Mercury: Host-side Flash Caching for the Data Center

Steve Byan, James Lentini, Anshul Madan, Luis Pabón
Michael Condict, Jeff Kimmel, Steve Kleiman, Christopher Small, Mark Storer
NetApp, Inc.
{*byan,jlentini,anshul,lpabon,mcondict,kimmel,srk,casmall,mwstorer*}*@netapp.com*

***Abstract*—The adoption of flash memory in high volume consumer products such as cell phones, tablet computers, digital cameras, and portable music players has driven down flash costs and increased flash quality. This trend is pushing flash memory into new applications, including enterprise computing. In enterprise data centers, servers containing flash-based Solid-State Drives (SSDs) are becoming common. However, data center architects prefer to deploy shared storage over direct-attached storage (DAS). Shared storage offers superior manageability, availability, and scalability compared to DAS. For these reasons, system designers want to reap the benefits of direct attached flash memory without decreasing the value of shared storage systems. Our solution is Mercury, a persistent, write-through host-side cache for flash memory. By designing Mercury as a hypervisor cache, we simplify integration and deployment into host environments. This paper presents our experience building a host-side flash cache, an architectural analysis of possible cache attachment points, and a performance evaluation using enterprise workloads. Our results show a 26% improvement in the bandwidth observed by the Jetstress benchmark and a 500% improvement in the I/O rate of an enterprise workload.**

I. Introduction

Modern data centers generally share two key characteristics. The first is their extensive use of server virtualization. Virtual machine technology enables better utilization of hardware resources and therefore reduces data center cost. By separating the physical machine from the operating system and application software, virtual machine technology allows dynamic allocation of hardware resources. Data center administrators have the flexibility to move workloads from one server to another for load balancing, hardware maintenance, and high availability.

The second characteristic common to most modern data centers is the extensive use of shared storage, both network attached storage (NAS) and storage area networks (SAN). Both NAS and SAN technologies allow for unified and centralized data management. This makes it easier for data center administrators to manage a dataset. Administrators can choose the level of data protection (e.g. none, RAID, NetApp® RAID-DP®, etc.), enable mirroring for disaster recovery, and carefully configure the backup policies. Another advantage of shared storage is its scalability. Shared storage systems allow additional storage space to be added dynamically and for storage space to be reassigned. Finally, centralization provides opportunities for deduplication to achieve greater storage efficiency.

The emergence of direct attached flash-based SSDs into this environment poses some interesting opportunities. Flash is generally accepted as new tier in the memory hierarchy between DRAM and magnetic hard disks. In terms of cost per gigabyte (GB), DRAM capacity is more expensive than flash capacity, which is more expensive than hard disk capacity. At the same time, DRAM latencies are less than flash, and flash latencies are less than hard disk. As a result, flash's cost per I/O operation is between DRAM and magnetic hard disks. The question is therefore how to effectively employ flash devices within a virtualized, shared storage data center.

With Mercury, we have built a system that uses flash memory as a write-through cache. A write-through cache allows the use of a shared storage system's reliability, availability, and data management features. This design is also consistent with the use of virtual machine technology since guest state is not tied to a particular server. Mercury evaluates read operations that miss the cache and all write operations that are successfully completed by the shared storage system for insertion into the cache. Cache insertions are done in the background to minimize overheads on the I/O path. For read hits, Mercury reduces read latencies by using a fast storage medium, in this case flash, and locating it close to the application. In a shared storage environment, write latency can be mitigated using other techniques, such as a low latency NVRAM buffer [7].

II. Architecture

As discussed in Section I, we see two major technology trends impacting storage systems in the near future: terabyte-scale second-level storage caches, and pervasive deployment of hypervisors. Along with the customer requirements discussed in Section II-A below, these dual technology trends shaped the architecture of our Mercury prototype.

A. Customer Requirements

Customers of enterprise storage systems expect them to meet a number of requirements. The storage system must be *highly available*. It must provide *correct and consistent access to data*. In a storage system with a distributed cache, the caches must be kept consistent. The storage system must provide *consistently high performance*. The customer expects

978-1-4673-1747-4/12/$31.00 © 2012 IEEE

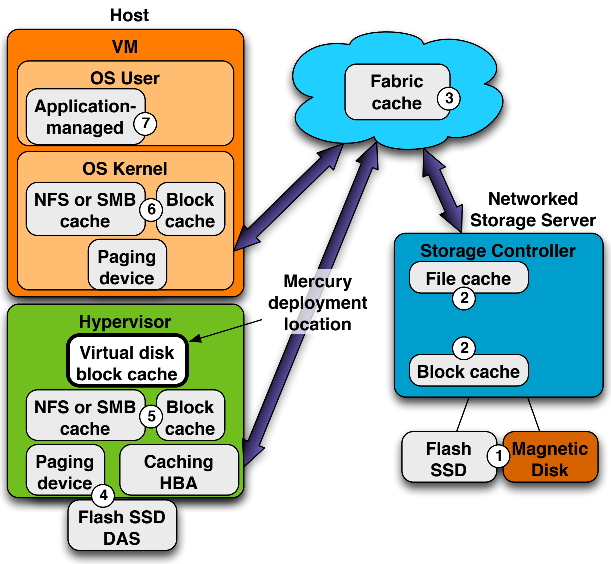

Fig. 1. Flash deployment locations

the system to meet her desired *Service Level Objective* (SLO). It should meet the SLO even after it is restarted from a scheduled shutdown or a crash or power-failure. Finally, the customer expects the storage system to integrate with her data-center management tools, which increasingly is coming to mean *integration with a hypervisor* and its data-management interfaces.

B. Consequences of the Requirements

There is a broad range of possible deployment locations for flash in a networked storage system attached to virtualized host processors. Figure 1 illustrates the range of choices.

Flash could be deployed as a peer to disks in the back end of the storage system (see ① in Figure 1) and managed either as a cache or as a higher-performance hierarchical storage tier. It could be integrated with the storage system controller as a logical (file) or physical (block) second-level buffer cache (②), behind the first-level DRAM buffer cache. It could be deployed in the network fabric itself (③).

On the host side, flash could be deployed simply as directly-attached file storage or as a paging device (④), occupying a niche similar to drum memory in the early virtual memory paging systems. It could be deployed as a block- or file-level cache (⑤) in a hypervisor or bare-metal OS installation. In virtualized guest machines, the same deployment choices are presented recursively (⑥). Finally, flash could be deployed as application-managed storage (⑦).

In the next section we analyze customer requirements to determine the best location to deploy flash from among these possibilities.

1) High performance: Our first goal is that a storage system using flash offer high peak performance. We wish to minimize the overhead on the cache-hit path. The cost of a network hop is about the same as the access latency of flash storage. Consequently locating a flash cache on the storage controller side of the network doubles the latency for a cache hit. Therefore flash storage targeted at providing high IOPS to a host application is best deployed on the host as directly-attached local storage.[1]

Our next consideration is to provide consistent performance, even after a system crash, power failure, or scheduled down-time. A terabyte-sized cache takes several hours to warm up to a high hit-rate. The point of a large flash cache is to help meet the customers performance SLO. If the customer must wait an inordinate amount of time after restarting the system before meeting her desired SLO, the cache is not benefiting the customer. We therefore conclude that the cache must at least be *persistent* across scheduled shutdown and restarts, and ideally should be *durable* in the face of system crashes or power failures.

Finally, in a virtualized data-center, virtual machines may be *live-migrated* from one physical host to another. That is, a running virtual machine may be transparently moved to another physical host without disturbing its state and with minimal delay in its processing. The storage system must offer consistent performance even after migration of a virtual machine to another physical host. By the argument presented above, taking a network hop back to the original physical host to fetch data from its flash cache does not deliver adequate performance. Consequently the cache must either re-warm itself quickly, perhaps with the aid of a list of the most-frequently-accessed blocks from the old cache, or migrate the most frequently accessed portion of the cache as part of the virtual machine state. Therefore the flash cache must be integrated with the hypervisor such that it can participate in the live-migration of a virtual machine.

2) High Availability: Our customers require a *recovery point objective* (RPO) of 0, meaning no data is lost from a crash or a data-center power-failure. To minimize the chance of data loss and maximize data availablity, data written to a direct-attached local flash cache must be protected by storing a redundant copy or erasure code in some off-node location.

We first considered a symmetric peer-to-peer architecture with a mirrored write-back cache policy. In such an architecture each physical host would have to discover and negotiate a pairing agreement with a buddy node to obtain storage for mirroring its writes. This reduces the amount of flash available to each host, since space has to be reserved for the mirrored writes. The off-node mirror write removes much of the benefit of a write-back cache policy, since to protect against either a node failure or a data-center-wide power failure, every write must be synchronously written-through to persistent storage on the buddy via a network hop before it can be reported complete to the application. This distributed peer-to-peer approach also imposes the implementation complexity of quorum and distributed agreement algorithms. In return it may offer better scaling at very large scales with a very high I/O demand. While the implementation complexity is not insurmountable and the possibility of operating at very large

[1]There are uses for flash in the storage controller itself that are not directly related to servicing requests from the host application, for example metadata caching or write buffering.

scale is attractive, we discarded this architecture due to the doubling of the already-high cost of flash storage imposed by the mirroring.

As a better alternative, we selected an asymmetric client-server architecture with a write-through cache policy. In this architecture the network storage controller providing the disk backing store also provides the redundant (mirror) storage for high availability. This may not be an optimal architecture for most storage arrays due to the high write load imposed on the network storage controller. However, NetApp Data ONTAP® storage systems have been architected and refined to excel at writes due to the log-structured write disk access pattern of the WAFL® (Write Anywhere File Layout) file system [7] and the provision of an NVRAM write buffer. The resulting latency for a write is as good or better than that of a peer-to-peer architecture and scales well up to the point where the I/O write demand exceeds the sequential write bandwidth of the back-end disk subsystem. Moreover, the mirroring of written data is accomplished using a small NVRAM buffer and low-cost sequential disk write bandwidth, rather than costly mirrored flash storage.

One consequence of the asymmetric client-server with write-through architecture is that the flash can be simply managed as a read cache rather than as a more-complicated hierarchically-tiered storage system. The storage system never has to migrate data from flash to disk nor to forward read requests to flash on another host; the disk backing store always contains an up-to-date copy. Thus the asymmetric client-server write-through architecture simplifies maintaining consistency between caches and between a cache and the disk backing store; see Section II-B3.

Customer acceptance of a non-zero RPO (loss of recently-written data on a crash) would open the architecture to a write-back flash cache without the necessity of synchronous mirroring; this could be implemented in either architectural model. To provide crash consistency, the write-back policy has to maintain the original write order, including that of all overwrites. Consequently its performance benefit is not as much as one would naively expect. We did not consider non-zero RPO further; it remains an area for future research.

3) Correct and Consistent Data: Multiple hosts with direct-attach flash caches introduce a pair of cache consistency issues (see Figure 2): *vertical cache consistency* between the flash cache on a physical host and the disk backing storage, and *horizontal cache consistency* between caches distributed on multiple physical hosts.

A system experiences loss of *vertical* cache consistency when the host cache contains data different from that on the backing store. There are two vertical consistency scenarios of concern. The first scenario has modified data in the host flash cache which has yet to be written to the disk backing store, and hence is not visible to the network storage controller; i.e., a write-back cache. This modified data is problematic if the storage controller attempts to snapshot or clone an object at the backing store; the object does not contain a consistent set of disk blocks and the modified blocks are not visible

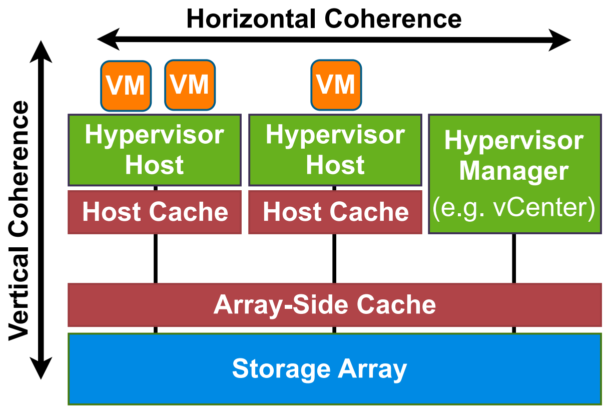

Fig. 2. Cache consistency dimensions

to the storage controller. The second scenario has modified data in the backing store which conflicts with older data in the host flash cache. Such a scenario might arise when the storage controller reverts to an older snapshot, or receives asynchronous mirror changes from a remote primary.[2]

We address the first scenario with our decision to implement a write-through caching policy. Addressing the second scenario requires a communication channel from the storage controller to the cache whereby the storage controller can force the invalidation of blocks that may be cached but which conflict with the new state of the backing store.

A system experiences loss of *horizontal* cache consistency when a host cache contains data different from newly-modified data on another host cache, or newly-modified data is sent to the backing store from a cache-less host.[3] In either case the result is similar to the second vertical cache consistency scenario; the state of the other cache(s) and the backing store diverge from that of the now-stale cache.

A write-through caching policy simplifies implementing horizontal cache consistency, since it is always correct to invalidate any cache entry at any time because cache blocks are never dirty. Any subsequent miss is guaranteed to return the correct data from the backing store. Consequently, to resolve horizontal cache conflicts we only require a cache directory entity with communication channels to each cache, through which it can invalidate the stale contents of a cache.

These cache consistency issues reduce to implementing some form of software distributed shared memory [4][10]. Aside from correctness issues, we are also concerned with minimizing the performance overhead of the cache consistency mechanism. Implementing a strict sequential consistency distributed shared memory model would be far too expensive both in performance and design complexity. It would introduce either a potential invalidate on each write, with the concomitant need to communicate with a lock manager on each write, or

[2]This second vertical consistency scenario could instead be considered as a form of the horizontal consistency problem, introduced by a "hidden" host embedded within the storage system.

[3]For example, a non-cached workstation may connect to the network storage to provision an existing virtual disk with an updated software installation; stale copies of this newly-written data must be purged from any caches.

else a complex scheme that implements exponential backoff in the size of the lock region in an attempt to efficiently discover the sets of non-shared disk blocks via a form of binary search.

On the other hand, implementing a release consistency distributed shared memory model [4] seems workable. The critical region for a virtual disk is acquired when a VM is started on a physical host and released when the VM is shutdown or migrated off the physical host. This provides a relaxed consistency model similar to the NFS *close-to-open* consistency model. The performance overhead is low as there is no consistency overhead in the data access path. However, implementing release consistency requires the identification and tracking of the set of blocks belonging to a virtual disk since they compose the critical region locked by the release consistency model.

This release consistency model supports the serial reuse of mutable virtual disks by VMs on different physical hosts. This feature supports the migration of a VM to another host, whether as a result of a live-migration, an off-line migration, or a high-availability fail-over. However, the model excludes concurrent caching of mutable virtual disks by VMs on different physical hosts. Such concurrent sharing arises when multiple VMs share a virtual disk using a cluster file system or a clustered database.[4]

We believe this last constraint has little practical import as caching may simply be disabled for any concurrently-shared mutable virtual disks. We choose instead to optimize for serially-reused access to a writeable virtual disk by a single VM at a time. Note that immutable virtual disks, such as so-called *golden master* disks, may be concurrently cached without difficulty since their contents never change.

4) Hypervisor Integration: As discussed above, to enable cache consistency through live virtual machine migrations and data management operations on virtual disks, we believe that a flash cache should be integrated with the hypervisor.

Within the hypervisor, the cache should be installed above the file system switch rather than in the block I/O stack beneath the file systems. We wish to support multiple networked storage protocols; a cache in the block I/O stack would not support network file systems such as NFS or SMB. A cache placed above the hypervisor file system switch plays nicely with both network file system clients and standard cluster file systems. It becomes "one cache to rule them all", covering SCSI pass-thru LUNs, virtual disk files on cluster file systems, and virtual disk files on NFS and SMB network file systems with one implementation. Additionally, a cache in the hypervisor block I/O stack would require sequential consistency between caches rather than release consistency, in order to support concurrent shared access to blocks containing cluster file system metadata.

[4]Note that hypervisors offer limited support in their I/O virtualization stack for VMs which concurrently access a single virtual disk. They do not typically support virtualization of a concurrently-shared virtual disk. They instead require shared virtual disks to be simply a pass-through to a physical SCSI LUN.

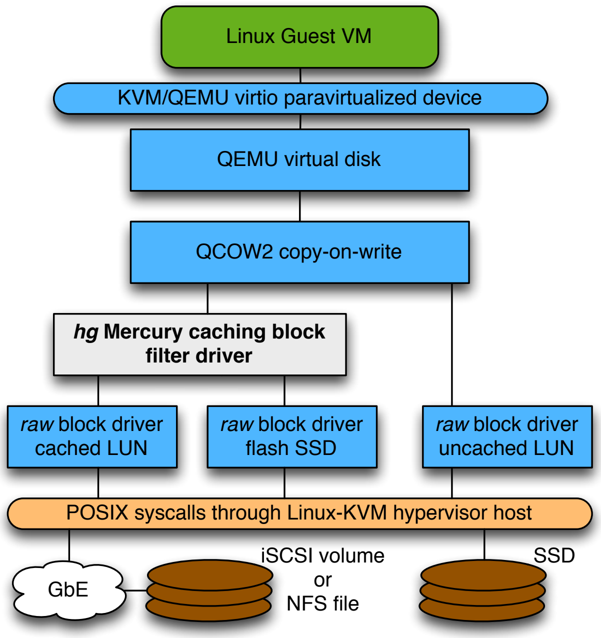
Linux Guest VM
KVM/QEMU virtio paravirtualized device
QEMU virtual disk
QCOW2 copy-on-write
hg Mercury caching block filter driver
raw block driver cached LUN
raw block driver flash SSD
raw block driver uncached LUN
POSIX syscalls through Linux-KVM hypervisor host
GbE
iSCSI volume or NFS file
SSD

Fig. 3. Hypervisor integration

Rather than trying to filter and cache the top edge of the hypervisor virtual file system switch, the cache should be placed in the hypervisor block I/O virtualization stack (which itself is located above the file system switch). This placement limits the interface to be the simpler block or SCSI interface rather than a full file system interface, and simultaneously restricts the filtered I/O to be only that issued by virtual machines. This avoids any confusion between I/O from a guest VM and that from the hypervisor itself.

Additionally, it allows the cache to associate blocks with the objects comprising a virtual disk. This association allows the cache to implement release consistency by managing exclusive access to the critical region composed of the blocks associated with the objects comprising a virtual disk. The association also allows the cache to migrate the cache data and/or frequency of access metadata for a migrating virtual machine's disks without having to track the virtual-to-physical mapping of all the individual disk blocks.

Most hypervisor block I/O virtualization stacks expose virtual disks which are composed of a hierarchical tree of block storage objects. This tree of storage objects is used to implement copy-on-write shared clones and snapshots. A copy-on-write layer sits above these block storage objects and exposes a single virtual disk block storage object to the guest virtual machine. An immutable lower-level storage object may be shared by many mutable virtual disk storage objects. The copy-on-write layer records overwrites of the shared immutable storage object in a unique writeable storage object associated with each mutable virtual disk.

If a flash cache were implemented above the copy-on-write layer, it would view each clone of an immutable lower-level storage object as a unique virtual disk. It would be forced to either cache multiple copies of the immutable storage object, or else discover the duplication introduced by these aliased virtual disk block addresses via a CPU-intensive signature-based inline deduplication algorithm.

The flash cache should instead filter the interface below the copy-on-write-layer and above the block storage objects. This placement allows the cache to use the "free" deduplication gained from visibility of the references to shared storage objects. The cache may tag blocks with the name of their (possibly shared) storage object rather than the unique name of their virtual disk, thus unifying the addresses of shared immutable storage objects. This enables the cache to avoid caching multiple copies of shared data.

In our prototype, the Mercury flash cache is installed in QEMU as a block device filter. See Figure 3. It is installed below the QCOW2 copy-on-write filter and above the block "devices" that compose the QCOW2 virtual disk.[5]

C. Preliminary Performance Analysis

We investigated the hit rate of a second-level flash cache using a simple analytical model. We modeled the ranked-frequency distribution of block accesses to the disk backing store as a Zipf's Law distribution. Assuming a cache uses the *least-frequently-used* (LFU) replacement policy, and knowing the size of the backing store in blocks n and the size of the cache in blocks k, from [17] we can express its hit rate h as

$$h = \frac{\ln k}{\ln n} \tag{1}$$

The Mercury architecture has a two-level cache hierarchy. The hit-rate of the first-level buffer cache (located in the operating system or database application) follows Equation 1. Now we model the second-level Mercury cache as an inclusive cache. That is, all data in the first-level buffer cache is also contained in the second-level Mercury cache.[6] Consequently, the hit rate of the combined first- and second-level caches is also described by Equation 1.

However, the hit-rate of the second-level cache alone is much smaller than that predicted by Equation 1. The first-level cache satisfies the most frequent requests, leaving only requests from the long flat tail of the Zipf distribution to be satisfied by the second-level cache. Consequently the second-level cache sees a request stream with little locality and hence has a low hit rate.

We find the hit rate h_2 of the second-level cache by subtracting the hit rate h_1 of the first-level cache from the hit rate h of the combined cache and scaling it by the miss rate $1 - h_1$ of the first level cache, since the second-level cache sees only the misses from the first level cache:

[5]In QEMU, each QCOW2 storage object is represented by a virtual QEMU device. Such "devices" may be either a file or a physical device in the underlying hypervisor.

[6]The final Mercury implementation may differ somewhat from this model. We have experimented with write-around policies and may in the future implement heuristics such as sequential I/O bypass which would make the cache not strictly inclusive. However, we do not anticipate implementing Mercury as a victim cache, and so the model still provides an approximate upper-bound on the hit rate.

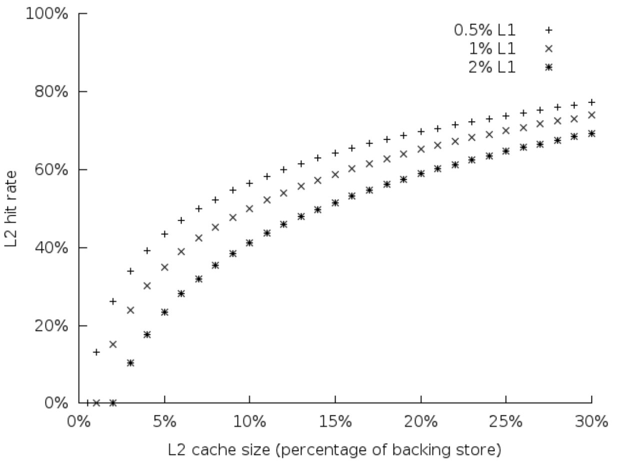

Fig. 4. L2 Hit Rate *vs.* Cache Size

$$h_2 = \frac{h - h_1}{1 - h_1} \tag{2}$$

Figure 4 shows the hit rate of the second-level cache as a function of its size for first-level cache sizes of 0.5%, 1%, and 2% of the backing store. For reasonable second-level cache sizes, the hit rate is alarmingly low, between 40% and 60%.

The overall mean service time t_s of the second-level cache is just the linear combination of the hit and miss service times weighted by the hit and miss rates:

$$t_s = h_2 t_h + (1 - h_2) t_m \tag{3}$$

In general $t_m \gg t_h$. Consequently, in analogy to Amdahl's Law, as the secondary cache hit rate falls below 100% the overall system service time quickly becomes dominated by the cost of the cache misses, even though the majority of requests are serviced as cache hits. It is therefore essential to minimize the cache overhead in order to minimize the harm introduced by the cache. At the low hit-rates predicted by Zipf's Law, a cache with significant hit and miss overhead could be slower than no cache at all.

This preliminary performance analysis informed the detailed design process, where care was taken to minimize the cache overheads. The resulting trade-offs are described below in Section III.

D. Prototype

Our initial prototype is focused on the *performance* and *persistence* requirements and on implementing the correct placement within the hypervisor. Although the architecture and design takes them into consideration, we leave the implementation of *durability* and *cache consistency*, as well as further performance improvements, as future work.

III. Design

The cache is divided into two sub-modules: the Operations Manager (OM) and the Storage Manager (SM). The OM is responsible for the management of the cache's in-memory data structures and contents. The SM maintains the on-flash data structures.

The cache processes an I/O command depending on the command's type, either a read or a write, and whether it is a cache hit or miss. Figure 5 shows a simplified flow of I/O through the cache. Briefly, a read hit can be serviced from the cache device. A read miss is forwarded to the backing device, and may be inserted into the cache when it completes. A write hit invalidates the cache's old version of the targeted blocks. Both a write hit and miss are forwarded to the backing device. As with a read miss, a write may be inserted into the cache.

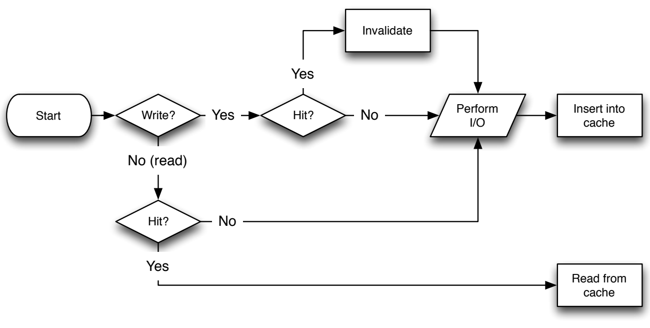

Fig. 5. Simplified I/O Flow Chart

We wait for a successful acknowledgment from the backing device before inserting data into the cache, which keeps the cache's contents consistent with the backing device. Once a successful acknowledgment is received, an I/O command's data is copied to an in-memory buffer, called a *log chunk*, and the command is completed to the upper layer. The log chunk is written to the cache device when full. This process allows I/O commands to complete without waiting for the cache device to be updated.

Data from a read miss or write is not inserted under certain conditions. In our results, we evaluate one of these policies, *write around caching*. In write around caching, all writes bypass the cache under the assumption that data written will not be read in the near future.

There are some subtle details that complicate the situation. One issue is that an I/O command can be unaligned with our cache. We term a backing device's smallest addressable unit a sector. Typically devices have 512 byte sectors, but our design allows for other sector sizes. An I/O command is defined by its device offset in sectors, called the logical block address (LBA), and its length in sectors. In contrast, our cache is divided into *cache blocks*. A cache block is one or more contiguous sectors. By default, we use a cache block size of 8 sectors (4 kilobytes (KB)). By unaligned, we mean that the LBA of the start or end of the I/O is not aligned on a cache block boundary. For these operations, the unaligned blocks bypass the cache, with unaligned write hits generating an invalidate to their containing cache block. Another complication is that an I/O command can target multiple blocks, some of which are hits and some of which are misses. To address this, we combine runs of hits and misses into a single operation.

A. *In-Memory Data Structures*

The cache's in-memory data structures are used to determine if an I/O command is a cache hit. Since this query is performed for every I/O command, the cache needs to efficiently answer this question. We use two in-memory data structures to identify cache hits: a *cache headers* array and an *address map* hash table.

Each cache block is tracked in the cache headers array. A cache header describes its corresponding block's backing device (16-bits), backing device LBA (48-bits), a checksum of the block's contents (32-bits), and information on the block's usage (32-bits). The memory used by a single cache header is 128-bits (16 bytes).

To determine if an I/O is a hit, the cache could linearly scan the cache headers array. An I/O that overlapped a header's LBAs would be a hit. This algorithm would produce correct results, but, at $O(n)$ time for a cache with n entries, the overhead is too high for a large cache.

Instead, we use a dictionary data structure, called the address map, to quickly determine if an I/O targets any cached blocks. Our dictionary maps (backing device, LBA) keys to cache header array indices. The address map is implemented as an open address hash table with linear probing. Each lookup on this data structure runs in $O(1)$ expected time.

An important design choice is whether to keep these cache data structures entirely in main memory, or to store them on the cache device and page portions into main memory. As we explain above, the cache is only effective if it imposes a minimal amount of overhead. Therefore, we must keep these data structures entirely in main memory so that determining if an I/O is a hit or miss runs at memory speed.

To calculate the memory overhead of these data structures, we use the following example. Suppose we have a 512GB cache device and use a 4KB cache block size. The memory overhead is approximately equal to the sum of the cache headers array size and the address map size. The 512GB cache in our example would require 134,217,728 entries in the cache headers array. Since each cache header entry is 16 bytes, the total size of the cache header array would be 2GB. The address map is implemented using an array of 4 byte entries. To make hash collisions infrequent, this array is 1.5 times larger than the number of cache headers. This results in a 768MB address map in our example. Therefore, the total memory overhead is around 2.75GB for a 512GB cache. From this example, we see that the ratio of memory to cache space is about 0.5%. Another way to look at this is that every 4KB cache block requires 22 bytes of memory: 16 bytes in the cache headers array and 6 bytes in the address map.[7] This memory overhead can be decreased by using a larger cache block size (e.g., using 16KB instead of 4KB would decrease the memory overhead by a factor of 4), but further experiments are needed to understand how this effects the cache's hit rate.

[7]Each entry is a four-byte index into the header array. We over-provision by 50% to reduce the expected number of probes; hence, $1.5 \times 4 = 6$ bytes per entry.

B. Block Replacement Policies

When the cache is full (which is the steady state), the cache must evict cached blocks. The Mercury cache supports two block replacement policies: First In First Out (FIFO) and CLOCK [2].

The FIFO replacement algorithm evicts all blocks in the oldest chunk regardless of the number of times a block has been used in the past. Since FIFO will be evicting all blocks, the cache can optimize the cleaning process and eliminate reading the oldest chunk into memory. By eliminating this I/O operation, the FIFO policy decreases the I/O load on the flash device. In our experience, use of the FIFO algorithm results in a lower hit rate than CLOCK. The trade-off between FIFO's reduction in cleaning overhead versus its lower hit rate are shown in Figures 7, 8, 9, 10, 11, and 15.

Mercury also supports a version of the CLOCK replacement algorithm. Our CLOCK implementation maintains a usage bit for each block in the cache. As the cache cycles through the chunks, blocks that have been read since the last pass are retained, and blocks that have not are evicted. While the CLOCK algorithm results in a higher hit rate, it does require reading each chunk of the cache during the cleaning process.

C. On-Flash Data Structures

The flash device contains cached data from one or more backing devices. When designing Mercury's on-flash data structures, we had the following goals in mind:

a) Cache Persistence: From the beginning, we recognized the importance of supporting a persistent cache. As our results demonstrate, the amount of time necessary to warm a large second level cache is significant. Without support for persistence, the cache would need to be re-warmed on each restart. To support persistence, each cached data block has an associated metadata entry. Mercury uses this metadata to reconstruct its in-memory data structures on a reboot.

b) Flash Awareness: Another goal was to use a flash-friendly access pattern. Flash devices generally perform best when written sequentially, rather than randomly [1]. For this reason, our on-flash data structures use log-structured updates. This minimizes the flash write amplification, achieving better performance and less wear-out.

c) Space Efficiency: An additional goal was to maximize space on the cache device for data blocks by using the smallest possible metadata entries. The metadata needed for each data block is contained in a cache header. One of the items in a cache header, the backing device description, needed to be compressed. A backing device description may be hundreds of bytes in length (e.g. a SCSI logical unit name or NFS export with DNS name and path). Although only a few backing devices are cached at one time, there may be millions of blocks in the cache for a single backing device. Obviously, duplicating these large backing device descriptions in every block's metadata would be wasteful. In our design, the backing device description is stored once, in a single location and assigned a unique small integer identifier. The compact integer identifier, rather than the large backing device description, is stored in the metadata for each cache block. A useful side effect of this arrangement is that all cache blocks for a given backing device can be quickly invalidated by invaliding the backing device's integer identifier. Mercury uses this technique to quickly invalidate cached blocks when caching is disabled for a backing device.

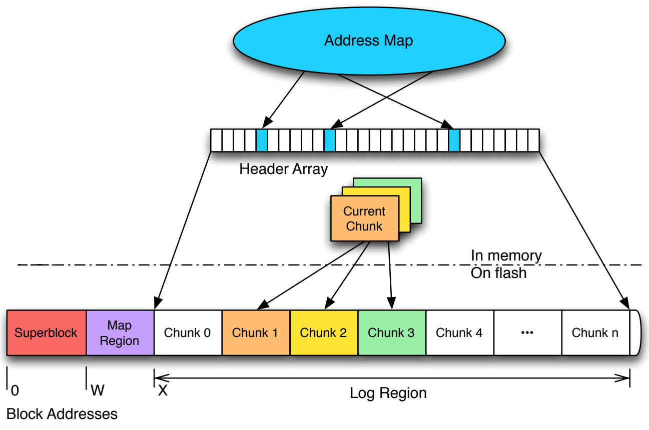

Fig. 6. Cache Data Structures

To achieve the goals above, the Mercury SM provides two on-flash data structures: a map for storing backing device descriptions and a log for storing cache blocks and their metadata. The flash device is split into three regions (see Figure 6). The first region is a superblock that contains the cache's configuration. The second region contains a map of (key,value) pairs. The size of the map is rounded up so that the start of the next region is aligned to a flash erase block size boundary (offset X = an erase block size multiple). The third region contains a circular log of metadata and data.

When the cache is rebooted, the OM and the SM cooperatively replay the log and rebuild the in-memory data structures. If multiple metadata entries exist for the same (backing device identifier, block index) pair, the entry in the newest log chunk is the correct one. A metadata entry with an invalid backing device identifier is for a backing device that is no longer being cached, and therefore can be ignored.

As with any device, it is possible for the cache device to malfunction. The on-flash data structures are designed to detect the full spectrum of possible device errors [12]:

- Latent Sector Errors
- Corruptions
- Torn writes
- Lost writes
- Misdirected writes

Correcting cache data in the presence of these device errors is not a goal of the on-flash data structures. Since the cache is write-through, a malfunctioning cache device is ignored. The cache disables itself, and notifies the system administrator of the error. While the cache is disabled, I/O commands are transparently directed to the backing store.

Our current implementation does not support durability, but we have designed our cache with this in mind. A durable cache is able to terminate execution at an arbitrary point without corrupting cached data. Events like a software fault or power loss would results in such a termination. The challenge is to persistently invalidate cached data on a write hit before sending the write to the backing device. The invalidation must be made persistent before the write is sent to the backing device because the cache might terminate at any time between the write being issued to the backing device and the new data being inserted into the cache. Our design for durability is to add the invalidation to the chunk metadata and flush the chunk metadata region to the cache device. The cache devices we are targeting have battery-backed write buffers. Therefore this operation will be analogous to writing the invalidation to an NVRAM device.

D. *Optimizations*

While implementing Mercury, we identified several performance optimizations to our original design.

One optimization was related to inserting data into the cache. Originally, a single buffer was used for insertions to the cache device. We expected all devices to have sufficient internal memory to buffer the write data, complete the write command, and asynchronously move the write data to their internal flash chips. In practice, this did not prove to be the case. We encountered devices that required multiple, parallel write commands in order to achieve their maximum performance levels. In light of this, we enhanced our cache insertion algorithms to use a configurable number of write buffers.

Related to our write buffering optimization, we also added chunk read-ahead. Some block replacement algorithms, such a CLOCK, retain previously cached data during log cleaning. These algorithms require the oldest log chunk be read into main memory for processing. To accelerate their performance, the cache anticipates requests for these chunks by performing log read-ahead. Log read-ahead is aided by the fact that log writes are performed sequentially. A cache log chunk that has been read will not be written until after it has been cleaned, and therefore the in-memory copy cannot become inconsistent with the contents of the cache device before it is used.

Another optimization was consolidating multiple contiguous operations into a single operation. Initially the cache issued individual I/O operations for each block. For example, a 5 block I/O command might hit on 3 blocks, and miss on the other 2. Rather than fetching the 3 cache hits using 3 cache reads, our implementation detects contiguous cache blocks and issues a single I/O. This optimization was necessary to achieve high performance with multi-block I/Os.

IV. Implementation

To test our approach in a real system, we integrated Mercury with the Linux KVM/QEMU hypervisor. We implemented our cache in user-space by creating a dynamically loadable Mercury library that is linked to QEMU. This approach allowed us to insert Mercury just below the copy on write (COW) layer in QEMU's I/O stack without modifying QEMU source code. Our implementation contains 41,000 source lines of code (SLOC [20]) consisting of the Mercury cache, utilities, and unit tests. In this implementation, there is a separate cache for each VM. In the future, we plan to develop a single cache shared across VMs.

V. Evaluation

We evaluate Mercury with the help of two widely used benchmarks. The first benchmark we use is Microsoft Exchange Jetstress. The second is an enterprise workload, which simulates OLTP workloads [3]. For both benchmarks, we use the experimental setup described in Section V-A below.

A. *Experimental Setup*

For our experiments, we used one host system which was an x3550 IBM Server with two 6 core Intel® Xeon® CPU E5645 processors running at 2.40GHz, a 1 Gbps network interface card, and 48GB of RAM. The system runs Red Hat Enterprise Linux 6.1 with the 64-bit Linux® kernel, KVM version 2.6.32-131.17.1, and QEMU version 0.15.1. All experiments were run inside a single virtual machine. A NetApp FAS3270 with 7200 RPM 1695.4GB SATA disks was used as the iSCSI target. The LUN size was 1 TB and it was created on an 11 disk RAID-DP aggregate. Table I shows the cache devices used in our tests.

B. *Jetstress*

Microsoft® Exchange Jetstress is a benchmark used for simulating an Exchange Server workload on a storage system. We run the Jetstress benchmark inside a Windows® 2008 64-bit VM, with 8 virtual processors, and 4GB virtual DRAM. The database has a maximum cache size of 256MB, and averages approximately 241MB. The dataset is between 800-900GB, 73% of the transactional database I/Os are 32 KB, and the read percentage is around 63%. The benchmark measures the peak storage subsystem throughput by adjusting the I/O intensity based on the observed IOPS of the system.

1) Methodology: As a write-through cache, Mercury does not accelerate the performance of a 100% write workload. Since Jetstress only issues writes to the log disk, we use a dedicated flash drive without a Mercury cache for the log disk.

Before each experiment, we cleared the NetApp storage system's iSCSI cache, rebooted the VM to clear the guest buffer cache, used the NetApp Snapshot^{TM} feature to revert the Jetstress database to its initial state, and cleared the Mercury cache by formatting it. For experiments that varied the cache size, we partitioned the cache device to the desired size, and filled the other unused partition with junk to trigger garbage collection.

We configured QEMU to use the Linux libaio asynchronous I/O interface with the Linux CFQ I/O scheduler. Previous work using SSDs as a swap device found that flash performed best with the Linux NOOP I/O scheduler [15]. However, this previous work used synchronous I/Os, rather than asynchronous I/Os. For synchronous I/Os, the Linux CFQ

TABLE I
FLASH DEVICES USED

Name	*Capacity*	*Type*	*Sequential Read*		*Sequential Write*		*Random Read*		*Random Write*	
			latency	*bandwidth*	*latency*	*bandwidth*	*latency*	*bandwidth*	*latency*	*bandwidth*
PCIe Flash Device X	320GB	SLC	25 us	376.5 MB/s	30 us	390 MB/s	65 us	231 MB/s	275 us	92.5 MB/s
SSD Y	100GB	SLC	100 us	135 MB/s	105 us	115 MB/s	240 us	110 MB/s	225 us	39 MB/s
SSD Z	300GB	MLC	105 us	185 MB/s	103 us	125 MB/s	245 us	155 MB/s	450 us	55 MB/s

scheduler optimizes for disk devices by doing unexplicit anticipatory scheduling. In contrast, CFQ batches asynchronous I/Os together into a single queue (one queue per priority). For the Mercury I/O access pattern, we did not find any advantage of NOOP over CFQ. The only difference in how they handle asynchronous I/O requests is that NOOP merges adjacent requests. Given our workload of random reads and sequential writes to the block device, most adjacent read requests don't qualify for merging, and adjacent writes are already buffered.

2) Results:

a) Accelerates performance: For PCIe Flash Device X, as cache size increases from 0% to 30% in Figure 7, we observe a 26% improvement in the bandwidth, and a cache read hit rate of up to 73%. As can be seen in Figure 12, Mercury succeeds in offloading traffic from the back-end storage system. Thus in spite of being a write through cache, it improves both the read and write bandwidth. The read latency improves by up to 21% and the write latency worsens marginally by up to 6% (Figure 8) because every write to the storage system also results in a write to the flash cache. A 10% PCIe Flash Device X cache has around 3.5% better bandwidth than a 10% SSD Y cache and 6.6% lower read latency.

b) Large Sequential Reads: From Figure 9, we observe that sequential reads by the application for the purpose of background database maintenance are polluting the cache. The hit rate is reduced from 73% to 65% for a 30% cache, thereby reducing the IOPS by approximately 13.5%. Caching these sequential reads is not generally useful because their contents are seldom read again. Thus heuristics to detect sequential reads and bypass the cache in such cases would be useful.

c) Write Through vs. *Write Around:* We observe that the write through policy is significantly better than a write around policy. As shown in Figure 11, the steady state hit rate after warmup is 90% better, although the warmup time is around 30 minutes for both. With the FIFO page replacement policy, read IOPS, write IOPS, and read latency is 35-50% better with write through (see Figure 9 and Figure 10). There is also a small amount of improvement in the write latency.

d) CLOCK vs. *FIFO Replacement Policy:* The performance observed by the application is nearly equivalent with FIFO and CLOCK (see Figure 9 and Figure 10). With the write through policy, the warmup time and steady-state hit rate is the same for both (see Figure 11).

e) Cache Overhead and Back-end Read Response Time: From Figure 13, we observe that the read response time improves by 31% or 2.2 ms when caching using Mercury at up to a 73% read hit rate. However, given that the random read latency of the flash device is 65 us (Table V-A), we

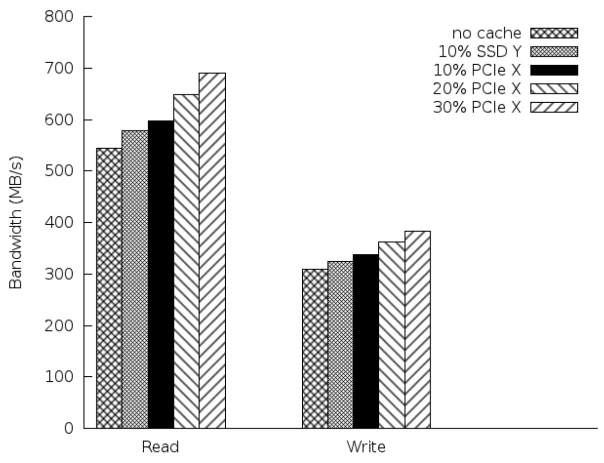

Fig. 7. Bandwidth variation with cache size, device type

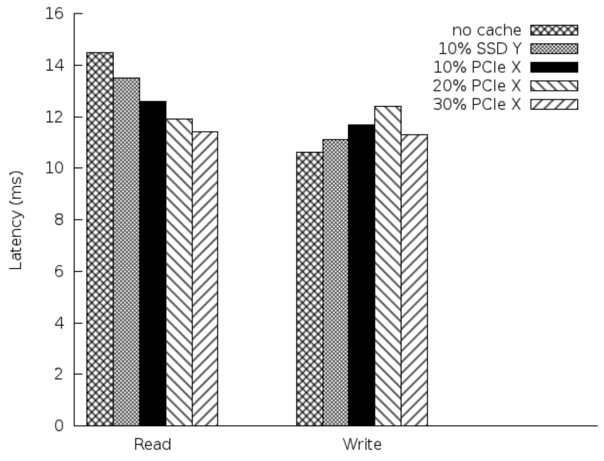

Fig. 8. Latency variation with cache size, device type

would expect the response time to be much less than 5 ms if the storage system's read response time was unchanged. In fact, we observe that the storage system's read response time increases to 17.8 ms from 7.3 ms, an increase of approximately 144%, and in Figure 12 we observe that the storage system's read bandwidth decreases by about 66%. The cache changed the storage system's workload, and, as a result, decreased both read IOPS and read latency. We believe, but have not proven, that this is a result of greater seek time over-head on the storage system.

C. Enterprise Workload

In addition to the Jetstress evaluation, we used an enterprise workload to exercise Mercury. The experimental setup described in Section V-A was used with a Fedora 16 Linux guest running kernel version 3.1.1-fc16.x86_64.

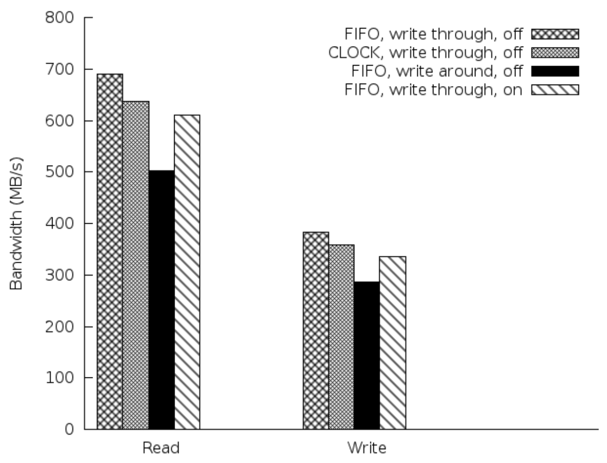

Fig. 9. Bandwidth variation with replacement policy, write policy, background maintenance (on/off), with PCIe Flash Device X, 30% cache

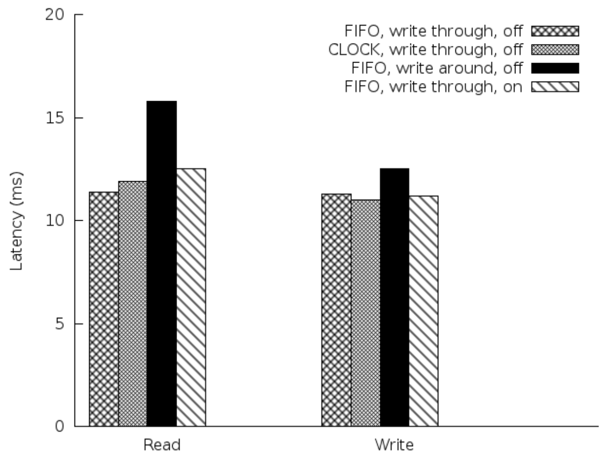

Fig. 10. Latency variation with replacement policy, write policy, background maintenance (on/off), with PCIe Flash Device X, 30% cache

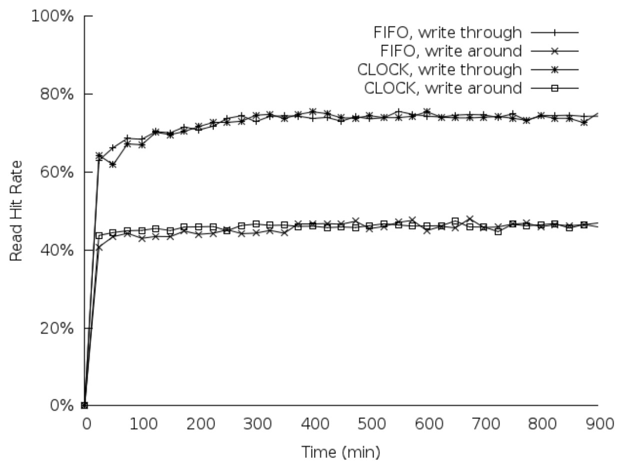

Fig. 11. Jetstress warmup time with PCIe Flash Device X

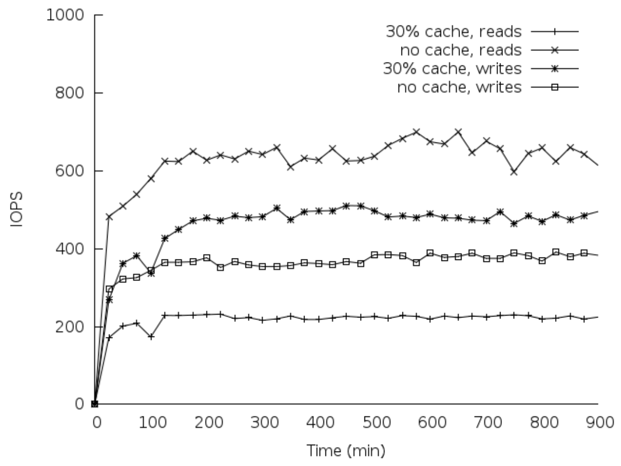

Fig. 12. Jetstress bandwidth between host and storage system with PCIe Flash Device X

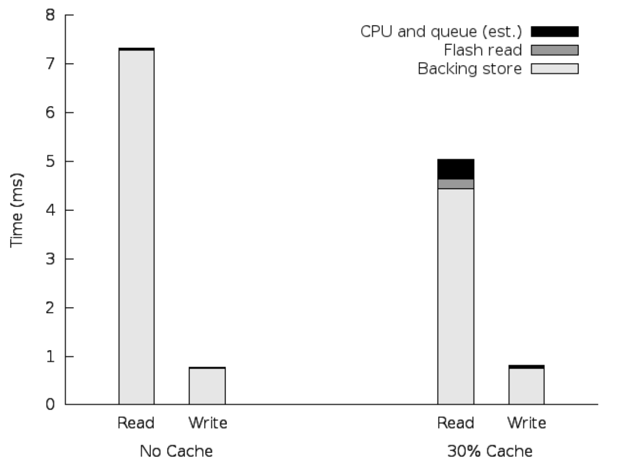

Fig. 13. Jetstress response time overheads (flash writes done asynchronously) with PCIe Device X

This test used twenty four iSCSI LUNs on the NetApp FAS3270 with a RAID-DP aggregate consisting of 12 SATA disks. Eight LUNs were used for each of the three types of Application Storage Units (ASUs). The *Data Store* (ASU 1) and the *User Store* (ASU 2) LUNs where cached using Mercury, while the *Log* (ASU 3) was not cached. The test consisted of a twenty four hour cache warmup stage using 50 Business Scaling Units (BSUs), where each BSU generated 50 I/Os per second. For comparison, we also included test results limiting the size of the PCIe Flash Device X to the capacity of the SSD Y device.

Once the caches were warmed up as shown in Figure 14, we then compared their average latencies using different numbers of BSUs. Without the cache enabled, the test was only able to reach about 2000 IOPS before having a latency larger than 30ms. Figure 16 shows how Mercury increases the I/O Rate for a guest by more than 500%. The figure also shows the I/O rate's dependency on cache size. Lower latency does help the PCIe flash device to achieve about a 20% increase over

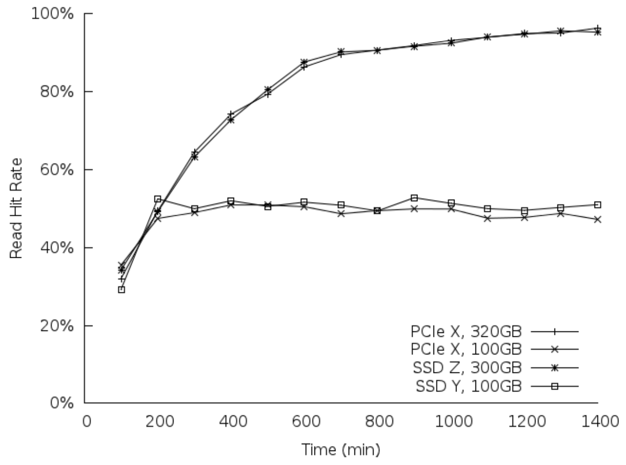

Fig. 14. Enterprise workload warmup times for multiple cache device types and sizes using a write-through policy and a CLOCK replacement algorithm.

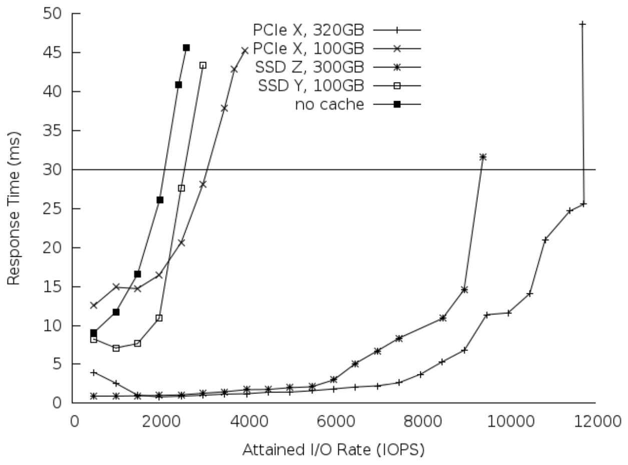

Fig. 16. Enterprise workload average response time for BSU values ranging from 10 to 300 using a cache write-through policy and a CLOCK replacement algorithm. A response time of over 30ms fails the tests.

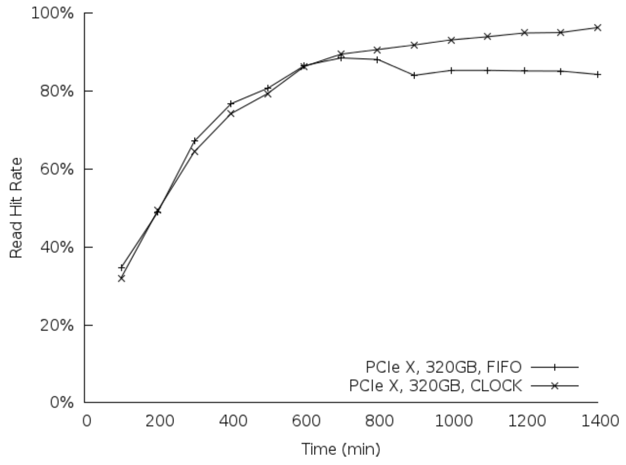

Fig. 15. Enterprise workload comparison of FIFO and CLOCK block replacement policies using a PCIe flash device over a twenty four hour period.

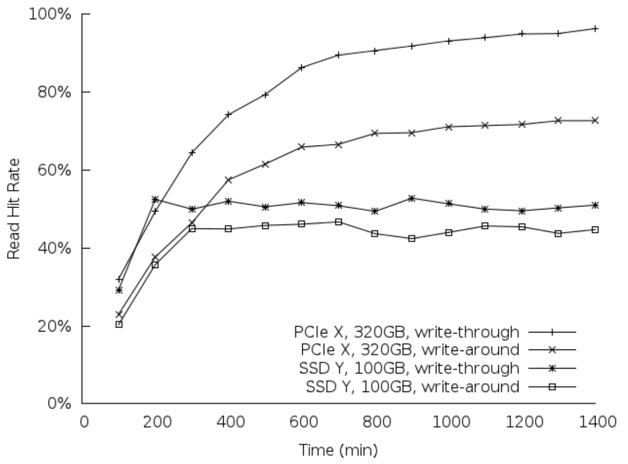

Fig. 17. Comparison of write through versus write around policies for enterprise workloads.

the equally sized SSD Y, but cache size has a more profound effect on the maximum I/O rate.

VI. Related Work

Caches have been studied in a number of different contexts. Previous research has investigated the benefits of locally caching remote data to improve performance. The Andrew File System [8] and the Coda file system [11] are two prominent examples of distributed file systems that use this approach. A more recent example in this area is the Linux FS-Cache [9], which may be used to cache data from an NFS server. An analogous system in the area of block storage is iCache [5], which locally caches iSCSI data. In contrast to these systems, the Mercury cache is protocol agnostic. As described above, our cache is capable of caching data from a variety of different sources including NAS protocols (e.g. NFS, SMB), SAN protocols (e.g. iSCSI, FCP), and locally attached devices (e.g. HDDs). This attribute distinguishes the Mercury cache from these systems.

Several studies have investigated second level caching. The Capo [18] cache is similar to Mercury. Like Mercury, Capo is a persistent, second level cache that attaches to a hypervisor. Unlike Mercury, Capo is designed to store cached data on a rotating magnetic hard disk drive rather than a solid state flash device. Another second level cache using magnetic media is Disk Caching Disk (DCD) [13]. In the case of DCD, the cache did not operate in a virtualized environment. Mercury's advantage over these approaches is that its cache data structures were designed from the ground up to work with the unique characteristics of flash memory devices.

Other researchers have combined caching and solid state flash devices. FlashVM [15] enables the Linux memory management subsystem to use a flash drive as a swap device. FlashVM stores anonymous pages, which are inherently volatile and not persistent. By definition, file-backed pages are not stored on the swap device. Therefore, unlike Mercury, FlashVM is not an I/O cache and is not persistent. Another system that combines caching and flash is FlashTier [16].

Instead using the interface of commodity SSDs, FlashTier is designed to work with a specialized solid-state cache device.

Other block caching systems have also been proposed. The DM-Cache [6] system transparently caches blocks using a local disk device. DM-Cache plugs into the Linux operating system's device-mapper layer. This system was later extended by Facebook for use with solid state devices [14] and renamed FlashCache. FlashCache, as a device mapper plug-in, supports only SAN protocols and locally-attached disk. Unlike Mercury, it cannot cache NAS protocols. The set-associative cache lookup employed by FlashCache results in contiguous extents in the backing store address space being broken up into widely-separated 4KB blocks on the flash device. Thus inserting or retrieving multiple 4KB blocks requires multiple I/O commands to the flash device. The Mercury cache instead inserts extents contiguously, resulting in less I/O overhead when accessing the flash device. FlashCache performs cache insertion synchronously, resulting in a high read-miss penalty as the slow write to flash is completed before FlashCache signals the completion of the read I/O to the application. Mercury performs cache insertion asynchronously on read-misses, providing a lower read-miss penalty.

Similarly, Bcache [19] integrates with the Linux block device layer and stores data on flash storage devices. Like FlashCache, it supports SAN protocols and locally-attached disk, but, unlike Mercury, it cannot cache NAS protocols. Bcache maintains the contiguity of extents on flash via a B-tree structure rather than Mercury's extent-preserving logging. Like Mercury, Bcache performs read-miss insertions asynchronously, giving a low read-miss penalty.

There are also several commercial products that target host-side caching using flash storage devices. Notable among these are Fusion-io's IOTurbine, Marvell's DragonFly, FlashSoft, and EMC Lightning. Unfortunately, detailed public specifications are unavailable, so contrasting their approach with Mercury is difficult at this time.

VII. Conclusion

Flash memory devices represent a new tier between DRAM and magnetic media in terms of both price and performance. The challenge is how to efficiently use this new tier of the memory hierarchy. We believe that flash memory devices are best utilized as persistent caches. We have explored several different attachment points and concluded that a hypervisor's virtual I/O stack represents the best location for such a cache. At this layer, the cache is transparent to applications and operating system software, capable of caching a variety of protocols (NAS, SAN, and local devices), and deployment is simpler than a per-operating system or per-application approach. We have described our experience designing and implementing this cache in the Linux KVM/QEMU hypervisor. Our results demonstrate the benefits of this approach, and evaluate different configurations of our cache with two important macro-benchmarks: Jetstress and an enterprise workload.

VIII. Acknowledgments

We thank Al Andux and Jeffrey Heller for their advice and support. We also thank members of NetApp's Advanced Technology Group (ATG), our anonymous reviewers, and Mohit Saxena for their valuable comments and suggestions.

References

[1] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *SIGMETRICS/Performance*, 2009, pp. 181–192.

[2] F. Corbato, "A paging experiment with the multics system," in *Festschrift: In Honor of P. M. Morse*, 1969, pp. 217–228.

[3] S. Daniel and R. E. Faith, "A portable, open-source implementation of the SPC-1 workload," in *Workload Characterization Symposium*, 2005.

[4] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *ISCA*, 1990, pp. 15–26.

[5] X. He, Q. Yang, and M. Zhang, "A caching strategy to improve iSCSI performance," in *LCN*, 2002, pp. 278–288.

[6] E. V. Hensbergen and M. Zhao, "Dynamic policy disk caching for storage networking," in *IBM Research Report (RC24123)*, 2006.

[7] D. Hitz, J. Lau, and M. A. Malcolm, "File system design for an NFS file server appliance," in *USENIX Winter*, 1994, pp. 235–246.

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, pp. 51–81, 1988.

[9] D. Howells, "FS-Cache: A network filesystem caching facility," in *Proceedings of the Linux Symposium*, 2006.

[10] P. J. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *ISCA*, 1992, pp. 13–21.

[11] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 3–25, 1992.

[12] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Parity lost and parity regained," in *FAST*, 2008, pp. 127–141.

[13] T. Nightingale, Y. Hu, and Q. Yang, "The design and implementation of a dcd device driver for unix," in *USENIX Annual Technical Conference, General Track*, 1999, pp. 295–307.

[14] P. Saab, "Releasing Flashcache in MySQL at Facebook Blog," http://www.facebook.com/note.php?note_id=388112370932 Retrieved April 27, 2010.

[15] M. Saxena and M. M. Swift, "FlashVM: Virtual memory management on flash," in *USENIX Annual Technical Conference*, 2010.

[16] M. Saxena, M. M. Swift, and Y. Zhang, "FlashTier: A lightweight, consistent and durable storage cache," in *EuroSys*, 2012.

[17] D. N. Serpanos, G. Karakostas, and W. H. Wolf, "Effective caching of web objects using zipf's law," in *IEEE International Conference on Multimedia and Expo (II)*, 2000, pp. 727–730.

[18] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield, "Capo: Recapitulating storage for virtual desktops," in *FAST*, 2011, pp. 31–45.

[19] W. Stearns and K. Overstreet, "Bcache: Caching beyond just RAM," in *LWN.net*, July 2, 2010.

[20] D. A. Wheeler, "More than a gigabuck: Estimating gnu/linux's size," http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html Retrieved March 12, 2012.