multiple heterogeneous virtual machines with different workloads; The other disadvantage is scalability. Since caching is usually implemented with a fine-grained cache line size (e.g., $4KB$, $8KB$), it requires a large number of CPU cycles for operations such as cache lookup, eviction, page mapping, etc., a large amount of memory space for maintaining cache metadata such as mapping table, LRU list, hash table, etc., and a fair amount of IOs to update the contents in Flash [3]. As the size of Flash storage grows to hundreds of GB or even several TB, the high cost of CPU, memory and IO bandwidth reduces the benefit of virtualization, where virtual machines are contending the same pool of resources from host. Even worse, it hinders the deployment of Flash resources in large scale.

To address these problems, we explore the Flash usage model from the hypervisor's point of view, and define a new set of goals: maximize the performance gain, and minimize the incurred cost for CPU, memory and IO bandwidth [4], [5]. With redefined goals of using Flash, we first design <u>V</u>Mware <u>F</u>lash <u>R</u>esource <u>M</u>anager (VFRM) to manage Flash resources in the virtual machine environment [4], [5] for a dedicated enterprise workload and then develop the global version GLB-VFRM to wisely allocate Flash resources among multiple heterogeneous workloads. Based on long-term observation of the IO access patterns, VFRM uses the heating and cooling concepts from thermodynamics to model the variation of IO popularity of individual blocks. With better understanding of the variation of IO popularity, it predicts the most popular blocks in the future and places them into the Flash drive to maximize the IO absorption ratio on Flash, which eventually maximizes the performance benefits from Flash resources. In addition, VFRM and GLB-VFRM use bins with large spacial granularity (e.g., $1MB$) as migration units to update the placement of data blocks between Flash and magnetic disks (MDs) in a lazy and asynchronous manner, which leads to a great saving in memory space for keeping the metadata, and a significant reduction in IOs that are needed for updating the contents in Flash.

A lot of SSD tiering solutions have been done in industry [6], [7]. Compared to these existing solutions, VFRM and GLB-VFRM are more cost-effective, because our designs adopt a finer tiering granularity (i.e., $1MB$) which reduces the chance of having cold data on Flash tier and thus makes more cost-effective use of Flash. For all of the external solutions we have examined, the tiering granularity ranges from $16MB$ to $1GB$ or even to an entire volume. Additionally, VFRM and GLB-VFRM allow users to customize their hybrid storage systems by choosing different types and capacities of storage devices, e.g., Flash memory and hard disks. Finally, as a new resource management solution, VFRM and GLB-VFRM can be well integrated with the existing features of VMware vSphere Datacenter such as resources scheduling, DRS [8], and VMotion, etc.

The remainder of this paper is organized as follows. Section 2 discusses some related work. Section 3 presents the goals and metrics of leveraging Flash technology in the virtual machine environment and analyzes some IO traces of real workloads to motivate the design of VFRM and GLB-VFRM. Section 4 describes the details of our designs. Section 5 evaluates VFRM and GLB-VFRM and compares existing caching solutions. Finally, we summarize our work and discuss the future work in Section 6.

2 Related Work

Host-side caches are being widely accepted in modern storage systems. Memcached is a distributed memory caching system by adding a scalable object caching layer to speed up dynamic Web applications and alleviate database load [9]. However, Memcache is more like an in-memory data store rather than a caching strategy in storage system. Flashcache is a kernel module which is built using the Linux Device Mapper (DM) and works primarily as a write back block cache in general purpose [10]. Recently, Facebook announced a new data management infrastructure, called TAO, in which its caching layer is designed as a globally distributed in-memory cache running on a large collection of geographically distributed server clusters [11].

Many efforts have focused on how to best utilize the Flash resources as a cache-based secondary-level storage system or integrated with HDD as a hybrid storage system. Some conventional caching policies [1], [12]–[14] such as LRU and its variants maintain the most recent accessed data for future reuse while some other works intended to design a better cache replacement algorithm by considering frequency in addition to recency [2], [15]. These caching algorithms compute the cache admission and eviction on each data access which is independent of the practical IO behavior. [16] uses Flash resources as a disk cache and adopt wear-level aware replacement policy based on LRU. SieveStore [17] presented a selective and ensemble-level disk cache by using SSDs to store the popular sets of data.

Flash-based multi-tiered storage systems have been recently studied in the literature [18]–[22]. For example, [18] presented a multi-tier SSD-based solution to perform dynamic extent placement using tiering and consolidation algorithms. To fit SSDs into a storage hierarchy, Hystor [19] and its related product Fusion Drive [20] provide a hybrid storage system for identifying performance- and semantically-critical data and timely retaining these data in SSDs. However, these approaches do not allow multiple entities to share SSDs. A hypervisor-based design, named “S-CAVE”, was presented in [21]. By identifying cache demands of each VM, S-CAVE dynamically adjusts the cache allocation among different VMs. This can be plugged in vSphere ESX directly. [23] proposed an optimized flash allocation algorithm based on both the cacheability of different traces’ IO activities and tiered storage characteristics like speed and price. Recently, [22] proposed a new allocation model based on the notion of per-device bottleneck sets. In this model, clients that are bottlenecked on the same storage device receive throughputs in proportion to their fair shares while allocation ratios among clients in different bottleneck sets are chosen to maximize overall system utilization. [24] proposed a CPU cache partitioning solution, whose perspective is mainly focusing on resource (both compute and storage) constrained. We notice that most of these approaches focus on how to exploit and improve traditional caching algorithms in a multi-tiered storage system, which still update contents of Flash in a fine-grained mode (like LRU, ARC, CAR). In contrast, our new resource manager mainly focuses on reducing operational IO costs by managing Flash in a coarse-grained manner, with respect to both temporal (e.g., $5min$) and spatial (e.g., $1MB$) granularities. Under VFRM, memory space for keeping the metadata can be greatly saved and IOs that are needed for updating the contents in Flash can be significantly reduced as well.

The benefits of VFRM are mainly motivated by three key observations of IO access patterns from workload studies. The effective workload studies can imply the accurate modeling, simulation, development and implementation of storage systems. [25] introduced twelve sets of long-term storage traces from various Microsoft production servers and analyzed trace characterizations in terms of block-level statistics, multi-parameter distributions, file access frequencies, and other more complex analyses. [26] presented an energy proportional storage system by effectively characterizing the nature of IO access on servers using workloads from three production systems. [27] created a mechanism for accelerating cache warmup based on detailed analysis of block-level data-center traces. They examined traces to understand the behavior of IO reaccesses in two dimensions, e.g., temporal and spatial behaviors. [28] is another good example of technique design motivated by workload analysis in which they proposed a write offloading design to save energy in enterprise storage by a better understand of IO patterns.

A lot of SSD tiering solutions have been done outside VMware [6], [7], [29], [30]. Compared with these solutions, vFRM has several advantages:

(1) Better cost-effectiveness: It is likely that the pages of one block on Flash tier are not all hot pages. Therefore the coarser the tiering granularity, the more cold data could reside on Flash tier as well as more waste of Flash resource. For all of the external solutions we have examined, the tiering granularity ranges from $16MB$ to $1GB$ or even to an entire volume. In contrast, vFRM manages Flash resources in the granularity of $1MB$, which is much finer and greatly reduces the chance of having cold data occupying the costly Flash.

(2) Heterogeneity: All of those external tiering solutions have a fixed and strict requirement on the model/type of the devices of Flash tier and the spinning disk tier. The tiering management software is also running on the storage array side which is transparent to the user. As a result, the user has no control on the building blocks of hybrid tiered storage. In contrast, vFRM can work with any type of Flash-SSDs and storage array.

(3) vSphere friendly: As vFRM solves problems from a resource management's perspective, it enables better integration with the existing vSphere features such as resources scheduling, DRS, VMotion, etc. Therefore, our solution is easy to be plugged into any vSphere-based systems.

3 Motivation

Flash resources are usually deployed as host-side cache for data centers. The most significant benefit by deploying Flash in systems is mainly in the consideration of performance improvement, i.e., increasing IO throughput and reducing IO latency. However, such kind of deployment inevitably introduces extra operational cost to the system. Motivated by this challenging issue, we strive to develop a new Flash management scheme, which is able to leverage the knowledge of real workload patterns to maximize utilization of flash resources and minimize operational costs incurred by Flash management.

3.1 Goals and Metrics

Instead of focusing on improving IO performance of an individual VM, we aim to maximize the utilization of Flash resources and minimize the cost incurred in managing Flash resources.

Maximizing Flash Utilization: When people buy an SSD, they are actually paying for performance rather than storage space. Therefore, we consider Input/Output Operations Per Second (IOPS), a common performance measurement, as the metric of Flash utilization and redefine one of our primary goals as maximizing IOPS utilization. As IOPS capabilities of Flash devices vary across different models, we alternatively use *IO hit ratio* as the metric of Flash utilization. *IO hit ratio* is defined as the fraction of IO requests that are served by Flash. The higher the *IO hit ratio*, the better the utilization of Flash resources. In order to achieve high *IO hit ratio*, the most frequently accessed data should be put on Flash media. As *IO hit ratio* increases, the processing efforts required for these IO requests are offloaded from the back-end storage array to the Flash tier and the storage array can thus allocate more processing power to serve other IO requests, which actually improves the IOs that are not served by Flash. This further improves the total cost of ownership (TCO) in terms of financial (IOPS/$) and power (IOPS/KWH) efficiencies of storage systems.

Minimizing CPU, Memory and IO Cost in managing Flash: The CPU, Memory and IO bandwidth are needed in Flash resource management. Today, a single Flash-based SSD can easily reach up to 1TB and the Flash resources are usually managed at a fine granularity (e.g., $4KB$ or $8KB$). Hence, it is fairly likely to incur a high fraction of in-memory footprint for the Flash related metadata. For example, if the memory footprint equals to 1% of Flash space, then it requires $10GB$ metadata for a SSD with $1TB$ size. Such a large memory footprint limits the scalability of deploying Flash resources with large capacity. Therefore, our second goal is to minimize the other cost incurred in managing and operating Flash resources.

3.2 IO Access Patterns

To understand volume access patterns in production systems, we first study a suite of one week block IO traces which were collected by MSR Cambridge in 2007 [28] from SNIA repository. In these IO traces, each data entry describes an IO request, including timestamp, disk number, logical block number (LBN), number of blocks and the type of IO (i.e., read or write). There are 36 traces from MSR-Cambridge in total, which includes a variety of workloads. In this paper, we select eight of them as representative and summarize the statistics of these traces in Table 1.

For each workload, we calculate IO hit ratios using the LRU caching algorithm with fully associative cache, $4KB$ cache line and $1GB$ cache size. The results in Table 1 show that the conventional caching algorithms (e.g., LRU) cannot always perform well. For example, the IO hit ratio is less than

TABLE 1
Statistics for Selected MSR-Cambridge Traces. Volume size denotes the maximum LBN accessed in disk volume. Working set size denotes the amount of data accessed. Re-accessed ratio denotes the percentage of IOs whose re-access time is within $5min$.

Category	Name	Server	Volume Size (GB)	Working Set Size (GB)	Hit Ratio by LRU	Re-access Ratio
Friendly	mds0	Media Serv.	33.9	3.23	90.84%	95.35%
	src12	Source Control Serv.	8.0	2.80	85.64%	94.81%
	stg0	Web Staging Serv.	10.8	6.63	89.28%	92.71%
	usr0	User Home Dir.	15.9	4.28	88.25%	96.03%
Unfriendly	stg1	Web Staging Serv.	101.7	81.5	34.60%	90.94%
	usr2	User Home Dir.	530.4	382.7	19.49%	95.50%
	web2	Web SQL Serv.	169.6	76.4	6.20%	95.45%
	src21	Source Control Serv.	169.6	22.0	2.82%	96.04%

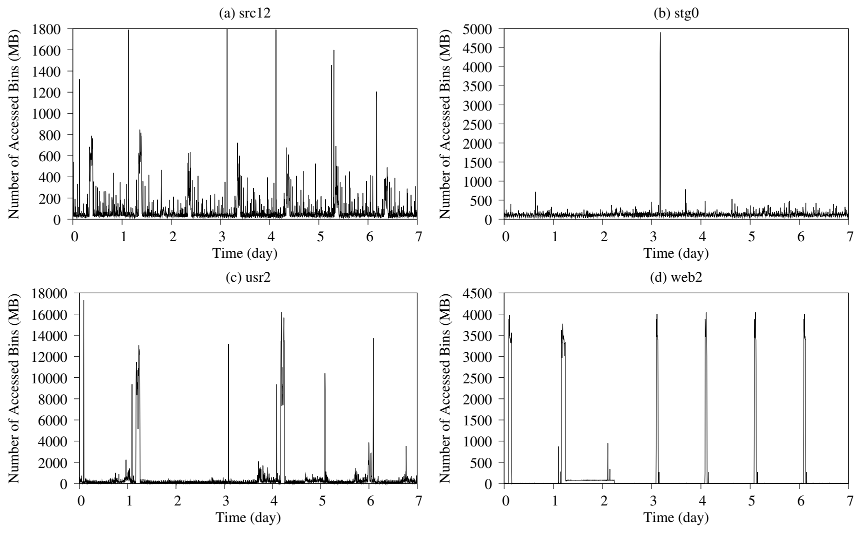


Fig. 1. Number of accessed bins per $5min$ of selected Cambridge traces.

3% under the "src21" workload. We thus coarsely classify the workloads into two categories: "cache-friendly" workloads (e.g., mds0, src12, stg0 and usr0) and "cache-unfriendly" workloads (e.g., stg1, usr2, web2 and src21). As shown in Table 1, cache-friendly workloads always obtain higher IO hit ratios (around 90%) under conventional caching algorithms, while cache-unfriendly workloads have relatively lower hit ratios (less than 40%). We interpret these results by observing that cache-unfriendly workloads often have larger volume sizes and working set sizes (see the third and the fourth columns in Table 1) than cache-friendly workloads, where volume size indicates the maximum LBN accessed in disk volume and working set size indicates the amount of data accessed. This means that the effectiveness of a cache is decided by its size to some extent. A small cache can only hold a small amount of data such that most of the cached data might be evicted or flushed out from the cache before it is reused if the actual working set size is large. Consequently, it is highly likely that the most recent or frequent data are not buffered in the cache which thus incurs low IO hit ratio.

To further investigate the differences between cache-friendly and cache-unfriendly workloads, we partition the entire LBNs address space of each workload into bins (with an equal width of $1MB$) and count the number of accessed bins per $5min$ over a period of seven days. Figure 1 shows the results of two representative workloads from each category. We observe that the cache-unfriendly workloads, (see Figure 1 (c) and (d)), have more IO spikes than the cache-friendly workloads, (see Figure 1 (a) and (b)). We also observe that these spikes in cache-unfriendly workloads are much stronger and longer (ranges from $1800MB$ to $16000MB$ of accessed data size), which can dramatically degrade IO hit ratios due to the first-time cache miss and even worse pollute the critical data in Flash. This motivates us to design a new Flash resource manager which can perform well for both cache-friendly and cache-unfriendly workloads.

To better understand IO access patterns, we further count the number of IO accesses for each bin in every hour over a period of seven days. Figure 2 plots the distribution of IO popularity of each bin and its variation over time, where the x-axis represents the LBN range, the y-axis represents the time and the z-axis represents the IO popularity (i.e., number of IO accesses) of each bin. The IO popularity of each bin was also represented in greyscale. A darker scale

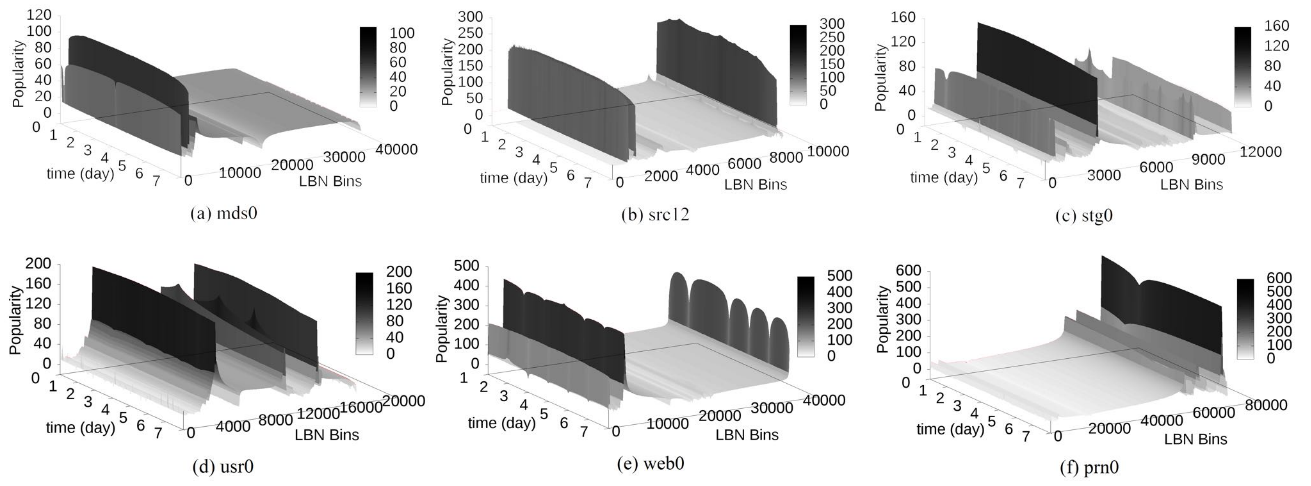


Fig. 2. IO popularity analysis of selected Cambridge traces.

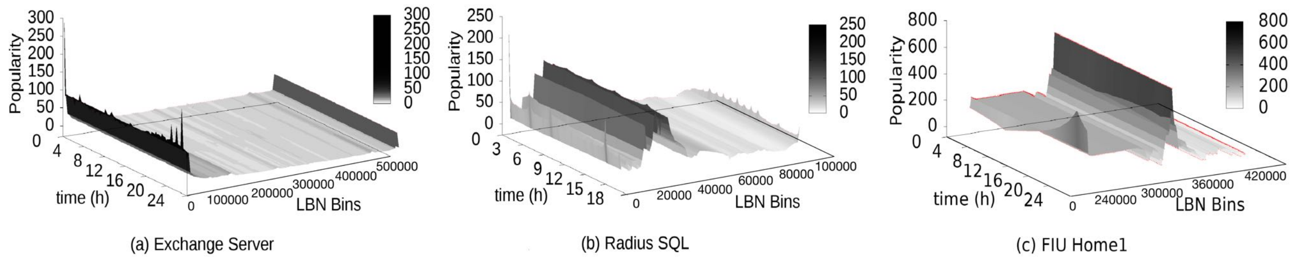


Fig. 3. IO popularity analysis of three traces.

represents a greater popularity. To further validate our observations in IO access patterns, we select other three real workload variants. The first trace is collected from Microsoft Exchange Server 2007 SP1 using the event tracing for a duration of 24 hours. The second one is Microsoft production server trace from RADIUS Back-end SQL Server for a duration about 17 hours [25]. The third one is the trace collected by Florida International University (FIU) from the first of four different end-user and developer home directories for a duration of 24 hours [26]. All these three traces are block level disk IOs with the same IO properties of MSR traces. Figure 3 shows the distribution of IO popularity of the three trace variants.

3.3 Observations

Among a number of interesting findings, we have three key observations that inspire the design of vFRM:

[Obv. 1] The block access frequency exhibits a bimodal distribution. Most of the bins are accessed rarely (i.e., less than 10 times a day), while a small fraction of the bins are accessed extremely frequently (i.e., more than thousands of times a day). This implies that only a small number of bins are popular enough to be placed on flash tier, while most of the remaining bins are not deserved for the high performance yet expensive flash resource. This observation also motivates that vFRM is suitable to be managed in a coarse granularity (i.e., $1MB$ bin).

[Obv. 2] The distribution of IO popularity does not vary significantly over time. This implies that vFRM does not need to actively and frequently update contents of Flash. The reaccess ratio in Table 1 further verifies that most of IO re-accesses happen in $5min$. Thus, a lazy and asynchronous approach should be sufficient for minimizing operational cost.

[Obv. 3] The distribution of IO popularity varies across workloads and volumes. This implies that different applications lead to diverse distributions of popular bins and thus need different amount of Flash resources.

4 vFRM Design and Algorithms

Inspired by the above observations, we design vFRM, a Flash resource manager to manage data blocks at the granularity of hypervisor file system block. vFRM dynamically relocates the data blocks between the Flash tier and the spinning disk tier to gain the most performance benefits from Flash. Additionally, it does the data block relocation lazily and asynchronously, which significantly reduces the cost for CPU, memory and IO incurred in managing Flash resources. By having the Flash tier absorbing more IO requests from VMs, vFRM lessens the contention for the IO bandwidth of the underlying storage, which in turn accelerates the IO access for data on the spinning disk tier. Note that we intentionally skip the availability problem of locally attached Flash device, which is beyond of the goals of this paper. In this paper, we assume that the Flash device already has a high availability.

4.1 Main Architecture

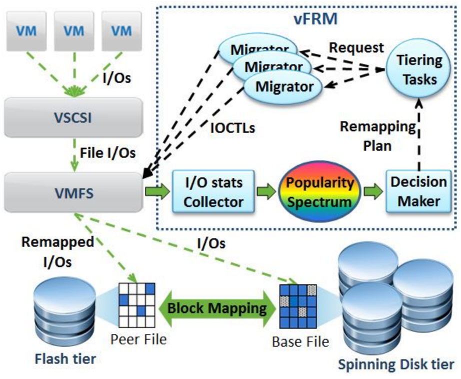


Fig. 4. vFRM 's architecture overview.

Figure 4 shows the architecture overview of vFRM, which consists of three major components: (1) a modified VMware Virtual Machine File System (VMFS) that allows composing a hybrid file with mixed blocks from both the spinning disk tier and the Flash tier via block mapping; (2) a tiering manager that monitors IO activities, makes migration decisions, and then generates tiering tasks for migrating hot blocks into the Flash tier and cold blocks out to the spinning disk tier; and 3) a pool of migrator threads that execute the migration tasks.

4.2 Hybrid File

A Virtual Machine Disk (VMDK) is essentially a file on a VMFS volume with all of its blocks allocated from the same VMFS volume [31]. In this paper, we propose a new type of file, called hybrid file, to extend the VMDK from spinning media to Flash media. A hybrid file comprises two files: a base file and a peer file. As such, the hybrid file can span across both tiers with the hot blocks in its peer file on the Flash tier and the cold ones in its base file on the spinning disk tier.

The peer file is a sparse file and its internal blocks keep the same logical offset in VMDK as their corresponding blocks in the base file. When overlapping these two files, we get a hybrid file with the mixed blocks from both the spinning media and the Flash media. The VMFS file block address resolution mechanism is designed to identify the location of a requested block (i.e., in the peer file or in the base file) and to seamlessly re-direct the IO to the right tier. Although the peer file has the same size of address space as its base file, it does not necessarily occupy the same size of Flash resources. In fact it is mostly sparse, because only a small portion of the blocks are allocated as hot blocks on the Flash tier. As each hot block keeps the same logical offset in both files, there is no need to add an extra mapping table to store the location mapping information of hot blocks between the Flash tier and the spinning disk tier. Moreover, we can use the inode pointer cache of the peer file as the block look up table, which further eliminates the need for an extra lookup table. If a block has been migrated to the Flash tier, the corresponding block will have been allocated and the inode pointer cache of the peer file can indicate the existence of this block. As a result, we have another saving of the memory space for the lookup table. During the migration of Flash resource, the dirty blocks on the Flash tier of the source host need to be migrated to the Flash tier of the destination host if the Flash tier cannot be accessed by both source and destination hosts. If the Flash tier is not shared and there is not Flash on the destination host, vFRM will collapse this hybrid file via writing the dirty blocks back to the spinning disk tier. In the virtualized environment, a virtual disk is a file on VMFS, the design of hybrid file automatically enables hybrid storage for VMs.

4.3 Basic Data Structure

Heat map: Heat map is used to represent the IO popularity statistics. Each $1MB$ block of the files on VMFS has on-flash metadata associated with it as heat map. The per-block metadata contains 16 bytes to record the number of IO accesses in which each 2 Bytes denotes the IO access count that happened in one epoch (e.g., $5min$). In our implementation, we store the IO statistics for the previous 8 epochs. The details of the usage of IO statistics to predict the IO popularity can be found in Section 4.4.2. In addition, we have 8 bytes of metadata to represent the logical address of the file descriptor and 4 bytes for the logical offset of the block. So that each $1MB$ block requires 28 bytes to hold the popularity statistics, which is only 0.0027% of the size of VMDK. And more importantly, heat map does not necessarily need to be pinned in memory. It only needs to be retrieved in memory for every $5min$ when we want to use it to figure what blocks need to be migrated into Flash tier and what blocks need to be migrated out. We will discuss more of the details in the following sections.

Tiering map: Tiering map is used to represent placement of the blocks between two tiers. A tiering map is specifically associated with a file and saved alongside the VMDK descriptor. It can be used to quickly warmup the hot blocks after migration of Flash resources. In the tiering map, one bit represents in which tier a block is located. Therefore the metadata footprint overhead is only about 0.00001% of the size of VMDK. The same as the heat map, tiering map does not need to be pinned in memory permanently.

4.4 Temperature-based Tiering Manager

The main task of a tiering manager is to migrate data blocks between spinning disk tier and Flash tier to gain the most performance benefit from Flash.

4.4.1 Four Steps of Auto-Tiering

There are four steps to place a block on the right tier.
Step 1: The IO stats collector collects the IO activities at runtime and periodically flushes the IO popularity statistics to disk.
Step 2: The tiering manager identifies the most popular blocks in the scope of all VMDK files based on a temperature-based model. We will discuss the temperature-based model in the following section.
Step 3: The tiering manager further generates a set of migrate-in (i.e., hot data into Flash) and migrate-out (i.e., cold data out of Flash) tasks.
Step 4: The migrators finally execute migration tasks. As a block migration involves modifying the file inode, all migration tasks are performed in the context of transactions to ensure the consistency of VMFS in case of host crash.

4.4.2 IO Popularity Prediction Model

We now define a temperature-based model for predicting the IO popularity of each block. In this model, we apply the concepts of heating and cooling from thermodynamics to represent the variation of IO popularity with time passing. When IO requests flow to a block, that particular block gets heated. With time passing, the heated block cools down. In general, we consider m minutes (e.g., $m = 5$ in our experiments) as an epoch and let $T(i)$ denote the estimated (or predicted) temperature of a block during the i^{th} epoch. Assume that for each epoch, we always have N previous epochs available. We then use the following equation to calculate a block's temperature:

$$T_i = \sum_{j=1}^{N} H(M_{i-j}) \cdot C(j), \tag{1}$$

where M_{i-j} is the number of IO requests to that block in the past $(i-j)^{th}$ epoch. $H(M_{i-j})$ and $C(j)$ denote the heating contribution and cooling factor respectively that are from the IO requests in the past $(i-j)^{th}$ epoch. Specifically, we define $H(M_{i-j})$ as a linear function, such that the heating temperature in the $(i-j)^{th}$ epoch is proportional to the number of IO requests during that epoch.

$$H(M_{i-j}) = \lambda \cdot M_{i-j}. \tag{2}$$

Here, λ is a tunable constant that determines how important one workload is relative to other workloads. The greater the λ is, the faster the block gets warmed up with the same number of IO requests. We define the cooling factor $C(j)$ as a function of the time distance (i.e., j epochs) from the current epoch.

$$C_j = \begin{cases} \frac{N+1-j}{N}, & 1 \le j < \frac{N}{2}+1 \\ \frac{1}{2^{j-3}}, & \frac{N}{2}+1 \le j \le N \end{cases} . \tag{3}$$

Such a cooling factor represents the declining heating effects with time passing. Currently we adopt a cooling scheme that linearly cools down in the first half of epochs and exponentially cools down in the second half of epochs. The heuristic behind this cooling scheme is that recent IO activities have more influence than the ones in the past. Using the above equations, we update instant and cumulative temperatures for all blocks for each epoch (i.e., every m minutes) based on their history temperatures in recent N epochs and IO request numbers during the current epoch. Moreover, as we adopt the concept of heating and cooling from thermodynamics to age the old epochs, we consider the fact that recent IO activities have more influence than the ones in the past by assigning different weights to the temperatures of recent N epochs. We then re-order all the blocks according to their cumulative temperatures to determine the popularity of these blocks. The most popular blocks (with highest temperatures) should be placed in the Flash tier based on the available capacity of Flash resources while the remaining blocks will be kept on the spinning disk tier.

4.5 Global vFRM among Multiple Heterogeneous VMs

In a virtualization environment, multiple VMs often share storage services and each VM has its own workload pattern and caching requirement. In most of such shared virtualization platforms, Flash is statically pre-allocated to each virtual disk (VMDK) for simplicity and the caching algorithm decides the cache admission and eviction for each VM only based on IO requests to that particular VM regardless of IOs to the others. Therefore, it is difficult for the hypervisor to cost-effectively partition and allocate Flash resources among multiple heterogeneous VMs, particularly under diverse IO demands. In this section, we further investigate the benefits of VFRM for managing Flash resources among multiple heterogeneous VMs. Our goal is to fully leverage the outstanding performance of shared Flash resources under the global view of caching management. The basic idea of the global version of VFRM is to divide Flash resources among multiple VMs with the goals of fully utilizing Flash and minimizing the operational cost. Intuitively, there are two straightforward approaches which simply allocate Flash resources among VMs by either equally assigning Flash to each VM or managing Flash resources in a fair competition mode. In the former approach, all VMs are purely isolated in using their own Flash resource and the caching management is fully affected by their own workload changes, while the second approach allows all VMs to freely use or share the entire Flash, such that the caching management is centrally interfered by the intensity of all workload changes.

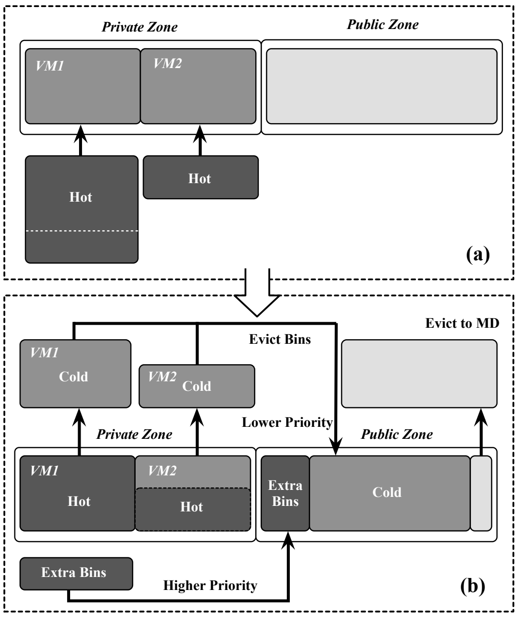


Fig. 5. Flash contents updating procedure of GLB-vFRM.

Unfortunately, these two straightforward approaches cannot fully utilize the benefits of Flash, particularly when the workloads frequently change and bursts or spikes of IOs occur from time to time. If Flash is equally reserved and assigned to all VMs, then VMs with bursty IOs or strict SLAs (Service-Level Agreement) cannot obtain more Flash resources. On the other hand, the second approach solves this issue by allowing all VMs to preempt or compete the Flash based on their present IO demands. Thus, VMs with higher IO demands can occupy more Flash resources by evicting less-accessed data from other VMs. However, under this approach, VMs with bursty IOs might occupy almost all the Flash resources and thus pollute the critical caching of other VMs. It is even worse that bursty workloads usually have less re-accesses in the long term. To wisely allocate Flash resources among all VMs, we develop the global version of vFRM which takes the dynamic IO demands of all VMs into consideration and divides Flash into a private zone and a public zone. Specially, the private zone is designed for reserving Flash for each VM in order to cache their recently accessed working sets, while the public zone is used to absorb and handle bursty IOs by being fairly competed among VMs according to their data popularities. We first implement a global vFRM algorithm, named "GLB-vFRM", such that all VMs are assigned the equal portion of Flash that is pre-allocated in the private zone. Algorithm 1 shows the pseudo code of GLB-vFRM. Figure 5 illustrates the Flash contents updating procedure. To manage each VM's private Flash, we sort its recently accessed bins (i.e., $1MB$) in the non-increasing order of their IO popularities. The top bins (i.e., with highest IO popularities) are then assigned to private Flash, see Figure 5(a). This procedure is denoted as *UpdatePrivateZone* in Algorithm 1. Meantime, both the residual of the recently accessed bins that cannot be cached in the private zone due to the limited space (i.e., *extraBin* in Algorithm 1) and the bins that are evicted from the private zone with less recency (i.e., *evictBin* in Algorithm 1) are then flushed into the public zone, see Figure 5(b). The public zone collects these data sets from all VMs and stores the critical data as much as possible according to their IO popularities, see the procedure of *UpdatePublicZone* in Algorithm 1. By this design, if some VMs receive higher IO demands than others, they can then occupy more Flash resources in the public zone (e.g., the extra bins of $VM1$ in Figure 5(b)), especially to handle their bursty demands. More importantly, bursty VMs cannot arbitrarily pollute the critical data of other VMs because each VM now owns their isolated Flash in the private zone which cannot be preempted by other VMs and thus guarantees the performance to some extent.

5 Evaluation

In this section, we present our experimental results to demonstrate the effectiveness of vFRM for a single enterprise workload and GLB-vFRM for multiple enterprise workloads with respect to our primary goals: maximizing Flash utilization and minimizing IO cost incurred in managing Flash. We first introduce the performance metrics and how they are measured to evaluate the effectiveness of our Flash managing algorithms. We then present the evaluation by implementing vFRM and GLB-vFRM as a trace-replay simulation program. For comparison, we also treat Flash as a second-level cache and implement the LRU, ARC [2] and CAR [32] caching solutions in our simulation.

5.1 Performance Metrics

In this section, we first introduce two performance metrics: IO hit ratio and IO cost. We consider a combination of these two metrics as a criterion to evaluate the effectiveness of our Flash managing algorithms. We also discuss the approaches which we used to calculate the overall IO cost under both the proposed and the conventional Flash managing algorithms.

IO Hit Ratio: IO hit ratio is defined as the fraction of IO requests that are served by Flash. An IO request might contain more than one page. We say an IO

TABLE 2
The necessary SSD and MD operations for all caching conditions.

(a) Operations for IO Access Cost

	Read Hit	Read Miss	Write Hit	Write Miss
LRU/ARC/CAR ($4KB$)	SSD Read	MD Read + SSD Write	SSD Write	SSD Write
vFRM/GLB-vFRM ($128KB$)	SSD Read	MD Read	SSD Write	MD Write

(b) Operations for Flash Update Cost

LRU/ARC/CAR ($4KB$)	Evict Dirty Page	
	SSD Read + MD Write	
vFRM/GLB-vFRM ($128KB$)	Admin Hot Bin	Evict Cold & Dirty Bin
	MD Read + SSD Write	SSD Read + MD Write

TABLE 3
Measured average IO response times of various types of IO operations at Flash and spinning disk.

Latency	$T_{SsdRead}$ (μs)	T_{SsdWrt} (μs)	T_{MdRead} (μs)	T_{MdWrt} (μs)
$4K$ Sequential	53	59	63	92
$128K$ Sequential	558	1242	1070	1104
$4K$ Random	135	58	7671	3922
$128K$ Random	790	1241	8665	4942

request to be Flash hit only when all of its associated pages are cached in Flash. Higher IO hit ratio indicates that more IOs can be accessed from Flash directly which accelerates the overall IO performance. Thus, one of our primary targets is to increase IO hit ratio for improving Flash utilization.

IO Cost: IO cost consists of two parts: IO access cost and Flash contents updating cost. Specifically, IO access cost can be represented as IO response time or IO throughput (e.g., IOPS). For example, in the case of read miss, LRU reads missed pages from MD and caches them in Flash. Thus, the corresponding IO access cost is the time spent during this procedure. Moreover, extra time is needed to flush (or evict) dirty pages when newly accessed pages are administrated but Flash is full. We here consider such data movements between Flash and MD as Flash contents updating cost and include this cost in the overall IO cost. We use Eq.(4) to calculate the overall IO cost C_{IO}, where C_{IOResp} and $C_{FlashUpdate}$ represent the IO access cost and the Flash contents updating cost, respectively. All N terms indicate the access numbers of SSD Read (N_{SsdRd}), SSD Write (N_{SsdWrt}), MD Read (N_{MdRd}), and MD Write (N_{MdWrt}), while all T terms (e.g., $T_{SsdRead}$ and T_{MdRead}) show the corresponding average IO latency for each operation.

$$\begin{aligned} C_{IO} &= C_{IOAccess} + C_{FlashUpdate} \\ &= N_{SsdRd} \cdot T_{SsdRd} + N_{SsdWrt} \cdot T_{SsdWrt} \\ &\quad + N_{MdRd} \cdot T_{MdRd} + N_{MdWrt} \cdot T_{MdWrt} \quad (4) \end{aligned}$$

In our evaluation, we use bins of large spacial granularity (i.e., $1MB$) as migration unit and choose an epoch of $5min$ to update the placement of data in Flash. By default, a single block size is set to $1MB$ in VMware VMFS [33], [34], such as the newly created VMFS-5 datastore. We thus set a bin size to $1MB$ as well in order to be compatible with VMware VMFS, making our approach pluggable. Additionally, users do not often change a single block size as different block sizes may cause some problems. For example, neither VMware vSphere Storage APIs – Array Integration (VAAI) nor VMware Consolidated Backup (VCB) using hot-add backup work in a VMFS datastore with different block sizes, as discussed in [35]. The choice of $5min$ temporal granularity is motivated by the observations shown in Section 3.2. We found that more than 95% of bins are re-accessed within $5min$ and the distribution of the IO popularity does not vary significantly over time. Our approach thus opts to update contents of Flash every $5min$ such that vFRM can capture the accesses of most IOs with minimum operational cost. In contrast, short temporal granularity (i.e., less than $5min$) might incur extra operational cost, without any increasing of IO hit ratio.

The basic IO sizes for the conventional caching algorithms and vFRM/GLB-vFRM are specified as $4KB$ and $128KB$, respectively. Since our Flash resource manager uses bins of large spacial granularity (i.e., $1MB$) as migration unit, large IOs (e.g., $128KB$) can be employed in operation to improve disk IO performance. Therefore, all T terms for the conventional caching algorithms and vFRM/GLB-vFRM are the corresponding disk performance of $4KB$ and $128KB$ IOs, respectively. Table 2 further presents the related IO operations for IO access (see (a) in the table) and Flash contents updating (see (b) in the table) under both the conventional caching algorithms and our Flash resource managers (vFRM and GLB-vFRM) when we are in four different scenarios, i.e., read hit, read miss, write hit, and write miss. As shown in Table 2(a), when we have a read or write miss, our

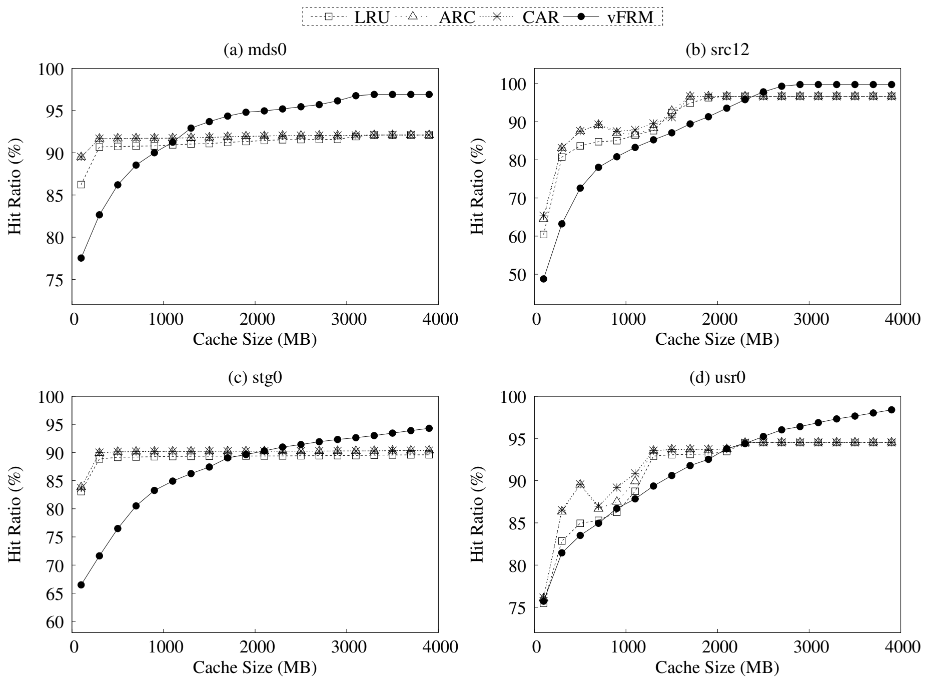


Fig. 6. IO hit ratios of vFRM, LRU and ARC.

Flash managers always redirect IOs to the spinning disk without updating the contents in Flash, and thus only trigger the operation of MD read/write, which is different from the conventional caching algorithms. As shown in Table 2(b), the conventional caching algorithms need a SSD read and a MD write to evict a dirty page from Flash to spinning disks. While our Flash managers only trigger move-in (for hot bins) and move-out (for cold bins) operations every epoch (e.g., $5min$). Thus, we count the number of hot and cold bins and consider 8 IOs of MD Read and SSD Write (resp. SSD Read and MD Write) for administrating (resp. evicting) a hot (resp. cold) bin in Flash as each IO operation is $128KB$ and the bin size is $1MB$. Table 3 illustrates the actual average IO response times (in microseconds) of various types of IO operations at both Flash and spinning disk devices. These results were measured from an `Intel DC S3500 Series` SSD with the capacity of $80GB$ and a `Western Digital WD20EURS-63S48Y0` hard drive with $2TB$ and 5400 RPM. As the conventional caching algorithms use $4KB$ as the cache line size while vFRM and GLB-vFRM set the bin size of $1MB$ and update Flash contents using the IO size of $128KB$, we present in Table 3 the measured response times for two levels of granularity (i.e., $4KB$ and $128KB$) in both sequential and random modes. These results will be used to calculate the overall IO cost as shown in Eq.(4).

5.2 Performance Evaluation for a Single Enterprise VM

In this section, we conduct experiments to verify the effectiveness of vFRM for a single enterprise VM case.

5.2.1 IO Hit Ratio

We first evaluate the IO hit ratio (i.e., the fraction of IO requests that are served by Flash) under vFRM using the representative MSR-Cambridge traces introduced in Section 3.2. Each trace represents the workload from a dedicated VM in the virtualized storage systems. For simplicity, we treat every workload as equally important (i.e., setting λ equal to one). We will evaluate the impact of λ in the clustering environment in our future work. The IO hit ratios with the conventional caching schemes (e.g., LRU, ARC and CAR) are also measured. We conduct experiments with various Flash sizes ranging from $100MB$ to $4GB$ and replay each trace separately. Figure 6 clearly shows that as the size of Flash increases, the IO hit ratio of vFRM catches up or even outperforms those of LRU, ARC and CAR for most of the workloads. As the capacities of Flash devices are usually large, vFRM is practically better in improving Flash utilization (e.g., IOPS) than classical caching solutions.

5.2.2 IO Cost

For both vFRM and existing caching solutions, internal IO costs are needed for both IO response and Flash contents updating, which is another type of performance criterion incurred in managing and operating Flash resources. vFRM only updates the contents every migration epoch (e.g., $5min$). In contrast, conventional caching updates the contents on every cache miss. Figure 7 shows the overall IO costs under both vFRM and LRU/ARC/CAR caching schemes. Here, Flash size is set to $4GB$. The numbers on top of each vFRM bar denote the relative improvement of the

Algorithm 1: Initial Task Assignment

```
Input: $n$: the number of VMs, $popBin[i]$: accessed bins
    of the $i^{th}$ VM in last epoch (e.g., $5min$),
    $prvBin[i]$: cached bins of the $i^{th}$ VM in private
    zone, $pubBin$: cached bins of all VMs in public
    zone
Output: $flashBin$: bins need to be cached in Flash
Procedure G1-vFRM()
    UpdatePrivateZone();
    UpdatePublicZone();
    for $i \leftarrow 1$ to $n$ do
        $flashBin += prvBin[i]$;
    $flashBin += pubBin$;
    return $flashBin$;
Procedure UpdatePrivateZone()
    for $i \leftarrow 1$ to $n$ do
        $popDiff$ = bins of $popBin[i]$ which are not in $prvBin[i]$;
        $prvDiff$ = bins of $prvBin[i]$ which are not in $popBin[i]$;
        if $len(popBin[i]) < len(prvBin[i])$ then
            $j = len(popDiff)$;
            $itemL$ = number of $j$ bins in $prvBin[i]$ with lowest IO popularity;
            $evictBin += itemL$;
            $prvBin[i] -= itemL$;
            $prvBin[i] += popDiff$;
        else
            $evictBin += prvDiff$;
            $j = len(prvBin[i])$;
            $prvBin[i]$ = number of $j$ bins in $popBins[i]$ with highest IO popularity;
            $extraBin +=$ the remaining bins of $popBins[i]$ which are not in $prvBin[i]$;
    return;
Procedure UpdatePublicZone()
    if $len(extraBin) \geq len(pubBin)$ then
        $j = len(pubBin)$;
        $pubBin$ = number of $j$ bins in $extraBin$ with highest IO popularity;
    else if $len(extraBin) + len(evictBin) \geq len(pubBin)$ then
        $j = len(pubBin) - len(extraBin)$;
        $itemH$ = number of $j$ bins in $evictBin$ with highest IO popularity;
        $pubBin = extraBin + itemH$;
    else
        $j = len(extraBin) + len(EvictBin)$;
        $itemL$ = number of $j$ bins in $pubBin$ with lowest IO popularity;
        $pubBin -= itemL$;
        $pubBin += extraBin + evictBin$;
    return;
```

number of IOs in relative of LRU. Lower percentage implies more reduction. We observe that in all cases, the IO costs of vFRM is far less than those of the other three classic caching solutions. In fact, most of them are order of magnitude better than the costs with LRU, ARC or CAR. For example, IO costs for *mds0* workload is only 31.87% of that of LRU solution. With such a great saving, vFRM can have more Flash IO bandwidth serving the IO requests, which further improves the VM's IO performance.

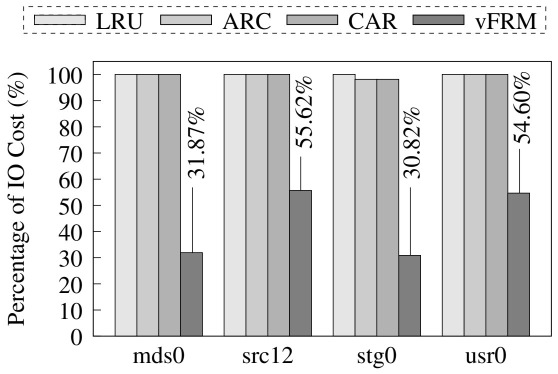


Fig. 7. IO costs by using MSR-Cambridge traces. The relative IO costs with respect to LRU are also shown on the bars of vFRM.

5.3 Performance Evaluation for Multiple VMs

In this section, we evaluate the effectiveness of our GLB-vFRM algorithm on allocating Flash resources among multiple enterprise applications (or VMs). The evaluation is conducted by using trace-replay simulations with 8 selected MSR-Cambridge IO traces (see Table 1). As shown in Section 3.2, these MSR-Cambridge IO traces can be classified into two categories, cache-friendly and cache-unfriendly. Thus, we generate three workloads ("$cf4$", "$cuf4$", and "$all8$") by mixing 4 cache-friendly traces, 4 cache-unfriendly traces, and all 8 traces, respectively. The timestamps of IO requests in each trace are normalized by a unified simulation start time and then used to determine the arrival times for each IO request in the workload.

The metrics considered in our evaluation include Flash utilization (in terms of IO hit ratio) and Flash managing overhead (with respect with IO cost). For comparison, we also present the results under three conventional caching algorithms, e.g., LRU, ARC and CAR. We also conduct experiments with various Flash sizes ranging from $1G$ to $32G$. In our GLB-vFRM algorithm, the entire Flash is statically divided into a private zone and a public zone; for example, 50% of Flash space is assigned to each of these two zones in our evaluation. Meanwhile, GLB-vFRM dynamically adjusts the partitioning of the private zone among different VMs.

5.3.1 Hit Ratio

Figure 8 illustrates IO hit ratios as a function of Flash size under three workloads (i.e., "$cf4$", "$cuf4$", and "$all8$"). We first observe that all these algorithms (including our GLB-vFRM) achieve high IO hit ratios when we have 4 cache-friendly traces (or VMs), see plot (a) in Figure 8. More importantly, under this cache-friendly workload, GLB-vFRM gains better Flash utilization than the conventional caching algorithms; IO hit ratios under GLB-vFRM keep rising to 99% as the capacity of Flash increases, while IO hit ratios under the conventional ones stop at around

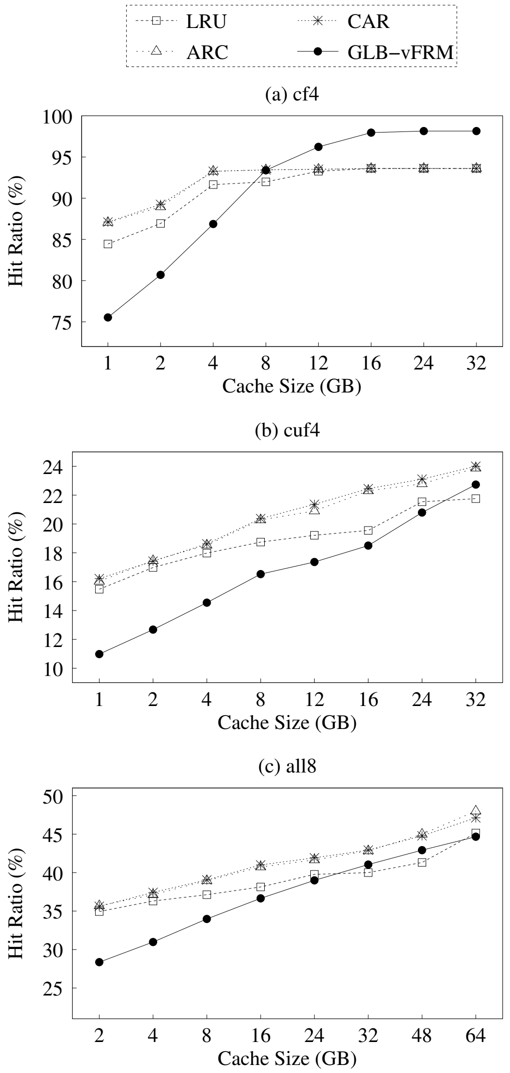


Fig. 8. IO hit ratios under three workloads (a)"$cf4$", (b)"$cuf4$", and (c)"$all8$".

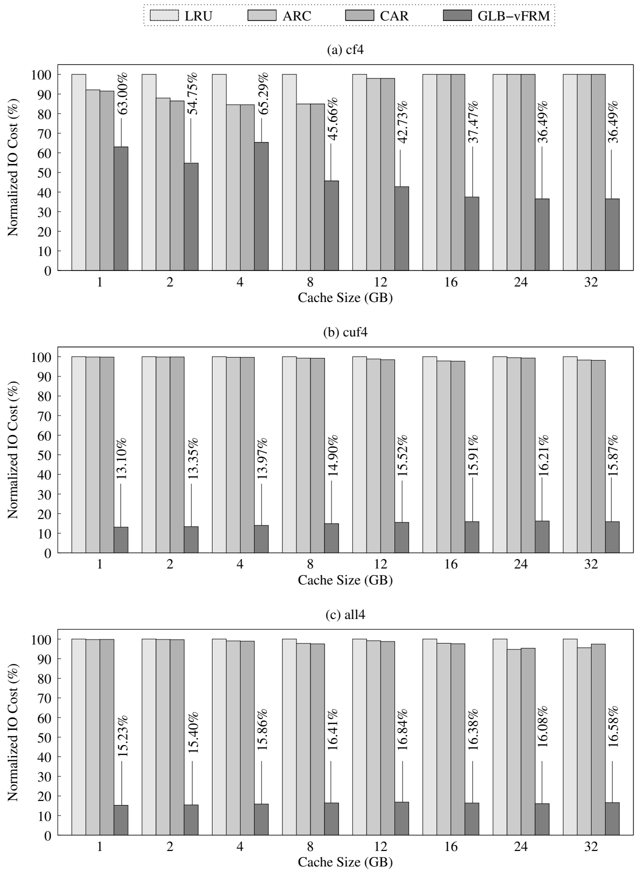


Fig. 9. Normalized IO costs (with respect to LRU) under three workloads (a)"$cf4$", (b)"$cuf4$", and (c)"$all8$".

93% when Flash size is larger than $4GB$. We also observe that under the cache-friendly ("$cuf4$") and mixed ("$all8$") workloads, the IO hit ratios of GLB-VFRM catch up and even slightly overcome some of the conventional algorithms as Flash size increases. We further look closely at IO accesses in these three workloads. As illustrated in Figure 1, IO spikes frequently appear in most traces such that a large number of bins are accessed during a short period which thus degrades IO hit ratios due to the first-time cache miss. Moreover, as the conventional caching algorithms cache data once there is a cache miss, it is highly likely that those IO spikes pollute the critical data of other applications (VMs) in Flash, especially if bins in these spikes are rarely reaccessed in near future. Our GLB-VFRM algorithm attempts to avoid such cache pollution by reserving private Flash for each VM and further improve IO hit ratio by caching data blocks in both private and public zones based on their IO popularities. Consequently, as long as Flash has enough capacity to hold active working sets of all VMs, GLB-VFRM is able to improve IO hit ratio (or Flash utilization) although GLB-VFRM does not update Flash contents upon every IO miss as the conventional caching algorithms do. On the other hand, when Flash size is relatively small, especially for those cache-unfriendly traces which have relatively large working sets (see Table 1), the conventional caching algorithms obtain higher hit ratios than GLB-VFRM by using small cache line size (e.g., $4KB$) and on-the-fly updating Flash contents for each cache miss. However, the cost of such caching algorithms is higher as well, which will be discussed in the following subsection. Therefore, GLB-VFRM can save lots of IO costs while keeps the similar hit ratios compared with conventional caching algorithms.

5.3.2 *IO Cost*

Figure 9 illustrates the normalized overall IO costs with respect to LRU under both GLB-VFRM and the conventional caching algorithms when we have 4 cache-friendly traces in "$cf4$", 4 cache-unfriendly traces in "$cuf4$", and 8 mixed traces in "$all8$". Consistently with the results for a single VM shown in Section 5.2, GLB-VFRM significantly reduces the overall IO costs for allocating Flash among multiple VMs compared to the conventional caching solutions. For example, under the "$cuf4$" workload (see Figure 9(a)), the overall IO cost under GLB-VFRM is decreased up to 65.29% and the relative reduction is increasing as Flash size increases. There are two main reasons for GLB-VFRM to have such low IO costs. First, instead of updating Flash contents upon each cache miss, GLB-VFRM, like VFRM, generates move-in/move-out tasks for both private and public zones in Flash every epoch (e.g., $5min$). Such a lazy and synchronize way allows GLB-VFRM to reduce the number of extra IOs for Flash contents updating. Secondly, GLB-VFRM adopts $1MB$ as the size of each bin and uses 8 IOs, each of which has the size of $128KB$, to move a bin into (or from) Flash, which reduces the number of IOs and shorten the latency for migrating a bin as well. More importantly, GLB-VFRM consumes much less IO cost for managing Flash resources when we have the "$cuf4$" and "$all8$" workloads (see Figure 9(b) and (c)) although the IO hit ratios of GLB-VFRM are slightly lower. We thus conclude that under the consideration of both Flash utilization (i.e., IO hit ratio) and Flash managing overhead (i.e., IO cost), GLB-VFRM is more effective than the conventional caching algorithms.

6 Conclusion

Effectively leveraging Flash resources in enterprise storage systems is highly important. Techniques for best usage of Flash resources should take into account both performance and the incurred cost for managing Flash resources. In this paper, we first designed a new Flash Resource Manager, named to VFRM, to make a cost-effective use of Flash resources in the virtual machine environment while reducing the cost for CPU, Memory, and Flash device IO bandwidth. Simulation results showed that VFRM not only outperforms traditional caching solutions in terms of performance utilization, but also incurs orders of magnitude lower cost for memory and Flash device IO bandwidth. In addition, VFRM effectively avoids cache pollution and eventually yields more improvement in IO performance. We further developed an extended version of VFRM which supports Flash resource management among multiple heterogeneous VMs. This global version (GLB-VFRM) divides Flash into two zones: a private zone is designed for reserving Flash for each VM in order to cache their recently accessed working sets, while the public zone is used to absorb and handle bursty IOs by being fairly competed among VMs according to their data popularities. Trace-replay simulations with the selected MSR-Cambridge IO traces show that GLB-VFRM obtains IO hit ratios even slightly better than some of the conventional algorithms as Flash size increases and meanwhile consumes much less IO cost for managing Flash resources. In the future, we plan to adopt a proactive approach to predict the IO temperature of data blocks. We also plan to apply VFRM technology to VMware VDI workload and explore the Flash resource management problem in a clustering environment in cooperation with VMware DRS. Moreover, we will also consider other block IO traces, such as Microsoft Production Server Traces and FIU Traces, from the SNIA repository to evaluate the effectiveness of our approaches. Additionally, we notice that the key idea of our designs can be applied to non-VM environments as well. Thus, we plan to refine our Flash resource managers to further support effective data placement in a non-VM storage cluster. The file system (instead of VMFS) will be modified to allow a hybrid file with mixed blocks from different storage devices. Finally, we will also consider to refine our GLB-VFRM approach to support the adjustment of two zone sizes on-the-fly in our future work.

Acknowledgements

This work was completed during Jianzhe Tai's internship at VMware. This project is partially supported by NSF grant CNS-1251129 and AFOSR grant FA9550-14-1-0160.

References

[1] Y. Zhou, J. Philbin, and K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," in *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, 2001, pp. 91–104.

[2] N. Megiddo and D. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 2003, pp. 115–130.

[3] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer, "Mercury: Host-Side Flash Caching for the Data Center," in *IEEE 28th Symposium on Mass Storage Systems and Technologies*, Pacific Grove, CA, 2012, pp. 1–12.

[4] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," in *Proceedings of the 5th symposium on Operating systems design and implementation*, Boston, MA, 2002, pp. 181–194.

[5] E. Bugnion, S. Devine, and M. Rosenblum, "DISCO: Running Commodity Operating Systems on Scalable Multiprocessors," in *Proceedings of the 6th ACM symposium on Operating systems principles*, 1997, pp. 143–156.

[6] "EMC FAST VP," http://www.emc.com/collateral/white-papers/storage-wp.pdf.

[7] "Hitachi Dynamic Tiering Software," http://www.hds.com/assets/pdf/hitachi-datasheet-dynamic-tiering.pdf.

[8] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu, "VMware Distributed Resource Management: Design, Implementation and Lessons Learned," *VMware Technical Journal*, vol. 1, 2012.

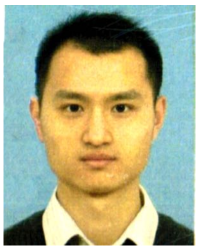
[9] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux Journal*, vol. 124, no. 5, 2004.

[10] "Facebook Flashcache," https://github.com/facebook/flashcache.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's Distributed Data Store for the Social Graph," in *Proceedings of the 2013 USENIX Annual Technical Conference on ATC'13*, San Jose, CA, 2013, pp. 49–60.

[12] E. O'Neil, P. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, Washington, DC, 1993, pp. 297–306.

[13] M. Kampe, P. Stenstrom, and M. Dubois, “Self-correcting LRU Replacement Policies,” in *Proceedings of the 1st conference on Computing frontiers*, Ischia, Italy, 2004, pp. 181–191.
[14] T. Johnson and D. Shasha, “2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm,” in *Proceedings of the 20th International Conference on Very Large Data Bases*, San Francisco, CA, 1994, pp. 439–450.
[15] D. Lee, J. Choi, J.-H. Kim, S. 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 Transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.
[16] T. Kgil, D. Roberts, and T. Mudge, “Improving NAND Flash Based Disk Caches,” in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, Bejing, China, 2008, pp. 327–338.
[17] T. Pritchett and M. Thottethodi, “SieveStore: A Highly-selective, Ensemble-level Disk Cache for Cost-performance,” in *Proceedings of the 37th annual international symposium on Computer architecture*, Saint-Malo, France, 2010, pp. 163–174.
[18] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, “Cost Effective Storage using Extent Based Dynamic Tiering,” in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, San Jose, CA, 2011.
[19] F. Chen, D. A. Koufaty, and X. Zhang, “Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems,” in *Proceedings of the International Conference on Supercomputing*, Tucson, Arizona, 2011, pp. 22–32.
[20] “Fusion Drive,” http://en.wikipedia.org/wiki/FusionDrive.
[21] J. Guerra, H. Pucha, J. S. Glider, W. Belluomini, and R. Rangaswami, “Cost Effective Storage using Extent Based Dynamic Tiering,” in *FAST*, 2011, pp. 273–286.
[22] H. Wang and P. J. Varman, “Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation,” in *FAST*, 2014, pp. 229–242.
[23] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and E. Schrock, “Janus: Optimal Flash Provisioning for Cloud Storage Workloads,” in *USENIX Annual Technical Conference*, 2013, pp. 91–102.
[24] N. Beckmann and D. Sanchez, “Jigsaw: Scalable Software-Defined Caches,” in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 213–224.
[25] S. Kavalanekar, B. Worthington, Z. Qi, and V. Sharda, “Characterization of Storage Workload Traces from Production Windows Servers,” in *Proceedings of the 2008 IEEE International Symposium on Workload Characterization*, Seattle, WA, 2008, pp. 119–128.
[26] A. Verma, R. Koller, L. Useche, and R. Rangaswami, “SRCMap: Energy Proportional Storage Using Dynamic Consolidation,” in *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, San Jose, CA, 2010.
[27] Y. Zhang, G. Soundararajan, M. W. Storer, L. N. Bairavasundaram, S. Subbiah, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Warming Up Storage-Level Caches with Bonfire,” in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, San Jose, CA, 2013, pp. 59–72.
[28] D. Narayanan, A. Donnelly, and A. Rowstron, “Write Off-Loading: Practical Power Management for Enterprise Storage,” *ACM Transactions on Storage*, vol. 4, no. 3, pp. 10:1–10:23, 2008.
[29] “The Architectural Advantages of Dell Compellent Automated Tiered Storage,” http://i.dell.com/sites/content/shared-content/data-sheets/en/Documents/dell-compellent-tiered-storage.pdf.
[30] “IBM Easy Tier,” http://pic.dhe.ibm.com/easytier.html.
[31] S. B. Vaghani, “Virtual Machine File System,” *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 4, pp. 57–70, 2010.
[32] S. Bansal and D. S. Modha, “CAR: Clock with Adaptive Replacement,” in *Proceedings of the 2th USENIX Conference on File and Storage Technologies*, vol. 4, 2004, pp. 187–200.
[33] “VMware White Paper: Recommendations for Aligning VMFS Partitions,” www.vmware.com/pdf/esx3_partition_align.pdf.
[34] “EMC White Paper: VMware ESX Server Using EMC CLARiiON/Symmetrix Storage Systems Solutions Guide,” http://www.emc.com/collateral/software/white-papers/h10630-vmware-vasa-symmetrix-wp.pdf.
[35] “VMware Community Discussion about VMFS Block Size,” communities.vmware.com/docs/DOC-11920.



Jianzhe Tai is a Software Engineer at MTS-Core Storage team of VMWare. He obtained his PhD degree in Computer Engineering at Northeastern University in 2014. His research interests are Virtualization and Cloud Management, Multi-tiered Storage Systems, Operating Systems, Virtual Machine Migration and Load Balance, Performance Isolation in Virtualized Systems, and Server Consolidation.

Deng Liu is currently a software engineer at Twitter Inc. Before that he was a software engineer at VMware Inc. Deng has a broad research and development experience in virtuaization, big data, distributed systems, and highly scalability storage systems. He received M.S. degree in Computer Science from the University of Wisconsin-Madison.

Zhengyu Yang is a PhD candidate at the Northeastern University. He graduated from the Hong Kong University of Science and Technology with a M.S. in Telecommunication in 2011, and he obtained his B.S. in Communication Engineering from Tongji University in China. His current research area is mainly on caching algorithm, cloud computing, deduplication, and performance simulations.

Xiaoyun Zhu is a Staff Engineer in the Cloud Resource Management group of VMware. She has worked on the development and performance improvement for VMware's key resource management features including DRS, DPM, and Storage DRS. Xiaoyun has co-authored over 50 technical papers in peer-reviewed journals and conferences, and holds over 20 patents. Xiaoyun received her B.S. in Automation from Tsinghua University in China, and her M.S. and Ph.D. in Electrical Engineering from California Institute of Technology.

Jack Lo is an Vice President of R&D of Core Storage and Availability VMware. He manages the Core Storage and Availability R&D team, responsible for vSphere storage technologies (file system, storage stack, storage management, etc.) and availability (HA, fault tolerance, replication, and backup). Previously he has managed the virtual machine platform and CPU virtualization teams at VMware.

Ningfang Mi is an Assistant Professor at Northeastern University, Department of Electrical and Computer Engineering, Boston. She received her Ph.D. degree in Computer Science from the College of William and Mary, VA in 2009. She received her M.S. in Computer Science from the University of Texas at Dallas, TX in 2004 and her B.S. in Computer Science from Nanjing University, China, in 2000. Her current research interests are capacity planning, MapReduce/Hadoop scheduling, cloud computing, resource management, performance evaluation, workload characterization, simulation and virtualization.