A Trace-driven Analysis of Solid-State Caching in Cloud Computing Systems

Dulcardo Arteaga, Ming Zhao  
{darte003, ming}@cs.fiu.edu  
Florida International University

Pim Van Riezen, Lennard Zwart  
{pi, lennard}@cloudvps.com  
Cloud VPS

Abstract

Distributed storage systems (e.g., SAN, iSCSI) are commonly used in the emerging cloud computing systems to provide virtual machine (VM) storage, for efficient storage utilization and fast VM migration. However, as the size of cloud systems and the number of hosted VMs rapidly grow, the scalability of shared VM storage systems becomes a serious issue. Client-side solid-state-based caching has the potential to improve the performance of cloud VM storage by employing solid-state drives (SSDs) available on the client-side of the storage system to exploit the locality inherent in VM IOs. However, there are several key questions to effectively use SSD caches in clouds. First, because of the limited capacity and high cost of SSDs, it is important to determine the proper size and configuration of the caches. Second, because of the diversity of cloud workloads, it is also critical to properly allocate the limited SSD cache capacity among concurrently hosted VMs. This paper provides answers to these questions by studying hundreds of GBs of and months long block IO traces collected from real-world private (FIU) and public (CloudVPS) cloud systems. The overall analysis shows that cloud computing systems are good target for SSD caching, while write-back caching is important to cache performance. The results from CloudVPS and FIU show an average of 74% and 78% hit ratio and 104% and 68% speedup in IO latency respectively. This analysis has also studied the use of dynamic cache allocation based on working-set size (WSS) analysis. The results show that the WSSes of the traces can be accurately predicted online (with less than 2% relative prediction error for 90 percentile) and the cache allocation can be adjusted dynamically to improve the performance of the competing workloads.

1 Introduction

Distributed storage systems such as SAN [15] and IP-SAN (e.g., iSCSI [11], NBD [6]) are commonly used in the emerging cloud computing systems to store virtual machine (VM) images for a set of VM hosts (e.g., [1, 7]). Such a shared storage system allows efficient storage utilization by consolidating separate VM storage resources into a single shared pool. It also enables fast, live VM migration which need to transfer only VMs’ in-memory state across hosts during the migrations. However, as the size of cloud systems and the number of hosted VMs rapidly grow, the scalability of the shared VM storage becomes a serious issue. In a production cloud, a single host can host up to 100 VMs, while a cluster of hosts can have up to 1000 VMs. As a result, the VM storage system can easily become the bottleneck while a VM can get its desired performance even it is provisioned with the necessary CPUs and memory.

Client-side storage caching can improve the performance of cloud storage by harnessing the storage available on the client-side of the storage system, the VM hosts, and the locality inherent in VM IOs. With local cache storage, each VM can get faster IOs if they are preformed locally on the cache device, whereas the load on the storage system can also be reduced. With the emergence of SSDs, the benefit of client-side caching becomes even more significant as the speed of an SSD cache can substantially outperform the HDD-based storage server. Various solutions [3, 8, 5, 4] have been proposed to implement SSD-based client-side caching. In particular, dm-cache has been deployed in the production systems of CloudVPS, a public cloud service provider since October, 2012.

However, there are several key questions that need to be answered in order to make effective use of SSD caches in cloud storage systems. First, how to size the SSD caches? Given the capacity and cost constraints of SSDs, there need to be enough locality in VM IOs in
order to make SSD-based caching cost effective. Otherwise, cloud may not be a good target for SSD caching. Second, how to configure the cache policies? Policies on cache replacement and prefetching are important to the cache performance; the policy of how to handle writes in the cache has implications on not only write performance but also data durability. Third, how to share the shared cache capacity among concurrent VMs? A single SSD cache may be shared by up to a hundred VMs running on the same host, while the VM workloads vary in terms of locality, read/write ratio, sequentiality, and burstiness. The lack of understanding of all these characteristics may lead to a cache sharing policy that unfairly treat the competing workloads and underutilize the cache resources.

This paper provides answers to the above questions by studying hundreds of GBs of block IO traces collected from the Florida International University private cloud and the Cloud VPS public cloud systems. The FIU trace contains 233GB of block IO traces collected from production servers (web, moodle, and file servers) at School of Computing Sciences during several weeks. The Cloud VPS trace contains 14GB of block IO traces from production systems of Infrastructure-as-a-Service during couple of days.

The analysis first summarizes the key characteristics including read/write ratio, intensity/burstiness, sequentiality, of all the collected traces. It then studies the impact of cache sizes and policies (write handling, prefetching) on cache hit rate and application performance for the different workloads through both simulations and real experiments. After determining that cloud can be a good target for block-level caching, this analysis continues to study the performance of caching sharing policies, including no-partitioning, static allocation, and dynamic allocation, through both simulations and experiments.

The results show that cloud computing systems are a good target for SSD caching, while write-back caching is important to cache performance. Traces collected at CloudVPS and FIU show an average of 74% and 78% hit rates respectively, using the write-back policy, whereas the use of write-through with write-allocate does not improve the hit rates much. The high hit rates of write-back caching translate to 104% and 64% speedup in IO latency for the CloudVPS and FIU workloads, respectively. This analysis has also studied the use of dynamic cache allocation based on working-set size (WSS) analysis. The results show that the WSSes of the traces can be accurately predicted online (with less than 2% relative prediction error for 90 percentile) and the cache allocation can be adjusted dynamically to improve the performance of the competing workloads.

Overall the work described in this paper has made the following contributions: 1) It has collected and analyzed a substantial amount of traces from production private and public cloud systems; 2) To the best of our knowledge, it provides the first comprehensive analysis of the effectiveness of SSD caching and the impacts of different caching policies and cache allocation approaches based on real-world traces; 2) It presents the implementation and evaluation of these cache management techniques based on dm-cache, an SSD caching solution used by production cloud systems.

The rest of this work is organized as follow: Section 2 describes the background and related work; Section 3 presents the methodology of the trace-driven analysis; Section 4 describes the cacheability analysis; Section 5 presents the working-set size analysis; Section 6 studies proportional cache allocation; Section 7 concludes the paper and discusses the future work.

2 Background and Related Work

Client-side disk caching can improve the performance of cloud storage by harnessing the storage available on the client-side of the system, the VM hosts, and the locality inherent in VM I/Os. With the emergence of SSDs, the benefit of client-side caching is potentially more significant as the speed of an SSD cache substantially outperforms the storage server. This potential has motivated several SSD caching solutions (e.g., dm-cache [3], io-Cache [5], Mercury [8]). In this paper, we focus on dm-cache based SSD caching, which has been successfully deployed in production cloud computing systems and motivated the designs of other SSD caching solutions (e.g., FlashCache [4]).

There is also related work on improving various aspects of SSD caching. For example, FlashTier proposed cache-specific SSD management to enhance the performance of an SSD device dedicated for caching uses [14]; Koller et al. proposed several techniques to improve the durability of write-cached data for SSD caches [10]. However, without a thorough understanding of the effectiveness of SSD caching based on real-world IO workloads, it is unclear how much impact these proposed techniques will really generate. This paper complements the existing work by offering such a much needed study based on production cloud traces.

dm-cache [3, 9] is a generic block-level disk caching utility for distributed storage systems. It is created upon block-level storage virtualization by interposing a virtual block device between the storage client and server, and can be transparently deployed on the VM hosts to provide client-side caching. To support the use in cloud computing systems, dm-cache is augmented to support shared block-level caching which allows multiple co-hosted VMs to use the same cache device without causing data corruption. It can maximize the cache utilization as any VM can use any part of the available cache.
Figure 1: Architecture of Shared SSD Cache for VMs in a Cloud System

Figure 1 illustrates the architecture of *dm-cache* based shared SSD cache. In this example we have multiple VMs each with its own virtual disk stored directly on the logical volumes (LVs) remotely accessed through iSCSI/SAN. The local storage device (/dev/sdc) available on the VM host is used to provide block-level caching for the VM images. In order for the VMs to share the cache device, we create a virtual cache for each VM’s virtual disk, which is only another level of mapping between the virtual disk and cache device. All virtual caches are in the end mapped to the same physical cache device. The per-VM virtual cache helps the shared-cache differentiate the IOs issued by different VMs. It is also necessary to handle VM migration. When a VM migrates across hosts, its data stored in the cache needs to be flushed. Tagged blocks allow the shared-cache to flush only the data of the migrating VM but not the entire cache.

## 3 Methodology

To support this SSD caching study, real-world block IO traces were collected from production cloud systems using using two different mechanisms: *blktrace*, a Linux block-layer IO tracing mechanism which provides detailed information about request queue operations, and, *dtrace* a Solaris dynamic tracing framework. Using these two mechanisms we were able to collect block IO traces with insignificant overhead to the systems.

The first group of traces were collected from a private cloud at Florida International University (FIU). Several production servers (web, moodle, and file servers) at the School of Computing and Information Sciences (SCIS) were traced for weeks. Table 1 summarizes the information about these FIU traces. The FIU *webserver* hosts the SCIS School’s website which provides information on the School’s news, programs, courses, and people; the FIU *moodle* server hosts the Moodle online learning system to provide course materials for many SCIS undergraduate and graduate courses; *Bear* and *Buffalo* are the file system servers for strong the user data of faculty/graduate students and undergraduate students, respectively.

The second group of traces were collected from the production system of a public Infrastructure-as-a-Service cloud computing provider (*CloudVPS* [2]). A random set of VMs were selected to be traced for a couple of days. Table 2 summarizes the information about these CloudVPS traces collected at CloudVPS.

Analysis was conducted on these traces by replaying them on a separate testbed using both simulations and real experiments. For simulations, a user-level cache simulator, *dm-cache-sim*, was created to analyze cache hit rates given different cache configurations and policies. It is able to model the cache management of dm-cache and use block-level traces to drive the simulations. Because it does not simulate the time behavior of dm-cache, it is faster than real experiments and good for exploring the impact of different cache parameters on hit rate.

To obtain IO performance metrics such as latency and throughput of SSD caching, the traces were also replayed on a real iSCSI-based storage system equipped with dm-cache based SSD caching. The node with a computer cluster are set up as the storage client and server. Each node has has two six-core 2.4GHz Opteron CPUs, 32GB of RAM, and one 500GB 7.2K RPM SAS disk. The client runs Xen-based VMs which has 1 vcpu, 1024GB of memory, run 2.6.32-5-xen-amd64 in Ubuntu 8.0.4. The traces were replayed from inside the VMs using *btreplay* with accelerated speed to shorten the experiment duration. Dm-cache was used on the client to provide caching using an SSD with 120GB capacity.

<table>
<thead>
<tr>
<th>Server</th>
<th>Time (days)</th>
<th>Trace type</th>
<th>Size (GB)</th>
<th>Number of IOs</th>
</tr>
</thead>
<tbody>
<tr>
<td>webserver</td>
<td>79</td>
<td>blktrace</td>
<td>18.2</td>
<td>142,078,746</td>
</tr>
<tr>
<td>moodle</td>
<td>54</td>
<td>blktrace</td>
<td>150</td>
<td>1,090,865,215</td>
</tr>
<tr>
<td>bear</td>
<td>64</td>
<td>dtrace</td>
<td>62</td>
<td>6,152,561,191</td>
</tr>
<tr>
<td>buffalo</td>
<td>16</td>
<td>dtrace</td>
<td>3.4</td>
<td>2,068,564,485</td>
</tr>
</tbody>
</table>

Table 1: FIU Servers Traces

<table>
<thead>
<tr>
<th>Host Name</th>
<th>Time (days)</th>
<th>Number of VMs</th>
<th>Size (GB)</th>
<th>Number of IOs</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhine</td>
<td>1</td>
<td>90</td>
<td>7</td>
<td>99,142,316</td>
</tr>
<tr>
<td>seine</td>
<td>2</td>
<td>18</td>
<td>3.3</td>
<td>41,946,745</td>
</tr>
<tr>
<td>heinen</td>
<td>3</td>
<td>62</td>
<td>4.5</td>
<td>25,270,874</td>
</tr>
</tbody>
</table>

Table 2: CloudVPS VM Traces
4 Cacheability Analysis

This section analyzes the patterns and locality of the traces and the SSD cache performance given different cache configurations and policies.

4.1 Trace Patterns

The first subsection summarizes the general patterns of the collected traces, including the overall IO intensity and the ratio between reads and writes. Figure 2 shows the total number of IOs and the numbers of reads and writes for the FIU webserver trace per week for a total of nearly three months. There is a spike in IO intensity during the second week of November when SCIS held its 25th anniversary which generated more visits to the website. The IO intensity reduces during the winter break and increases again at the beginning of the spring term. Overall across the entire trace there are 32% of reads and 68% of writes. The IOs are dominated by writes because the website’s workload is mostly browsing of a set of commonly viewed web pages which can be largely cached by the webserver’s memory.

Figure 3 shows the patterns of the FIU moodle trace per week for a total of nearly two months. Although this trace is collected also from a website, its patterns are quite different from the FIU webserver. First, the overall intensity is an order of magnitude higher than the FIU webserver trace, because the Moodle website services contents such as course slides and assignments which in total are much larger than the data served by the FIU webserver. Second, because the overall data set is much larger, it becomes less cacheable by the memory and the IO workload is thus dominated by reads (82% overall), in contrast to the FIU webserver which is dominated by writes.

Figure 4 shows the IO patterns for the FIU buffalo trace per day for almost a complete month, this is a file server that store student’s any kind of data, due to its nature the IO workload is different than the previous traces. We can observe that both IO operations reads and writes present a similar percentage 53% and 46% respectively, making the trace as evenly distributed. Another file server traced for almost three months is FIU bear, that also provides storage for students and faculty members, the pattern in this trace is different than the other file server, here the read operations are considerable dominant with 80% reads and 20% writes.

Figure 6 is based on the traces of a subset of VMs hosted by CloudVPS, where every data point corresponds to the one-day trace of one of the VMs. These VMs exhibit diverse IO characteristics in terms of intensity and read/write ratio. As CloudVPS is an Infrastructure-as-a-Service provider, the guest systems of the VMs are owned by the users and we can only observe their behaviors from outside of the VMs.

4.2 Write Caching Policies

4.2.1 Cache Hit Rate

This subsection studies the cache hit rate given different write caching policies. The choice of a write caching policy has implications on both performance and data durability. If there is enough locality in writes, a policy that retains writes in cache can speed up the IOs including both reads and writes that hit the cached blocks. Otherwise, the limited cache capacity can be wasted. Moreover, if writes are delayed in cache without immediately submitted to the storage server, the writes may be lost if the SSD fails. As shown in our traces, a workload (e.g., FIU webserver) can be dominated by writes. This
observation is also confirmed by related work [12, 13], which can be attributed to the fact that modern computer systems are getting larger memories which can cache a substantial amount of reads in memory but do not buffer writes for too long due to durability concerns.

We consider different write caching policies, including write-through without write-allocate, and write-through with write-allocate, and write-back. When write-allocate is used together with write-through (denoted as write-allocate for simplicity), the write is stored in cache and at the same time submitted to the storage server. When write-allocate is not used with write-through (denoted as write-allocate for simplicity), the write is only submitted to the storage server without being cached. The use of write-allocate allows the writes to be cached without losing durability. It can increase the hit rate of reads because they can be directly serviced by the cache without contacting the server, but it does not help the hit rate of writes because writes have to always be sent to the server.

With the write-back policy, the write is stored in cache and submitted to the storage server later (when it is replaced by another data block or flushed periodically). In order to get the maximum amount of hits possible in the analysis, we generated the miss-rate curves (MRCs) for the traces using our dm-cache-sim simulator by varying the cache size until the capacity is able to fit the all possible hits.

Figure 7 shows the number of cache hits for the FIU webserver using different write caching policies, where we can clearly appreciate that write-back policy works substantially better than the other two policies. There is not much difference in terms of number of hits between the write-through and write-allocate policies, which indicates that the cached writes are not reused by the following read much. Figure 8 shows the different hits for the FIU moodle trace, which also shows higher hits for write-back. But the difference with write-back and the other policies is not as significant as in the webserver case, because the majority of this workload is read operations and the number of writes hits is smaller that in the FIU webserver trace.

Figure 11 shows the cache hit rate for the CloudVPS traces, where each trace is replayed separately. In general we can see that write-back still achieves the highest number of hits.

Table 3 summarizes the overall hit rates for all the traces. Again, it strengthens the observations that write-back caching can substantially improve the cache performance, although it is considered less safe than the other policies. Write-through with write-allocate does not improve the hit rate although it makes the writes available in the cache for future reads to reuse. These results underlines the importance of having a write-back cache. More research (such as the related work [10]) is necessary to improve the durability of write-back caching while benefiting its performance improvement.

### 4.2.2 IO Latency

With the understanding of how different write caching policies affect hit rate, we further study their impact on actual IO performance by replaying the traces in a real environment using dm-cache and real SSD-based cache (as described in Section 3). During the replay, we collected the IO latency from inside of the VM using iostat, which measures the average latency of IO request every 5 seconds. Because real replay is time-consuming (e.g.,
it takes 14 hours to replay a week of the FIU webserver trace, we only selected a subset of the available traces to evaluate here.

Figure 12 shows the average latency of every 5 seconds for replay the first week of the FIU webserver trace using write-through (without write-allocate) and write-back policies compared to the case where SSD caching is disabled (no-cache). The results confirm the previous analysis: using write-back policy achieves much lower IO latency than the other policies. The interesting new observation is that using write-through actually hurts the performance as compared to no caching, because there is not enough locality in reads to be exploited and compensate the overhead of read-only caching for such a write-intensive workload.

Figure 13 shows the IO latency when replaying one of the CloudVPS VM’s trace (vps26485 in Figure 11) in real, where again write-back is slightly better than write-through. In this trace, the use of write-through caching outperforms the no-cache case because this trace is dominated by reads and there is enough read locality to be exploited.

Table 4 shows a summary of average latency for the entire traces using the different policies. The results confirm that the write-back policy provides lower latency than the other policies.

4.3 Sequentiality

Last in this section, we analyze the sequentiality of the traces. Although sequentiality is not as important to the performance of SSDs as to HDDs, it can have a significant impact to the performance of an SSD cache. The sequentiality of reads and writes affects the effectiveness of prefetching for caching reads when using write-through and writes when using write-back. Even without write caching, the sequentiality of writes also affects the effectiveness of write aggregation, i.e., combining several consecutive writes in a single request to the storage server. Good sequentiality in the workload will lead to better improvement on performance; otherwise, it can actually hurt the performance by wasting cache space and delaying the IOs. Although these optimizations are not currently implemented in dm-cache, we use the simulation analysis here to understand their effectiveness based on the collected traces.

In this sequentiality analysis, we count the number of consecutive reads or writes in the unit of 4KB block, the common size of block IOs issued by file systems. Figure 14 shows the histogram of different sizes of sequentiality for the FIU webserver trace normalized according to the total of each operation (reads with respect to the total reads operations and write with respect to the total reads operations). We find that more than 50% of write operations are at least 128KB in size, and more than 50% are at least 512KB in size, both of which show good sequentiality. The same analysis was perform on FIU moodle traces (Figure 15) which shows similar sequentiality as the FIU webserver. The results confirm the potential benefits from prefetching and write aggregation, which will be further studied in our future work.
5 Working Set Size Analysis

The analysis in previous sections has demonstrated that real-world IO workloads have substantial locality while an SSD cache can exploit it to achieve significant performance improvement for the IOs. However, the limited size of an SSD cache capacity must be properly allocated across the many co-hosted VMs because the size of all the VM images is often orders of magnitudes larger than the SSD’s size. Based on the working-set model, the key is to find out the working-set size (WSS) of the competing workloads and allocate the cache capacity based on the WSSes. In this section, we conduct the WSS analysis on the traces in order to determine, first, whether it is possible to accurately estimate a workload’s WSS online, and second, whether it is feasible to fit the WSSes of up to a hundred co-hosted VMs using an SSD that is of the common commodity size.

To calculate the WSS, we need to first set a window size, \( \delta \), and then count the number of unique block addresses made by the workload during \( \delta \). Traditionally, when estimating the WSS for a workload such as memory accesses, \( \delta \) is often specified by the number of accesses, because the recently accessed data is more like to accessed again. However, a storage IO workload can be highly spiky—a VM may often become idle for a considerable amount of time without issuing much IO activity, which is a characteristic quite different from the access patterns typically seen by a processor or memory cache. If the \( \delta \) is determined by IO accesses and does not capture when they are issued at all, the cache share allocated to the VM based on the \( \delta \) may be often wasted while it can be used by another competing VM. Therefore, it may be more appropriate to specify the \( \delta \) using the elapsed time. In this WSS analysis, we considered both of these two approaches.

The WSS analysis for the entire three month FIU webserver trace is presented in Figure 16. Figure 16a and Figure 16b show the WSSes calculated using a \( \delta \) specified based on the number of accesses, one with a \( \delta \) of 1024 and the other a \( \delta \) of 4096. In both cases we can appreciate certain stability of WSS throughout the entire three month of the trace and an evident change of WSS occurred in the middle of the trace.

Figure 16c and Figure 16d show the WSS calculated for FIU webserver using a \( \delta \) based on the elapsed time, 10 minutes and 30 minutes, respectively. Compared to the WSS analyzed using access-based \( \delta \), we can make several interesting observations. First, there is also a good amount of stability of the WSS whereas the two-phase behavior of the workload is also evident. Second, when examining WSS based on time, it is clear that the workload is highly spiky. On one hand, when the IO activity is low, it may take a while to fill up an access-based \( \delta \), which confirms that if cache is allocated based on the latter then the share may not be well utilized. On the other hand, when the IO activity is in
the middle of a high spike, it may fill up the delta rather quickly, which means that if the cache is allocated using access-based delta the share may soon become unnecessary.

Figure 18 shows the WSS analysis for the FIU moodle trace. Compared to the above results form the FIU webserver traces, the WSSes are much larger and the patterns are also more spiky throughout the course of the two months of the trace.

Figures 17a and 17b show the WSS calculated for one CloudVPS VM sample using the access-based delta of 1024 and 16384, which demonstrate the transition among several different WSSes. Figure 17c and Figure 17d show the WSS calculated using time-based delta of 10 minutes and 30 minutes. Both of them show similar patterns of the WSSes, which exhibit a relatively constant WSS of below 20K blocks most of the time and a repetitive pattern of peaks with a WSS of 60K. These patterns are only observable when the analysis is done on the time scale, which may lead to more efficient cache allocation than the access-based analysis.

With the understanding of how to perform WSS analysis, we continue to investigate whether it is feasible to accurately predict WSS online. Based on WSS historic calculations observed online, we try to predict the WSS for the next window using either exponential smoothing or double exponential smoothing. In Tables 5 and 6, we evaluate how accurate our prediction models are. The level of accuracy will determine how effective we predict the future cache demand of a particular IO workload based on online WSS analysis. The tables show the minimum, average, and 90th percentile, summary grouped by delta, of the relative error for every predicted WSS. We can observe that the 90th percentile error is below 2%, which indicate good accuracy for using these prediction techniques.

6 Proportional Cache Allocation

With the understanding of VMs’ cache needs, we further investigate how to enforce the cache allocation among the possibly many VMs sharing the same SSD cache. Cache allocation can be enforced statically for competing VMs’ workloads; however, as shown in the previous section, due the dynamism inherent in the workloads, the WSSes change over time, so the allocation may also need to be updated dynamically.

Different from traditional processor and memory cache management, the change of cache allocation should not require extensive data copying and flushing, which are much slower and more expensive on a disk cache compared to a processor or memory cache. Hence, we propose replacement-time enforcement of cache al-
location and implemented it in \textit{dm-cache}. This approach does not physically partition the cache across VMs. Instead, it enforces logical partitioning at replacement time: if the cache is not full, a VM can use empty slots in the cache beyond its allocated share; when it is full, a VM that has not used up its share takes its space back by replacing a block from another VM that has borrowed its share.

In order to realize dynamic cache allocation, we enhanced \textit{dm-cache} to allow it to analyze the running workloads online, without interference with the any IO request. This \textit{wss-analyzer} keeps track the information (block address and time) of recently received IOs, and calculates WSSes for each workload online. Based on the observed WSSes, it also predict the WSSes of the workloads for the next window. Using the predicted

WSSes of all the workloads, \textit{dm-cache} reallocates the cache among the competing VMs and enforce the allocation at replacement time. When the total size of the estimated WSSes is greater than the cache capacity, the cache is allocated proportionally to the WSSes in order to achieve proportional slow down for the competing workloads. When the total size of the WSSes is less than the cache capacity, the spare capacity is also allocated evenly to the competing workloads.

In the rest of this section, we study the performance of different cache allocation approaches. A set of experiments were conducted using two of the most IO intensive traces from \textit{CloudVPS}. We created an environment with two co-hosted VMs that at the time of creation each gets 50\% of the cache. We compare the performance of the workloads using three different cache management approaches, 1) with static cache allocation which gives each VM 50\% of the cache throughout the experiment, 2) with dynamic cache allocation which allocates the cache based on the WSSes predicted online (using double exponential smoothing based prediction and a delta of 10 minutes for online WSS analysis).

Table 7 shows the average latency of real trace replay using different allocation policies, using the \textit{FIU web-server} and \textit{FIU moodle} traces. We observe that dynamic allocation provides half of the latency for the \textit{FIU moodle} trace compared to the static allocation.
7 Conclusions and Future Work

Caching is one of the most widely used techniques for improving the performance of data accesses in any computer systems. Its effectiveness is largely determined by the available locality in the workload that can be exploited by the cache, and the speedup that can obtained by serving it from the cache vs. from the next layer in the storage hierarchy. The emergence of SSDs has motivated the consideration of client-side caching in a distributed storage system because the speed of SSDs is substantially faster than the network and the HDDs on the storage server. It also comes in time to address the serious scalability issues that cloud computing systems are facing now as the number and size of VMs quickly increase on a shared storage system. However, the existing literature does not provide any answers to the key questions on whether there is good locality in typical cloud IO workloads and whether SSDs can effectively utilize the locality to achieve good performance speedup.

To the best of our knowledge, this paper is the first to provide answers to the above questions through a trace-driven analysis. We have collected large amounts of traces from both production public and private clouds. To effective analyze these traces and understand the impacts of various key caching policies and cache allocation approaches, we have developed a user-space cache simulator. Finally, we have also implemented these techniques in real based on the dm-cache framework. The results confirm that cloud computing systems are good target for SSD caching, but the locality in writes must be effectively utilized. The results also show that the working-set sizes of real cloud workloads can be accurately predicted online and used to guide efficient dynamical cache allocation.

In our ongoing work, we are collecting more traces from the product public cloud system. The overhead of our currently used tracing tool, blktrace, can be an issue when it simultaneously traces a large number of VMs, because it collects comprehensive block IO information. To reduce this overhead, we have implemented tracing capability within dm-cache, which collects much less data and is experimentally proven to be much more efficient. With this updated dm-cache deployed in CloudVPS, we expect to start the tracing of a large number of VMs in the near future.

The results of these papers have raised many more interesting research questions. For example, how to utilize the sequentiality in the workloads to improve cache performance? How to combine access-based and time-based WSS analysis in order to better estimate the workload locality? How to improve the durability of write-cached data to harness the good write locality observed in the workloads without risking the data? These questions will be addressed in our future work.

References


