RELIABLE WRITEBACK FOR CLIENT-SIDE FLASH CACHES

by

Dai Qin

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Enginering
University of Toronto

Abstract

Reliable Writeback for Client-side Flash Caches

Dai Qin

Master of Applied Science

Graduate Department of Electrical and Computer Enginering

University of Toronto

2014

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, because these barriers are the only reliable method for storing data durably on storage media. Our evaluation shows that these policies achieve performance close to write-back caching, while providing stronger guarantees than vanilla write-though caching.

Acknowledgements

I would like to thank my committee members, Professor Amr Helmy, Professor Ding Yuan and Professor Michael Stumm, for their valuable comments and positive feedback. I like to thank to my supervisors, Professor Ashvin Goel and Professor Angela Demke Brown, for their support, patience and encouragement. It was their encouragement made me realize my own potential and value. Without them, I could hardly survive these two years of graduate study. I would like to thank Daniel Fryer, Jack Sun and Dhaval Giani, for their help and the best working environment they provided. Working with them is a joy.

Without my friends and family, I would not have come this far. I want to thank Tianzheng Wang, Wei Huang, Wenbo Dai, Xu Zhao, Yongle Zhang and Junji Zhi, who cheered me up during my hard time. I would like to thank Feifei Chen for her comfort and love. Without her, it would have been much more difficult to finish my study. Finally, I would like to thank my parents, who had always been helpful and supportive to me, no matter what happened.

Contents

List of Tables

List of Figures

Chapter 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 simpler, centralized management and better scalability over directly attached storage. Shared storage allows unified data protection and backup policies, storage to be allocated dynamically, and deduplicated for better storage efficiency [4, 12, 24, 20].

Shared storage can however suffer from resource contention issues, providing low throughput when serving many clients [24, 16]. 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 [17]. Thus many hybrid storage solutions have been proposed that use the flash devices as a high capacity, caching layer to help reduce contention to shared storage [11, 23, 8].

While server-side flash caches improve storage performance, clients accessing shared storage may still observe high I/O latencies due to network accesses (both at the link level and the protocol level (e.g., iSCSI)) as 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 systems [4, 6]. These systems use write-through caching because it simplifies the cache consistency model. All writes are sent to shared storage and cached on flash devices before being acknowledged to the client. As a result, any failure at the client or the flash device does not affect consistency on the storage side.

While write-through caching works well for read-heavy workloads, write-heavy workloads, which are commonly deployed [12, 22], observe network latencies and contention on the storage side. Furthermore, write traffic contends with the read traffic, and thus small changes in the cache hit rate may have significant impact on read performance [11]. 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. [11] propose a write-back based policy, called ordered write-back, for providing storage consistency. Ordered write-back flushes data blocks to storage 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 point in the past. However, it does not ensure durability, because on a client failure, a write that is acknowledged to the client may never make it to storage.

Furthermore, the consistency guarantee provided by the ordered write-back policy currently 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 [10] or by physical devices on the storage system [13]. The reason is that the physical devices themselves have disk caches and uses 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 special cache flush command to flush dirty buffers from the disk cache. This command enables implementing barrier semantics for writes [2], 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 discuss this issue in more detail in Chapter 2.

In this work, we propose two simple 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 running 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.

The 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 either destroyed, or is unavailable, or there is insufficient time for recovering data from the flash 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 has crashed or is unavailable either temporarily or permanently, but the cached data on the flash device in still accessible and can be used for recovery. This type of failure can occur, for example, due to a power outage at the client.

The write-back flush caching policy is designed to handle destructive failure. When an application issues a barrier request, e.g., by calling the `fsync(fd)` system call, this policy simply flushes all dirty blocks cached on the flash device to storage and then acknowledges the barrier. This policy provides durability and consistency because applications are expected to handle any storage inconsistency caused by out-of-order writes that 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 atomically persists the cache metadata in memory to the flash device. The cache metadata, consisting of mappings from storage block locations to flash block locations, helps find blocks on the flash device. This policy provides durability and consistency because the flash device is still available on a failure, and the cache metadata on the device enables accessing a consistent snapshot of the cached data at the last barrier request. 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.

The contributions of this work are the following: 1) we take advantage of the write barrier interface to greatly simplify the design of client-side flash caching policies, 2) these policies provide both durability and consistency guarantees in the presence of destructive and recoverable failure, 3) we discuss various design optimizations that help improve the performance of these policies, and 4) we implement these policies and show that they provide good performance.

The rest of this thesis is organized as follows. Chapter 2 discusses previous work on write-back flash caching in more detail, providing motivation for this work. Chapter 3 describes our caching policies and then Chapter 4 describes the design of our caching system and the various optimizations that improve the performance of our policies. Chapter 5 provides details about our implementation, and Chapter 6 shows the results of our evaluation. Chapter 7 describes related work and Chapter 8 presents our conclusions and future work directions.

Chapter 2

Motivation and Background

Storage applications that require durability and consistency already implement their own atomicity scheme (e.g., atomic rename, write-ahead logging, copy-on-write, etc.) or durability scheme (e.g., using `fsync`) using write barriers. The key insight in this work is that write-back caching policies that leverage these application-specified barriers can provide both durability and consistency, and they can be implemented efficiently because they can take advantage of applications having no storage reliability expectations between barriers.

Next, we motivate our approach by providing background on existing caching policies and then provide background on write barriers.

2.1 Current Caching Policies

Current client-side flash caches uses write-through caching [4, 6] 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 server. Write-through caching trivially provides durability and consistency on destructive 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 write-back policies that aim to reduce this overhead.

2.2 Ordered Write-back Caching

As mentioned earlier, 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 [11]. 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."
       "Your order will be delivered soon");
```

The `fdatasync()` system call requires writing file contents to storage media durably and thus the file system issues a write barrier request. However, with ordered write-back caching, `fdatasync()` would be 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 would have to be issued on every write on the storage side. Thus simple write ordering for ensuring consistency is both expensive, and unnecessary (as shown later with our polices).

2.3 Journaled Write-Back Caching

Koller et al. present a second caching policy called journaled write-back that improves performance over ordered write-back. This policy allows coalescing of writes in the cache, thus improving cache utilization and reducing network traffic [11]. Journaled write-back provides point-in-time consistency guarantees at a system-defined epoch granularity. This approach allows writes to the same location within an epoch to be coalesced on the client side. All writes within an epoch are then 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 the journaled write-back, 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 consistency or durability expectations between barriers.

2.4 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. 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.

Modern storage complicates the block interface further by allowing IO operations to be issued asynchronously and queued in hardware [1]. Next, we describe the semantics of the cache flush command because it helps implement write barriers.

1. Any write request that has been *acknowledged* by the device before a cache flush command is issued is guaranteed to be durable by the time the command is acknowledged.

2. The behavior 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 write will be guaranteed by the next flush command.

Based on this behavior, we define epochs that start when a cache flush command is issued. We fix the unspecified behavior described above by ensuring that all writes *issued* before the cache flush command become durable. As a result, writes issued within an epoch may be reordered, but they are not reordered across epochs. Writes within an epoch become durable when the cache flush command issued at the end of the epoch is acknowledged. We call the start of each epoch a write barrier.

Chapter 3

Caching Policies

Our caching design is motivated by a simple principle, that the caching device should provide exactly the same interface as the physical device, as described earlier in Chapter 2.4. 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 chapter, 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.

3.1 Write-Back Flush

The write-back flush policy implements the write barrier semantics by flushing dirty data to storage. On a cache flush command, this policy flushes all dirty blocks on the flash device. 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 on the flash device, or on the storage side, 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 Chapter 2.1). As compared to write-back

Policy	Recoverable Failure		Destructive 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 have the same properties. These policies were proposed in previous work [11]. Low[3]: Barriers are ignored and hence they don't introduce any additional latency.

Table 3.1: Comparison of Different Caching Policies

caching, this policy will delay barrier requests, thus primarily affecting sync-heavy workloads.

3.2 Write-back Persist

The write-back persist policy implements the write barrier semantics by atomically flushing dirty data to the flash device. The dirty blocks are already cached on the flash device. However, to ensure that they can be found on the flash device after a failure, we need to atomically persist cache metadata in client memory to the flash device. This metadata consists of mappings from block locations on storage to block locations on the flash device, helping find blocks 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 helps present a consistent state of data in the cache, at the time the last barrier was issued.

The main advantage of this approach is that its performance approaches write-back performance. The barrier request adds some latency, but given the size of the flash device, the cache metadata is fairly small, and thus persisting it to the flash device has low overhead. Similar to write-back, the main drawback is that large amounts of data may be cached on the flash device, and so destructive failure cannot be tolerated. Furthermore, if the client fails permanently, then recovery time may be significant because it will involve moving data from the flash device using either an out-of-band channel (e.g., live CD), or by physically moving the device to another client.

Table 3.1 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.

Chapter 4

Design of the Caching System

In this chapter, we describe the design of our caching system for supporting the write-back flush and persist policies. We start by presenting the basic operation of our system. Then we describe our design for storing the cache metadata and the allocation information. Finally, we describe our flushing policy.

4.1 Basic Operation

The design of our block-level caching system is shown in Figure 4.1. We maintain 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 to storage in the background.

The mapping and allocation information operations are shown in Table 4.1. As discussed later in Chapter 4.2.2, we separate mappings for clean and dirty blocks into two maps, called the clean and dirty maps. Table 4.2 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, and write the block contents there. We also insert a mapping

Operation	Description
`find_mapping`	find a mapping entry in either clean and 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 4.1: Mapping and Allocation Operations

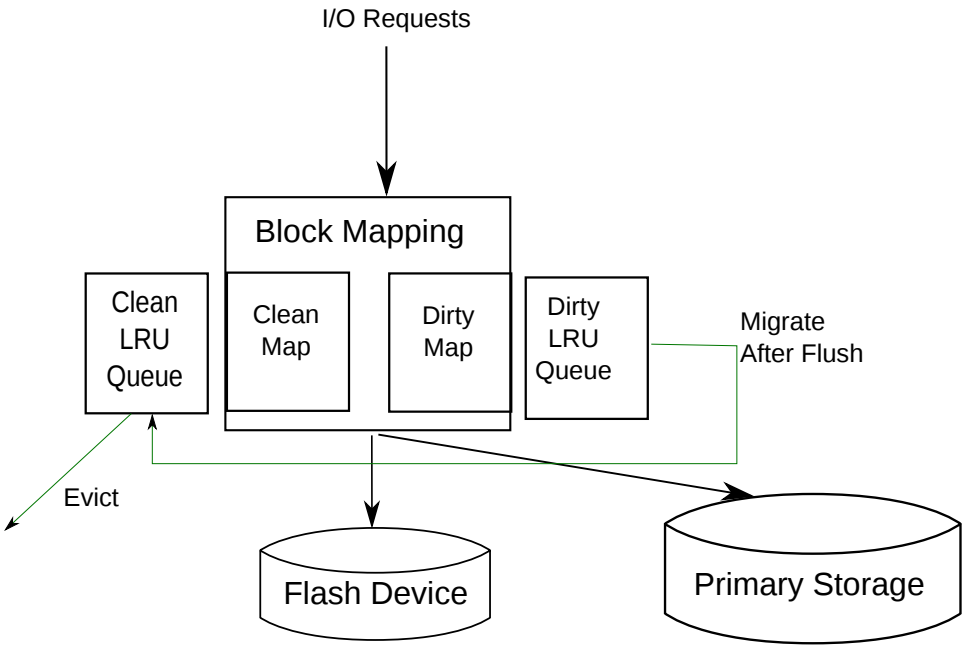


Figure 4.1: Design of the Caching System

entry for the newly cached block in the clean map.

On a write request, instead of overwriting a cached block, we allocate a new block on flash, write the block contents there, and insert a mapping entry in the dirty map. With this no-overwrite approach, writes can occur concurrently with flushing. In particular, a write is not blocked while a previous version of the block is being flushed, which also simplifies our design.

We avoid delaying read and write requests when the cache is full (the cache will be full after it has been in use for some time) by only evicting clean blocks from the cache. Currently, we use the standard LRU replacement policy for evicting clean blocks. 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 the `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 pro-actively so that write requests avoid hitting the `max_dirty_blocks` threshold. After writing a block, it moves the block from the dirty map to the clean map. The flush thread waits when the number of dirty blocks reaches a `min_dirty_blocks` threshold value (not shown in Table 4.2), unless it is signaled by the write-back flush policy to flush all dirty blocks.

The write-back flush and write-back persist policies are implemented on a barrier request. The flush policy waits for all the dirty blocks to be stored durably on storage, while the persist policy atomically persists the dirty map on the flash device. These policies share much of the caching functionality. The next two sections describe

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()`
`  insert_mapping(clean_map, bnr, flash_bnr)`
`  write(flash, flash_bnr, data)`
`  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()`
	`insert_mapping(dirty_map, bnr, flash_bnr)`
	`write(flash, flash_bnr, data)`

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)`
	`} else if (policy == PERSIST) {`
	`  persist_map(dirty_map, flash)`
	`}`

Table 4.2: IO Request Processing

certain differences in the mapping and allocation operations, shown in Table 4.1, 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 data structure for several reasons. First, the BTree provides fast lookup when large numbers of entries are present. Second, it can be persisted efficiently on flash. Third, it naturally sorts entries by key, which can be used to improve flushing performance. Next, we discuss these issues.

4.2.1 Write-back Flush

The mapping information is kept entirely in client memory for the write-back flush policy, because the cache contents are not needed after a client failure, as explained in Chapter 3.1. On a client restart, the flash cache is

empty, and the mapping information is repopulated on IO requests. Thus a client restart requiring warming the cache [26].

4.2.2 Write-back Persist

The mapping information needs to be persisted atomically for the write-back persist policy, as explained in 3.2. This `persist_map` operation is performed on a barrier, as shown in Table 4.2. We implement atomic persist by using a copy-on-write BTree, similar to the approach used in the Btrfs file system [21].

Separating Clean and Dirty Mappings

One option for persisting the mapping information is to persist all mappings. The benefit of this approach is that a client restart will not affect performance because the cache will be warm. A second option is to persist the mappings for dirty blocks only. This option works because only the dirty blocks need to be flushed to storage. The clean blocks can be retrieved from storage on a client restart. The benefit of this option is that it reduces barrier latency because the clean mappings do not have to be persisted.

We have chosen to persist the dirty mapping information only. 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, and we call it 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 mapping spread across the map, requiring many blocks to be persisted. An additional benefit is that recovery, which needs to read the persisted dirty map (See Chapter 3.2), 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 4.2. This is an in-memory operation, but it does update the dirty map, which will then be persisted at the next barrier.

Journal for Copy-on-Write BTree

The copy-on-write BTree data structure updates a block by making a copy of the block. The parent block pointing to the original block needs to be updated to point to the new copy of the block. As a result, a copy is made of the parent block as well. This copying continues until a copy of the root block is made. On a barrier, all the modified (copy) blocks need to be persisted. This can affect the performance of sync-heavy workloads because the updated dirty map blocks could generate many more writes than the writes for the dirty data blocks. This problem is solved by combining a journal with the copy-on-write data structure (e.g., log-tree in Btrfs [21], ZIL in ZFS [3]).

We have adopted a similar journal-based solution for the dirty map, and we allocate a static region on flash for the journal. When an entry is inserted in the dirty map, we also append an entry in the journal. On a barrier request, unless the journal is full, we persist either the journal or the BTree, depending on whichever one has fewer dirty blocks. If we choose to persist the BTree and the journal is not empty, then we need to clear the journal by reinitializing the current journal entry pointer to the beginning of the journal.

4.3 Allocation Information

We use a bitmap to record block allocation information on the flash device. The bitmap indicates that 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 on client failure, as discussed in Chapter 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 need to separate out the clean and dirty allocation information and only persist the dirty allocation information to ensure consistency of the mapping and allocation information during recovery. This complexity is not needed, because during recovery, we read in the dirty map anyway, which allows rebuilding the dirty allocation information.

One complication with the write-back persist policy is that 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. Say 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 map to Block F, which contains Block B.

We solve this write-back persist 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 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

The basic operation of the flush thread is described in Chapter 4.1. 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 order. The flush thread shown in Table 4.2 is flushing dirty blocks in ascending order. 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 default write-back flush policy, barrier request processing is a slow operation because all the dirty blocks need to be flushed to storage. Suppose a thread of an application has issued a barrier, e.g., by calling `fdatasync()`, but before the barrier finishes, another thread issues new writes. If the barrier processing accepts these writes, it will take even longer to finish the barrier request. In the worst case, new writes may be issued at the same rate as dirty blocks are flushed, and the barrier processing would 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 goes against our goal of reducing write latencies using a write-back policy.

We can take advantage of the barrier semantics, described in Chapter 2.4, to minimize delaying new writes. Each cache flush request starts a new barrier epoch, and writes issued within the epoch must become durable when the flush request issued at the end of the epoch completes. This flushing of dirty blocks within an epoch (epoch-based flushing) requires two changes to the default write-back flush policy. First, the dirty mappings are split by epoch. Second, instead of waiting for all dirty blocks in the dirty map to be flushed (as shown in Table 4.2), the barrier request only waits until all 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 to limit the maximum number of concurrent epochs. If new writes are issued beyond this limit, then the writes are delayed.

Chapter 5

Implementation

We implemented a prototype caching system using the Linux device mapper framework. This framework is a Linux kernel module that enables creating virtual block devices. The virtual block device is transparent to the storage client so minimal configuration is required to use the system.

The device mapper framework is a block layer framework that allows intercepting block IO requests and implementing the IO request processing shown in Table 4.2. After we receive an IO request, we split the large block requests into 4KB blocks, and issue them to the flash device or to the storage device, as shown in Figure 5.1. Next, we describe our implementation in more detail.

5.1 Mapping and LRU Information

The copy-on-write BTree for the dirty map is implemented using the `dm-bufio` device mapper module, which provide a simple buffer manager in kernel space. Since the dirty map is fairly small, I/O accesses to the map are generally cached in memory. On a barrier request, the copy-on-write BTree is persisted to the flash device, as

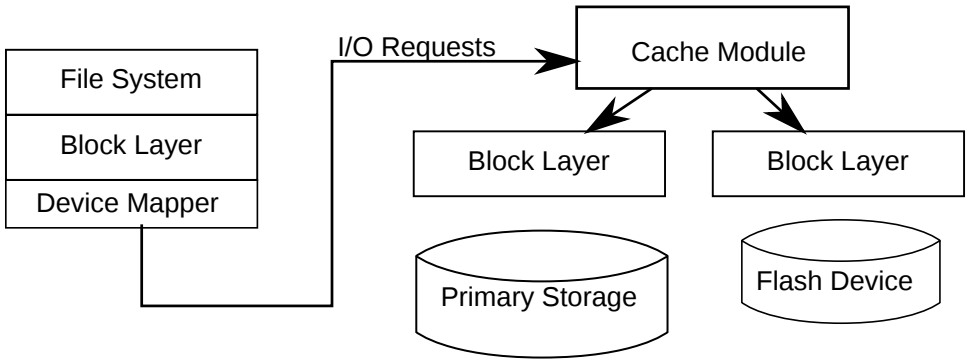


Figure 5.1: Block Layer Caching Implementation

explained in Section 4.2.2.

We use separate LRU lists for the clean and dirty blocks. This allows fast eviction of clean blocks, which happens on each cache miss, since the flash device is generally full after the cache has been warmed (See Table 4.2). Our implementation uses a priority queue sorted by a last access timestamp. When the flush thread removes a block from the dirty LRU list, it places it in the order correct in the clean LRU list, so that logically, the two lists form a single global LRU list.

We associate a block header for each block on the flash device on startup. This avoids any memory allocation requests after startup, and it doesn't waste memory because there are rarely any free blocks on the device. The block header maintains LRU information, IO status (e.g., whether read or write request is pending), and a reference count. The reference count ensures that a block in use is not freed. A block can be referenced by 1) the clean or the dirty map, 2) the flush thread, and 3) the IO request processing thread, as explained next.

5.2 IO Request Processing

While the flash device has lower latency than storage, its latency is still considerably higher than DRAM latency. In our environment, the latency of a sequential read to the flash device is roughly 70 μs, while the latency of a sequential read to storage is roughly 460 μs. To reduce IO latency, we can use asynchronous IO or multi-threaded IO. We have chosen to use single-threaded asynchronous IO processing, because it scales well with the number of concurrent IO requests, without requiring tuning of the number of concurrent worker threads. In particular, we use a single worker thread, associated with a Linux workqueue, for issuing asynchronous IO requests to the block layer. This thread lies in the IO critical path, issuing read requests to the flash device on a cache hit or to storage on a cache miss, write requests to flash, and performs barrier processing, as shown in Table 4.2.

We 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.

For simplicity, a single global lock protects our data structures, i.e. the clean and dirty maps and the clean and dirty LRU lists. Interestingly, we found that if we associated multiple worker threads with the workqueue, and used all of them to issue the asynchronous IO requests, then they would queue on the lock, delaying user programs that were issuing IO requests, causing a significant drop in performance. We plan to revisit this implementation if it becomes a bottleneck for higher-throughput flash devices.

Operation Type	Key	Value
Low 12 bits	**13:64 bits**	**64:128 bits**
Insert	key » 12	value
Delete	key » 12	—
Commit	—	generation number

Table 5.1: Journal Entry Format

5.3 Journal for Copy-on-Write BTree

Recall from Section 4.2.2 that we use a journal, in addition to the copy-on-write BTree, for persisting the dirty mapping information. The journal size is 4MB in our implementation. We use a logical journal because it reduces the size of the journal entries. Each logical journal entry contains 16 bytes: a flag indicating the name of the operation, a key of the mapping (block number), and a value of the mapping (block number on flash device). Table 5.1 shows this format. To ease implementation, every entry is aligned to 16 bytes.

Similar to Btrfs[21], we add a generation number, a 64 bit monotonically increasing counter indicating the barrier epoch, in the root block of the COW BTree. When we commit the COW BTree to the flash device, we persist the current generation number as well, and then increment the generation number by one. This generation number indicates the last epoch when the dirty map was persisted.

Table 5.1 shows that we also commit this generation number in the logical journal. In this way, we can determine whether the journal or the BTree has the latest dirty mapping information. In particular, during recovery, we only replay the journal entries if the generation number in the journal is greater than the generation number in the COW BTree. This ensures idempotent replay, so that any failure during recovery is handled correctly.

Chapter 6

Evaluation

We now evaluate the performance of our client-side flash cache designs, comparing them to write through, write back, and no cache baselines. After presenting overall results, we analyze the effects of several optimizations. Our primary metric is average IOPS, however we also consider average request latency.

6.1 Setup

Our experimental setup consists of a primary storage server and a storage client equipped with a flash cache. The primary storage server is equipped with 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. The primary storage is served as an iSCSI target using Linux 3.11.2's LIO in-kernel iSCSI implementation with buffer disabled. The storage client runs Linux 3.6.10 and has 2 Xeon(R) E5-2650 processors and a 120GB Intel 510 Series SSD. We only use 8GB of the Flash device as the client cache in our experiments. We limit the memory on the storage client to 2GB using a boot parameter, so that our test data set will not fit in storage client buffer cache. The storage server and client are connected by 1 Gbps Ethernet.

Our client setup is configured so that the size of memory and Flash device are roughly proportional to a mid-end real-world storage server. For example, the FAS3270[18] from Netapp has 32GB RAM and a 100GB SSD when attaching a DS4243 shelf.

6.1.1 Workload

We evaluate the performance of four write policies: write through, write-back flush, write-back persist, and write-back. We include a baseline no-cache policy for comparison (which is just the primary storage without any Flash

device cache). The write-back policy is similar to write-back persist, except that it ignores the barrier requests, which is unsafe to use in practice. Write-Back is included to show the performance cost of persisting the cache metadata (i.e., the mapping table) on a barrier.

We use Filebench 1.4.9.1 to generate workloads on the storage client. We run Filebench with the following five different workloads:

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 consists of fairly sequential reads and a very small amount of write.

We use two versions of this workload: webserver is a smaller version with only 4GB of data, which can fit entirely onto Flash device; webserver-large is a larger version, with around 14GB of data, which will cause cache capacity misses during our experiments.

ms_nfs and ms_nfs-small

ms_nfs is a metadata-intensive and write-intensive 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[15] by Microsoft Research.

Similar to webserver, we also have a compact version of ms_nfs. ms_nfs-small consists of only 6.5GB of data, while the original ms_nfs is around 22GB. In ms_nfs-small, 84% of requests are writes, making this a write-intensive workload. In the full-sized ms_nfs, only 58% of requests are write, which is still fairly write-intensive, although somewhat more balanced than the ms_nfs-small workload.

We also note that ms_nfs-small is a fairly dynamic workload: it keeps creating new files and deleting old files, and the file system keeps writing to newly allocated blocks and then freeing them. However, from the caching layer's perspective, this is similar to polluting the cache with dirty blocks that are never used, because the caching layer is not aware of blocks that are freed by the file system. This could lead to cache misses in ms_nfs-small when the Flash device becomes filled with these polluting blocks.

varmail

Varmail simulates a mail server. A mail server application usually requires very strong write ordering, and therefore the mail server issues `fsync()`calls quite frequently. Between each `fsync()`, there is a very small number of writes and very few reads. For varmail, we only use the default configuration with around 4GB of data, which can

fit entirely on Flash device, because a mail server does not typically have a huge active working set in practice,

6.1.2 Methodology and Metrics

Our primary evaluation metric is average IOPS, as reported by Filebench. However, we observe that it typically takes some time for the IOPS to stabilize after starting an experiment. Initially we may see both higher and lower IOPS than the eventual stable behavior in different settings. This variability happens for two reasons: (1) the buffer cache on the storage client could affect the performance; and (2) the Flash cache needs to be warmed up. As a result, we show graphs of the average IOPS in each 30 second interval during the experiments. We expect the steady-state behavior to be more representative of the performance in a real-world deployment. Thus, we use the IOPS after the workload stabilizes to calculate the overall average IOPS for each experiment. To be precise, we use only the last 10 minutes of each run, which contains the average IOPS from twenty 30-second intervals.

To make our IOPS number stabilize faster, we use the following procedure to run the experiments. First, we use Filebench to create the files needed to run the benchmark on the storage client with the cache module. Some data will be cached on the Flash device during this create phase. Second, we pause Filebench and wait for the flush thread to stop, which means either that it has flushed every dirty block on the Flash device in the case of write-back flush, or that it has gone under the lower dirty block threshold in the case of write-back persist. This pause ensures that when the experiment starts to run, there is no heavy background IO traffic from the flush thread. Finally, we run the experiments and continuously collect the average IOPS number for every 30 second interval.

6.2 Raw Hardware Performance Numbers

To help understand the performance of our caching solution, we first evaluate the raw performance of the hardware we have in our experimental environment. Centralized storage usually has larger capacity but higher latency, especially using iSCSI, while the locally attached Flash device should have much lower latency compared to primary storage.

We use FIO[9] benchmark to test the raw capability of our hardware. FIO is a block level benchmark and it bypasses the buffer cache. All tests were performed under ext4 file system. In Table 6.1 and Table 6.2, we show the average throughput and latency. Peak write throughput of the primary storage is only 114.8MB/s on average, while the Flash device achieves three times higher write throughput.

Our latency for sequential access to the primary storage is a bit high, because the link we are using is 1 Gbps Ethernet. We also found high latency for random read and write access, which we believe is due to the high latency of disk seeks. For the cache on the storage client, we are using an Intel 510 Series SSD. According to the Intel specifications, this device has around 65 μs read latency and 85 μs write latency. We measured around 70

Device	Sequential Read	Random Read	Sequential Write	Random Write
Primary	459.85 μs	6869.61 μs	669.38 μs	15682.55 μs
Flash device	70.87 μs	204.37 μs	81.26 μs	94.17 μs

Table 6.1: Raw Latency of Hardware on Average

Device	Sequential Read	Random Read	Sequential Write	Random Write
Primary	114.8 MB/s	6.2 MB/s	111.1 MB/s	0.78 MB/s
Flash device	406.1 MB/s	78.1 MB/s	198.1 MB/s	61.7 MB/s

Table 6.2: Raw Throughput of Hardware on Average

μs for sequential read and 81 μs for write, but with higher latency for random read performance. We believe the internal flash translation layer (FTL) in the Intel SSD is responsible for the reduced random read performance, as previous research[5] has found similar performance characteristics.

6.3 Experimental Results

6.3.1 Average IOPS After Stabilization

We compare write-back flush and write-back persist policy against no-cache and write through policy here. We enabled all optimizations mentioned previously: flushing in ascending block number, using multiple epochs (with a maximum of 4 epochs), and using a logical journal for fast commit.

The average IOPS (during the last 10 minutes of each run) for each write policy are shown in Figure 6.1.

Unsurprisingly, unsafe write-back has the best performance for all five workloads, since it does not make any guarantees about the consistency or durability of the data. Of the other policies, write-back persist has the best performance in four out of the five workloads, coming close to write-back in all cases except varmail. This result shows that persisting the cache metadata to Flash device at barriers does not incur a significant performance penalty, except when barriers are occur with high frequency. The write-back flush policy, which maintains the same reliability as write through, achieves significantly higher IOPS than write through in two out of five workloads, and matches the performance of write through for another two workloads. The write through policy, which a lot of previous systems have been built on, only out-performs no-cache for two of the five workloads.

Next, we take a closer look at how these policies compare on different workloads.

First, webserver is a small, read intensive workload that completely fits into the Flash cache. As a result, all caching policies boost the performance significantly compared to no-cache. Since there is a very small amount of write traffic, the choice of write through or write back with any of the write-back policies does not make a significant difference. We do see a tiny performance decrease in write-back persist, however. This might be

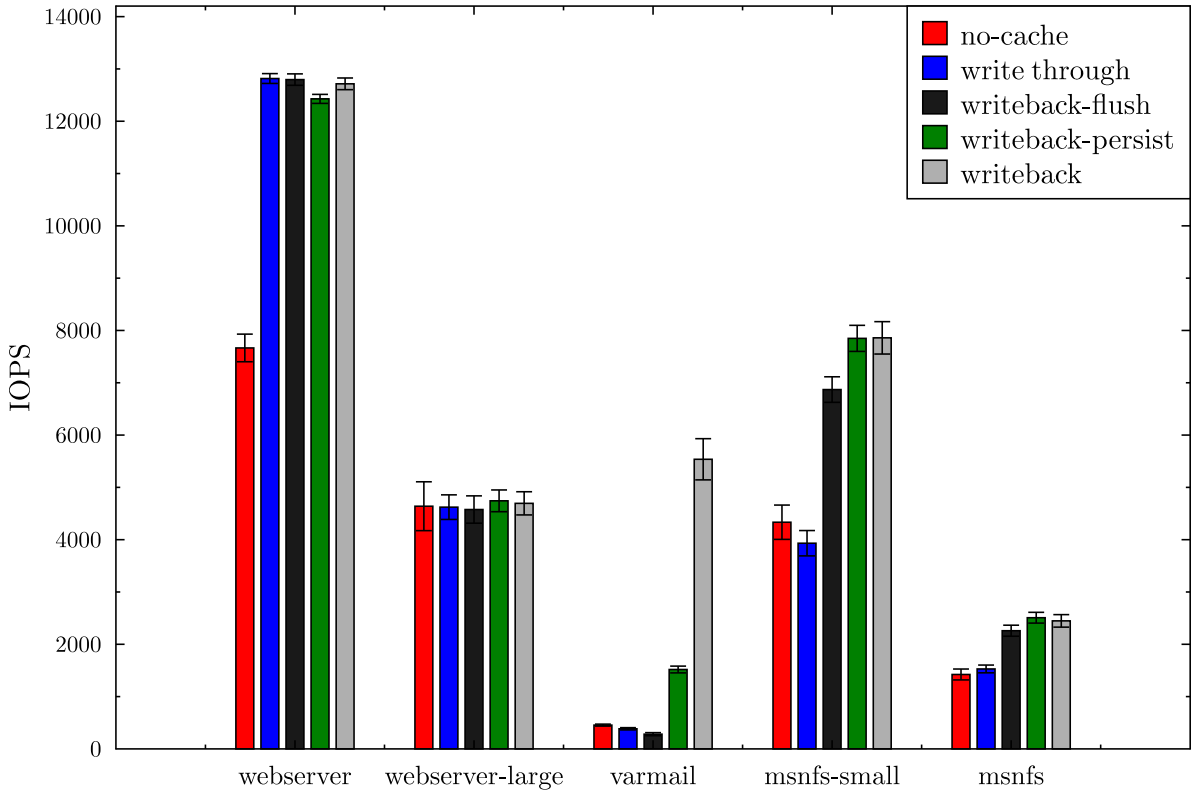


Figure 6.1: Average IOPS After Stabilization

because of the cost of extra writes to the Flash device to persist the mapping table. In the experiments, we found that the bandwidth to Flash device is saturated, and therefore, extra writes to Flash device could affect the foreground read performance by a small amount.

In contrast, webserver-large contains a much larger data set. It is still read intensive, but with heavier write traffic compared to webserver. More importantly, it is saturating on the primary storage: the random file selection causes frequent cache misses on the Flash device and heavy seeks on the primary storage. In this workload, only write-back persist can get marginally better performance by buffering writes to the Flash device. Neither write-back flush nor write through boosted performance compared to no-cache, because a low hit rate on the Flash device cache combined with heavy seeks led to the primary storage becoming the bottleneck.

The varmail workload is unusual because it is issuing sync requests very frequently. We found that in varmail, unsafe write-back showed significantly better performance than write-back persist because varmail is very latency sensitive. At each sync, varmail must wait for the barrier to finish. Thus, the latency to process a disk barrier becomes the bottleneck. In write-back, barriers are ignored completely, while write-back persist has to persist the mapping entry, which involves writing to Flash device.

Between each of these frequent syncs in varmail, there is a very small amount of random writes. These

characteristics make optimizations done by write-back flush useless: varmail is issuing sync frequently, so there is not enough time to finish flushing dirty blocks in the background; each barrier dirty epoch is very random and small, so write reordering or write coalescing within a dirty epoch is useless. We also flush a batch of dirty blocks to the primary storage in our flush thread, but because the dirty epoch of varmail is so small that can fit into the buffer cache, buffer cache is also able to batch the write to the primary storage when using no-cache or write through. This means that batching in our flush thread does not bring any benefit to varmail.

In fact, because the write-back flush policy has to write to Flash device first, and then flush thread needs to read back from Flash device and write to the primary storage, this extra turn-around actually decreases the performance. In Figure 6.1, the average IOPS achieved by write-back flush is decreased by 26% compared to write through.

ms_nfs-small is a write-intensive but dynamic workload. As we mentioned earlier, it will cause cache pollution as it runs – it keeps creating and deleting files, and this means that the caching layer will pollute the cache with writing blocks that will never get used. This cache pollution effect causes cache misses to happen after running for 200 – 600 seconds, depending on the write policy. After cache miss begins to happen, the Flash cache hit rate is around 78% for write-back persist and only 49% for write-back flush. Cache misses also bring an additional cache fill penalty: we need to read the block from the primary storage and then perform an extra write to Flash device. In Figure 6.1, this cache penalty reduces the IOPS of write through by 9% compared to no-cache. Fortunately, 84% of the requests by ms_nfs-small are write requests, and both write-back flush and write-back persist can boost performance for write requests, making the average IOPS much higher than write through and no-cache.

In contrast, ms_nfs has a much larger data set, and this causes access to the primary storage more often than ms_nfs-small. Also, the 22GB data set increases the seek latency on the primary storage, which makes the latency on the primary storage a bigger performance bottleneck compared to the miss penalty in ms_nfs-small. For these reasons, write through can boost performance by around 7% compared to no-cache, even with a lower hit rate of around 46%. Also, because ms_nfs is less write-intensive compared to ms_nfs-small (only 58% of the requests are writes), ms_nfs benefits less on write-back flush and write-back persist policies than ms_nfs-small.

6.3.2 IOPS Over Time

As we mentioned in Chapter 6.1.2, the IOPS can be unstable at the beginning of each experiment because of cold cache effects from both the file system buffer cache and the Flash cache at the storage client. For this reason, we only include the IOPS from the last 10 minutes of each run to calculate average IOPS. In this section, we will discuss why the IOPS is unstable at the beginning of the experiment by analyzing the graph of IOPS over time. We modified Filebench to output the average IOPS every 30 seconds (rather than just at the end of the run). The

statistics are cleared after every snapshot so that the data points represent the average in each 30 second interval, rather than the average from the beginning of the experiment.

Figure 6.2 shows the average IOPS over time for the 5 workloads. We found that IOPS for almost all runs are unstable at the beginning, but then become stable in less than 10 minutes. There are two major factors that we believe are affecting the IOPS at the beginning of the experiments. First, the file system buffer cache on the storage client could keep some blocks in memory during the benchmark setup phase, so that at the beginning of the workload, most read requests were served by the buffer cache. As the experiments run, the buffer cache might no longer hold the hot data that the workload needs, requiring it to be read from the storage. At this point, IOPS begin to drop and finally stabilize. Second, the Flash cache hit rate changes over time could also affect the IOPS in a similar way. For example, at the beginning of the experiments, some data might not be found on the Flash device, requiring it to be fetched from the primary storage, and this could reduce the IOPS.

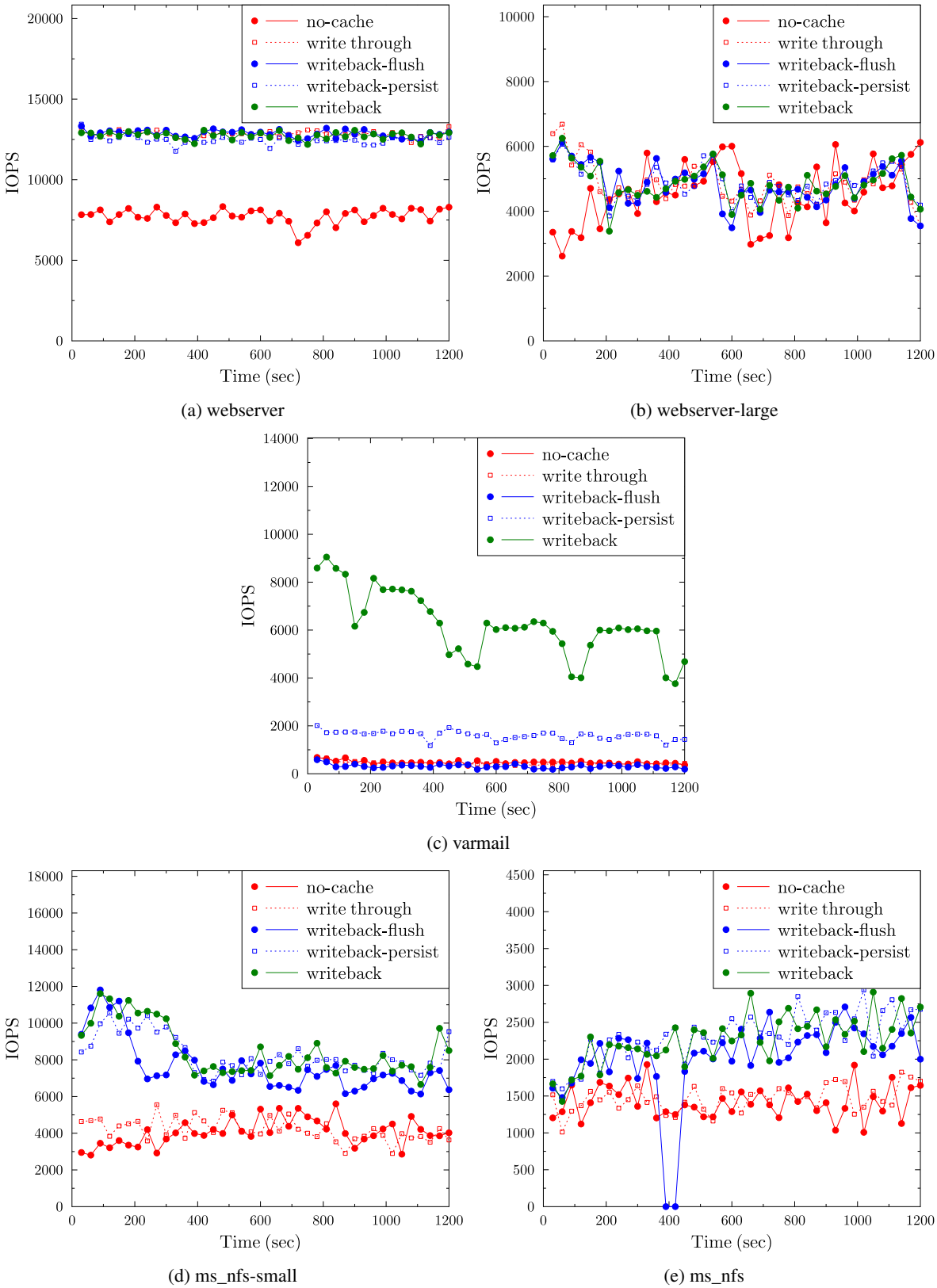
The IOPS for webserver stabilizes very quickly. Since webserver fits entirely into the Flash device, no Flash cache misses occur during the experiments. Only at the very beginning of the workload, the buffer cache on the storage client was able to serve the read requests by webserver and thus boost the IOPS, but this effect lasted for less than 30 seconds. Soon after that, webserver began to issue reads to the Flash device, and the IOPS stabilized.

In contrast, webserver-large does not fit entirely into Flash device, and thus it takes longer (around 200 seconds) to stabilize, as shown in Figure 6.2(b). This effect occurs because some of the data is initially cached on the Flash device during the benchmark setup phase. At the beginning of the experiment, webserver-large obtains some benefit from cache hits on that data, but then as it runs, it begins to read more data that are not cached on the Flash device, leading to more Flash cache misses and a decrease in the average IOPS. It takes about 200 seconds for this effect to stabilize, however the IOPS for webserver-large continue to show high variability throughout the experiment.

The IOPS for varmail also stabilized very quickly. Similar to webserver, at the beginning of the experiment, the buffer cache on the client side boosts the performance. In fact, we found at the beginning of the experiment, with a higher IOPS, varmail issued around 15% less reads than when the IOPS became stable.

The case for ms_nfs-small is rather complicated. Unlike other workloads, which only have two phases (either higher or lower IOPS at the beginning and then stable behavior), ms_nfs-small has three phases for the write-back policies. For example, consider the write-back flush policy. First, starting from around 10,000 IOPS, it climbs up to 13,000 IOPS in 120 seconds; second, IOPS begin to drop, falling below 8,000 at around 200 seconds after the experiment starts to run; finally, it begins to show more regular behavior after 200 second.

We found that in the first phase, IOPS are increasing because initially the data needed by ms_nfs-small is not in the buffer cache. These misses are handled by fetching them from the Flash device. After about 60 seconds of fetching from Flash device, the buffer cache is able to serve most of the read requests issued by ms_nfs-small,



(a) webserver

(b) webserver-large

(c) varmail

(d) ms_nfs-small

(e) ms_nfs

Figure 6.2: IOPS over time

maximizing the IOPS. We found that with write-back flush policy, 58% fewer reads are issued from 60 to 120 seconds compared to the first 60 seconds, but at the same time, IOPS were much higher than during the first 60 seconds.

In the second phase, the cache pollution effect we mentioned in Chapter 6.1.1 begins to happen. The Flash cache becomes full with dirty blocks, some of which are free blocks in the file system. At this point, the Flash cache begins evicting blocks and cache misses begin to happen rather intensively. The Flash cache miss keeps dragging the IOPS down, and eventually after around 200 seconds, IOPS reach their stable point.

A similar scenario happens to write-back persist, but not for write through. This is because ms_nfs-small is a write-intensive workload and IOPS of write through are bottlenecked by writes to the primary storage.

Less complicated than ms_nfs-small, ms_nfs exhibits classical Flash cache warm up behavior, as seen in Figure 6.2(e). As the experiments run, Flash cache misses are generated, and ms_nfs needs to fetch data from primary storage.

IO Stall in ms_nfs Aside from the different IOPS numbers at the beginning of the experiments, we also found one interesting issue with ms_nfs when using the write-back flush policy. As the benchmark runs, ms_nfs experienced a significant IO stall lasting around 1 to 2 minutes. During that stall, the IOPS of the last 30 second intervals drops completely to zero. This can be seen in Figure 6.2(e) around the 400s point.

To understand what causes the IO stall, we investigated what is happening during this time. First, the application (filebench in our case) is issuing write() system calls. This data is written into the buffer cache first, and later flushed to storage by the file system. To maintain metadata consistency, the file system needs to maintain certain write-before relations when it is flushing dirty buffers [7]. To do so, the file system needs to flush some buffers, issue a barrier request and wait for the barrier to return, and then flush some other buffers. While the file system is waiting for the barrier to return, any writes from the application will be buffered into the buffer cache.

During the IO stall, we found that the storage client holds around 129MB dirty buffers in the buffer cache, and there are around 1.3GB of buffer cache in total. In Linux, there is an upper limit on the ratio of dirty buffers in the buffer cache [1]. When the size of dirty buffers exceeds this limit, the operating system will simply block the write() system call from the application. This parameter is set to 10% by default, which suggests that, in our case, the write() system call was being blocked by the operating system because of too many dirty buffers.

These 129MB of dirty buffers are waiting for the file system to flush them. During the IO stall, we did not see any IO traffic issued by the file system. However, we do see that the file system issued the barrier, and our sync-thread was in the process of flushing dirty blocks to the primary storage. Right after the sync-thread finished flushing that epoch and acknowledged the barrier request to the file system, the IOPS began to recover. This

[1] vm.background_dirty_ratio

suggests that while dirty buffers are waiting for the file system to flush them, the file system is waiting for the barrier to finish. Thus, the root cause of the IO stall is that the barrier takes too long to finish, causing the write() system call to become blocked.

To address this IO stall issue, we tried to tune two different thresholds: (1) increase the dirty ratio limit in the buffer cache from the default 10% value to 40%; (2) increase the upper limit on the maximum number of dirty blocks cached on the Flash device for the sync-thread from 4GB to 1GB. By increasing either one of these two thresholds, we were able to decrease the IO stall from 1-2 minutes to around 30 seconds.

6.4 Analysis of Optimizations

In this section, we measure each of our optimizations mentioned in our design (Chapter 4.4), to see whether they are really boosting performance, and if so, how much benefit results from each optimization.

6.4.1 Flushing Policies

We mentioned in Chapter 4.4 that we need to consider two policies: flushing in LRU order or flushing in order of ascending block number. Flushing in ascending order optimizes for flushing throughput while flushing in LRU order optimizes for hit rate. We expect that flushing in LRU order could increase the hit rate on large workloads like ms_nfs and flushing in ascending order can improve performance of write-back flush policies in general because of higher throughput to flush dirty blocks. In this section, we will compare how these two different policies affect the performance with both write-back persist policy and write-back flush policy.

Using write-back persist Policy

In Figure 6.3, we see that flushing in ascending order is slightly better than flushing in LRU order in three out of five workloads, and flushing in LRU order is slightly better in the remaining two workloads.

We found that webserver and ms_nfs-small benefit slightly from flushing in LRU order. However, we do not see any hit rate improvements compared to flushing in ascending order. These two workloads are both saturating the Flash device bandwidth: webserver is read-intensive and ms_nfs-small is write-intensive. We found that flushing in LRU order achieved a much lower write-back throughput (<10MB/s) compared to flushing in ascending order. We were surprised to find that flushing in ascending order is actually too fast, leading to a negative impact on the foreground read traffic.

In contrast, varmail is sensitive to barrier latency. By flushing in ascending order, we are able to migrate mapping entries from the dirty map to the clean map more rapidly due to the higher flushing throughput. As

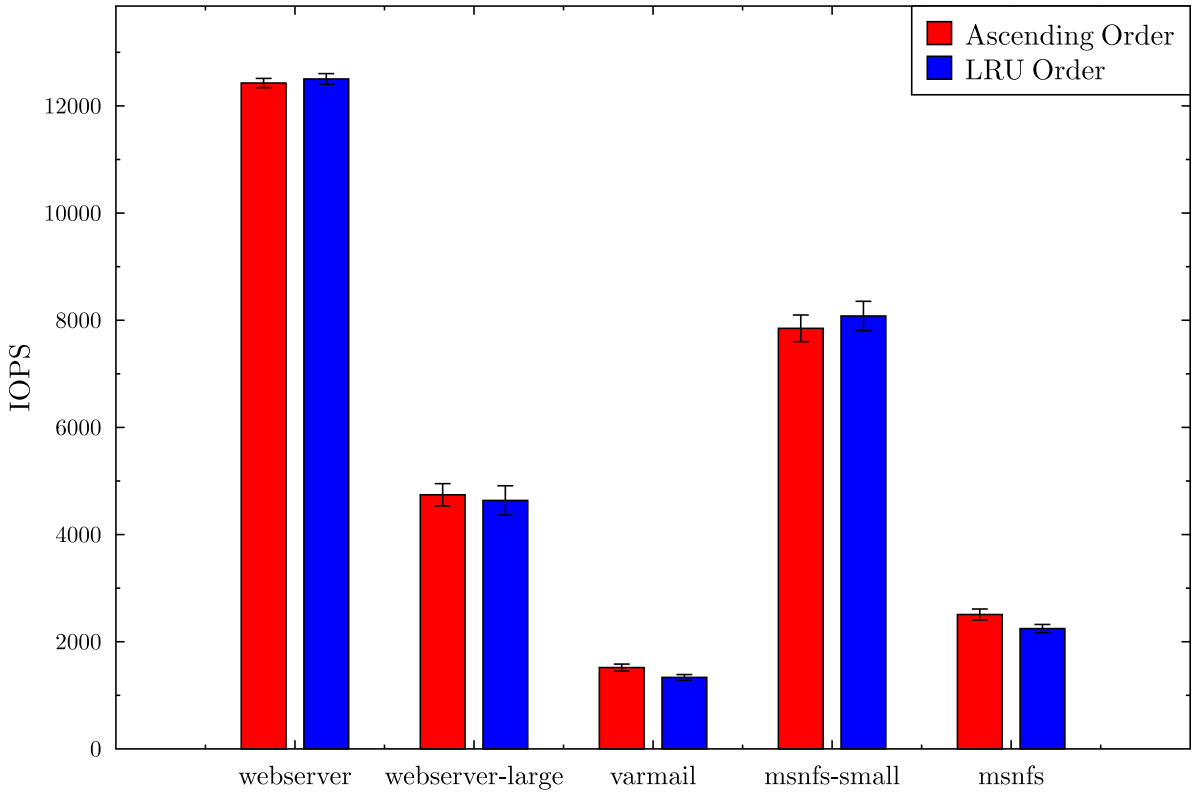


Figure 6.3: Average IOPS for write-back persist policy with different flush orders

a result, there are fewer entries in the dirty map to persist at barriers, leading to 13% higher IOPS on average compared to flushing in LRU order.

For webserver-large and ms_nfs, as we showed in Figure 6.1, there is little cost to persist the mapping entry to the Flash device. Instead, the benefit of flushing in ascending order is on the primary storage. In this case, primary storage receives a more sequential write stream with less randomness and therefore the seek overhead is reduced.

In all five cases, we did not find a significant difference in hit rate. This might be because for all five workloads that we chose, flushing in either order is quite efficient. In either case, dirty blocks can be written back to primary storage before they become cold, meaning that there are very few cold blocks in the dirty queue that cannot be evicted. The order of flushing hot blocks is not critical to the overall cache hit rate since the blocks move into the clean LRU queue after flushing anyway, and have time to become cold (or to be reused if they are still hot) there before being evicted from the cache in LRU order.

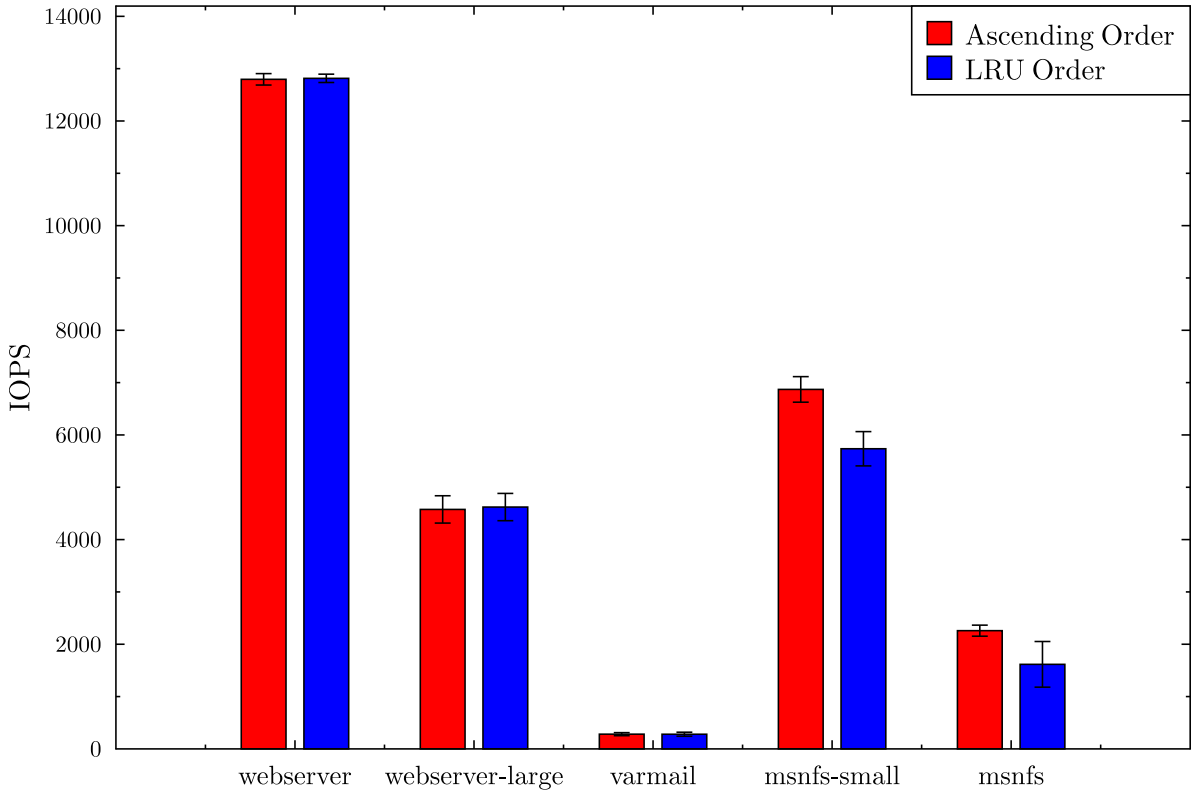


Figure 6.4: Average IOPS for write-back flush policy with different flush orders

Using write-back flush Policy

We perform the same experiments for the write-back flush policy. In write-back flush policy, the sync-thread is more likely to be a bottleneck because it needs to flush every dirty block in the previous barrier epoch to the primary storage. In Figure 6.4, we present the average IOPS after the IOPS stabilized. As can be seen, flushing in ascending order gives better performance for two of the five workloads, but the other three workloads do not show any significant difference.

This result is expected, as webserver and webserver-large are read intensive and flushing dirty blocks might not be a bottleneck for them. For varmail, the workload performs frequent syncs and only contains a few random writes per epoch. The added barrier latency due to the write-back flush policy is dominated by the cost of writing these blocks at the primary storage but the order in which they are flushed does not have much impact since there are only a few random writes in either case.

In the two write-intensive workloads, ms_nfs-small and ms_nfs, flushing in ascending order provides better performance. This result indicates that flushing is truly the bottleneck and flushing in ascending order reduces disk seeks on the primary storage during flushing. Compared to flushing in LRU order, flush in ascending order improves average IOPS by 19% and 39% with ms_nfs-small and ms_nfs, respectively.

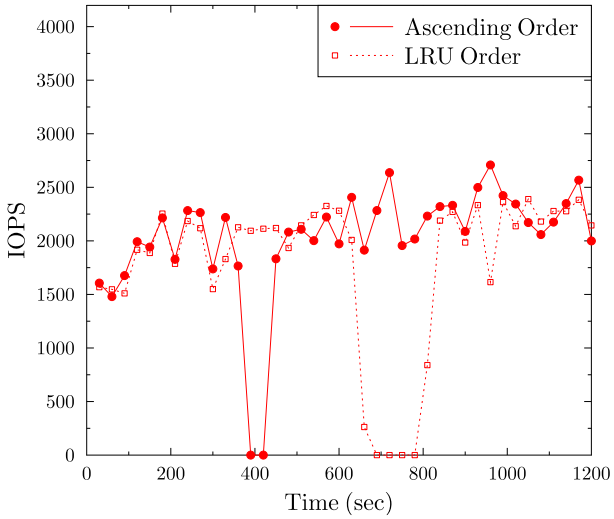


Figure 6.5: ms_nfs

Figure 6.6: IOPS over time with write-back flush policy on ms_nfs

Similar to write-back persist, in all five workloads we did not find significant hit rate differences. The reasoning is the same as with write-back persist. Together, these results show that flushing in LRU order does not achieve the benefit for which it was designed.

IO Stall in ms_nfs We observed exactly the same IO stall as we saw before in Figure 6.2(e) for ms_nfs with both policies for flush order. As can be seen in Figure 6.6, there are length IO stalls in both cases, during which the IOPS drops to 0 before recovering after 1 to 3 minutes. With flushing in LRU order, this stall lasts 3 minutes, which is longer than with flushing in ascending order where it only lasts for 1 minute. As we explained above, the reason behind the stall is that the barrier takes too long to finish. This further demonstrates that flushing in LRU order yields much lower throughput on flushing dirty data, which in turn causes the barrier to take even longer to finish compared to flushing in ascending order.

6.4.2 Multiple Epoch

In our write-back flush policy, barrier request processing is slow because of the need to flush all dirty blocks back to primary storage. As we noted in Chapter 4.4.2, there could be new writes arriving between the time when a barrier was issued and when it is completed, requiring further delay of either the barrier or the new write operation. To solve this problem we introduced multiple dirty epochs. This optimization is not needed for write-back persist because it only persists the dirty mapping table on the Flash device upon a barrier request. In this section, we evaluate the effectiveness of using multiple dirty epochs.

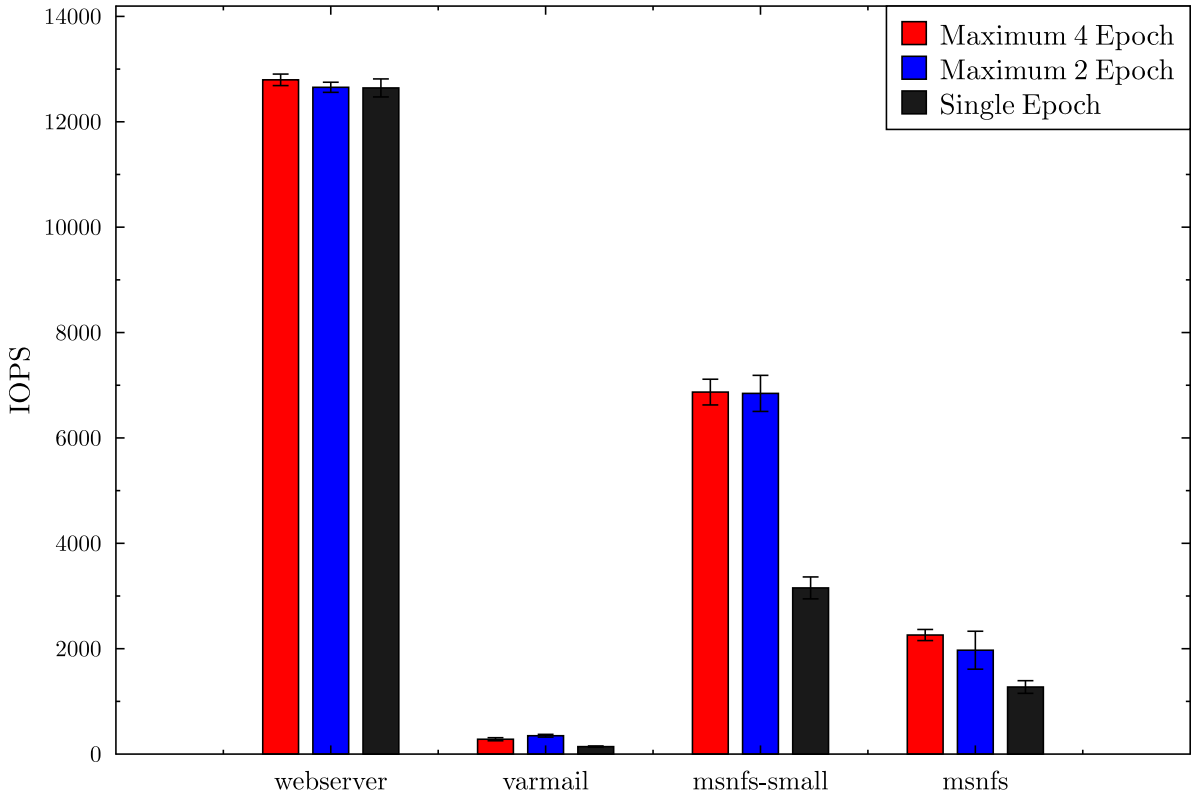


Figure 6.7: Average IOPS after stabilization with varying numbers of dirty epochs

In Figure 6.7, we show that with multiple epochs, we can increase performance significantly for ms_nfs, ms_nfs-small, and varmail. Only webserver does not show any improvements, because it is read intensive. We were unable to run webserver-large with either the single dirty epoch or two dirty epoch configurations, so this workload is not shown in Figure 6.7 (previously presented results for webserver-large with the write-back flush policy show the four dirty epoch configuration). On investigation, we found that Filebench was exiting when the file system returned an error in response to some request. Specifically, Filebench needed to access some metadata that was being flushed by the file system, and the file system was waiting for a barrier to complete the flush. After waiting for several minutes for the barrier, the file system code timed out the Filebench request and returned an error. This is evidence of exactly the sort of indefinite barrier delay that multiple dirty epochs were designed to avoid. For webserver-large, we found that four dirty epochs were needed to prevent the problem. This problem is similar to the IO stall issue we have with ms_nfs, except here, we can fix the problem by increasing the maximum number of epochs.

We also found that having multiple epochs has a downside, which is that we have to search more BTrees on the read and write path to locate a mapping entry. This could be the reason why the performance of varmail drops with a maximum of four epochs instead of two, because we now have to search up to four BTrees instead of two

to find a block on the Flash device. Thus, we may double the time to find a block on the Flash device compared to having only two BTrees.

6.4.3 Logical Journaling

In write-back persist, we choose to optimize for sync-frequent workloads by adding a logical journal to speed up the writing of dirty mapping entries on the Flash device. Among the five workloads we tested, varmail is the only sync-frequent workload where this optimization is expected to make a difference.

For varmail, we found that using logical journaling does not significantly increase the IOPS. Without logical journaling, we measured 1506 IOPS on average after stabilization, while with logical journal the IOPS marginally improves to 1517, which is only a 0.7% improvement.

This is because varmail is a latency sensitive benchmark, and logical journal is trying to optimize the number of block writes to the Flash device. On a high performance device like Flash, reducing the number of blocks written does not have a big impact on the IO latency, and thus, varmail does not benefit from this optimization either.

For the remaining workloads in our study, the difference between write-back and write-back persist shown in Figure 6.1 demonstrated that persisting the mapping table on the Flash device has negligible overhead. Thus, even if other workloads trigger logical journaling, it is not expected to improve the performance.

6.5 Summary of Results

We first conclude that by sacrificing consistency and durability on destructive failure by using write-back persist, it is possible to boost performance. However, sacrificing consistency and durability on recoverable failure is not a worthwhile tradeoff, given the low overhead of write-back persist compared to unsafe write-back.

We found that using the write-back flush policy to provide consistency and durability for destructive failure can yield higher performance than write through, especially on write-intensive workloads, such as ms_nfs-small and ms_nfs. We conclude that write-back persist and write-back flush achieve higher IOPS than write through for two major reasons. First, both write-back policies can batch the dirty blocks on Flash device and flush them in the background. This hides the latency to access the primary storage, and provides a big advantage to write-back flush compared to write through in a bursty write workload like ms_nfs. Second, reordering the writes during flush can boost the flushing throughput by generating a primary storage friendly access pattern, and therefore increase the write bandwidth overall.

Chapter 7

Related Work

With the price-per-capacity of flash storage declining rapidly recently, there have been many proposals for using flash devices to improve IO performance. We discuss recent, related work in the area.

Koller et al. [11] present a client-side flash-based caching system that can provide consistency on both recoverable and destructive failures. They present two write-back policies, ordered write-back and journaled write-back, and their evaluation showed 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 cannot guarantee consistency on storage failure. While their evaluation showed that write coalescing improves performance for write-back policies, we have found that batching and reordering requests, e.g., in ascending order, has significant benefits.

Holland et al. [8] present a detailed study on using a flash cache on the storage client. They use trace-driven simulations to explore a large number of cache design decisions, including write policies in both buffer cache and flash cache, unification of buffer cache and flash cache, cost of cache persistent, etc. Their simulation showed write-back policies do not significantly improve performance, and write-through is sufficient. They argue that the main reason is that both the buffer cache and the flash cache are able to flush the dirty blocks to storage in time, and therefore all the writes issued by any application are asynchronous.

Their results suggest that a write-back caching policy is unnecessary for client-side flash caches. However, their simulation results have two significant limitations. First, their simulation does not consider synchronous IO operations like barrier requests. In fact, their traces contain reads and writes but no barrier requests. Second, their simulation does not consider batching requests to storage, which offers significant performance improvements, as shown in our evaluation.

They also evaluated the overhead of persisting mapping tables on flash, and their simulation results showed that this overhead is minimal. This result is confirmed by our evaluation as well.

Mercury [4] presents client-side flash caching for data centers. It focuses on virtualization clusters in a data center. 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 [23] presents an interface for caching on RAW flash memory chips, rather than on flash storage (SSD) with a flash translation layer. Their approach benefits from integrating wear-level management and garbage collection with cache eviction, and using 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.

Similar to our work, they separate the clean and dirty mapping. The clean mappings are committed asynchronously because losing them on a failure does not affect correctness. However, the dirty mappings are committed synchronously because they need to be durable. We separate the mappings for a similar reason but we also use this separation to reorder writing dirty blocks in the flush thread.

Bcache [19] is a Linux kernel block layer cache similar to our system. It implements the write-through and our write-back persist policy but not our write-back flush policy.

Previous work on flash caching [8, 11] has suggested that the flash cache hit rate, and thus the replacement policy, is crucial to performance. Our work focuses on the write-back policy and reliability, and hence we have chosen to use a simple LRU algorithm for the replacement policy.

There is a huge body of work cache replacement algorithms. The Adaptive Replacement Caching [14] algorithm is effective because it uses an algorithm that takes access frequency and recency into account compared to LRU which only considers recency. This makes the algorithm scan-resistant, i.e. it resists cache pollution on full table scans, and helps it adapt to the workload to improve cache hit rates.

An important aspect of caching is warming the caches after system reboot. Recent research suggests using prefetching for warming up the cache after a system reboot. Bonfire [26] shows that on-demand cache warm up takes a long time because of growing flash caches. By monitoring I/O traces, they are able to warm up the cache by loading hot data in bulk, which speeds up the cache warm up process. Their results show that this approach 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.

Windows SuperFetch prefetches commonly used data from storage into the buffer cache. It decreases boot

time and application launch time compared to warming up the cache on demand. Similar to Bonfire, SuperFetch also monitors accesses to the buffer cache to determine hot data.

Chapter 8

Conclusions and Future Work

This work has shown that a high-performance write-back caching system can support strong reliability guarantees. The key insight is that storage write barriers are the only reliable method for storing data durably on storage media. Storage applications that require durability and consistency already implement their own atomicity and durability schemes using these write barriers. 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 have designed two flash-based caching policies called write-back flush and write-back persist. The flush policy provides reliability under flash failure. The persist policy provides higher performance but makes assumptions about failure-free flash operation.

Our evaluation showed three results. First, read-heavy workloads, all caching policies, write-through or write-back, perform comparably. Second, for write-heavy workloads, write-back flush provides the same reliability guarantees as write-through while offering higher performance. For these workloads, the write-back persist performance is indistinguishable from write-back. Third, for sync-heavy workloads, write-back persist can significantly improve performance over write-back flush by reducing the latency of the barrier request.

We have several plans for future work. While the write-back flush policy works well, under certain workloads, it can stall IO processing on the client for several minutes. We have analyzed this issue and have found that tuning the virtual memory parameters of the operating system or the thresholds in our caching scheme can eliminate this stall. We plan to understand this issue in more detail and tune these parameters automatically so that these stalls can be avoided.

We plan to compare the performance of our system against existing write-back caching systems such as Bcache [19] and the journaled write-back policy [11]. The journaled write-back policy does not provide durability

and so this comparison will show the cost of providing durability in our system.

Finally, we plan to integrate our write-back caching policies in virtualized environments [4]. This will require understanding how barriers are handled, and their costs, in these environments. For example, live virtual machine migration will need to ensure that all data is synchronized before and after migration. Also, caching performance could be improved in these environments by cache partitioning [25], which prevents cache pollution across virtual machines.

Bibliography

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

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

[3] J. Bonwick and B. Moore. ZFS - The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

[4] 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 *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12, 2012.

[5] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

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

[7] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proceedings of Twenty -first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 307 –320, New York, NY, USA, 2007. ACM.

[8] David A. Holland, Elaine Angelino, Gideon Wald, and Margo I. Seltzer. Flash caching on the storage client. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 127–138, Berkeley, CA, USA, 2013. USENIX Association.

[9] Jens Axboe. http://freecode.com/projects/fio.

[10] Jens Axboe and Suparna Bhattacharya. http://lxr.free-electrons.com/source/Documentation/block/biodoc.txt#L826.

[11] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 2013.

[12] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *Trans. Storage*, 6(3):13:1–13:26, September 2010.

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

[14] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[15] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[16] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron. Everest: scaling down peak loads through i/o off-loading. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 15–28, 2008.

[17] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: Analysis of tradeoffs. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 145–158, New York, NY, USA, 2009. ACM.

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

[19] Kent Overstreet. bcache. `http://bcache.evilpiepirate.org/`, 2013.

[20] R. Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. Snapmirror: File-system-based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[21] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[22] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of the USENIX Technical Conference*, June 2000.

[23] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[24] Mohammad Shamma, Dutch T. Meyer, Jake Wires, Maria Ivanova, Norman C. Hutchinson, and Andrew Warfield. Capo: Recapitulating storage for virtual desktops. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[25] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

[26] Yiying Zhang, Gokul Soundararajan, Mark W. Storer, Lakshmi N. Bairavasundaram, Sethuraman Subbiah, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *Proceedings of the 11th Conference on File and Storage Technologies (FAST 2013)*, February 2013.