CloudCache: On-demand Flash Cache Management for Cloud Computing

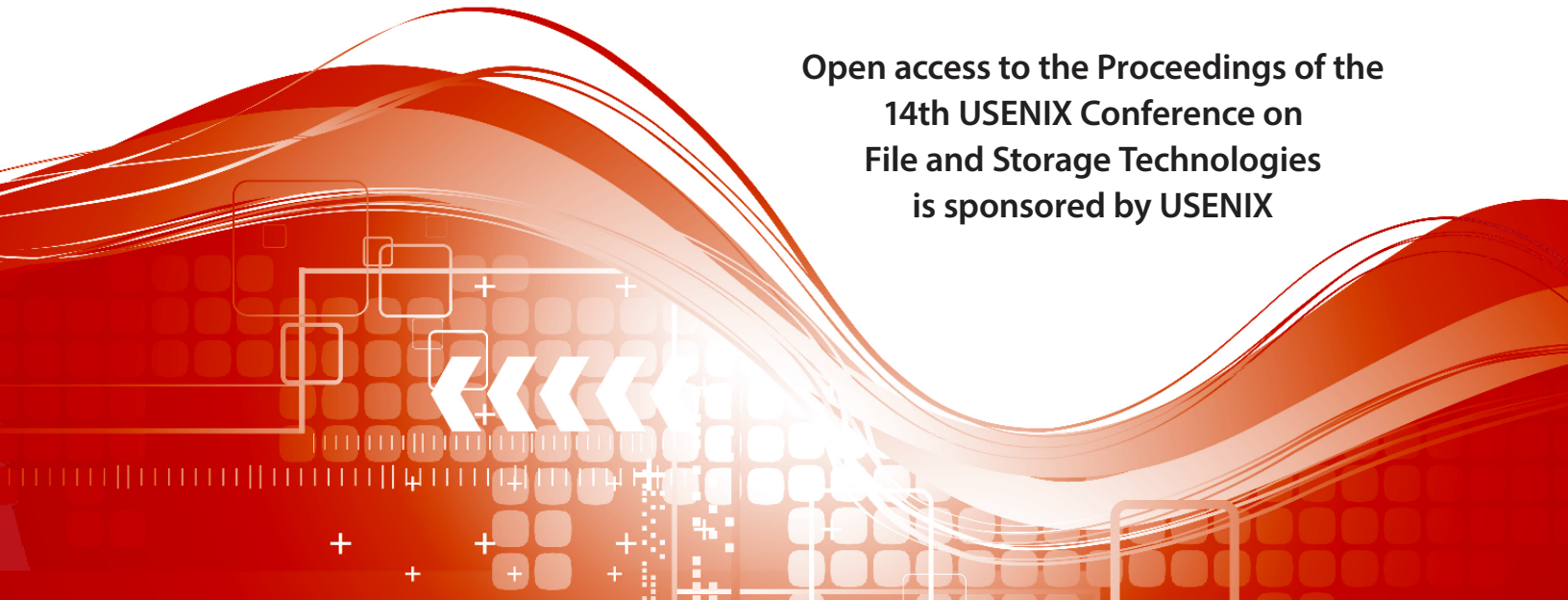
Dulcardo Arteaga and Jorge Cabrera, *Florida International University;* Jing Xu, *VMware Inc.;*
Swaminathan Sundararaman, *Parallel Machines;* Ming Zhao, *Arizona State University*

https://www.usenix.org/conference/fast16/technical-sessions/presentation/arteaga

This paper is included in the Proceedings of the
14th USENIX Conference on
File and Storage Technologies (FAST '16).

February 22–25, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-28-7



Open access to the Proceedings of the
14th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX

CloudCache: On-demand Flash Cache Management for Cloud Computing

Dulcardo Arteaga Jorge Cabrera
Florida International University

Jing Xu
VMware Inc.

Swaminathan Sundararaman
Parallel Machines

Ming Zhao
Arizona State University

Abstract

Host-side flash caching has emerged as a promising solution to the scalability problem of virtual machine (VM) storage in cloud computing systems, but it still faces serious limitations in capacity and endurance. This paper presents CloudCache, an on-demand cache management solution to meet VM cache demands and minimize cache wear-out. First, to support on-demand cache allocation, the paper proposes a new cache demand model, Reuse Working Set (RWS), to capture only the data with good temporal locality, and uses the RWS size (RWSS) to model a workload's cache demand. By predicting the RWSS online and admitting only RWS into the cache, CloudCache satisfies the workload's actual cache demand and minimizes the induced wear-out. Second, to handle situations where a cache is insufficient for the VMs' demands, the paper proposes a dynamic cache migration approach to balance cache load across hosts by live migrating cached data along with the VMs. It includes both on-demand migration of dirty data and background migration of RWS to optimize the performance of the migrating VM. It also supports rate limiting on the cache data transfer to limit the impact to the co-hosted VMs. Finally, the paper presents comprehensive experimental evaluations using real-world traces to demonstrate the effectiveness of CloudCache.

1 Introduction

Host-side flash caching employs flash-memory-based storage on a virtual machine (VM) host as the cache for its remote storage to exploit the data access locality and improve the VM performance. It has received much attention in recent years [10, 1, 14, 7], which can be attributed to two important reasons. First, as the level of consolidation continues to grow in cloud computing systems, the scalability of shared VM storage servers becomes a serious issue. Second, the emergence of flash-memory-based storage has made flash caching a promising option to address this IO scalability issue, because accessing a local flash cache is substantially faster than accessing the remote storage across the network.

However, due to the capacity and cost constraints of flash devices, the amount of flash cache that can be employed on a host is much limited compared to the dataset sizes of the VMs, particularly considering the increasing data intensity of the workloads and increasing number of workloads consolidated to the host via virtualization. Therefore, to fulfill the potential of flash caching, it is important to allocate the shared cache capacity among the competing VMs according to their actual demands. Moreover, flash devices wear out by writes and face serious endurance issues, which are in fact aggravated by the use for caching because both the writes inherent in the workload and the reads that miss the cache induce wear-out [33, 15]. Therefore, the cache management also needs to be careful not to admit data that are not useful to workload performance and only damage the endurance.

We propose CloudCache to address the above issues in flash caching through *on-demand cache management*. Specifically, it answers two challenging questions. First, *how to allocate a flash cache to VMs according to their cache demands?* Flash cache workloads depend heavily on the dynamics in the upper layers of the IO stack and are often unfeasible to profile offline. The classic working set model studied for processor and memory cache management can be applied online, but it does not consider the reuse behavior of accesses and may admit data that are detrimental to performance and endurance. To address this challenge, we propose a new cache demand model, *Reuse Working Set (RWS)*, to capture the data that have good temporal locality and are essential to the workload's cache hit ratio, and use the RWS size (*RWSS*), to represent the workload's cache demand. Based on this model, we further use prediction methods to estimate a workload's cache demand online and use new cache admission policies to admit only the RWS into cache, thereby delivering a good performance to the workload while minimizing the wear-out. Cloud-

Cache is then able to allocate the shared cache capacity to the VMs according to their actual cache demands.

The second question is *how to handle situations where the VMs' cache demands exceed the flash cache's capacity.* Due to the dynamic nature of cloud workloads, such cache overload situations are bound to happen in practice and VMs will not be able to get their desired cache capacity. To solve this problem, we propose a *dynamic cache migration* approach to balance cache load across hosts by live migrating the cached data along with the VMs. It uses both on-demand migration of dirty data to provide zero downtime to the migrating VM, and background migration of RWS to quickly warmup the cache for the VM, thereby minimizing its performance impact. Meanwhile, it can also limit the data transfer rate for cache migration to limit the impact to other co-hosted VMs.

We provide a practical implementation of CloudCache based on block-level virtualization [13]. It can be seamlessly deployed onto existing cloud systems as a drop-in solution and transparently provide caching and on-demand cache management. We evaluate it using a set of long-term traces collected from real-world cloud systems [7]. The results show that RWSS-based cache allocation can substantially reduce cache usage and wear-out at the cost of only small performance loss in the worst case. Compared to the WSS-based cache allocation, the RWSS-based method reduces a workload's cache usage by up to 76%, lowers the amount of writes sent to cache device by up to 37%, while delivering the same IO latency performance. Compared to the case where the VM can use the entire cache, the RWSS-based method saves even more cache usage while delivering an IO latency that is only 1% slower at most. The results also show that the proposed dynamic cache migration reduces the VM's IO latency by 93% compared to no cache migration, and causes at most 21% slowdown to the co-hosted VMs during the migration. Combining these two proposed approaches, CloudCache is able to improve the average hit ratio of 12 concurrent VMs by 28% and reduce their average 90th percentile IO latency by 27%, compared to the case without cache allocation.

To the best of our knowledge, CloudCache is the first to propose the RWSS model for capturing a workload's cache demand from the data with good locality and for guiding the flash cache allocation to achieve both good performance and endurance. It is also the first to propose dynamic cache migration for balancing the load across distributed flash caches and with optimizations to minimize the impact of cache data transfer. While the discussion in the paper focuses on flash-memory-based caches, we believe that the general CloudCache approach is also applicable to new nonvolatile memory (NVM) technologies (e.g., PCM, 3D Xpoint) which will likely be used as a cache layer, instead of replacing DRAM, in the storage hierarchy and will still need on-demand cache allocation to address its limited capacity (similarly to or less than flash) and endurance (maybe less severe than flash).

The rest of the paper is organized as follows: Section 2 and Section 3 present the motivations and architecture of CloudCache, Section 4 and Section 5 describe the on-demand cache allocation and dynamic cache migration approaches, Section 6 discusses the integration of these two approaches, Section 7 examines the related work, and Section 8 concludes the paper.

2 Motivations

The emergence of flash-memory-based storage has greatly catalyzed the adoption of a new flash-based caching layer between DRAM-based main memory and HDD-based primary storage [10, 1, 7, 24]. It has the potential to solve the severe scalability issue that highly consolidated systems such as public and private cloud computing systems are facing. These systems often use shared network storage [20, 5] to store VM images for the distributed VM hosts, in order to improve resource utilization and facilitate VM management (including live VM migration [11, 25]). The availability of a flash cache on a VM host can accelerate the VM data accesses using data cached on the local flash device, which are much faster than accessing the hard-disk-based storage across network. Even with the increasing adoption of flash devices as primary storage, the diversity of flash technologies allows the use of a faster and smaller flash device (e.g., single-level cell flash) as the cache for a slower but larger flash device (e.g., multi-level cell flash) used as primary storage.

To fulfill the potential of flash caching, it is crucial to employ *on-demand cache management*, i.e., allocating shared cache capacity among competing workloads based on their demands. The capacity of a commodity flash device is typically much smaller than the dataset size of the VMs on a single host. How the VMs share the limited cache capacity is critical to not only their performance but also the flash device endurance. On one hand, if a workload's necessary data cannot be effectively cached, it experiences orders of magnitude higher latency to fetch the missed data from the storage server and at the same time slows down the server from servicing the other workloads. On the other hand, if a workload occupies the cache with unnecessary data, it wastes the valuable cache capacity and compromises other workloads that need the space. Unlike in CPU allocation where a workload cannot use more than it needs, an active cache workload can occupy all the allocated space beyond its actual demand, thereby hurting both the performance of other workloads and the endurance of flash device.

S-CAVE [22] and vCacheShare [24] studied how to

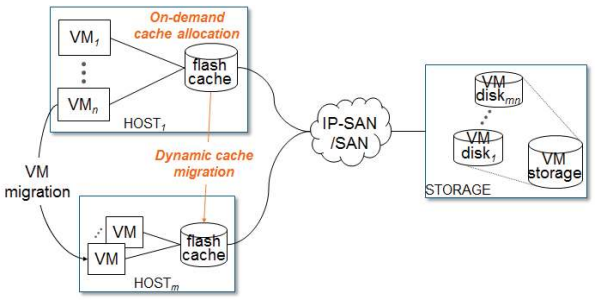


Figure 1: Architecture of CloudCache

optimize flash cache allocation according to a certain criteria (e.g., a utility function), but they cannot estimate the workloads' actual cache demands and thus cannot meet such demands for meeting their desired performance. HEC [33] and LARC [15] studied cache admission policies to reduce the wear-out damage caused by data with weak temporal locality, but they did not address the cache allocation problem. Bhagwat *et al*. studied how to allow a migrated VM to access the cache on its previous host [9], but they did not consider the performance impact to the VMs. There are related works studying other orthogonal aspects of flash caching, including write policies [17], deduplication/compression [21], and other design issues [14, 7]. A detailed examination of related work is presented in Section 7.

3 Architecture

CloudCache supports *on-demand cache management* based on a typical flash caching architecture illustrated in Figure 1. The VM hosts share a network storage for storing the VM disks, accessed through SAN or IP SAN [20, 5]. Every host employs a flash cache, shared by the local VMs, and every VM's access to its remote disk goes through this cache. CloudCache provides on-demand allocation of a flash cache to its local VMs and dynamic VM and cache migration across hosts to meet the cache demands of the VMs. Although our discussions in this paper focus on block-level VM storage and caching, our approaches also work for network file system based VM storage, where CloudCache will manage the allocation and migration for caching a VM disk file in the same fashion as caching a VM's block device. A VM disk is rarely write-shared by multiple hosts, but if it does happen, CloudCache needs to employ a cache consistency protocol [26], which is beyond the scope of this paper.

CloudCache supports different write caching policies: (1) *Write-invalidate*: The write invalidates the cached block and is submitted to the storage server; (2) *Write-through*: The write updates both the cache and the storage server; (3) *Write-back*: The write is stored in the cache immediately but is submitted to the storage server later when it is evicted or when the total amount of dirty data in the cache exceeds a predefined threshold. The write-invalidate policy performs poorly for write-intensive workloads. The write-through policy's performance is close to write-back when the write is submitted to the storage server asynchronously and the server's load is light [14]; otherwise, it can be substantially worse than the write-back policy [7]. Our proposed approaches work for all these policies, but our discussions focus on the write-back policy due to limited space for our presentation. The reliability and consistency of delayed writes in write-back caching are orthogonal issues to this paper's focus, and CloudCache can leverage the existing solutions (e.g., [17]) to address them.

Trace	Time (days)	Total IO (GB)	WSS (GB)	Write (%)
Webserver	281	2,247	110	51
Moodle	161	17,364	223	13
Fileserver	152	57,887	1037	22

Table 1: Trace statistics

In the next few sections, we introduce the two components of CloudCache, *on-demand cache allocation* and *dynamic cache migration*. As we describe the designs, we will also present experimental results as supporting evidence. We consider a set of block-level IO traces [7] collected from a departmental private cloud as representative workloads. The characteristics of the traces are summarized in Table 1. These traces allow us to study long-term cache behavior, in addition to the commonly used traces [4] which are only week-long.

4 On-demand Cache Allocation

CloudCache addresses two key questions about on-demand cache allocation. First, *how to model the cache demand of a workload?* A cloud workload includes IOs with different levels of temporal locality which affect the cache hit ratio differently. A good cache demand model should be able to capture the IOs that are truly important to the workload's performance in order to maximize the performance while minimizing cache utilization and flash wear-out. Second, *how to use the cache demand model to allocate cache and admit data into cache?* We need to predict the workload's cache demand accurately online in order to guide cache allocation, and admit only the useful data into cache so that the allocation does not get overflown. In this section, we present the CloudCache's solutions to these two questions.

4.1 RWS-based Cache Demand Model

Working Set (WS) is a classic model often used to estimate the cache demand of a workload. The working set $WS(t,T)$ at time t is defined as the set of distinct (address-wise) data blocks referenced by the workload during a time interval $[t-T,t]$ [12]. This definition uses

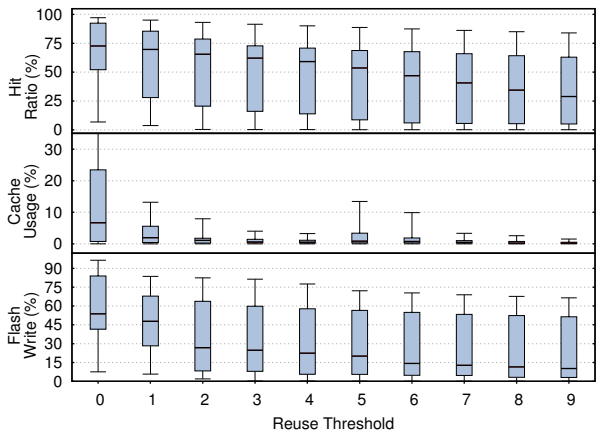


Figure 2: RWS analysis using different values of N

the principle of locality to form an estimate of the set of blocks that the workload will access next and should be kept in the cache. The *Working Set Size* (WSS) can be used to estimate the cache demand of the workload.

Although it is straightforward to use WSS to estimate a VM's flash cache demand, a serious limitation of this approach is that it does not differentiate the level of temporal locality of the data in the WS. Unfortunately, data with weak temporal locality, e.g., long bursts of sequential accesses, are abundant at the flash cache layer, as they can be found in many types of cloud workloads, e.g., when the guest system in a VM performs a weekly backup operation. Caching these data is of little benefit to the application's performance, since their next reuses are too far into the future. Allowing these data to be cached is in fact detrimental to the cache performance, as they evict data blocks that have better temporal locality and are more important to the workload performance. Moreover, they cause unnecessary wear-out to the flash device with little performance gain in return.

To address the limitation of the WS model, we propose a new cache-demand model, *Reuse Working Set*, $RWS_N(t,T)$, which is defined as the set of distinct (address-wise) data blocks that a workload has *reused* at least N times during a time interval $[t-T,t]$. Compared to the WS model, RWS captures only the data blocks with a temporal locality that will benefit the workload's cache hit ratio. When $N = 0$ RWS reduces to WS. We then propose to use *Reuse Working Set Size* (*RWSS*) as the estimate of the workload's cache demand. Because RWSS disregards low-locality data, it has the potential to more accurately capture the workload's actual cache demand, and reduce the cache pollution and unnecessary wear-out caused by such data references.

To confirm the effectiveness of the RWS model, we analyze the MSR Cambridge traces [4] with different values of N and evaluate the impact on cache hit ratio, cache usage—the number of cached blocks vs. the number of IOs received by cache, and flash write ratio—the number of writes sent to cache device vs. the number of IOs received by cache. We assume that a data block is admitted into the cache only after it has been accessed N times, i.e., we cache only the workload's RWS_N. Figure 2 shows the distribution of these metrics from the 36 MSR traces using box plots with whiskers showing the quartiles. Increasing N from 0, when we cache the WS, to 1, when we cache the RWS_1, the median hit ratio is reduced by 8%, but the median cache usage is reduced by 82%, and the amount of flash writes is reduced by 19%. This trend continues as we further increase N.

These results confirm the effectiveness of using RWSS to estimate cache demand—it is able to substantially reduce a workload's cache usage and its induced wear-out at a small cost of hit ratio. A system administrator can balance performance against cache usage and endurance by choosing the appropriate N for the RWS model. In general, N = 1 or 2 gives the best tradeoff between these objectives. (Similar observations can be made for the traces listed in Table 1.) In the rest of this paper, we use N = 1 for RWSS-based cache allocation. Moreover, when considering a cloud usage scenario where a shared cache cannot fit the working-sets of all the workloads, using the RWS model to allocate cache capacity can achieve better performance because it prevents the low-locality data from flushing the useful data out of the cache.

In order to measure the RWSS of a workload, we need to determine the appropriate time window to observe the workload. There are two relevant questions here. First, how to track the window? In the original definition of process WS [12], the window is set with respect to the process time, i.e., the number of accesses made by the process, instead of real time. However, it is difficult to use the number of accesses as the window to measure a VM's WS or RWSS at the flash cache layer, because the VM can go idle for a long period of time and never fill up its window, causing the previously allocated cache space to be underutilized. Therefore, we use real-time-based window to observe a workload's RWSS.

The second question is how to decide the size of the time window. If the window is set too small, the observed RWS cannot capture the workload's current locality, and the measured RWSS underestimates the workload's cache demand. If the window is set too large, it may include the past localities that are not part of the workload's current behavior, and the overestimated RWSS will waste cache space and cause unnecessary wear-out. Our solution to this problem is to profile the workload for a period of time, and simulate the cache hit ratio when we allocate space to the workload based on its RWSS measured using different sizes of windows. We then choose the window at the "knee point" of this

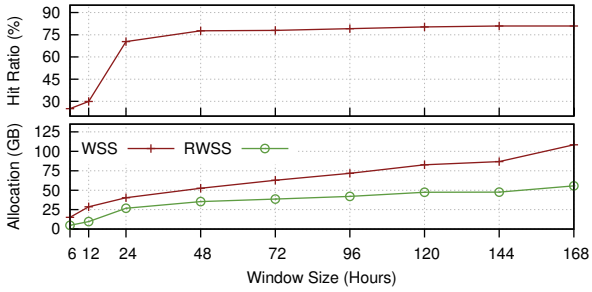


Figure 3: Time window analysis for the Moodle trace

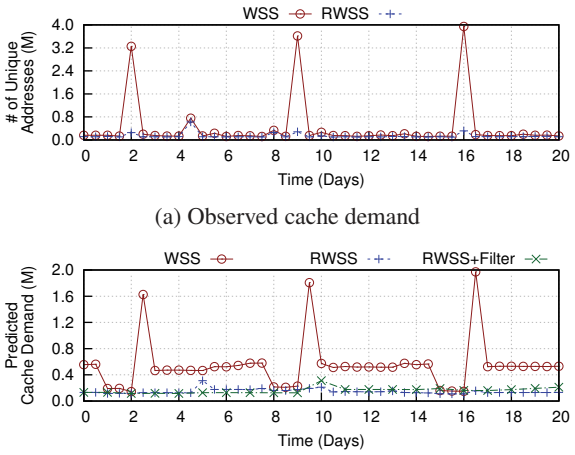
hit ratio vs. window size model, i.e., the point where the hit ratio starts to flatten out. This profiling can be performed periodically, e.g., bi-weekly or monthly, to adjust the choice of window size *online*.

We present an example of estimating the window size using two weeks of the Moodle trace. Figure 3 shows that the hit ratio increases rapidly as the window size increases initially. After the 24-hour window size, it starts to flatten out, while the observed RWSS continues to increase. Therefore, we choose between 24 to 48 hours as the window size for measuring the RWSS of this workload, because a larger window size will not get enough gain in hit ratio to justify the further increase in the workload's cache usage, if we allocate the cache based on the observed RWSS. In case of workloads for which the hit ratio keeps growing slowly with increasing window size but without showing an obvious knee point, the window size should be set to a small value because it will not affect the hit ratio much but can save cache space for other workloads with clear knee points.

4.2 Online Cache Demand Prediction

The success of RWSS-based cache allocation also depends on whether we can accurately predict the cache demand of the next time window based on the RWSS values observed from the previous windows. To address this problem, we consider the classic exponential smoothing and double exponential smoothing methods. The former requires a smoothing parameter α, and the latter requires an additional trending parameter β. The values of these parameters can have a significant impact on the prediction accuracy. We address this issue by using the self-tuning versions of these prediction models, which estimate these parameters based on the error between the predicted and observed RWSS values.

To further improve the robustness of the RWSS prediction, we devise filtering techniques which can dampen the impact of outliers in the observed RWSS values when predicting RWSS. If the currently observed RWSS is λ times greater than the average of the previous n observed values (including the current one), this value is replaced with the average. For example, n is set to 20 and λ is set to 5 in our experiments. In this way, an outlier's impact in the prediction is mitigated.



(a) Observed cache demand

(b) Predicted demand

Figure 4: RWSS-based cache demand prediction

Figure 4 shows an example of the RWSS prediction for three weeks of the Webserver trace. The recurring peaks in the observed WSS in Figure 4a are produced by a weekly backup task performed by the VM, which cause the predicted WSS in Figure 4b to be substantially inflated. In comparison, the RWSS model automatically filters out these backup IOs and the predicted RWSS is only 26% of the WSS on average for the whole trace. The filtering technique further smooths out several outliers (e.g., between Day 4 and 5) which are caused by occasional bursts of IOs that do not reflect the general trend of the workload.

4.3 Cache Allocation and Admission

Based on the cache demands estimated using the RWSS model and prediction methods, the cache allocation to the concurrent VMs is adjusted accordingly at the start of every new time window—the smallest window used to estimate the RWSS of all the VMs. The allocation of cache capacity should not incur costly data copying or flushing. Hence, we consider *replacement-time enforcement* of cache allocation, which does not physically partition the cache across VMs. Instead, it enforces logical partitioning at replacement time: a VM that has not used up its allocated share takes its space back by replacing a block from VMs that have exceeded their shares. Moreover, if the cache is not full, the spare capacity can be allocated to the VMs proportionally to their predicted RWSSes or left idle to reduce wear-out.

The RWSS-based cache allocation approach also requires an *RWSS-based cache admission policy* that admits only reused data blocks into the cache; otherwise, the entire WS will be admitted into the cache space allocated based on RWSS and evict useful data. To enforce this cache admission policy, CloudCache uses a small

portion of the main memory as the staging area for referenced addresses, a common strategy for implementing cache admission [33, 15]. A block is admitted into the cache only after it has been accessed N times, no matter whether they are reads or writes. The size of the staging area is bounded and when it gets full the staged addresses are evicted using LRU. We refer to this approach of staging only addresses in main memory as *address staging*.

CloudCache also considers a *data staging* strategy for cache admission, which stores both the addresses and data of candidate blocks in the staging area and manages them using LRU. Because main memory is not persistent, so more precisely, only the data returned by read requests are staged in memory, but for writes only their addresses are staged. This strategy can reduce the misses for read accesses by serving them from the staging area before they are admitted into the cache. The tradeoff is that because a data block is much larger than an address (8B address per 4KB data), for the same staging area, data staging can track much less references than address staging and may miss data with good temporal locality.

To address the limitations of address staging and data staging and combine their advantages, CloudCache considers a third *hybrid staging* strategy in which the staging area is divided to store addresses and data, and the address and data partitions are managed using LRU separately. This strategy has the potential to reduce the read misses for blocks with small reuse distances by using data staging and admitting the blocks with relative larger reuse distances by using address staging.

4.4 Evaluation

The rest of this section presents an evaluation of the RWSS-based on-demand cache allocation approach. CloudCache is created upon block-level virtualization by providing virtual block devices to VMs and transparently caching their data accesses to remote block devices accessed across the network (Figure 1). It includes a kernel module that implements the virtual block devices, monitors VM IOs, and enforces cache allocation and admission, and a user-space component that measures and predicts RWSS and determines the cache shares for the VMs. The kernel module stores the recently observed IOs in a small circular buffer for the user-space component to use, while the latter informs the former about the cache allocation decisions. The current implementation of CloudCache is based on Linux and it can be seamlessly deployed as a drop-in solution on Linux-based environments including VM systems that use Linux-based IO stack [8, 2]. We have also created a user-level cache simulator of CloudCache to facilitate the cache hit ratio and flash write ratio analysis, but we use only the real implementation for measuring real-time performance.

The traces described in Section 3 are replayed on a real iSCSI-based storage system. One node from a compute cluster is set up as the storage server and the others as the clients. Each node has two six-core 2.4GHz Xeon CPUs and 24GB of RAM. Each client node is equipped with the CloudCache modules, as part of the Dom0 kernel, and flash devices (Intel 120GB MLC SATA-interface) to provide caching to the hosted Xen VMs. The server node runs the IET iSCSI server to export the logical volumes stored on a 1TB 7.2K RPM hard disk to the clients via a Gigabit Ethernet. The clients run Xen 4.1 to host VMs, and each VM is configured with 1 vCPU and 2GB RAM and runs kernel 2.6.32. The RWSS window size for the Webserver, Moodle, and Fileserver traces are 48, 24, and 12 hours, respectively. Each VM's cache share is managed using LRU internally, although other replacement policies are also possible.

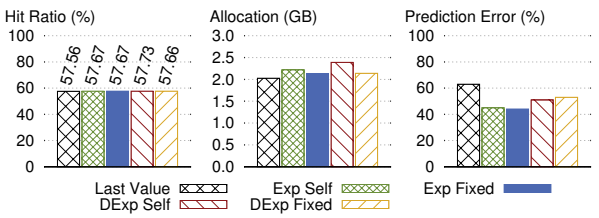


Figure 5: Prediction accuracy

4.4.1 Prediction Accuracy

In the first set of experiments we evaluate the different RWSS prediction methods considered in Section 4.2: (1) *Exp fixed*, exponential smoothing with $\alpha = 0.3$, (2) *Exp self*, a self-tuning version of exponential smoothing, (3) *DExp fixed*, double-exponential smoothing with $\alpha = 0.3$ and $\beta = 0.3$, (4) *DExp self*, a self-tuning version of double-exponential smoothing, and (5) *Last value*, a simple method that uses the last observed RWSS value as predicted value for the new window.

Figure 5 compares the different prediction methods using three metrics: (1) *hit ratio*, (2) *cache allocation*, and (3) *prediction error*—the absolute value of the difference between the predicted RWSS and observed RWSS divided by the observed RWSS. Prediction error affects both of the other two metrics—under-prediction increases cache misses and over-prediction uses more cache. The figure shows the average values of these metrics across all the time windows of the entire 9-month Webserver trace.

The results show that the difference in hit ratio is small among the different prediction methods but is considerable in cache allocation. The last value method has the highest prediction error, which confirms the need of prediction techniques. The exponential smoothing methods have the lowest prediction errors, and *Exp Self* is more preferable because it automatically trains its parameter. We believe that more advanced prediction methods are possible to further improve the prediction accuracy and

(a) Staging area size (b) Staging strategies

Figure 6: Staging strategy analysis

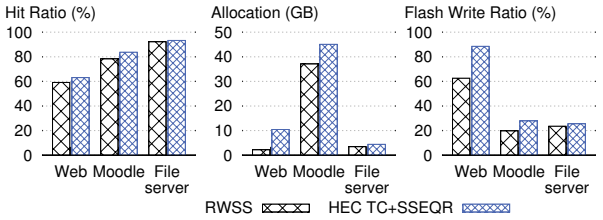


Figure 7: Comparison to HEC

our solution can be extended to run multiple prediction methods at the same time and choose the best one at runtime. But this simple smoothing-based method can already produce good results, as shown in the following experiments which all use *Exp Self* to predict cache demand.

4.4.2 Staging Strategies

In the second set of experiments, we evaluate CloudCache's staging strategies. First, we study the impact of the staging area size. In general, it should be decided according to the number of VMs consolidated to the same cache and the IO intensity of their workloads. Therefore, our approach is to set the total staging area size as a percentage, e.g., between 0.1% and 1%, of the flash cache size, and allocate the staging area to the workloads proportionally to their flash cache allocation. Figure 6a gives an example of how the staging area allocation affects the Webserver workload's hit ratio when using address staging. The results from data staging are similar. In the rest of the paper, we always use 256MB as the total staging area size for RWSS-based cache allocation. Note that we need 24B of the staging space for tracking each address, and an additional 4KB if its corresponding data is also staged.

Next we compare the address, data, and hybrid staging (with a 1:7 ratio between address and data staging space) strategies with the same staging area size in Figure 6b. Data staging achieves a better read hit ratio than address staging by 67% for the Webserver trace but it loses to address staging by 9% for Moodle. These results confirm our discussion in Section 4.3 about the tradeoff between these strategies. In comparison, the hybrid staging combines the benefits of these two and is consistently the best for all traces. We have tested different ratios for hybrid staging, and our results show that the hit ratio difference is small (<1%). But a larger address staging area tracks a longer history and admits more data into the cache, which results in more cache usage and flash writes. Therefore, in the rest of this paper, we always use hybrid staging with 1:7 ratio between address and data staging space for RWSS-based allocation.

We also compare to the related work High Endurance Cache (HEC) [33] which used two cache admission techniques to address flash cache wear-out and are closely related to our staging strategies. HEC's Touch Count (TC) technique uses an in-memory bitmap to track all the cache blocks (by default 4MB) and admit only reused blocks into cache. In comparison, CloudCache tracks only a small number of recently accessed addresses to limit the memory usage and prevent blocks accessed too long ago from being admitted into cache. HEC's Selective Sequential Rejection (SSEQR) technique tracks the sequentiality of accesses and rejects long sequences (by default any longer-than-4MB sequence). In comparison, CloudCache uses the staging area to automatically filter out long scan sequences.

Because HEC did not consider on-demand cache allocation, we implemented it by using TC to predict cache demand and using both TC and SSEQR to enforce cache admission. Figures 7 shows the comparison using the different traces, which reveals that on average HEC allocates up to 3.7x more cache than our RWSS-based method and causes up to 29.2% higher flash write ratio—the number of writes sent to cache device vs. the number of IOs received by cache. In return, it achieves only up to 6.4% higher hit ratio. The larger cache allocation in HEC is because it considers all the historical accesses when counting reuses, whereas the RWSS method considers only the reuses occurred in the recent history—the previous window. (If we were able to apply the same cache allocation given by the RWSS method while using HEC's cache admission method, we would achieve a much lower hit ratio, e.g., 68% lower for Moodle, and still a higher flash write ratio, e.g., 69% higher for Moodle.) The result also confirms that the RWSS method is able to automatically reject scan sequences (e.g., it rejects on average 90% of the IOs during the backup periods), whereas HEC needs to explicitly detect scan sequences using a fixed threshold.

4.4.3 WSS vs. RWSS-based Cache Allocation

In the third set of experiments, we compare RWSS-based to WSS-based cache allocation using the same prediction method, exponential smoothing with self-tuning. In both cases, the cache allocation is strictly enforced, and at the start of each window, the workload's extra cache usage beyond its new allocation is immediately dropped. This setting produces the *worst-case* result for on-demand cache allocation, because in practice Cloud-

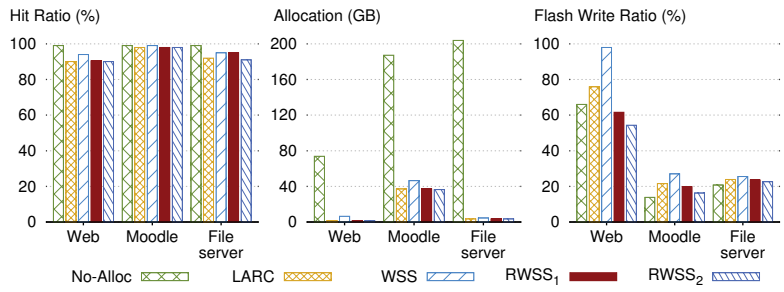


Figure 8: Allocation methods

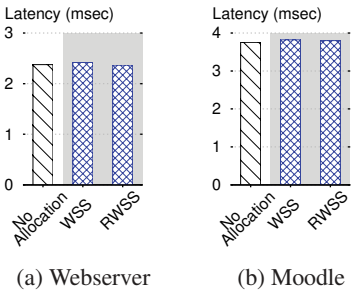


Figure 9: VM IO latency comparison

Cache allows a workload to use spare capacity beyond its allocation and its extra cache usage is gradually reclaimed via replacement-time enforcement. We also include the case where the workload can use up the entire cache as a baseline (*No Allocation*), where the cache is large enough to hold the entire working set and does not require any replacement.

Figure 8 shows the comparison among these different approaches. For RWSS, we consider two different values for the N in RWS_N, as described in Section 4.1. In addition, we also compare to the related cache admission method, LARC [15], which dynamically changes the size of the staging area according to the current hit ratio—a higher hit ratio reduces the staging area size. Like HEC, LARC also does not provide on-demand allocation, so we implemented it by using the number of reused addresses to predict cache demand and using LARC for cache admission.

$RWSS_1$ achieves a hit ratio that is only 9.1% lower than *No Allocation* and 4% lower than *WSS*, but reduces the workload's cache usage substantially by up to 98% compared to *No Allocation* and 76% compared to *WSS*, and reduces the flash write ratio by up to 6% compared to *No Allocation* and 37% compared to *WSS*. (The cache allocation of RWSS and LARC is less than 4GB for Webserver and Fileserver and thus barely visible in the figure). *No Allocation* has slightly lower flash write ratio than $RWSS_1$ for Moodle and Fileserver only because it does not incur cache replacement, as it is allowed to occupy as much cache space as possible, which is not a realistic scenario for cloud environments. Compared to *LARC*, $RWSS_1$ achieves up to 3% higher hit ratio and still reduces cache usage by up to 3% and the flash writes by up to 18%, while using 580MB less staging area on average. Comparing the two different configurations of RWSS, $RWSS_2$ reduces cache usage by up to 9% and flash writes by up to 18%, at the cost of 4% lower hit ratio, which confirms the tradeoff of choosing different values of N in our proposed RWS model.

To evaluate how much performance loss the hit ratio reduction will cause, we replay the traces and measure their IO latencies. We consider a one-month portion of the Webserver and Moodle traces. They were replayed on the real VM storage and caching setup specified in Section 4.4. We compare the different cache management methods in terms of 95th percentile IO latency. Figure 9 shows that the *RWSS*-based method delivers the similar performance as the alternatives (only 1% slower than *No Allocation* for Moodle) while using much less cache and causing more writes to the cache device as shown in the previous results.

The results confirm that our proposed RWSS-based cache allocation can indeed substantially reduce a workload's cache usage and the corresponding wear-out at only a small performance cost. In real usage scenarios our performance overhead would be much smaller because a workload's extra cache allocation does not have to be dropped immediately when a new time window starts and can still provide hits while being gradually replaced by the other workloads. Moreover, because the WSS-based method requires much higher cache allocations for the same workloads, cloud providers have to either provision much larger caches, which incurs more monetary cost, or leave the caches oversubscribed, which leads to bad performance as the low-locality data are admitted into the cache and flush out the useful data.

5 Dynamic Cache Migration

The *on-demand cache allocation* approach discussed in the previous section allows CloudCache to estimate the cache demands of workloads online and dynamically allocate the shared capacity to them. To handle scenarios where the cache capacity is insufficient to meet all the demands, this section presents the *dynamic cache migration* approach to balance the cache load across different hosts by dynamically migrating a workload's cached data along with its VM. It also considers techniques to optimize the performance for the migrating VM as well as minimize the impact to the others during the migration.

5.1 Live Cache Migration

Live VM migration allows a workload to be transparently migrated among physical hosts while running in its VM [11, 25]. In CloudCache, we propose to use live

VM migration to balance the load on the flash caches of VM hosts—when a host's cache capacity becomes insufficient to meet the local VMs' total cache demands (as estimated by their predicted RWSSes), some VMs can be migrated to other hosts that have spare cache capacity to meet their cache demands.

VM-migration-based cache load balancing presents two challenges. First, the migrating VM's dirty cache data on the migration *source* host must be synchronized to the *destination* host before they can be accessed again by the VM. A naive way is to flush all the dirty data to the remote storage server for the migrating VM. Depending on the amount of dirty data and the available IO bandwidth, the flushing can be time consuming, and the VM cannot resume its activity until the flushing finishes. The flushing will also cause a surge in the storage server's IO load and affect the performance of the other VMs sharing the server. Second, the migrating VM needs to warm up the cache on the destination host, which may also take a long time, and it will experience substantial performance degradation till the cache is warmed up [14, 7].

To address these challenges, CloudCache's dynamic cache migration approach uses a combination of reactive and proactive migration techniques:

On-Demand Migration: When the migrated VM accesses a block that is dirty in the source host's cache, its local cache forwards the request to the source host and fetches the data from there, instead of the remote storage server. The metadata of the dirty blocks, i.e., their logical block addresses, on the source host are transferred along with VM migration, so the destination host's local cache is aware of which blocks are dirty on the source host. Because the size of these metadata is small (e.g., 8B per 4KB data), the metadata transfer time is often negligible. It is done before the VM is activated on the destination, so the VM can immediately use the cache on the destination host.

Background Migration: In addition to reactively servicing requests from the migrated VM, the source host's cache also proactively transfers the VM's cached data—its RWS—to the destination host. The transfer is done in background to mitigate the impact to the other VMs on the source host. This background migration allows the destination host to quickly warm up its local cache and improve the performance of the migrated VM. It also allows the source host to quickly reduce its cache load and improve the performance of its remaining VMs. Benefiting from the RWSS-based cache allocation and admission, the data that need to be transferred in background contain only the VM's RWS which is much smaller than the WS, as shown in the previous section's results. Moreover, when transferring the RWS, the blocks are sent in the decreasing order of their recency so the data that are most likely to used next are transferred earliest.

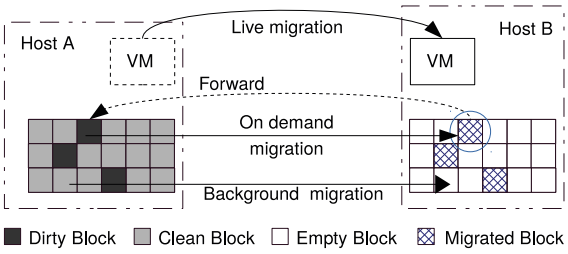


Figure 10: Architecture of dynamic cache migration

On-demand migration allows the migrated VM to access its dirty blocks quickly, but it is inefficient for transferring many blocks. Background migration can transfer bulk data efficiently but it may not be able to serve the current requests that the migrated VM is waiting for. Therefore, the combination of these two migration strategies can optimize the performance of the VM. Figure 10 illustrates how CloudCache performs cache migration. When a VM is live-migrated from Host *A* to Host *B*, to keep data consistent while avoiding the need to flush dirty data, the cached metadata of dirty blocks are transferred to Host *B*. Once the VM live migration completes, the VM is activated on Host *B* and its local flash cache can immediately service its requests. By using the transferred metadata, the cache on Host *B* can determine whether a block is dirty or not and where it is currently located. If a dirty block is still on Host *A*, a request is sent to fetch it on demand. At the same time, Host *A* also sends the RWS of the migrated VM in background. As the cached blocks are moved from Host *A* to Host *B*, either on-demand or in background, Host A vacates their cache space and makes it available to the other VMs.

The CloudCache module on each host handles both the operations of local cache and the operations of cache migration. It employs a multithreaded design to handle these different operations with good concurrency. Synchronization among the threads is needed to ensure consistency of data. In particular, when the destination host requests a block on demand, it is possible that the source host also transfers this block in background, at the same time. The destination host will discard the second copy that it receives, because it already has a copy in the local cache and it may have already overwritten it. As an optimization, a write that aligns to the cache block boundaries can be stored directly in the destination host's cache, without fetching its previous copy from the source host. In this case, the later migrated copy of this block is also discarded. The migrating VM needs to keep the same device name for its disk, which is the virtual block device presented by CloudCache's block-level virtualization. CloudCache assigns unique names to the virtual block devices based on the unique IDs of the VMs in the cloud system. Before migration, the mapping from the

virtual block device to physical device (e.g., the iSCSI device) is created on the destination host, and after migration, the counterpart on the source host is removed.

5.2 Migration Rate Limiting

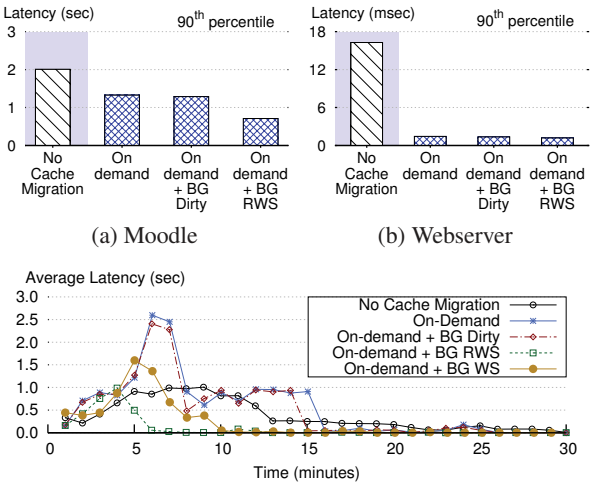
While the combination of on-demand and background migrations can optimize the performance of a migrating VM, the impact to the other VMs on the source and destination hosts also needs to be considered. Cache migration requires reads on the source host's cache and writes to the destination host's cache, which can slow down the cache IOs from the other co-hosted VMs. It also requires network bandwidth, in addition to the bandwidth already consumed by VM memory migration (part of the live VM migration [11, 25]), and affects the network IO performance of the other VMs.

In order to control the level of performance interference to co-hosted VMs, CloudCache is able to limit the transfer rate for cache migration. Given the rate limit, it enforces the maximum number of data blocks that can be transferred from the source host to the destination host every period of time (e.g., 100ms), including both on-demand migration and background migration. Once the limit is hit, the migration thread will sleep and wait till the next period to continue the data transfer. If on-demand requests arrive during the sleep time, they will be delayed and served immediately after the thread wakes up. The rate can be set based on factors including the priority of the VMs and the RWSS of the migrating VM. CloudCache allows a system administrator to tune the rate in order to minimize the cache migration impact to the co-hosted VMs and still migrate the RWS fast enough to satisfy the cache demands.

5.3 Evaluation

We evaluate the performance of CloudCache's dynamic cache migration using the same testbed described in Section 4.4. Dynamic cache migration is implemented in the CloudCache kernel module described in Section 4.4. It exposes a command-line interface which is integrated with virt-manager [3] for coordinating VM migration with cache migration. We focus on a day-long portion of the Moodle and Webserver traces. The Moodle one-day trace is read-intensive which makes 15% of its cached data dirty (about 5GB), and the Webserver one-day trace is write-intensive which makes 85% of its cached data dirty (about 1GB).

We consider four different approaches: (1) *No Cache Migration:* the cached data on the source host are not migrated with the VM; (2) *On-demand:* only the on-demand cache migration is used to transfer dirty blocks requested by the migrated VM; (3) *On-demand + BG Dirty:* in addition to on-demand cache migration, background migration is used to transfer only the dirty blocks of the migrated VM; (4) *On-demand + BG RWS:* both



(a) Moodle (b) Webserver

(c) The migrating VM's performance (average IO latency per minute) for Moodle. The migration starts at the 5th minute.

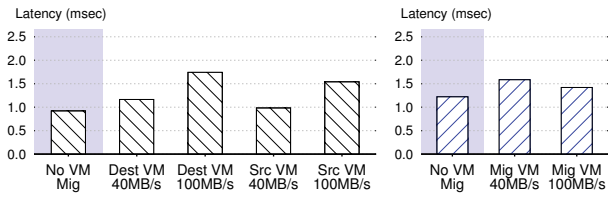
Figure 11: Migration strategies

on-demand migration of dirty blocks and background migration of RWS are used. In this experiment, we assume that the cache migration can use the entire 1Gbps network bandwidth, and we study the impact of rate limiting in the next experiment. For on-demand cache migration, it takes 0.3s to transfer the metadata for the Moodle workload and 0.05s for the Webserver workload.

Figure 11a shows that for the Moodle workload, on-demand cache migration decreases the 90th percentile latency by 33% and the addition of background migration of dirty data decreases it by 35%, compared to *No Cache Migration*. However, the most significant improvement comes from the use of both on-demand migration of dirty data and background migration of the entire RWS, which reduces the latency by 64%. The reason is that this workload is read-intensive and reuses a large amount of clean data; background migration of RWS allows the workload to access these data from the fast, local flash cache, instead of paying the long network latency for accessing the remote storage.

For the Webserver workload, because its RWS is mostly dirty, the difference among the three cache migration strategies is smaller than the Moodle workload (Figure 11b). Compared to the *No Cache Migration* case, they reduce the 90th percentile latency by 91.1% with on-demand migration of dirty data, and by 92.6% with the addition of background migration of RWS.

Note that the above results for the *No Cache Migration* case do not include the time that the migrated VM has to wait for its dirty data to be flushed from the source host to the remote storage before it can resume running again, which is about 54 seconds for the Moodle workload and 12 seconds for the Webserver workload, assuming it can use all the bandwidths of the network and storage server.



(a) Co-hosted VMs (b) Migrating VM

Figure 12: Impact of different cache migration rate

In comparison, the VM has zero downtime when using our dynamic cache migration.

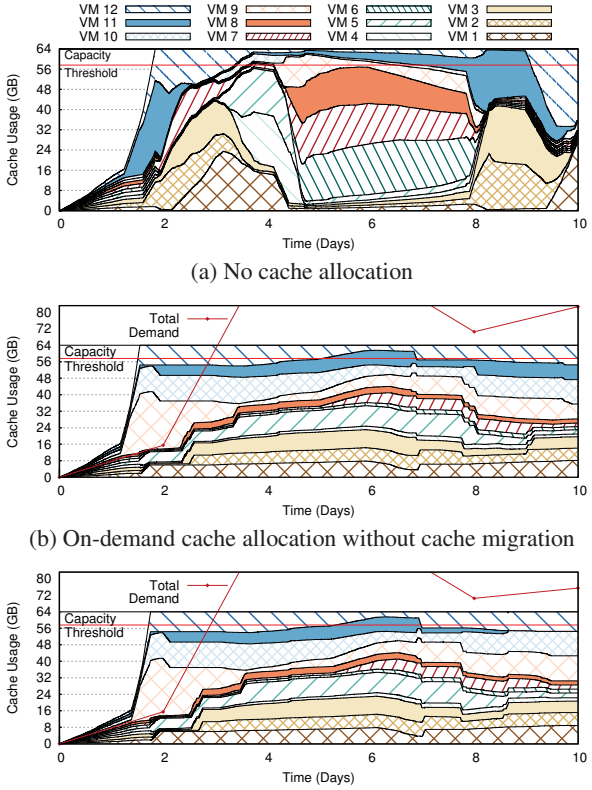
Figure 11c shows how the migrating VM's performance varies over time in this Moodle experiment so we can observe the real-time performance of the different migration strategies. The peaks in *On-demand* and *On-demand + BG Dirty* are caused by bursts of on-demand transfer of clean data blocks requested by the migrated VM. We believe that we can further optimize our prototype implementation to avoid such spikes in latency.

In Figure 11c, we also compare our approach to an alternative cache migration implementation (*On-demand + BG WS*) which migrates the VM's entire working set without the benefit of our proposed RWS model. Using the same Moodle trace, at the time of migration, its RWSS is 32GB and WSS is 42GB. As a result, migrating the WS takes twice the time of migrating only the RWS (6mins vs. 3mins) and causes a higher IO latency overhead too (71% higher in 90th percentile latency).

In the next experiment, we evaluate the performance impact of rate limiting the cache migration. In addition to the migrating VM, we run another IO-intensive VM on both the source and destination hosts, which replays a different day-long portion of the Webserver trace. We measure the performance of all the VMs when the cache migration rate is set at 40MB/s and 100MB/s and compare to their normal performance when there is no VM or cache migration. Figure 12 shows that the impact to the co-hosted VMs' 90th percentile IO latency is below 16% and 21% for the 40MB/s and 100MB/s rate respectively. Note that this is assuming that the co-hosted VMs already have enough cache space, so in reality, their performance would actually be much improved by using the cache space vacated from the migrating VM. Meanwhile, the faster migration rate reduces the migrating VM's 90th percentile IO latency by 6%. Therefore, the lower rate is good enough for the migrating VM because the most recently used data are migrated first, and it is more preferable for its lower impact to the co-hosted VMs.

6 Putting Everything Together

The previous two sections described and evaluated the RWSS-based on-demand cache allocation and dynamic cache migration approaches separately. In this section,



(a) No cache allocation

(b) On-demand cache allocation without cache migration

(c) On-demand cache allocation with dynamic cache migration

Figure 13: Cache usages of 12 concurrent VMs

we present how to use them together to realize on-demand cache management for multiple VM hosts. Consider the flash cache on a single host. If its capacity is sufficient to satisfy the predicted cache demands for all the local VMs, it is simply allocated to the VMs according to their demands. The spare capacity is distributed to the VMs proportionally to their demands, or left idle to minimize wear-out. If the cache capacity is not sufficient, then cache migration needs to be considered in order to satisfy the demands of all the VMs.

When considering the use of cache migration, there are three key questions that need to be answered, *when to migrate*, *which VM to migrate*, and *which host to migrate it to?* To answer the first question, CloudCache reserves a certain percentage (e.g., 10%) of the cache capacity as a buffer to absorb the occasional surges in cache demands, and it starts a migration when the total cache demand exceeds the 90% threshold for several consecutive RWSS windows (e.g., three times). This approach prevents the fluctuations in cache workloads from triggering unnecessary cache migrations which affect the VMs' performance and the system's stability.

To answer the second and third questions, CloudCache's current strategy is to minimize the imbalance of cache load among the hosts in the system. The host that requires cache migration queries every other host's cur-

rent cache load. It then evaluates all the possible migration plans of moving one of its local VMs to a host that can accommodate the VM's RWS under the 90% threshold. It then chooses the plan that minimizes the variance of the hosts' cache load distribution.

We use a real experiment to illustrate the use of our approaches for meeting the cache demands of dynamically changing workloads. We consider two VM hosts each with 64GB of flash cache. Host *A* ran 12 VMs, and Host *B* ran three VMs, concurrently. Each VM replayed a different 10-day portion of the Webserver trace. The cache allocation was adjusted every 2 days on both hosts. The first time window is the warm-up phase during which the VMs were given equal allocation of the cache capacity. Afterwards, the cache was allocated to the VMs proportionally to their estimated RWSSes. Moreover, a VM could take more than its share if there was idle capacity from the other VMs' shares because our approach is work-conserving. The experiment was done on the real VM storage and caching setup specified in Section 4.4.

Figure 13a shows how the cache space is distributed among the VMs on Host A when *(a)* there is no cache allocation, *(b)* on-demand cache allocation but without cache migration, and *(c)* on-demand cache allocation with dynamic cache migration. Comparing (a) and (b), we can see how our RWSS-based on-demand allocation improves the fairness among the competing VMs. For example, between Days 4 and 8, VMs 6, 7, 8 dominated the cache space in (a), but in (b), every VM got a fair share of the cache space proportionally to their estimated RWSSes. Notice that VMs 7 and 8 were allocated much less in (b) than what they got in (a), which is an evidence of how the RWS-based cache demand model filtered out the VMs' low-locality data and kept only those that are useful to their performance. As a result, comparing the average performance of all 12 VMs across the entire experiment, (b) is better than (a) by 17% in terms of hit ratio and 13% in terms of 90th percentile IO latency.

In (c) dynamic cache migration was enabled in addition to on-demand cache allocation. After the total demand—the sum of the 12 VMs' RWSSes—exceeded the threshold for three consecutive windows, CloudCache initiated cache migration on Day 8 and chose to move VM 11, the one with the largest predicted RWSS at that time, and its cached data to Host *B*. As VM 11's RWS was moved to Host *B*, the remaining 11 VMs took over the whole cache on Host *A*, proportionally to their estimated RWSSes. As a result, comparing the average performance of all 12 VMs after Day 8, (c) is better than (b) by 49% in terms of hit ratio and 24% in terms of 90th percentile IO latency. Across the entire experiment, it outperforms (a) by 28% in hit ratio and 27% in 90th percentile IO latency, and outperforms (b) by 10% in hit ratio and 16% in 90th percentile IO latency.

Although this experiment involved only two VM hosts and the migration of only one VM, the above results are still representative for the migration of any VM and its cache data between two hosts in a large cloud computing environment. But we understand in such a large environment, more intelligence is required to make the optimal VM migration decisions. There is a good amount of related work (e.g., [31, 32]) on using VM migration to balance load on CPUs and main memory and to optimize performance, energy consumption, etc. CloudCache is the first to consider on-demand flash cache management across multiple hosts, and it can be well integrated into these related solutions to support the holistic management of different resources and optimization for various objectives. We leave this to our future work because the focus of this paper is on the key mechanisms for on-demand cache management, i.e., on-demand cache allocation and dynamic cache migration, which are missing in existing flash cache management solutions and are non-trivial to accomplish.

7 Related Work

There are several related flash cache management solutions. S-CAVE [22] considers the number of reused blocks when estimating a VM's cache demand, and allocates cache using several heuristics. vCacheShare [24] allocates a read-only cache by maximizing a unity function that captures a VM's disk latency, read-to-write ratio, estimated cache hit ratio, and reuse rate of the allocated cache capacity. Centaur [18] uses MRC and latency curves to allocate cache to VMs according to their QoS targets. However, these solutions admit all referenced data into cache, including those with weak temporal locality, and can cause unnecessary cache usage and wear-out. Moreover, none of them considers dynamic cache migration for meeting the demands when a cache is overloaded. These problems are addressed by CloudCache's on-demand cache allocation and dynamic cache migration approaches.

HEC and LARC studied cache admission policies to filter out data with weak temporal locality and reduce the flash wear-out [33, 15]. However, they do not consider the problem of how to allocate shared cache capacity to concurrent workloads, which is addressed by CloudCache. Moreover, our RWSS-based approach is able to more effectively filter out data with no reuses and achieve good reduction in cache footprint and wear-out, as shown in Section 4.4.

Bhagwat *et al*. studied how to allow a migrated VM to request data from the cache on its previous host [9], in the same fashion as the on-demand cache migration proposed in this paper. However, as shown in Section 5.3, without our proposed background cache migration, on-demand migration alone cannot ensure good per-

formance for the migrated VM. It also has a long-lasting, negative impact on the source host in terms of both performance interference and cache utilization. When the migrated VM's data are evicted on the source host, the performance becomes even worse because a request has to be forwarded by the source host to the primary storage. VMware's vSphere flash read cache [6] also supports background cache migration. Although its details are unknown, without our proposed RWS model, a similar solution would have to migrate the VM's entire cache footprint. As shown in Section 5.3, this requires longer migration time and causes higher impact to performance. In comparison, CloudCache considers the combination of on-demand migration and background migration and is able to minimize the performance impact to both the migrated VM and the other co-hosted VMs.

In the context of processor and memory cache management, ARC addresses the cache pollution from scan sequences by keeping such data in a separate list (*T1*) and preventing them from flooding the list (*T2*) of data with reuses [23]. However, data in *T1* still occupy cache space and cause wear-out. Moreover, it does not provide answers to how to allocate shared cache space to concurrent workloads. Related work [19] proposed the model of effective reuse set size to capture the necessary cache capacity for preventing non-reusable data from evicting reusable data, but it assumes that all data have to be admitted into the cache. There are also related works on processor and memory cache allocations. For example, miss-rate curve (MRC) can be built to capture the relationship between a workload's cache hit ratio and its cache sizes, and used to guide cache allocation [27, 34, 28, 29, 30]. Process migration was also considered for balancing processor cache load on a multicore system [16].

Compared to these processor and memory caching works, flash cache management presents a different set of challenges. Low-locality data are detrimental to not only a flash cache's performance but also its lifetime, which unfortunately are abundant at the flash cache layer. While VM migration can be used to migrate workloads across hosts, the large amount of cached data cannot be simply flushed or easily shipped over. CloudCache is designed to address these unique challenges by using RWSS to allocate cache to only data with good locality and by providing dynamic cache migration with techniques to minimize its impact to VM performance.

8 Conclusions and Future Work

Flash caching has great potential to address the storage bottleneck and improve VM performance for cloud computing systems. Allocating the limited cache capacity to concurrent VMs according to their demands is key to making efficient use of flash cache and optimizing VM performance. Moreover, flash devices have serious endurance issues, whereas weak-temporal-locality data are abundant at the flash cache layer, which hurt not only the cache performance but also its lifetime. Therefore, on-demand management of flash caches requires fundamental rethinking on how to estimate VMs' cache demands and how to provision space to meet their demands.

This paper presents CloudCache, an on-demand cache management solution to these problems. First, it employs a new cache demand model, Reuse Working Set (RWS), to capture the data with good temporal locality, allocate cache space according to the predicted Reuse Working Set Size (RWSS), and admit only the RWS into the allocated space. Second, to handle cache overload situations, CloudCache takes a new cache migration approach which live-migrates a VM with its cached data to meet the cache demands of the VMs. Extensive evaluations based on real-world traces confirm that the RWSS-based cache allocation approach can achieve good cache hit ratio and IO latency for a VM while substantially reducing its cache usage and flash wear-out. It also confirms that the dynamic cache migration approach can transparently balance cache load across hosts with small impact to the migrating VM and the other co-hosted VMs.

CloudCache provides a solid framework for our future work in several directions. First, we plan to use flash simulators and open-controller devices to monitor the actual Program/Erase cycles and provide more accurate measurement of our solution's impact on flash device wear-out. Second, when the aggregate cache capacity from all VM hosts is not sufficient, CloudCache has to allocate cache proportionally to the VMs' RWSSes. We plan to investigate a more advanced solution which maps each VM's cache allocation to its performance and optimizes the allocation by maximizing the overall performance of all VMs. Third, although our experiments confirm that flash cache allocation has a significant impact on application performance, the allocation of other resources, e.g., CPU cycles and memory capacity, is also important. We expect to integrate existing CPU and memory management techniques with CloudCache to provide a holistic cloud resource management solution. Finally, while the discussion in this paper focuses on flash-memory-based caching, CloudCache's general approach is also applicable to emerging NVM devices, which we plan to evaluate when they become available.

9 Acknowledgements

We thank the anonymous reviewers and our shepherd, Carl Waldspurger, for the thorough reviews and insightful suggestions. This research is sponsored by National Science Foundation CAREER award CNS-125394 and Department of Defense award W911NF-13-1-0157.

References

[1] Fusion-io ioCache. http://www.fusionio.com/products/iocache/.

[2] Kernel Based Virtual Machine. http://www.linux-kvm.org/page/Main_Page.

[3] Manage virtual machines with virt-manager. https://virt-manager.org.

[4] MSR cambridge traces. http://iotta.snia.org/traces/388.

[5] Network Block Device. http://nbd.sourceforge.net/.

[6] Performance of vSphere flash read cache in VMware vSphere 5.5. https://www.vmware.com/files/pdf/techpaper/vfrc-perf-vsphere55.pdf.

[7] D. Arteaga and M. Zhao. Client-side flash caching for cloud systems. In *Proceedings of International Conference on Systems and Storage (SYSTOR 14)*, pages 7:1–7:11. ACM, 2014.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP 03)*. ACM, 2003.

[9] D. Bhagwat, M. Patil, M. Ostrowski, M. Vilayannur, W. Jung, and C. Kumar. A practical implementation of clustered fault tolerant write acceleration in a virtualized environment. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 287–300, Santa Clara, CA, 2015. USENIX Association.

[10] 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 *Proceedings of the 28th IEEE Conference on Massive Data Storage (MSST 12)*, Pacific Grove, CA, USA, 2012. IEEE.

[11] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation (NSDI 05)*, pages 273–286. USENIX Association, 2005.

[12] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[13] E. V. Hensbergen and M. Zhao. Dynamic policy disk caching for storage networking. Technical Report RC24123, IBM, November 2006.

[14] D. A. Holland, E. L. Angelino, G. Wald, and M. I. Seltzer. Flash caching on the storage client. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC 13)*. USENIX Association, 2013.

[15] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with lazy adaptive replacement. In *Proceedings of the 29th IEEE Symposium on Mass Storage Systems and Technologies (MSST 13)*, pages 1–10. IEEE, 2013.

[16] R. C. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.

[17] R. Koller, L. Marmol, R. Ranganswami, S. Sundararaman, N. Talagala, and M. Zhao. Write policies for host-side flash caches. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST 13)*, 2013.

[18] R. Koller, A. J. Mashtizadeh, and R. Rangaswami. Centaur: Host-side SSD caching for storage performance control. In *Proceedings of the 2015 IEEE International Conference on Autonomic Computing (ICAC 15), Grenoble, France, July 7-10, 2015*, pages 51–60, 2015.

[19] R. Koller, A. Verma, and R. Rangaswami. Generalized ERSS tree model: Revisiting working sets. *Performance Evaluation*, 67(11):1139–1154, Nov. 2010.

[20] M. Krueger, R. Haagens, C. Sapuntzakis, and M. Bakke. Small computer systems interface protocol over the internet (iSCSI): Requirements and design considerations. Internet RFC 3347, July 2002.

[21] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC 14)*, pages 501–512. USENIX Association, 2014.

[22] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou. S-CAVE: Effective SSD caching to improve virtual machine storage performance. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT 13)*, pages 103–112. IEEE Press, 2013.

[23] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies (FAST 03)*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[24] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu. vCacheShare: Automated server flash cache space management in a virtualization environment. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC 14)*, pages 133–144, Philadelphia, PA, June 2014. USENIX Association.

[25] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the USENIX Annual Technical Conference (ATC 05)*, pages 391–394. USENIX, 2005.

[26] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):134–154, 1988.

[27] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (ICPADS 01)*, pages 116–127, 2001.

[28] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 09)*, pages 121–132, New York, NY, USA, 2009. ACM.

[29] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, Feb. 2015. USENIX Association.

[30] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 335–349, Broomfield, CO, Oct. 2014. USENIX Association.

[31] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI 07)*, pages 17–17, Berkeley, CA, USA, 2007. USENIX Association.

[32] J. Xu and J. Fortes. A multi-objective approach to virtual machine management in datacenters. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC 11)*, pages 225–234, New York, NY, USA, 2011. ACM.

[33] J. Yang, N. Plasson, G. Gillis, and N. Talagala. HEC: improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR 13)*, page 10. ACM, 2013.

[34] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 04)*, pages 177–188, New York, NY, USA, 2004. ACM.