Reliable Writeback for Client-side Flash Caches

Dai Qin, Angela Demke Brown, and Ashvin Goel, ***University of Toronto***

https://www.usenix.org/conference/atc14/technical-sessions/presentation/qin

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

Reliable Writeback for Client-side Flash Caches

Dai Qin
University of Toronto

Angela Demke Brown
University of Toronto

Ashvin Goel
University of Toronto

Abstract

Modern data centers are increasingly using shared storage solutions for ease of management. Data is cached on the client side on inexpensive and high-capacity flash devices, helping improve performance and reduce contention on the storage side. Currently, write-through caching is used because it ensures consistency and durability under client failures, but it offers poor performance for write-heavy workloads.

In this work, we propose two write-back based caching policies, called write-back flush and write-back persist, that provide strong reliability guarantees, under two different client failure models. These policies rely on storage applications, such as file systems and databases, issuing write barriers to persist their data reliably on storage media. Our evaluation shows that these policies perform close to write-back caching, significantly outperforming write-through caching, for both read-heavy and write-heavy workloads.

1 Introduction

Enterprise computing and modern data centers are increasingly deploying shared storage solutions, either as network attached storage (NAS) or storage area networks (SAN), because they offer centralized management and better scalability over directly attached storage. Shared storage allows unified data protection and backup policies, dynamic allocation, and deduplication for better storage efficiency [2, 8, 16, 20].

Shared storage can however suffer from resource contention issues, providing low throughput when serving many clients [12, 20]. Fortunately, servers with flash-based solid state devices (SSD) have become commonly available. These devices offer much higher throughput and lower latency than traditional disks, although at a higher price point than disks [13]. Thus many hybrid storage solutions have been proposed that use the flash devices as a high capacity caching layer to help reduce contention on shared storage [6, 7, 19].

While server-side flash caches improve storage performance, clients accessing shared storage may still observe high I/O latencies due to network accesses (at the link level and the protocol level, such as iSCSI) compared to clients accessing direct-attached storage. Attaching flash devices as a caching layer on the client side provides the performance benefits of direct-attached storage while retaining the benefits of shared storage [2, 4]. Current systems use write-through caching because it simplifies the consistency model. All writes are sent to shared storage and cached on flash devices before being acknowledged to the client. Thus, a failure at the client or the flash device does not affect data consistency on the storage side.

While write-through caching works well for read-heavy workloads, write-heavy workloads observe network latencies and contention on the storage side. In these commonly deployed workloads [8, 18], the write traffic contends with read traffic, and thus small changes in the cache hit rate may have significant impact on read performance [7]. Alternatively, with write-back caching, writes are cached on the flash device and then acknowledged to the client. Dirty cached data is then flushed to storage when it needs to be replaced. However, write-back caching can flush data blocks in any order, causing data inconsistency on the storage side if a client crashes or if the flash device fails for any reason.

Koller et al. [7] propose a write-back based policy, called ordered write-back, for providing storage consistency. Ordered write-back flushes data blocks to the shared storage system in the same order in which the blocks were written to the flash cache. This write ordering guarantees that storage will be consistent until some time in the past, but it does not ensure durability because a write that is acknowledged to the client may not make it to storage on a client failure.

Furthermore, the consistency guarantee provided by the ordered write-back policy depends on failure-free operation of the shared storage system. The problem is that the write ordering semantics are not guaranteed by the block layer of the operating system [1], or by physical devices on the storage system [9], because the physical devices themselves have disk caches and use write-back caching. On a power failure, dirty data in the disk cache may be lost and the storage media can become inconsistent. To overcome this problem, physical disks provide a cache flush command to flush dirty buffers from the

disk cache. This command enables implementing barrier semantics for writes [22], because it waits until all dirty data is stored durably on media. The ordered write-back policy would need to issue an expensive barrier operation on each ordered write to ensure consistency. In essence, simple write ordering provides neither durability, nor consistency without the correct use of barriers.

We propose two write-back caching policies, called *write-back flush* and *write-back persist*, that take advantage of write barriers to provide both durability and consistency guarantees in the presence of client failures and power failure at the storage system. These policies rely on storage applications (e.g., file systems, applications on file systems, databases running on raw storage, etc.) issuing write barriers to persist their data, because these barriers are the only reliable method for storing data durably on storage media. For example, journaling file systems issue a barrier before committing a transaction, and applications invoke the `fsync(fd)` system call to flush all file data associated with the `fd` file descriptor. Write-back caching policies only need to enforce reliability guarantees at these barriers, since applications receive no stronger guarantees from the storage system. Our caching polices target files that are read and written on a single client, such as files accessed by a virtual machine (VM) running on the client. Thus, we do not consider coherence between client-side caches.

Our two caching policies are designed to handle two different client failure models that we call *destructive* and *recoverable* failure. Destructive failure assumes that the cached data on the flash device is unrecoverable, because either it is destroyed or there is insufficient time for recovering data from the device. This type of failure can occur, for example, due to flash failure or a fire at the client. Recoverable failure is a weaker model that assumes that the client is unavailable either temporarily or permanently, but the cached data on the flash device is still accessible and can be used for recovery. This type of failure can occur due to a power outage at the client.

The write-back flush caching policy is designed to handle destructive failure. When a storage application issues a barrier request, this policy flushes all dirty blocks cached on the client-side flash device to the shared storage system and then acknowledges the barrier. This policy provides durability and consistency because applications are already expected to handle any storage inconsistency caused by out-of-order writes that may have reached storage between barriers (e.g., undo or ignore the effect of these writes). The main overhead of the write-back flush policy, as compared to write-back caching, is that barrier requests may by delayed for a long time, thus affecting sync-heavy workloads.

The write-back persist caching policy is designed to handle recoverable failure. When an application issues a barrier request, this policy persists the in-memory cache metadata to the client-side flash device atomically. The cache metadata consists of mappings from storage block locations to flash block locations; it is needed to locate blocks on the flash device. Durability and consistency are provided by this policy, assuming that the flash device is still available on a failure, because the cache metadata on the device enables accessing a consistent snapshot of data at the last barrier. This policy has minimal overhead on a barrier because persisting the cache metadata to the flash device is a fast operation.

Our evaluation of the two caching policies shows the following results: 1) both policies perform as well as write-back for read-heavy workloads, 2) the write-back flush policy performs significantly better than write-through for write-heavy workloads, even though it provides the same reliability guarantees, 3) the write-back persist policy performs as well as write-back for write-heavy workloads, even though it provides much stronger reliability guarantees, 4) the write-back persist policy has significant benefits as compared to write-through or write-back flush for sync-heavy workloads.

We make the following contributions: 1) we take advantage of the write barrier interface to greatly simplify the design of client-side flash caching policies, providing both durability and consistency guarantees in the presence of destructive and recoverable failure, 2) we discuss various design optimizations that help improve the performance of these policies, and 3) we implement these policies and show that they provide good performance.

The rest of this paper is organized as follows. We discuss prior work on write-back flash caching in Section 2, providing motivation for our work. Section 3 describes our caching policies and Section 4 describes the design of our caching system and the optimizations that improve the performance of our policies. Section 5 shows the results of our evaluation. Section 6 describes related work and Section 7 presents our conclusions and future work.

2 Motivation

Current client-side flash caches use write-through caching [2, 4] because the client and the client-attached flash are considered more failure prone. This caching method also simplifies using existing virtual machine technology since guest state is not tied to a particular client. Write-through caching trivially provides durability and consistency on destructive client failures because storage is always up-to-date and consistent. However, write-through caching by itself doesn't provide any guarantees on storage failures, unless application-issued barriers are honored on the storage side. The main drawback of write-through caching is that it has high overhead for write-heavy workloads.

Next, we discuss two write-back policies that aim to reduce this overhead.

Ordered Write-back Caching Ordered write-back caching flushes data blocks to storage in the same order in which the blocks were written to the flash cache, thus ensuring point-in-time consistency on a destructive failure [7]. This approach does not provide durability because a write that is acknowledged to an application may never reach storage on a destructive failure.

However, durability is a critical concern in many environments. Consider a simple ordering system in which customers place an order and the system stores the order in a file. After confirming the order with the customer, the system persists the contents of the file by invoking the `fdatasync()` system call, and then notifies the customer that the order has been received.

```
int fd = open(...);
...
write(fd, your_order);
fdatasync(fd);
printf("We have received your order.");
```

The `fdatasync()` system call requires writing file contents to storage media durably and thus the file system issues a write barrier. However, with ordered write-back caching, `fdatasync()` is ignored, and data cached on the flash device may not be available on storage after destructive failure. As a result, recent writes may be lost even though the customer is informed otherwise.

Another serious issue with ignoring barriers is that point-in-time consistency can only be guaranteed under failure-free storage operation, since the storage can cache writes and issue them out of order. To avoid this problem, a barrier needs to be issued on every write on the storage side. Thus simple write ordering for ensuring consistency is both expensive, and unnecessary, as described later.

Journaled Write-Back Caching Koller et al. present a second caching policy called journaled write-back that improves performance over ordered write-back by coalescing writes in the cache [7]. Journaled write-back provides point-in-time consistency guarantees at a system-defined epoch granularity. Within an epoch, writes to the same location can be coalesced on the client. All writes within an epoch are written to a write-ahead log (journal) on the storage side, so that data can be committed atomically to storage at epoch granularity.

Although the paper does not mention it, this approach also requires issuing barriers at commit. The system-defined epoch granularity presents a trade-off, with frequent commits affecting performance, and infrequent commits risking more data loss. Furthermore, the system assumes that sufficient NVRAM is available on the storage side to avoid the overheads of journaling.

Unlike either ordered or journaled write-back caching, our write-back policies ensure durability by taking advantage of application-specified barriers. Also, we do not require any journaling on the storage side because applications have no reliability expectations between barriers.

3 Caching Using Barriers

Storage applications that require consistency and durability already implement their own atomicity scheme (e.g., atomic rename, write-ahead logging, copy-on-write, etc.) or durability scheme (e.g., using `fsync`) via write barriers. Our key insight is that write-back caching policies can efficiently provide both durability and consistency by leveraging these application-specified barriers. Since applications have no storage reliability expectations between barriers, the caching policies also only need to enforce these properties at barriers.

We assume that the client flash cache operates at the block layer (i.e., it is below the client buffer cache and independent of it) and caches data for the underlying shared storage system. We now describe the semantics of write barriers, and then describe our caching policies.

3.1 Write Barriers

The block-level IO interface is typically assumed to consist of read and write operations. However, a write operation to storage does not guarantee durability. In addition, multiple write operations are not guaranteed to reach media in order. All levels of the storage stack, including the block layer of the client or the storage-side operating system, the RAID controller, and the disk controller, can reorder write requests. Modern storage systems complicate the block interface further by allowing IO operations to be issued asynchronously and queued [21].

Durability and write ordering are guaranteed only after a cache flush command is issued by a storage application, making this command a critical component of the block IO interface. The cache flush command is supported by most commonly used storage protocols, such as ATA and SCSI, and is widely used by storage-sensitive applications, such as file systems, databases, source code control systems, mail servers, etc.

The cache flush command ensures that any write request that has been *acknowledged* by the device before a cache flush command is issued is durable by the time the flush command is acknowledged. The status of any write request acknowledged after the flush command is issued is unspecified, i.e., it may or may not be durable after the flush. However, the durability of this acknowledged write will be guaranteed by the next flush command.

The flush command enables the implementation of write barriers to ensure ordering and durability [22]. In particular, applications can issue writes concurrently

Policy	Recoverable Client Cache Failure		Destructive Client Cache Failure		Storage Failure	Latency	
	Consistency	Durability	Consistency	Durability	Consistency & Durability	Write	Barrier
Write-through	Yes	Yes	Yes	Yes	Yes[1]	High	Low
Write-back flush	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Low**	**High**
Ordered write-back[2]	Yes	No	Yes	No	No	Low	Low[3]
Write-back persist	**Yes**	**Yes**	**No**	**No**	**Yes**	**Low**	**Low**
Write-back	No	No	No	No	No	Low	Low

Yes[1]: The write-through policy handles storage failure when barriers are supported.
Ordered write-back[2]: Ordered and journaled write-back (proposed in previous work [7]) have the same properties.
Low[3]: Barriers are ignored and hence they don't introduce any additional latency.

Table 1: Comparison of Different Caching Policies

when no ordering is needed. To ensure these writes are all durable, the application waits for these writes to be acknowledged and then issues the cache flush command. When this command is acknowledged, further writes can be issued with the guarantee that they will be ordered after the previous writes.

3.2 Caching Policies

Our caching design is motivated by a simple principle: the caching device should provide *exactly* the same semantics as the physical device, as described in Section 3.1. This approach has two advantages. First, applications running above the caching device get the same reliability guarantees as they expect from the physical device, without requiring any modifications. Second, the caching policies can be simpler and more efficient, because they need to make the minimal guarantees provided by the physical device.

In this section, we present our write-back flush and write-back persist policies. Both policies essentially implement the semantics of the write barrier. The flush policy handles destructive failures in which the flash device may not be available after a client failure, while the persist policy handles recoverable failures in which the flash device is available after a client failure. The choice of these policies in a given environment depends on the type of failure that the storage administrator is anticipating.

3.2.1 Write-Back Flush

The write-back flush policy implements barrier semantics on a cache flush request by flushing dirty data on the flash device to storage. The flush process sends these blocks to storage, issues a write barrier on the storage side, and then acknowledges the command to the client.

Similar to write-through, the write-back flush policy does not require any recovery after failure. Any dirty data cached either on flash or storage since the last barrier may be lost, but it is not expected to be durable anyway. As a result, this policy is resilient to destructive failure.

The main advantage of this approach over write-through caching is that writes have lower latency because dirty blocks can be flushed to storage asynchronously. Moreover, it provides stronger guarantees than vanilla write-through caching because it handles storage failures as well (see Section 2). Compared to write-back caching, barrier requests will take longer with this policy, which primarily affects sync-heavy workloads.

3.2.2 Write-back Persist

The write-back persist policy implements barrier semantics on a cache flush request by atomically flushing dirty cache metadata to the flash device. The dirty file-system blocks are already cached on the flash device, but we also need to atomically persist the cache metadata in client memory to flash, to ensure that blocks can be found on the flash device after a client or storage failure. This metadata contains mappings from block locations on storage to block locations on the flash device, helping to find blocks that are cached on the device.

The write-back persist policy assumes that the flash device is available after failure. During recovery, the cache metadata is read from flash into client memory. The atomic flush operation at each barrier ensures that the metadata provides a consistent state of data in the cache, at the time the last barrier was issued.

The main advantage of write-back persist is that its performance is close to that of ordinary write-back caching. Some latency is added to barrier requests, but persisting the cache metadata to the flash device has low overhead, given the fairly small amount of metadata needed for typical cache sizes. The drawback is that destructive failure cannot be tolerated because large amounts of dirty data may be cached on the flash device, similar to write-back caching. Furthermore, if the client fails permanently, then recovery time may be significant because it will involve moving data from flash using either an out-of-band channel (e.g., live CD), or by physically moving the device to another client.

Operation	Description
`find_mapping`	find a mapping entry in either the clean or the dirty map
`insert_mapping, remove_mapping`	insert or remove mapping in the clean or dirty map
`persist_map`	atomically persist the dirty map to flash
`alloc_block, free_block`	allocate or free a block on flash
`evict_clean_block`	evict a clean block by freeing the block and removing its mapping

Table 2: Mapping and Allocation Operations

Table 1 provides a comparison of the different caching policies. The caching policies are shown in increasing order of performance, with write-through being the slowest and write-back being the fastest caching policy. The write-back flush policy provides the same guarantees as write-through, with low write latency, but with increased barrier latency. The write-back persist policy provides performance close to write-back, but unlike the write-back flush policy, it doesn't handle destructive failure.

4 Design of the Caching System

We now describe the design of our caching system that supports the write-back flush and persist policies. We first present the basic operation of our system, followed by our design for storing the cache metadata and the allocation information. Last, we describe our flushing policy.

4.1 Basic Operation

Our block-level caching system maintains mapping information for each block that is cached on the flash device. This map takes a storage block number as key, and helps find the corresponding block in the cache. We also maintain block allocation and eviction information for all blocks on the flash device. In addition, we use a flush thread to write dirty data blocks on the flash device back to storage. The mapping and allocation operations are shown in Table 2. As we discuss in Section 4.2.2, we separate the mappings for clean and dirty blocks into two maps, called the clean and dirty maps.

Table 3 shows the pseudocode for processing IO requests in our system, using the mapping and allocation operations. On a read request, we use the IO request block number (`bnr`) to find the cached block. On a cache miss, we read the block from storage, allocate a block on flash, write the block contents there, and then insert a mapping entry for the newly cached block in the clean map. On a write request, instead of overwriting a cached block, we allocate a block on flash, write the block contents there, and insert a mapping entry in the dirty map. This no-overwrite approach allows writes to occur concurrently with flushing – a write is not blocked while a previous block version is being flushed. Mappings are updated only after writes are acknowledged to maintain barrier semantics (see Section 3.1).

We avoid delaying read and write requests when the cache is full (which it will be, after it is warm) by only evicting clean blocks from the cache, using the standard LRU replacement policy. The clean blocks are maintained in a separate clean LRU list to speed up eviction. We ensure that clean blocks are always available by limiting the number of dirty blocks in the cache to a `max_dirty_blocks` threshold value. Once the cache hits this threshold, we fall back to write-through caching.

The flush thread writes dirty blocks to storage in the background. It uses asynchronous IO to batch blocks, and writes them proactively so that write requests avoid hitting the `max_dirty_blocks` threshold. The dirty map always refers to the latest version of a block, so only the last version is flushed when a block has been written multiple times. After a block is flushed, it is moved from the dirty map to the clean map. The flush thread waits when the number of dirty blocks reaches a low threshold value, unless it is signaled by the write-back flush policy to flush all dirty blocks (not shown in Table 3).

The write-back flush and write-back persist policies are implemented on a barrier request. The flush policy writes the dirty blocks to storage and waits for them to be durable by issuing a barrier request to storage. The persist policy makes the current blocks on storage durable and persists the dirty map on the flash device, performing the two operations concurrently and atomically.

These policies share much of the caching functionality. Next, we describe key differences in the mapping and allocation operations for the two policies.

4.2 Mapping Information

The mapping information allows translating the storage block numbers in IO requests to the blocks numbers for the cached blocks on flash. We store this mapping information in a BTree structure in memory because it enables fast lookup, and it can be persisted efficiently on flash.

4.2.1 Write-back Flush

The mapping information is kept solely in client memory for the write-back flush policy, because the cache contents are not needed after a client failure, as explained in Section 3.2.1. On a client restart, the flash cache is empty and the mapping information is repopulated on IO requests. Cache warming can help reduce this impact [23].

Read Request	Write Request

`entry = find_mapping(bnr)`
`if (entry):`
`  # cache hit`
`  return read(flash, entry->flash_bnr)`
`else:`
`  # cache miss`
`  data = read(storage, bnr)`
`  if (flash_is_full):`
`  evict_clean_block()`
`  flash_bnr = alloc_block()`
`  write(flash, flash_bnr, data)`
`  insert_mapping(clean_map, bnr, flash_bnr)`
`  return data`
	`entry = find_mapping(bnr)`
	`if (entry):`
	`  free_block(entry->flash_bnr);`
	`  remove_mapping(entry->mapping, bnr);`
	`if (flash_is_full):`
	`  evict_clean_block()`
	`  if nr_dirty_blocks > max_dirty_blocks:`
	`  # fallback to write_through`
	`  write_through();`
	`  return`
	`flash_bnr = alloc_block()`
	`write(flash, flash_bnr, data)`
	`insert_mapping(dirty_map, bnr, flash_bnr)`

Flush Thread	**Barrier Request**

`foreach entry in dirty_map:`
`  # read dirty block from flash`
`  # and write to storage`
`  data = read(flash, entry->flash_bnr)`
`  write(storage, bnr, data)`
`  # move dirty block to clean state`
`  remove_mapping(dirty_map, bnr)`
`  insert_mapping(clean_map, bnr,`
`  entry->flash_bnr)`
	`if (policy == FLUSH):`
	`  signal(flush_thread)`
	`  wait(all_dirty_blocks_flushed)`
	`  barrier()`
	`else if (policy == PERSIST):`
	`  barrier()`
	`  persist_map(dirty_map, flash)`

Table 3: IO Request Processing

4.2.2 Write-back Persist

The mapping information needs to be persisted atomically for the write-back persist policy, as explained in Section 3.2.2. This `persist_map` operation is performed on a barrier, as shown in Table 3. We implement atomic persist by using a copy-on-write BTree, similar to the approach used in the Btrfs file system [17].

Only the dirty mappings need to be persisted to ensure consistency and durability, since the clean blocks are already safe and can be retrieved from storage following a client restart. This option reduces barrier latency because the clean mappings do not have to be persisted. However, persisting all mappings may be beneficial because a client can restart with a warm cache.

We have chosen to persist only the dirty mapping information. To do so, we keep two separate BTrees, called the clean map and the dirty map, for the clean and dirty mappings. The dirty map is persisted on a barrier request; we call this the *persisted dirty map*. Compared to persisting all mappings using a single map, this separation benefits both read-heavy and random write workloads. In both cases, the dirty map will remain relatively small compared to the single map, which would either be large due to many clean mappings, or would have dirty mappings spread across the map, requiring many blocks to be persisted. An additional benefit is that recovery, which needs to read the persisted dirty map, is sped up.

When the flush thread writes a dirty block to storage, we move its mapping from the dirty map to the clean map, as shown in Table 3. This in-memory operation updates the dirty map, which is persisted at the next barrier.

4.3 Allocation Information

We use a bitmap to record block allocation information on the flash device. The bitmap indicates the blocks that are currently in use, either for the mapping metadata or the cached data blocks.

We do not persist the allocation bitmap to flash for several reasons. First, the bitmap does not need to be persisted at all for the write-back flush policy since the cache starts empty after client failure, as discussed in Section 4.2.1. Second, we separate the clean and dirty mapping information and only persist the dirty map for the write-back persist policy. As a result, we would also need to separate the clean and dirty allocation information and only persist the dirty allocation information to ensure consistency of the mapping and allocation information during recovery. Since we read in the dirty map during recovery anyway, which allows us to rebuild the allocation bitmap, this added complexity is not needed.

In the write-back persist policy, the cache blocks that are referenced in the persisted dirty map cannot be evicted even if they are clean, or else corruption may occur after recovery. For example, suppose that Block

A is updated and cached on flash Block F, and then the dirty map is persisted on a barrier. Now suppose Block B is updated and we evict Block A and overwrite flash Block F with the contents of Block B. On a failure, the dirty map would be reconstructed from the persisted dirty map, and so Block A would now map to Block F, which actually contains Block B. We solve this issue by maintaining a second in-memory bitmap, called the persist bitmap. This bitmap is updated on a barrier, and tracks the cache blocks in the persisted dirty map. A block is allocated only when it is free in both the allocation and the persist bitmaps, thus avoiding eviction of blocks referenced in the persisted dirty map.

4.4 Flushing Policies

Section 4.1 describes the basic operation of the flush thread. In this section, we describe two optimizations, flushing order and epoch-based flushing, that we have implemented for flushing dirty blocks.

4.4.1 Flushing Order

Our system supports flushing dirty blocks in two different orders, *LRU order* and *ascending order*. For the LRU order, the dirty blocks are maintained in a separate dirty LRU list. After the least-recently used dirty block is flushed, it is moved from the dirty LRU list to the clean LRU list. We use the last access timestamp to ensure that the flushed block is inserted in the clean LRU list in the correct order. As a result, after a cold dirty block is flushed, it is likely to be evicted soon.

We also support flushing dirty blocks in ascending order of storage block numbers. To do so, we use the dirty map (which stores its mappings in this sort order), as shown in the flush thread code in Table 3. Since the flash device can cache large amounts of data, we expect that flushing blocks in ascending order will significantly reduce seeks on the storage side compared to flushing blocks in LRU order. However, flushing in ascending order may have an affect on the cache hit rate because the flushed blocks may not be the least-recently used dirty blocks. As a result, warm clean blocks may be evicted before cold dirty blocks are flushed and evicted.

4.4.2 Epoch-Based Flushing

In the write-back flush policy, barrier request processing is a slow operation because all the dirty blocks need to be flushed to storage. Suppose an application thread has issued a barrier, e.g., by calling `fdatasync()`, but before the barrier finishes, another thread issues new writes. If barrier processing accepts these writes, it will take even longer to finish the barrier request, and with a high rate of incoming writes, barrier processing may never complete. Alternatively, new writes could be delayed until the completion of the barrier request. However, these writes may also incur high barrier latency, which defeats the goal of using a write-back policy to reduce write latencies.

We can take advantage of the barrier semantics, described in Section 3.1, to minimize delaying new writes, with *epoch-based flushing*. Each cache flush request starts a new epoch, and only the writes acknowledged within the epoch must become durable when the next flush request completes. This flushing of dirty blocks within an epoch requires two changes to the default write-back flush policy. First, the dirty mappings need to be split by epoch. Second, instead of waiting for all dirty blocks in the dirty map to be flushed (as shown in Table 3), the barrier request only waits until all the blocks associated with the dirty mappings in the current epoch are flushed. Since each barrier request starts a new epoch, and barrier processing can take time, multiple epochs may exist concurrently. To maintain data consistency, an epoch must be flushed before starting to flush the next epoch.

We maintain a separate BTree for all concurrent epochs in the dirty map. While epoch-based flushing increases concurrency because writes can be cached on the flash device while blocks are being flushed to storage on a barrier, it also increases the cost of the find and remove mapping operations because they need to search all BTrees. As a result, we have chosen a maximum limit of four concurrent epochs. If new writes are issued beyond this limit, then the writes are delayed.

5 Evaluation

To evaluate our caching policies, we have implemented a prototype caching system using the Linux device mapper framework. This framework enables the creation of virtual block devices that are transparent to the client, so minimal configuration is required to use the system.

Our implementation uses two Linux workqueues, serviced by two worker threads, for issuing asynchronous IO requests to the block layer. The first thread lies in the IO critical path and (i) issues read requests to the flash device on a cache hit or to storage on a cache miss, (ii) issues write requests to flash, and (iii) performs barrier processing, as shown in Table 3. The second thread is only used to issue write requests to flash to insert blocks into the cache following a read miss. We also use a flush thread to write dirty blocks to storage in the background. This thread issues read requests to flash, and write requests to storage. It uses asynchronous IO to batch requests, which helps hide network and storage latencies, thus improving flushing throughput.

Inspired by the journaled write-back policy [7], we implemented a variant of write-back flush called *write-back consistent* that flushes dirty data in each barrier epoch asynchronously. Similar to the flush policy, the consis-

tent policy ensures that data in each epoch is flushed to storage before data in the next epoch. However, the consistent policy acknowledges the barrier operation immediately, without waiting for the flush operation, so it provides consistency but no durability on destructive failure.

We evaluate our write-back flush and persist caching policies by comparing them with four baseline policies, no caching (no flash used), write-through, write-back consistent and write-back caching. Of these, only the persist policy issues barrier requests to the flash device because it needs to persist its mapping atomically to the device. All the policies issue barriers to storage when the application makes a barrier request (see Table 3), with the exception of the write-back policy, which provides no reliability guarantees. Next, we present our experimental methodology and then the performance results.

5.1 Experimental Methodology

Our experimental setup consists of a storage server connected to a client machine with a flash device over 1 Gb Ethernet. The storage server runs Linux 3.11.2, has 4 Intel E7-4830 processors (32 cores in total), 256GB memory and a software RAID-6 volume consisting of 13 Hitachi HDS721010 SATA2 7200 RPM disks. Storage is served as an iSCSI target, using the in-kernel Linux-IO implementation. We disable the buffer cache on the server so that we can directly measure RAID performance, and also because our Linux-IO implementation ignores barriers when the buffer cache is enabled.

The client has 2 Xeon(R) E5-2650 processors and a 120GB Intel 510 Series SATA SSD. We use 8GB of the flash device as the client cache, with 2M mapping entries (1 per 4KB page). Each entry is 16 bytes, which together with the BTree structure, leads to a memory overhead of about 40MB for the mapping information.

We limit the memory on the client to 2GB so that our test data set will not completely fit in the client's buffer cache. In this setup, the ratio of the memory size and flash device capacity is similar to a mid-end, real-world storage server. For example, the NetApp FAS3270 storage system has 32GB RAM and a 100GB SSD when attached to a DS4243 disk shelf [14]. The client runs Linux 3.6.10 and mounts an Ext4 file system in ordered journal mode using the iSCSI target provided by the storage server. Ext4 initiates journal commits (leading to barriers issued to the block layer) every five seconds.

5.1.1 Workloads

We use Filebench 1.4.9.1 [5] to generate read-heavy, write-heavy and sync-heavy workloads on the client. For the read- and write-heavy workloads, barriers are initiated due to Ext4's commit interval. More frequent barriers occur due to application-level sync operations in the sync-heavy workload.

Read-heavy: webserver and webserver-large Webserver consists of several worker threads, each of which reads several whole files sequentially and then appends a small chunk of data to a log file. Files are selected using a uniform random distribution. Overall, webserver mostly performs sequential reads and some writes. We use two versions of this workload: webserver is a smaller version with 4GB of data, which can fit entirely on flash; webserver-large is a larger version, with 14GB of data, which causes cache capacity misses in our experiments.

Write-heavy: ms_nfs and ms_nfs-small ms_nfs is a metadata-heavy workload that emulates the behavior of a network file server. It includes a mix of file create, read, write and delete operations. The directory width, directory depth, and file size distributions in the data set are based on a recent metadata study by Microsoft Research [11]. Similar to webserver, we also use a compact version of ms_nfs, consisting of 6.5GB of data, while the original ms_nfs has 22GB.

Sync-heavy: varmail Varmail simulates a mail server application that issues frequent sync operations to ensure write ordering. For varmail, we use a single, default configuration with 4GB of data, which fits on flash, because a mail server typically does not have a large working set.

5.1.2 Metric

Filebench starts with a setup phase in which it populates a file system before running the workload. During setup, data is cached on flash and flushed in the background. We pause Filebench after setup finishes until the flush thread has stopped flushing data, to avoid interference between the setup phase and the workload. Then, we run each workload for 20 minutes.

Filebench reports the average IO operations/second (IOPS) for a workload at the end of the run. We modified it to report average IOPS every 30 seconds during the run. We found that this IOPS value varies in the first 10 minutes and then stabilizes, due to cache warming effects at both the buffer cache and the flash cache. We present steady-state IOPS results, by averaging the 20 IOPS readings taken in the last 10 minutes of the run.

5.2 Experimental Results

We first present the overall performance results for all the caching policies. We have enabled all flushing optimizations for our write-back policies. We flush in ascending order for both policies, and we use epoch-based flushing with 4 epochs for the write-back flush and write-back consistent policies. Finally, we show the impact of these flushing optimizations.

Figure 1 shows the average IOPS for the different caching policies for the three types of workload. As expected, all write caching policies perform comparably

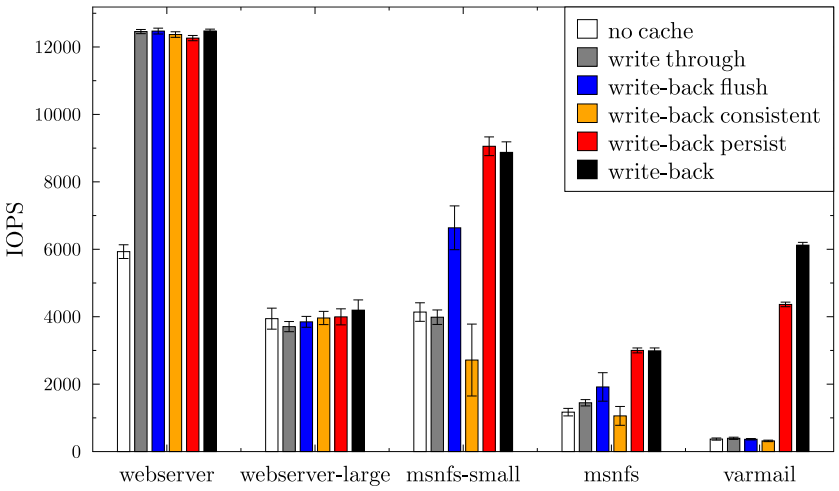

Figure 1: IOPS for different caching policies

for read-heavy workloads. For webserver, the caching policies increase IOPS by more than 2X compared to no-cache because the workload fits in the cache. We found that webserver saturates the flash bandwidth and hence write-back persist performs slightly worse because it needs to persist its mapping table to flash on barriers. In contrast, webserver-large has a low cache hit rate. Thus, it issues many reads to storage, making the storage a bottleneck, and so the no-cache policy performs as well as any caching policy.

The ms_nfs workloads create many dirty blocks and then free them, causing cache pollution and many cache misses, because the caching layer is unaware of freed blocks. Nonetheless, our write-back policies perform well, with write-back persist performing comparably to vanilla write-back because the workloads are write-heavy (84% and 58% write requests in ms_nfs-small and ms_nfs, respectively). With ms_nfs-small, write-through caching performs slightly worse than no-caching because the cache misses require filling the cache, however the difference is within the error bars. With the larger ms_nfs, storage becomes a bottleneck, and hence performance decreases for all workloads. However, this workload has a smaller ratio of write requests than ms_nfs-small and so the performance benefits of write-back caching over write-through caching are smaller.

In ms_nfs-small, the cache hit rate for write-back flush is 52%, while the hit rate for write-back persist is 79%, accounting for the difference in their performance. For write-back flush, the high barrier latency (due to flushing dirty blocks back to storage) causes filesystem journal commits to be delayed since the next transaction cannot commit until the previous one has completed. For ms_nfs-small, only 13 transactions were committed during the 20 minute run. This delay increases the epoch size (we observed a maximum of 4.6GB, with 2GB on average), which leads to dirty blocks occupying a large fraction of the flash cache. Read requests tend to be for clean blocks however, since recently written dirty blocks are more likely to be in the buffer cache, and the reduced number of clean blocks in the flash cache leads to a lower hit rate. We see a similar effect in the ms_nfs workload, although the hit rates are lower in both cases (45% for write-back flush vs. 60% for write-back persist).

Varmail is the most demanding workload for our policies. It issues `fsync()` calls frequently and waits for them to finish, making the workload sensitive to barrier latency. Write-back persist issues synchronous writes to flash and hence has significant overheads compared to the write-back policy. However, persist still performs much better than the other policies that issue synchronous writes to storage. The write-back flush policy performs worse than the write-through policy by 7%, because in our current implementation, the flush thread always performs an additional block read from flash to flush the block to storage.

Contrary to our expectation, the write-back consistent policy, which doesn't provide durability, performs worse than the write-back flush policy for the ms_nfs workloads. These workloads quickly consume all available epochs because each file-system commit issues a barrier that starts a new epoch, but the barriers themselves are not held up by flushing. When no epochs are available, all writes are blocked. We observed that epochs do not become large (as with write-back flush) but writes are frequently blocked due to having no available epochs. Increasing the number of epochs did not significantly improve performance. In contrast, the write-back flush policy delays barriers, but due to this delay it does not run out of epochs. This result suggests that the delay introduced by barrier-based flushing provides a natural control mechanism for avoiding other limits due to cache size, number of epochs, etc.

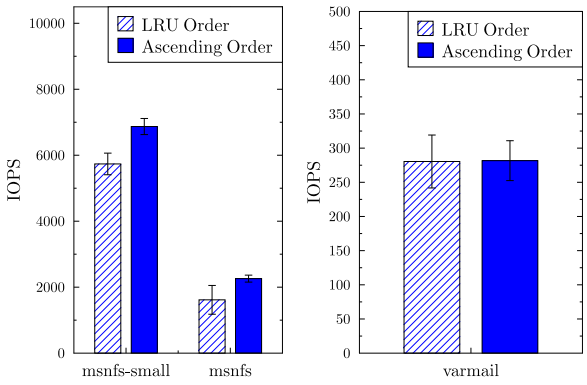

Figure 2: Effect of flushing order in the flush policy

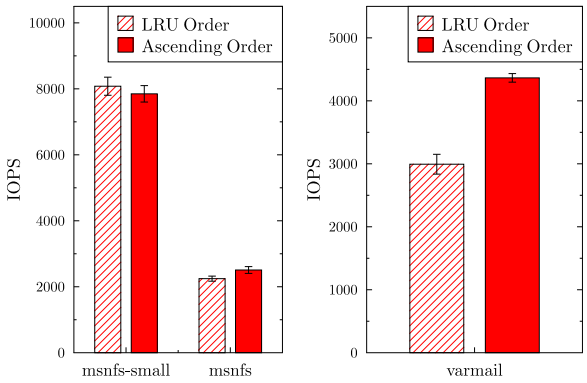

Figure 3: Effect of flushing order in the persist policy

5.2.1 Flushing Order Optimization

In this section, we evaluate the performance of flushing in two different orders, LRU order and ascending order, as described in Section 4.4.1. LRU order is expected to improve cache hit rate, while ascending order is expected to improve flushing throughput. For the read-heavy workloads, flushing is not a bottleneck and hence the flushing order does not affect performance.

Figure 2 shows the effect of flushing order for the write and sync-heavy workloads with the write-back flush policy. In this policy, the flushing operation can become a bottleneck because all the blocks dirtied in the last epoch need to be flushed on a barrier. Compared to LRU order, flushing in ascending order improves performance for the write-heavy workloads, by 19% for ms_nfs-small and 39% for ms_nfs. This result indicates that flushing is the bottleneck for write-heavy workloads and flushing in ascending order reduces disk seeks on storage. The varmail performance is not affected by the flushing order because there are few writes between barriers.

Figure 3 shows the effect of flushing order with the write-back persist policy. We measured the flushing throughput and found that flushing in ascending order performs significantly better than LRU order for both write and sync-heavy workloads. However, ms_nfs-small fits in the flash cache, making flash bandwidth the bottleneck, and hence the flushing order has minimal impact on performance. For ms_nfs, ascending order improves performance due to fewer seeks on storage.

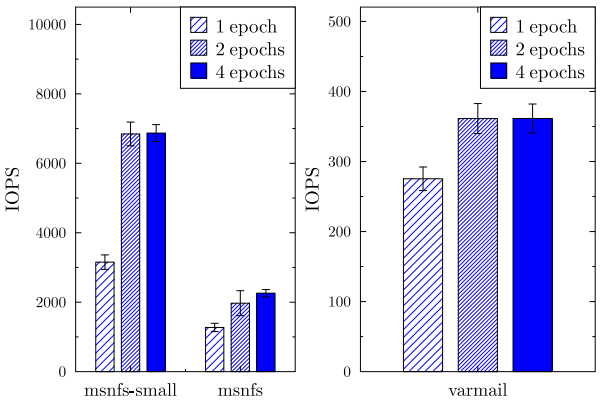

Figure 4: Effect of number of epochs in the flush policy

With varmail, flushing in ascending order helps migrate mapping entries from the dirty map to the clean map much more rapidly due to higher flushing throughput. Since varmail is very sensitive to barrier latency, and there are fewer entries in the dirty map to persist to flash at barriers, ascending order improves performance by 45% over LRU order.

We found that the hit rate did not change significantly when flushing in ascending order versus LRU order for write-back persist. We observed that the flushing operation is relatively efficient for these workloads, and as a result, cold blocks do not remain in the dirty LRU queue for long periods when flushing in ascending order. Once these blocks are flushed, they are moved to the clean queue, and then evicted. To assess the impact on the replacement policy, we logically combined the clean and the dirty mapping queues in timestamp order, and found that the tail of the clean queue is at most 8.5% from the tail of the combined queue in the worst case. As a result, ascending order flushing does not significantly affect the eviction decision and outperforms or is comparable to LRU flushing in all cases.

5.2.2 Epoch-Based Flushing Optimization

The write-back flush policy delays barrier requests because it needs to flush all dirty blocks back to storage. We implemented epoch-based flushing, described in Section 4.4.2, for the write-back flush policy to reduce this impact. We did not implement this optimization for the write-back persist policy because the dirty map can be persisted to the flash device efficiently on a barrier.

Figure 4 shows the benefits of using multiple epochs for flushing data. For both the write and sync heavy both workloads, using a maximum of two epochs improves performance significantly over using a single

epoch. However, performance does not necessarily improve any further with four epochs because there may not be enough write concurrency in the workloads. Also, we keep a separate BTree for each epoch and need to search all the BTrees for the mapping operations, which may have a small impact on performance.

6 Related Work

There have been many recent proposals for using flash devices to improve IO performance, as we discuss here.

Koller et al. [7] present a client-side flash-based caching system that provides consistency on both recoverable and destructive failures. They present ordered write-back and journaled write-back policies, and their evaluation shows that journaled write-back performs better because it allows coalescing writes in an epoch. Unlike our write-back policies, both ordered and journaled write-back do not provide durability because they ignore barrier-related operations, such as `fsync()`. They also ignore disk cache flush commands, and thus do not guarantee consistency on storage failure.

Holland et al. [6] present a detailed trace-driven simulation study of client-side flash caching. They explore a large number of design decisions, including write policies in both the buffer and flash caches, unification of the two caches, cost of cache persistence, etc. They showed that write-back policies do not significantly improve performance. Write-through is sufficient because the caches are able to flush the dirty blocks to storage in time, and thus all application writes are asynchronous. However, their traces do not contain barrier requests (only reads and writes), thus they do not consider synchronous IO operations. Also, their simulation does not consider batching or reordering requests, which offers significant performance benefits, as we have shown.

Mercury [2] provides client-side flash caching that focuses on virtualization clusters in data centers. It uses the write-through policy for two reasons. First, their customers cannot handle losing any recent updates after a failure. Second, a virtual machine accessing storage can be migrated from one client machine to another at any time. With write-through caching, the caches are transparent to the migration mechanism. Mercury can persist cache metadata on flash, similar to the write-back persist policy. However, their motivation for persisting cache metadata is to reduce cache warm up times on a client reboot. Thus the cache metadata is only persisted on scheduled shutdown or reboot on the storage client.

FlashTier [19] presents an interface for caching on raw flash memory chips, rather than on flash storage with a flash translation layer (i.e., an SSD). Their approach benefits from integrating wear-level management and garbage collection with cache eviction, and using the out-of-band area on the raw flash chip to store the mapping tables. FlashTier complements our work because it allows using the flash device more efficiently.

Bcache [15] is a Linux-based caching system that supports caching on flash devices, similar to our system. It implements write-through caching and allows persisting metadata to flash. A comparison of our write-back policies against Bcache would be interesting. Bcache, however, does not support barrier requests in the kernel version that we used for our implementation, so the results would not be comparable.

Previous work on flash caching [6, 7] suggests that the flash cache hit rate, and thus the replacement policy, is crucial to performance. The Adaptive Replacement Cache (ARC) [10] algorithm is effective because it takes access frequency and recency into account, which makes ARC scan-resistant (i.e., it resists cache pollution on full table scans) and helps it adapt to the workload to improve cache hit rates. We have focused on the write-back policy and reliability, and have used only a simple LRU replacement policy.

Bonfire [23] shows that on-demand cache warm up (after system reboot) is slow because of growing flash caches. They warm the cache by monitoring I/O traces and loading hot data in bulk, which speeds up warm up time by 59% to 100% compared to on-demand warm up. We could use a similar approach for warming our cache.

7 Conclusions and Future Work

We have shown that a high-performance write-back caching system can support strong reliability guarantees. The key insight is that storage applications that require reliability already implement their own atomicity and durability schemes using write barriers, which provide the only reliable method for storing data durably on storage media. By leveraging these barriers, the caching system can provide both consistency and durability, and it can be implemented efficiently because applications have no reliability expectations between barriers.

We designed two flash-based caching policies called write-back flush and write-back persist. The write-back flush policy provides the same reliability guarantees as the write-through policy. The write-back persist policy assumes failure-free flash operation, and provides improved performance by flushing data to the flash cache rather than to storage on barrier requests, thereby reducing the latency of the barrier request.

Our evaluation showed three results. First, for read-heavy workloads, all caching policies, write-through or write-back, perform comparably. Second, our write-back policies provide higher performance than write-through caching for bursty and write-heavy workloads because IO requests can be batched and reordered. The dirty

blocks in the flash cache can be batched and flushed asynchronously. They can also be reordered to improve the write access pattern and thus the flushing throughput on the storage side. Third, write-back persist performs comparably to write-back for all workloads, other than sync-heavy workloads, for which it still offers significant performance improvements over write-through caching.

In the future, we plan to use the trim command [3] to reduce cache pollution caused by freed, dirty blocks. We also plan to optimize the flush thread to avoid reading blocks from flash when they are available in the buffer cache. Finally, we plan to evaluate our write-back caching policies in virtualized environments [2].

References

[1] J. Axboe and S. Bhattacharya. `http://lxr.free-electrons.com/source/Documentation/block/biodoc.txt#L826`.

[2] 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 *Proc. of the 28th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, April 2012.

[3] J. Corbet. Block layer discard requests. `http://lwn.net/Articles/293658/`.

[4] EMC. `http://www.emc.com/collateral/hardware/data-sheet/h9581-vfcache-ds.pdf`, 2013.

[5] Filebench. `http://sourceforge.net/apps/mediawiki/filebench`.

[6] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer. Flash caching on the storage client. In *Proc. of the 2013 USENIX Annual Technical Conference*, ATC'13, pages 127–138, June 2013.

[7] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao. Write policies for host-side flash caches. In *Proc. of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 45–58, February 2013.

[8] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Trans. Storage*, 6(3):13:1–13:26, Sept. 2010.

[9] C. Mason. ext3[34] barrier changes. `http://lwn.net/Articles/283169/`, 2008.

[10] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, pages 115–130, Mar. 2003.

[11] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proc. of the 9th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–13, Feb. 2011.

[12] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: scaling down peak loads through I/O off-loading. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 15–28, Nov. 2008.

[13] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: Analysis of tradeoffs. In *Proc. of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 145–158, Mar. 2009.

[14] NetApp. `http://www.netapp.com/us/products/storage-systems/fas3200/fas3200-tech-specs-compare.aspx`.

[15] K. Overstreet. bcache. `http://bcache.evilpiepirate.org/`, 2013.

[16] R. H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. Snapmirror: File-system-based asynchronous mirroring for disaster recovery. In *Proc. of the First USENIX Conference on File and Storage Technologies (FAST)*, pages 117–129, Jan. 2002.

[17] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-tree filesystem. *ACM Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[18] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proc. of the USENIX Annual Technical Conference*, pages 1–14, June 2000.

[19] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: A lightweight, consistent and durable storage cache. In *Proc. of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, Apr. 2012.

[20] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield. Capo: Recapitulating storage for virtual desktops. In *Proc. of the 9th USENIX Conference on File and Storage Technologies (FAST)*, pages 29–43, Feb. 2011.

[21] Tagged command queuing. Wikipedia. `http://en.wikipedia.org/wiki/Tagged_Command_Queuing`.

[22] Write barriers. Fedora Documentation. `http://docs.fedoraproject.org/en-US/Fedora/14/html/Storage_Administration_Guide/writebarr.html`.

[23] Y. Zhang, G. Soundararajan, M. W. Storer, L. N. Bairavasundaram, S. Subbiah, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Warming up storage-level caches with Bonfire. In *Proc. of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 59–72, Feb. 2013.