Client-side Flash Caching for Cloud Systems

Dulcardo Arteaga
Florida International University
darte003@cs.fiu.edu

Ming Zhao
Florida International University
ming@cs.fiu.edu

ABSTRACT

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 flash-based caching has the potential to improve the performance of cloud VM storage by employing flash storage available on the client-side of the storage system to exploit the locality inherent in VM IOs. However, because of the limited capacity and durability of flash storage, it is important to determine the proper size and configuration of the flash caches used in cloud systems. This paper provides answers to the key design questions of cloud flash caching based on dm-cache, a block-level caching solution customized for cloud environments, and a large amount of long-term traces collected from real-world public and private clouds. The study first validates that cloud workloads have good cacheability and dm-cache-based flash caching incurs low overhead with respect to commodity flash devices. It further reveals that write-back caching substantially outperforms write-through caching in typical cloud environments due to the reduction of server IO load. It also shows that there is a tradeoff on making a flash cache persistent across client restarts which saves hours of cache warm-up time but incurs considerable overhead from committing every metadata update persistently. Finally, to reduce the data loss risk from using write-back caching, the paper proposes a new cache-optimized RAID technique, which minimizes the RAID overhead by introducing redundancy of cache dirty data only, and shows to be significantly faster than traditional RAID and write-through caching.

1. INTRODUCTION

Network storage systems such as SAN [29] and IP-SAN (e.g., iSCSI [24], NBD [12]) are commonly used in the emerging cloud computing systems to store virtual machine (VM) images for a set of VM hosts (e.g., [1, 13]). 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 migrations 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 shared storage becomes a serious issue. In a production cloud, a single host can run hundreds of VMs, while a cluster of hosts can have thousands of VMs sharing the same storage server. Consequently, the VM storage system can become the bottleneck while a VM cannot get its desired performance even if it is provisioned with the necessary CPUs and memory.

Client-side persistent-storage-based caching can improve the performance of cloud VM storage by harnessing the storage available on the client-side of the storage system, the VM hosts, and the locality inherent in VM IOs. Each VM can get faster IOs if they are performed on the local cache, instead of from the likely highly loaded remote storage. With the emergence of flash storage, the benefit of client-side caching becomes even more significant as the speed of a flash cache can substantially outperform the typically hard-disk-based storage server. Various solutions [5, 17, 9, 7] have been proposed to implement flash-based client-side caching.

There are several key questions that need to be answered in order to make effective use of flash caches in cloud storage systems. First, *how to size the flash caches?* Given the capacity and cost constraints of flash devices, there needs to be enough locality in VM IOs in order to make flash caching cost effective. Otherwise, cloud may not be a good target for flash caching. Second, *how to choose the write caching polices?* Although the non-volatile nature of flash storage allows writes to be served directly from the cache, how to synchronize the cache with the server has implications on both IO performance and data durability. Third, *is it necessary to make a flash cache persistent across client restarts and crashes?* A persistent cache requires both data and metadata to be persistently stored on the cache, which introduces additional writes that are detrimental to both IO performance and flash endurance. Finally, *how to improve the reliability of a flash cache so as to tolerate device-level failures?* If a flash cache retains locally modified data, it is critical that the cache can recover from flash device failures, but the employed fault-tolerance mechanism should not negate the performance benefit of write-back caching.

This paper studies client-side flash caching in cloud systems by providing answers to the above questions based on *dm-cache* [5], a block-level cache solution that provides transparent flash caching on cloud VM hosts and supports concurrent, dynamic VMs to efficiently share a cache. It has been adopted by cloud service providers for production use [3]. To facilitate this study, we have collected a substantial amount of block IO traces from a private cloud at Florida International University (FIU) and a public cloud from CloudVPS [3]. The FIU trace contains nearly one year of block IO traces collected from several production servers (Web serve, Moodle server, and network file system servers). The CloudVPS traces contains block IO traces

SYSTOR '14, June 10-12, Haifa, Israel
Copyright 2014 ACM 978-1-4503-2920-0 ...$15.00
DOI 10.1145/2611354.2611372.

from hundreds of VMs on the production systems of the Infrastructure-as-a-Service (Iaas) cloud for several days.

Our study first analyzes the basic characteristics of dm-cache based flash caching and the collected cloud traces. It reveals that dm-cache introduces small latency overhead which is around 23μs when using an SATA SSD device and 9μs when using a PCIe SSD device. It also validates that cloud VMs are good targets for flash caching by comparing the working set size (WSS) of the traces to the typical size of commodity flash devices.

After confirming the feasibility of flash caching, we further study the impact of different write caching policies. Our results show that retaining writes in the cache is beneficial to performance (48% to 321% speedup compared to a policy that only invalidate cache blocks upon writes). More importantly, delaying the synchronization with the server (i.e., write-back caching) can significantly improve the performance (74% to 1289% speedup) compared to the policy that synchronizes with the server upon every write (i.e., write-through caching). This improvement can be mainly attributed to the 52% to 94% reduction of server load) by exploiting the locality of cached writes, which was not considered in the related work [20].

Our study also reveals the tradeoff of making a flash cache persistent across client restarts and crashes. To store the metadata persistently, it introduces up to 0.06ms latency overhead, but it allows the client to work with a warm cache after it recovers, which saves the time (3 to 5 hours in our traces) to warm up the cache and increases the hit rate by up to 28%. Compared to the related work [20], we provide quantitative results on the cost and benefit of making flash cache persistent and show that this tradeoff should be carefully decided based on the cloud environment such as the expected client failure rate.

With the understanding of the importance of write-back caching, we further investigate how to make it reliable and affordable. Our solution is a new cache-optimized RAID technique which selectively provides data redundancy. It recognizes the fact that cached clean data already have redundant copies on the server, and employs additional flash devices only to provide fault tolerance to cached dirty data, thereby minimizing the overhead while maximizing utilization. The results show that this cache-optimized RAID can provide fault tolerance with negligible overhead (9.1μs), and substantially improves the performance by 135% and 72% compared to using traditional RAID and write-through caching, respectively, to achieve reliability.

The rest of the paper is organized as follow: Section 2 describes the background and related work; Section 3 presents the methodology of the trace-driven analysis; Section 4 analyzes the cacheability of cloud workloads using the collected cloud traces; Section 5 discusses the overhead of dm-cache-based flash caching; Section 6 analyzes the impact of different write caching policies; Section 7 analyzes the cost and benefit associated with making a cache persistent; Section 8 presents the cache-optimized RAID technique for reliable write-back caching; and Section 9 concludes the paper.

2. BACKGROUND AND MOTIVATIONS

Client-side persistent-storage-based caching can improve the performance of a distributed storage system by harnessing the persistent storage available on the storage client to exploit the locality within its IOs, thereby accelerating data accesses to the client and reducing IO load on the server. Earlier results from dm-cache show that HDD-based client-side caching can achieve a 15-fold speedup for an iSCSI-based system with 8 clients sharing one HDD-based server [19]. However, the use of client-side disk caching was not widely adopted, which can be attributed to at least the fact that the latency of an HDD-based cache is often comparable to the network latency to the storage server. The benefit of client-side caching hence exhibits only when the server is heavily loaded [19] or accessed through a wide-area network [31, 32].

While the emergence of flash-based storage is fundamentally transforming the landscape of computer storage field, it is also changing the perception on client-side caching, because the speed of a flash-based cache can be substantially faster than an HDD-based storage server. Even as flash storage gets increasingly adopted on the storage server side, the diversity of flash devices allows the use of faster flash storage (e.g., single-level cell flash) on the client side as the cache for the slower flash storage (e.g., multiple-level cell flash, hybrid flash/HDD) on the server side.

The great potential of flash caching has motivated several related solutions. For example, Mercury [17] provides a block-level flash cache in the hypervisor of a storage client, in order to provide caching to the VMs hosted on the client over a variety of networked storage protocols. ioCache [9] supports caching in the hypervisor or in the individual VMs on a storage client using custom-built flash hardware and management software. In this paper, we base our flash caching study on *dm-cache* [5], a open-source block-level caching solution. It is created upon block-device virtualization and can be transparently deployed on VM hosts. It has been successfully adopted by production cloud systems [5] and motivated the designs of other related solutions (e.g., FlashCache [7]). Details of dm-cache are introduced in Section 3.1.

Although the potential of client-side flash caching is well recognized, it is still unclear how much performance improvement that it can achieve for typical cloud workloads and how to best design and configure the cache given the many possible choices. Recently, Holland et al. [20] studied several key design considerations, including flash-RAM integration, write-back policy, cache persistency, and cache consistency, based on simulations. In particular, they found that write-through caching is good enough because the writes to the storage server can be submitted asynchronously without slowing down the client. However, they did not consider the impact on the server's load and its resulting effect on the client's performance, which are studied in this paper using a real flash cache implementation with real traces.

Previous work from Koller *et al.* also advocated the importance of write-back caching and studied new ordered and journaled write-back policies for flash caches, in order to improve the consistency of cached dirty data [23]. This paper complements the previous work by further studying the performance impact of write-back caching to both storage client and server using real workloads and proposing a new cache-optimized RAID technique to improve the reliability of write-back-based flash caches.

RAID is a classic technique to improve the reliability of storage and has also been considered in the context of flash storage [22]. A unique challenge of flash-based RAID is synchronous aging, which means that the flash devices used in a RAID group wear out at the same time and cannot be recovered by RAID. Diff-RAID was proposed to address this challenge by intentionally distributing parity blocks unevenly across the flash devices so that the writes caused by parity updates are also unevenly distributed, allowing the devices to wear out at different speed [15]. This related work is complementary to this paper's cache-optimized RAID technique which improves storage utilization and reduces wear out by providing redundancy to only the dirty data in a cache.

The importance of flash caching has also motivated related work on exploiting cache-specific characteristics to optimize the use of flash storage. FlashTier [28] studied a new flash device interface specialized for caching, which reduces the block management overhead by unifying the block address mappings done by the cache and device and reduces the device garbage collection overhead by silently evicting clean cache blocks. HEC [30] and LARC [21] studied new cache admission policies to address the flash wear-out issue by not caching data that are infrequently used or from backup workloads. These solutions are complementary to this paper's study which focuses on the design issues internal to cache while our discoveries also have an impact on flash performance and endurance.

3. METHODOLOGY

3.1 Dm-cache Block-level Cache

Dm-cache [5, 19] provides caching at block level 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 client-side of a distributed storage system to provide caching. Our current implementation of dm-cache is based on the Linux block device virtualization framework (device mapper) and can be seamlessly employed by any Linux-based environments including VM systems that use Linux-based IO stack (e.g., Xen [16] and KVM [11]). It is an open-source solution and has been adopted by production cloud systems [4]. Therefore, we use dm-cache as a representative flash caching solution and feed it with real-world traces to carry out this paper's study.

Dm-cache supports full associativity with LRU-based replacement and various write caching polices. It employs a radix tree for fast cache lookup and an LRU list for quickly finding a replacement block. (See Section 5 for the overhead analysis.) Although alternative cache replacement algorithms (e.g., ARC [26]) are available, cache replacement is not the focus of this paper and the use of LRU in our study offers at least a baseline performance from a commonly used algorithm.

To support the use in cloud computing systems, dm-cache allows multiple co-hosted VMs to safely share the same cache device in a work-conserving manner (but with isolated datasets, data sharing at block level requires a cluster file system [14, 10] and will be considered in our future work). It also allows the cache contents to be controlled on a per-VM basis in order to support dynamic VM life cycles and migrations. For example, when a VM is terminated or migrated to a different host, its cached data can be flushed without affecting the other VMs sharing the same cache.

Figure 1 illustrates the architecture of dm-cache-based flash caching in cloud environments. In this example there are multiple VMs each with its own virtual disk stored directly on the logical volumes (LVs) (*/dev/lv-disk#*) remotely accessed through SAN or IP SAN (e.g., iSCSI [24], NBD [12]).

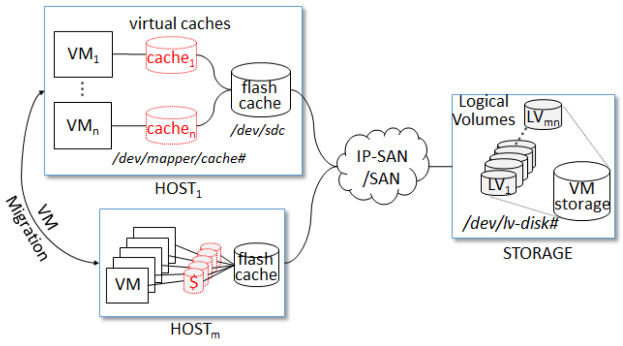


Figure 1: Architecture of Shared flash Caches for Cloud

The local storage device (*/dev/sdc*) on the client-side of this distributed storage system, the VM host, is used to provide block-level caching for the VM images. In order for the VMs to share the cache device, a virtual cache (e.g., */dev/mapper/cache1*) is created for each VM and presented to the VM as its virtual disk, while all the virtual caches are at the end mapped to the same physical cache device (a VM identifier is stored in every cached block to track the ownership of cache data). Each VM's IOs to its virtual disk are thereby handled by dm-cache and satisfied from the cache or remote LV.

3.2 Dm-cache-sim Cache Simulator

To facilitate the analysis of cache performance using long-term traces and with different cache configurations, we also created a user-level cache simulator, *dm-cache-sim*. It is able to flexibly model the cache management of dm-cache, use block-level traces to drive the simulations, and collect detailed statistics on cache usages and hit rates. Note that we do not use this simulator to gather IO latency or throughput which are always collected using dm-cache with real experiments. Because we do not attempt to simulate the time behavior of dm-cache, the simulator generally runs faster than real experiments while giving the same cache hit rate results as real experiments. It is therefore good for quickly exploring the impact of different cache configurations on cache hit rate using long-term traces.

3.3 Traces

To support this flash caching study, real-world block IO traces were collected from production cloud systems using *blktrace*, a Linux block-layer IO tracing mechanism [2], and, *dtrace* a Solaris dynamic tracing framework [6]. The statistics of the collected traces are summarized in Table 1. The first group of traces were collected from a private cloud at FIU. Several production servers (Web, Moodle, and network file system servers) were traced for months. The *Web server* hosts a departmental website; the *Moodle* server hosts the Moodle online learning system; the *Bear* and *Buffalo* servers are the file servers for storing the user data of faculty and students, respectively. These different types of servers represent services that are commonly hosted on cloud VMs. The second group of traces were collected from the production system of a public IaaS cloud provider (*CloudVPS*) [3]. A random set of 170 VMs were selected from three VM hosts and traced for up to three days.

Figure 2 shows the total number of IOs and the numbers of reads and writes for the *Webserver* trace for over 10 months. This workload is write-intensive because the reads

Server Name	Time (days)	IO Load (GB)	WSS (GB)	Write (%)
webserver	281	2,247	110	51
moodle	161	17,364	223	13
buffalo	90	39,128	638	41
bear	152	57,887	1037	22
CloudVPS (170+ VMs)	3	7 - 223	5 - 20	14 - 85

Table 1: Trace statistics

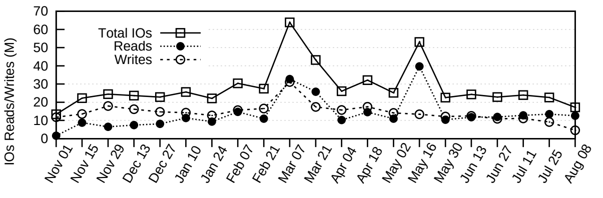


Figure 2: FIU Web server IO patterns

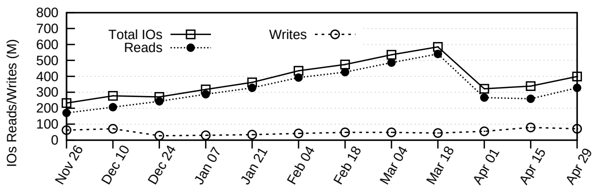


Figure 3: FIU Moodle server IO patterns

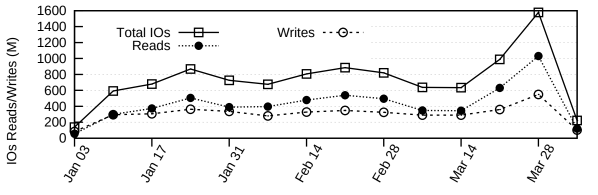


Figure 4: FIU Buffalo file server IO patterns

to the commonly visited web pages can be well captured by the webserver's memory cache, leaving the underlying storage layer a high ratio of writes. Figure 3 shows the patterns of the *Moodle* trace for nearly six months. Although this trace is collected also from a website, its patterns are quite different from the *Webserver*. First, the overall intensity is an order of magnitude higher than the *Webserver* trace, because the Moodle website services contents such as course slides and assignments which are much larger than the data served by the *Webserver*. Second, because the working set is much larger, a significant number of reads misses the memory cache and dominates the storage workload (82% overall). Figures 4 and 5 show the IO patterns for the two file server traces, which are both much more intensive than the *Webserver* and *Moodle* traces. Between these two file servers, *Bear* services a larger dataset and its storage workload has a greater percentage of reads than *Buffalo*.

The above four traces provide a good representation of cloud workloads with different levels of IO intensity and different read/write ratio. Figure 6 further illustrates commercial cloud workload patterns using a subset of the VM traces collected from *Cloud VPS*, where every group of bars corresponds to a one-day trace from one of the VMs. These VMs exhibit diverse IO characteristics in terms of intensity and read/write ratio. As Cloud VPS is an IaaS provider, the guest systems of the VMs are owned by the users and their behaviors can be only observed from outside of the VMs.

3.4 Experimental Testbed

To obtain IO performance metrics such as latency and throughput of flash caching, the collected traces were replayed on a real iSCSI-based storage system. One node from a compute cluster is set up as the storage server (iSCSI target) and the others as the clients (iSCSI initiators). Each node has two six-core 2.4GHz Opteron CPUs, 32GB of RAM, and one 500GB 7.2K RPM SAS disk, running 3.2.20 Linux-kernel in a Debian 6.0 OS. Each client node in addition is equipped with dm-cache and flash devices to provide caching. The server node runs iSCSI server to export the LVs stored on its SAS disk to the clients via a Gigabit Ethernet.

The performance of flash devices varies across different interfaces, vendors, and models. In this study, we consider two representative devices from major vendors and with different interfaces: a 120GB MLC SATA-interfaced flash device from Intel (Model: Intel C2CW120A3) and a 240GB MLC PCIe interfaced flash device from OCZ.

4. CACHEABILITY ANALYSIS

We start our study with a basic cacheability analysis by analyzing the working set size (WSS) of our collected cloud traces. Because a cache's performance for a given workload is largely determined by how well the cache can store the workload's working set, we try to understand whether the capacity of commodity flash devices is sufficient with respect to the working set size of a typical cloud workload. In the analysis below, we consider using the write-back caching policy and LRU-based cache replacement.

Table 1 lists the total WSS, i.e., the total number of unique block references, across the entire duration of every cloud trace. For the Web server, Moodle, and CloudVPS traces, their WSSes can be well stored by a typical commodity flash device. While the WSSes of the buffalo and bear file server traces are much larger, they can also be completely stored in a high-end flash device. However, in a cloud environment, the limited cache capacity has to be shared by many VMs hosted on the same client. Each VM only gets a portion of the flash cache, which is most unlikely to be sufficient for the total WSSes observed from these traces. Nonetheless, it is also unnecessary to keep the working set of an entire workload which lasts up to 9 months for the above traces, in the cache. If the cache can hold the working set observed at a smaller timer scale, say weeks, then it can still achieve good performance most of the time, except for when the workload transits across different working sets.

Figure 7 shows the WSS calculated per week (*Weekly WSS*) and the WSS calculated from the start of the trace (*Total WSS*) as they vary over time for the Web and Moodle server traces. The results show that the weekly WSS of the Web server workload is quite stable and stays below 20GB most of the time, although the entire WSS for 9 months can grow to 110GB. The weekly WSS of the Moodle server workload fluctuates over time but in average it is 100GB, which is less than half of the total WSS at the end of the 5-month trace.

Finally, to illustrate the potential cache performance with different cache sizes, we extract one-month-long segments of the Web server trace that exhibit different WSS, and show how well the cache performs in terms of hit rate in Figure 8. In general, the cache hit rate is well above 50% and exceeds 90% in many cases.

The above analysis reveals that the working sets of typical cloud workloads can be well cached in commodity flash de-

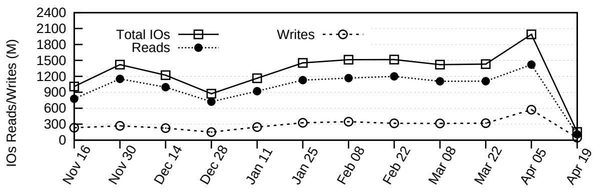


Figure 5: FIU Bear file server IO patterns

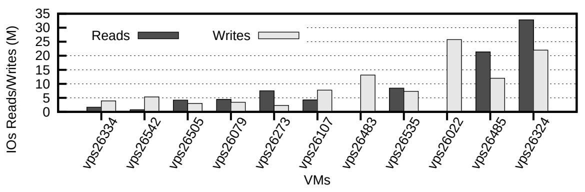


Figure 6: Cloud VPS VM IO patterns

vices, thereby verifying the feasibility of using flash devices as caches in cloud systems. Given the workloads that a VM host need to serve, the above analysis can also help determine the appropriate size of the flash cache. However, for unknown workloads, their WSSes have be estimated online, which will be studied in our future work.

5. CACHE OVERHEAD

To further investigate the feasibility of flash-based caching, we analyze its worst-case overhead using dm-cache. Cache overhead is directly associated with cache management operations including lookup, insertion, invalidation, and update. This overhead needs to be small, especially considering the fast speed of flash storage which can make any software-introduced overhead appear significant in the overall IO latency. We compare the IO latencies from when flash caching is not used to when it is used but with a cold cache, in order to evaluate the overhead of cache lookup and insertion. We compare the IO latencies from raw flash device (without using dm-cache) to the latencies from warm flash cache (using dm-cache) to evaluate the overhead of cache lookup and invalidation/update.

We use *fio* [8] to create basic read and write intensive workloads of sequential and random patterns. These workloads are issued to the raw storage device using direct IOs so that any potential optimization done by the file system and memory cache is bypassed in this performance analysis. Each workload exercises 1GB of data and is repeated four times. We consider three caching policies as defined below, which mainly differ in how they handle a write to the cache.

- *Write-invalidate (WI)*: The write invalidates the cached block and is submitted to the storage server.
- *Write-through (WT)*: The write updates both the cache and the storage server.
- *Write-back (WB)*: The write is stored in the cache immediately but is submitted to the storage server later. Before the storage server gets the write, the block is locally modified in the cache and considered *dirty*.

The write-through and write-back polices are well studied in the processor cache related literature [18], which provide a tradeoff between data coherence and performance. The write-invalidate policy sounds contrived, but it simplifies the handling of writes. There are also some variations in implementing the write-through and write-back policies. For write-through, the IO experiences a write stall if it waits for the write to complete in both the cache and back-end storage [18]. A common optimization is to allow the IO to be returned to the upper storage layer once it is stored in the cache, which allows the application to continue while the back-end storage is being updated. The dm-cache implementation adopts this optimization. For the write-back policy, a dirty block is written to the back-end storage when it is replaced. As an optimization for better data reliability, dm-cache also supports the automatic flushing of dirty blocks periodically or when the percentage of dirty blocks exceeds a threshold (similarly to the Linux *pdflush* policy) as well as manual flushing through a IOCTL signal.

Workload type	SATA-SSD (ms)	PCIe-SSD (ms)
Sequential read	0.14	0.23
Sequential write	0.07	0.03
Sequential read/write	0.08	0.16
Random read	0.18	0.23
Random write	0.07	0.04
Random read/write	0.10	0.19

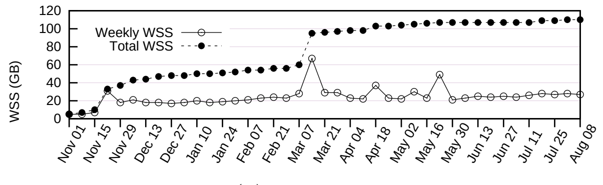
Table 2: Raw Flash Latencies

Workload	SATA-SSD (usec)		PCIe-SSD (usec)	
type	AVG	STD	AVG	STD
Seq read	96.2	28.5	240.1	10.1
Seq write	51.6	111.7	45.9	72.6
Seq read/write	94.6	132.0	171.5	148.9
Rand read	190.5	40.1	239.9	2.7
Rand write	54.5	124.1	57.1	76.5
Rand read/write	102.2	145.1	203.5	171.7

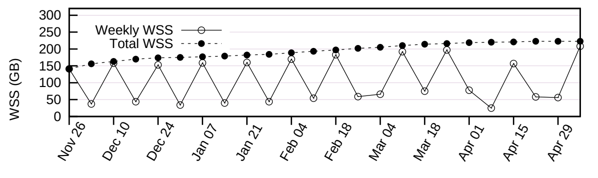
Table 3: Raw Flash Latencies

For a read workload (Figure 9), the average latency is about 0.3ms when there is no cache. When the SATA-flash cache is used the results show a small overhead of less than 0.023ms when the cache is cold, and for the PCI-e flash cache this overhead is less than 9μs. This overhead is mainly from looking up the requested block and finding the replacement block. Both operations are fast because dm-cache employs a radix tree for cache lookup which has a time complexity of $O(\log n)$ where n is the maximum number of blocks in the the cache, and it maintains a linked-list-based LRU list for replacement. Once the requested block is fetched from the server, it is immediately returned to the upper layer in the IO stack while being stored into cache. When the cache is warm, the average latency drops to 0.107ms for the SATA cache and 0.23ms for the PCI-e cache, both of which match the raw flash read latencies (<0.01ms slowdown) and are substantially faster than reading from the remote HDD.

For a sequential write workload (Figure 10), the average latency is around 0.4ms when flash cache is not used. When write-invalidate caching is used, the latencies are the same as when there is no cache since all the writes still need to be serviced by the remote server. When write-through or write-back caching is used, the latencies drop drastically, 0.06ms for the SATA flash and 0.05ms for the PCIe flash, because writes can be returned once they are stored by the flash cache. Their performance matches the raw device latencies with negligible difference (<0.008ms slowdown) because the overhead introduced by cache lookup and finding the replacement block is small. For a random write workload (Fig-



(a) Web server



(b) Moodle server

Figure 7: Working set size (WSS) variations over time

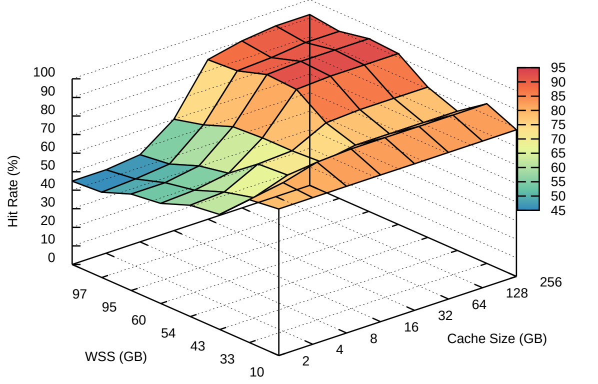


Figure 8: Cache hit rate given different cache Size and WSS

ure 10), the latency difference between write-through/write-back and no-cache/write-invalidate is even more drastic, because the HDD-based back-end performs much worse for random writes than sequential writes while the flash's performance remains almost the same.

In summary, the above results confirm that the overhead introduced by dm-cache is small and insignificant even compared to the raw latencies of flash devices, thereby further verifying the feasibility of flash-based caching with software-based cache management. In the rest of the paper, we use only the SATA SSDs for the flash caching experiments.

6. WRITE POLICY ANALYSIS

As shown in Section 3.3, a cloud workload can have a substantial amount of writes . This observation is also confirmed by related work [25, 27], 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. Therefore, the choice of a write caching policy is important and it has implications on both performance and data durability. This section studies the impact of different write cache policies, where we use dm-cache-sim to study the impact on cache hit rate using long-term traces and use dm-cache to evaluate the impact on IO performance using real experiments driven by shorter traces.

6.1 IO Latency

The various write caching policies impact IO latencies differently. If there is enough locality in writes, a policy that

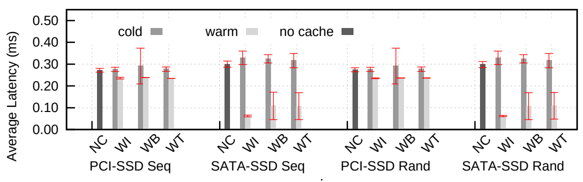


Figure 9: Dm-cache latency for read workloads

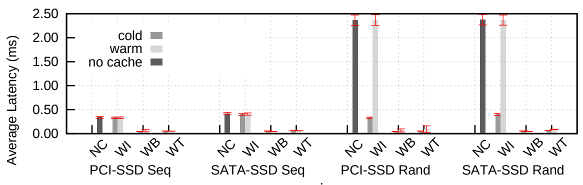


Figure 10: Dm-cache latency for write workloads

retains writes in cache (i.e., write-through or write-back) can speed up the IOs including both reads and writes that hit the cached blocks, compared to another policy that does not retain writes (i.e., write-invalidate). Otherwise, the limited cache capacity can be wasted which slows down the IOs that experience conflict misses. Comparing write-through policy to write-back policy, they exhibit the same behavior in terms of the cache hit rates but *not necessarily* the IO performance. Although writes can be returned as soon as they are stored in cache in both policies, the IOs that have to be serviced by the server experience different latencies. With the write-through policy, all the writes have to be sent to the server right away, while with the write-back policy, writes can be delayed and the following writes that hit the cached dirty data can be absorbed completely by the cache. Therefore, the server experiences a higher load under the write-through policy which in turn affects the performance of the clients. This difference can be significant in a highly consolidated environment such as a cloud system.

In order to evaluate the performance impact of different write caching policies, we consider two real workloads taken from the Web server and the Moodle server traces described in Section 3.3, which are relatively more write-intensive and read-intensive respectively. One typical day of workload was extracted from each trace and replayed using *btreplay* at a 20-fold speedup in the environment specified in Section 3.4. While the accelerated replay makes the replayed workload more intensive than the original one, it is still a reasonable setup because, *1)* on a typical cloud VM host there can be well above 20 VMs running concurrently; *2)* the original trace would have also been more intensive on its own if there was a flash caching deployed to speed up its IOs.

6.1.1 Read-intensive Trace

First, we replayed a one-day read-intensive workload extracted from the Moodle server trace, which has a 20GB total working set size and consists of 65% reads and 35% writes. Figure 11 shows the average IO latencies measured every 20 minutes during the experiment using dm-cache with different write policies. The latencies from native iSCSI without dm-cache are also provided as a reference.

Initially, it takes around 5 hours to warm up the cache, during which the different write policies offer similar performance in term of latency because the performance is dominated by reads that miss the cache and have to be serviced by the server. Note that the *No Cache* case also exhibits a warm-up phase, although it does not employ a client-side

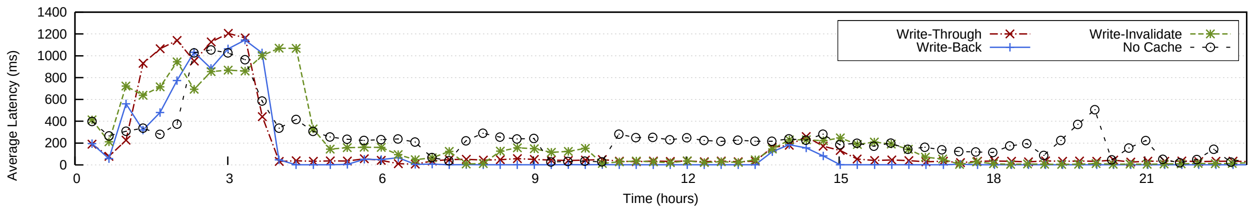
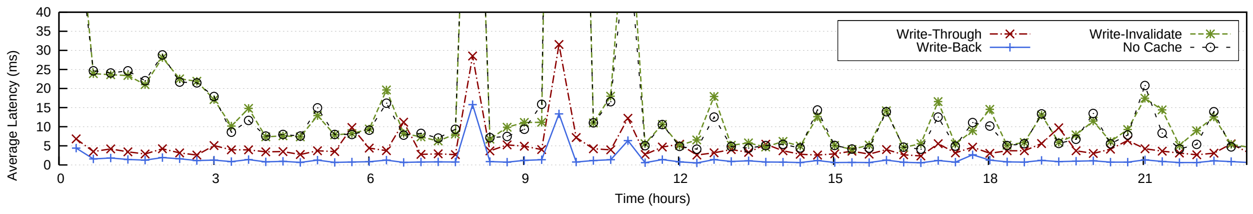
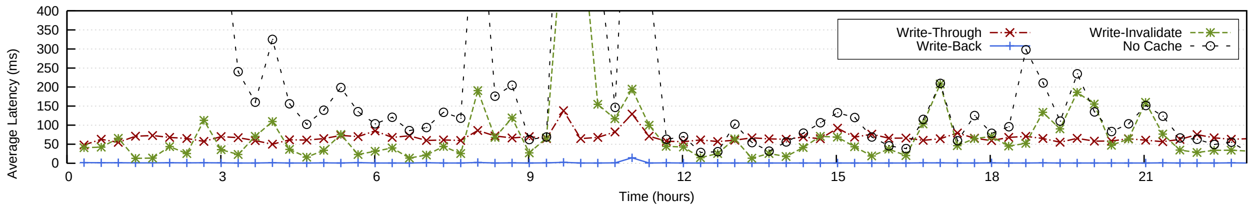


Figure 11: Performance of a read-intensive workload using different write caching policies



(a) Single Client



(b) Three Clients

Figure 12: Performance of a write-Intensive workload using different write caching policies

flash cache, because the memory caches involved in this distributed storage system also need to be warmed up initially.

During the rest of the experiment, as the flash cache is warmed up to serve the reads, the difference in the write policy shows up where the write-back policy consistently outperforms the other policies. The IO latencies from both write-back and write-through policies are lower than write-invalidate by 58ms and 23ms in average respectively, because writes can be returned immediately after they are stored in cache. However, because the write-through policy still submits all writes to server, it slows down the read misses that have to be serviced by the server, although the latencies of writes are hidden to the client. Hence, the IO latencies from the write-through policy are higher than the write-back policy by 35ms (247%) in average.

6.1.2 Write-intensive Trace

The second experiment considers a one-day write-intensive workload extracted from the Web server trace, which has a total of 10GB WSS and consists of 15% read and 85%writes. We expect to see a larger performance difference among the different write caching policies compared to the above read-intensive trace.

Figure 12a shows the IO latencies measured every 20 minutes during the trace replay. For the *No Cache* and *Write Invalidate* cases, it also takes around 5 hours to warm up the caches. In contrast, the *Write Through* and *Write Back* cases do not exhibit a warm-up phase, because most of the IOs in this write-intensive workload can be directly serviced from the flash cache. The *Write Back* and *Write Through* policies present latencies lower that the case of *Write Invalidate* by 19ms and 3ms respectively. More importantly, throughout the experiment, the *Write Back* policy's IO latencies are lower than the *Write Through* policy by 3.5ms (230%) in average, mostly because it effectively reduces the server IO load and allows the IOs that have be serviced by the server to complete faster.

In order to evaluate the different write caching policies in a highly consolidated cloud environment, we employ three storage clients that share the same storage server in the next experiment, where each client replays a different day of the write-intensive Web server trace. In this more typical scenario, we can appreciate the substantial improvement made by the write-back caching: it's IO latencies are lower than the write-through caching by 67ms (5991%) in average. In fact, the performance of write-through caching is slowed down to the same level of the much simpler write-invalidate caching, with only 17ms improvement.

6.2 Server Load

As shown in the above experimental results, the write caching policies do exhibit evident differences in their impacts to a workload's IO performance. In particular, the difference between write-through and write-back can be significant. Although both can hide the latency for writes, the difference in server IO load does impact the client-side performance substantially. As a further validation of these observations, we extend this write policy analysis to the entire traces using the dm-cache-sim simulator. However, instead of collecting hit rates, which are always the same between write-through and write-back, we collect the number of IO requests that are serviced by the server, which is the server IO load during these long-term traces.

Figures 13-16 illustrate how the server load varies over the entire duration of the four FIU traces. All of them show that the write-back policy always results in substantially lower server load than the write-through policy. The largest improvement is from the *Bear file server* trace (Figure 16),

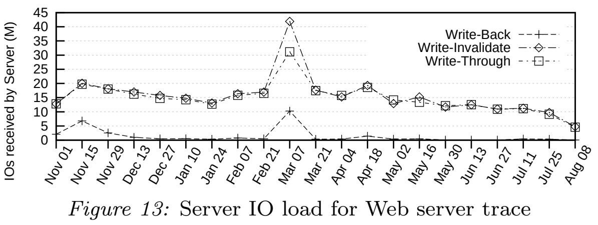


Figure 13: Server IO load for Web server trace

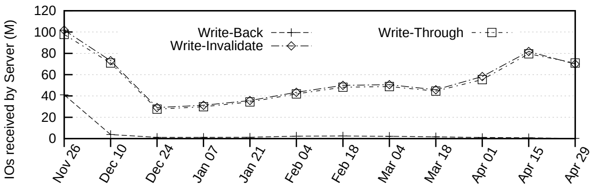


Figure 14: Server IO load for Moodle server trace

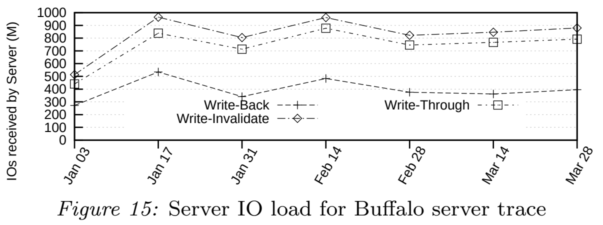


Figure 15: Server IO load for Buffalo server trace

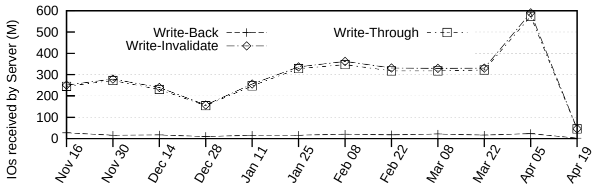


Figure 16: Server IO load for Bear server trace

which shows a 94% reduction on IO load. The smallest improvement is from the *Buffalo file server* trace (Figure 15), which shows a 52% reduction on IO load.

Figure 17 shows the IO load on the storage server for the *Cloud VPS* traces, where each trace is replayed separately. In general we can see that the write-back policy still achieves the lowest IO load on the server, and the reduction varies from 21% to 83% compared to the write-through policy.

7. PERSISTENCY ANALYSIS

7.1 Overhead of Persistency

With the understanding of the impact on hit rate and IO latency of the different cache policies, we want to further analyze the overhead associated with making the cache persistent—although cached data blocks are always persistently stored on flash, the metadata of these blocks, including the source-to-cache address mappings and valid and dirty bits, also need to be considered in terms of their persistency. Storing the metadata persistently on flash allows the cached data to be reused after the storage client reboots, but it incurs more overhead. Moreover, if the write-back policy is used, the metadata of dirty blocks must be stored persistently; otherwise, these locally modified data will be lost after a reboot. Note that even if the storage client is completely lost, a persistent flash cache can be physically moved to a different client to reuse or recover the cached data. Based on the above considerations, we study two different persistency configurations for a flash cache.

- *All-persistent*: The metadata of all cached blocks are persistently stored on the flash.
- *Write-back-persistent*: The metadata of only the dirty cache blocks are persistently stored on the flash.

To make a flash cache persistent, metadata updates need to be committed to the cache upon cache insertions, replacements, and invalidations. Our current implementation for making *dm-cache* persistent is quite straightforward. A metadata update is written to the flash at the same time of the corresponding cache insertion or replacement (but cache invalidations require only metadata updates and no data updates). The data and metadata updates are issued in parallel and the original IO request received by dm-cache is returned only when both are committed to the cache. The IO latency is hence determined by the slower one between the data and metadata updates. Although flash devices typically have good internal parallelism to handle concurrent IOs, additional writes introduced by the metadata updates may degrade the performance of flash caching because writes tend to be slower than reads and get amplified due to the need of garbage collection.

More efficient handling of metadata update is possible but not trivial. For example, it is possible to combine the data and metadata updates in a single write, but the metadata is typically small and requires the update on a partial page. Related work [28] proposed to store the metadata in the out-of-band (OOB) area of a flash page on the device, but it requires changing the device's FTL and occupies the limited OOB area which is commonly used for important error correction. In addition to the potential slowdown, storing metadata in flash cache also reduces the size available for data caching; in our experiments, using a 120GB flash device total size, 1GB of the flash capacity needs to be reserved for metadata storage.

Figure 18 shows the IO latencies for various persistency configurations when handling a random read/write (50% reads and 50% writes) workload of different sizes generated by the *fio* benchmark. The write-back policy is used for both persistency configurations. When the workload is small (4GB random reads/writes with 1GB of WSS), the overhead of persistency is small (around 0.03ms); but when the workload is larger (10GB random reads/writes with 5GB of WSS), the overhead grows to 0.06ms (101.8%) as the addition metadata updates slow down the other cache accesses.

7.2 Benefits of Persistency

Having a persistent cache allows the client to continue with a warm cache after it reboots or recover from a crash. In contrast, with a non-persistent cache, the client has to flush all the cached data after it comes back and warms up the cache from scratch, which may lead to substantial compulsory misses. We study this performance improvement from a persistent cache by analyzing the cache hit rate from the two different configurations, *all-persistent* and *write-back-persistent* using dm-cache-sim, while considering different reboot/crash frequencies (daily and hourly).

Figure 19a shows the results from a workload extracted from the Web server trace, assuming the client reboots or recovers upon the start of every day in the experiment. The results show that upon every reboot/recovery, the write-back-persistent configuration has to warm up the cache again, which in average takes 5 hours, whereas the all-persistent

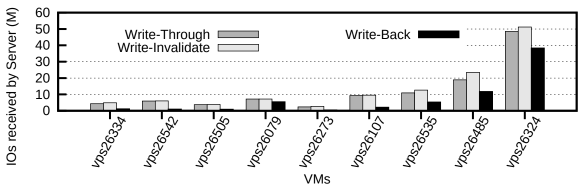
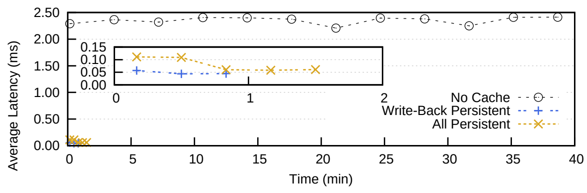
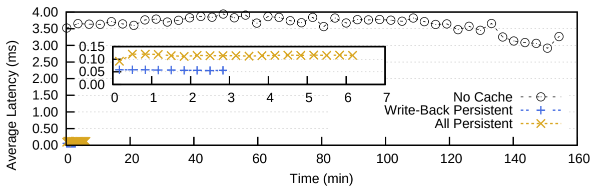


Figure 17: Server IO load for CloudVPS traces

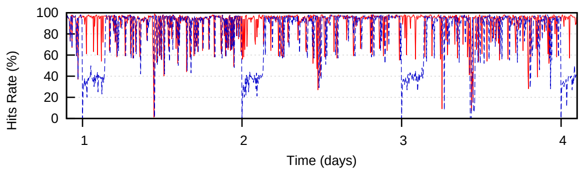


(a) 1GB random reads/writes

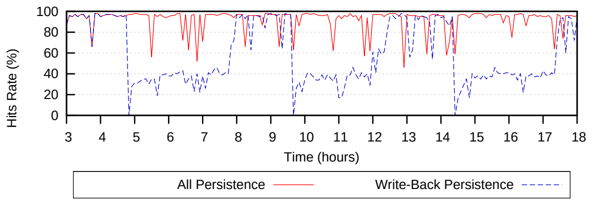


(b) 10GB random reads/writes

Figure 18: Persistency overhead with fio



(a) Daily crashes



(b) Hourly crashes

Figure 19: Cache hit rate changes over time with different persistency configurations

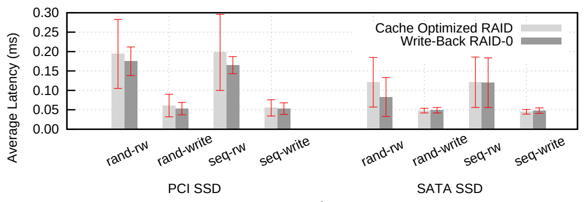


Figure 20: Overhead of cache-optimized RAID

configuration always enjoys a warm cache despite of the reboots or crashes. In average, the hit rate of all-persistent configuration is higher than the write-back-persistent configuration by 7.97% in this experiment. Note that the hit rate drops in the middle of day which happens to both configurations and is caused by the change of data locality.

Figure 19b shows the results from a workload extracted from the Moodle server trace, assuming the client reboots or recovers upon the start of every 5th hour in the experiment. For this workload, using the write-back-persistent configuration, it takes in average 3 hours to warm up the cache again after a reboot/crash, and as a result the hit rate is lower by 27.66% than the all-persistent configuration which has a warm cache persisting across reboots/crashes.

The above cost and benefit analysis shows a clear tradeoff. Making a flash cache entirely persistent slows down IO latencies during normal operations but improves hit rates after client reboots. This decision should be made based on the expected client failure rate for a given cloud system.

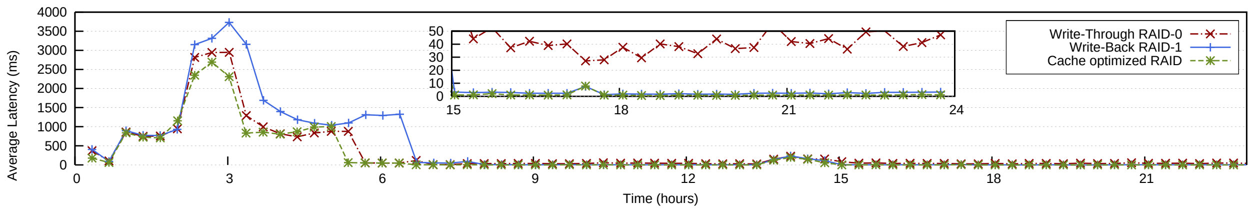
8. RELIABILITY

While the persistent flash cache discussed in the previous section allows the cache to tolerate client restarts and crashes, it does not protect data against flash device failures, including memory cell failures that cannot be masked by the device controller and catastrophic whole-device or whole-chip failures. This concern for data reliability is the reason why flash caching is commonly used in write-through mode, by submitting writes to the storage server while caching them on the flash device, instead of the write-back mode in which writes are delayed in cache without immediately submitted to server. However, as shown in Section 6, write-back caching can substantially improve the storage client's performance and reduce the storage server's load. This conflicts presents a challenge to the effective use of flash caching.

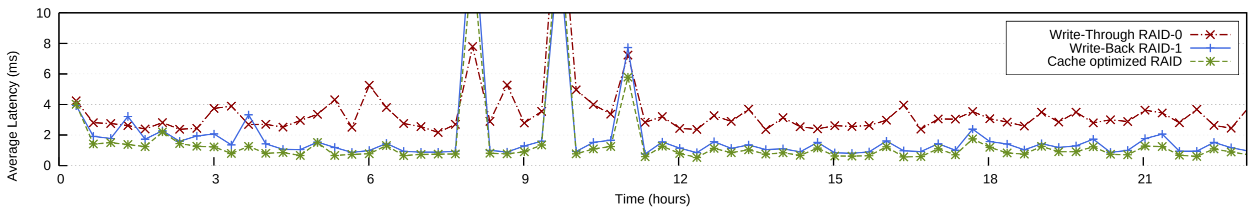
RAID is a classic technique used to tolerate catastrophic failures for hard-disk storage, and has been recently studied for flash storage [22, 15]. However, compared to the level of RAID extensively employed on the storage server, the use of RAID for flash caching faces two major limitations. *First*, the cost of using RAID for a flash cache is substantially more expensive than the cost on the storage server. Following the general principle of forming an effective storage hierarchy, for a storage layer to be fast enough as a cache for the underlying layer, it has to use a technology that is typically much more expensive in terms of per unit size cost. *Second*, using RAID to improve the reliability for a flash cache is at conflict with the other objectives, particularly performance—more redundancy leads to less capacity for storing data localities, and endurance—more redundancy also leads to more wear-out to the flash of the same size.

To address the above limitations, we propose a new *cache-optimized RAID* technique by exploiting different levels of reliability needs for clean data and dirty data to improve cache utilization and reduce its cost. On one hand, clean data in the cache do not require extra redundancy, and their flash pages can employ RAID-0 across the participating flash devices to provide only performance improvement via striping. On the other hand, dirty data in the cache must be provided the same level of reliability as the primary storage, so they will employ higher levels of RAID to tolerate different types of failures. In this way, the cost of using RAID to provide fault tolerance can be minimized by introducing only the necessary redundancy into a flash cache, and this approach has the potential to make a write-back cache reliable and affordable. For the same reason, the adversary impact of using RAID to cache performance and wear-out can also be minimized. Furthermore, the tradeoff between these conflicting objectives can be flexibly adjusted by tuning the amount of dirty data kept in cache.

We have implemented this cache-optimized RAID technique in dm-cache. In this implementation, reads are striped among the flash devices in a RAID-0 fashion for performance



(a) Read-intensive workload



(b) Write-intensive workload

Figure 21: Performance of different reliability configurations

improvement while writes are replicated among the devices in a RAID-1 fashion for reliability improvement. For replacement, a read replaces the LRU block considering all devices in the RAID group, a write is replicated across the devices using the LRU block on each device.

The rest of this section evaluates our proposed cache-optimized RAID technique. First we study the overhead by comparing the *cache-optimized RAID* with a vanilla write-back cache layered on top native Linux RAID-0 (*Write-back RAID-0*). Because the *Write-back RAID-0* does not provide any data redundancy, this experiment evaluates the overhead incurred by replicating the dirty cached blocks in our *cache-optimized RAID*. We employed two identical flash devices on the client for the RAID configurations and used *fio* to generate different workload patterns for the experiment. The results in Figure 20 show that this overhead is small (less than 9.1μs (9%) increase in IO latency).

We further analyze the cache-optimized RAID for real-world workloads and compare it to the alternative options for data reliability, including write-through caching on top of native Linux RAID-0 (*Write-through RAID-0*) and write-back caching on native RAID-1 (*Write-back RAID-1*). We consider the same two workloads used in Section 6.1, one read-intensive from the Moodle server trace and the other write-intensive from the Web server trace. Figure 21a shows the IO latencies for the read-intensive workload. In average, the *cache-optimized RAID* configuration's latencies are lower than the *Write-through RAID-0* and *Write-back RAID-1* configurations by 63ms (26%) and 172ms (72%) respectively. Because of the slower performance of the *write-back RAID-1*, from replicating every block and half-reduced capacity, its warm-up time is also stretched longer than the other two configurations. Figure 21b shows the latencies for the write-intensive workload, where the *cache-optimized RAID* configuration's latencies are again lower than *Write-through RAID-0* and *Write-back RAID-1* by 1.97ms (135%) and 0.23ms (23%) in average, respectively.

9. CONCLUSIONS

Caching is one of the most widely used techniques for improving the performance of data accesses in 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 versus from the next layer in the storage hierarchy. The emergence of flash storage has motivated the consideration of client-side caching in a network storage system because the speed of flash is substantially faster than the network and the mechanical disks 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 adequate answers to the key questions on whether there is good locality in typical cloud workloads and whether flash caches can effectively utilize the locality to achieve good speedup.

This paper provides answers to the above questions based on dm-cache, a block-level caching solution designed for cloud environments, and a large amount of real-world traces collected from both public and private clouds. Our study confirms that cloud workloads have good cacheability and dm-cache incurs low overhead with respect to commodity flash devices. The impact of different write caching policies is significant to cache performance. In particular, different from the conclusion from related work, our results show that write-back caching can substantially outperform write-through caching due to the reduction of server IO load. Our results also show that there is a tradeoff on making a flash cache persistent across client restarts which saves several hours of cache warm-up time but also incurs considerable overhead from committing metadata updates persistently. Finally, to address the reliability issue of write-back caching, we propose a new cache-optimized RAID technique which minimizes the RAID overhead by introducing redundancy to only cached dirty data and shows to be significantly faster than traditional RAID and write-through caching.

10. ACKNOWLEDGEMENT

The authors thank the anonymous reviewers and Swaminathan Sundararaman from Fusion-io for their helpful comments, and thank Lennard Zwart and Pim Van Riezen from CloudVPS and Eric Johnson from FIU for assisting the collection of traces. This research is sponsored by National Science Foundation under grant CCF-0938045 and CAREER award CNS-125394, Department of Homeland Security under grant 2010-ST-062-000039, and a gift from CloudVPS.

11. REFERENCES

[1] Amazon Elastic Block Store. `http://aws.amazon.com/ebs/`.

[2] *blktrace: Linuz block I/O traces.* `http://linux.die.net/man/8/blktrace`.

[3] Cloud VPS. `https://www.cloudvps.nl/`.

[4] Cloud VPS. `http://www.cloudvps.com/blog/cloudvps-activates-linux-ssd-caching-with-dm-cache`.

[5] dm-cache. `http://visa.cs.fiu.edu/dmcache`.

[6] *Dtrace: dynamic tracing framework by Sun Microsystems.* `http://en.wikipedia.org/wiki/DTrace`.

[7] Facebook Flashcache. `https://github.com/facebook/flashcache/`.

[8] *Fio - Flexible I/O Tester Synthetic Benchmark.* `http://git.kernel.dk/?p=fio.git`.

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

[10] GFS Project Page. `http://sourceware.org/cluster/gfs/`.

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

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

[13] Openstack Compute Documentation. `http://nova.openstack.org/index.html`.

[14] VMware VMFS. `http://www.vmware.com/products/vmfs/overview.html`.

[15] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi. Differential raid: rethinking raid for ssd reliability. *ACM Transactions on Storage (TOS)*, 6(2):4, 2010.

[16] 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*, volume 37, 5 of *Operating Systems Review*, pages 164–177, New York, Oct. 19–22 2003. ACM Press.

[17] 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.

[18] J. L. Hennessy and D. A. Patterson. *Computer architecture - a quantitative approach, 4th Edition.* Morgan Kaufmann, 2006.

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

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

[21] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with lazy adaptive replacement.

[22] N. Jeremic, G. Mühl, A. Busse, and J. Richling. The pitfalls of deploying solid-state drive raids. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, page 14. ACM, 2011.

[23] 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*, 2013.

[24] 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.

[25] A. Leung, S. Pasupathy, G. Goodson, and E. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proc. of USENIX ATC*, 2008.

[26] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[27] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *Proc. of USENIX FAST*, 2008.

[28] M. Saxena, M. M. Swift, and Y. Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[29] R. Thornburgh and B. Schoenborn. *Storage Area Networks.* Prentice Hall PTR, 2000.

[30] 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*, page 10. ACM, 2013.

[31] M. Zhao and R. J. Figueiredo. Application-tailored cache consistency for wide-area file systems. In *Proc. Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 41–41, 2006.

[32] M. Zhao, J. Zhang, and R. Figueiredo. Distributed file system virtualization techniques supporting on-demand virtual machine environments for grid computing. *Cluster Computing*, 9(1):45–56, January 2006.