# FlashTier: a Lightweight, Consistent and Durable Storage Cache

Mohit Saxena, Michael M. Swift and Yiying Zhang
University of Wisconsin-Madison
{msaxena,swift,yyzhang}@cs.wisc.edu

## Abstract

The availability of high-speed solid-state storage has introduced a new tier into the storage hierarchy. Low-latency and high-IOPS solid-state drives (SSDs) cache data in front of high-capacity disks. However, most existing SSDs are designed to be a drop-in disk replacement, and hence are mismatched for use as a cache.

This paper describes *FlashTier*, a system architecture built upon *solid-state cache* (SSC), a flash device with an interface designed for caching. Management software at the operating system block layer directs caching. The FlashTier design addresses three limitations of using traditional SSDs for caching. First, FlashTier provides a unified logical address space to reduce the cost of cache block management within both the OS and the SSD. Second, FlashTier provides cache consistency guarantees allowing the cached data to be used following a crash. Finally, FlashTier leverages cache behavior to silently evict data blocks during garbage collection to improve performance of the SSC.

We have implemented an SSC simulator and a cache manager in Linux. In trace-based experiments, we show that FlashTier reduces address translation space by 60% and silent eviction improves performance by up to 167%. Furthermore, FlashTier can recover from the crash of a 100 GB cache in only 2.4 seconds.

***Categories and Subject Descriptors*** D.4.2 [*Operating Systems*]: Storage Management; C.4 [*Computer Systems Organization*]: Performance of Systems

***Keywords*** Solid-State Cache, Device Interface, Consistency, Durability

## 1. Introduction

Solid-state drives (SSDs) composed of multiple flash memory chips are often deployed as a cache in front of cheap and slow disks [9, 22]. This provides the performance of flash with the cost of disk for large data sets, and is actively used by Facebook and others to provide low-latency access to petabytes of data [10, 32, 36]. Many vendors sell dedicated caching products that pair an SSD with proprietary software that runs in the OS to migrate data between the SSD and disks [12, 19, 30] to improve storage performance.

Building a cache upon a standard SSD, though, is hindered by the narrow block interface and internal block management of SSDs, which are designed to serve as a disk replacement [1, 34, 39]. Caches have at least three different behaviors that distinguish them from general-purpose storage. First, data in a cache may be present elsewhere in the system, and hence need not be durable. Thus, caches have more flexibility in how they manage data than a device dedicated to storing data persistently. Second, a cache stores data from a separate address space, the disks', rather than at native addresses. Thus, using a standard SSD as a cache requires an additional step to map block addresses from the disk into SSD addresses for the cache. If the cache has to survive crashes, this map must be persistent. Third, the consistency requirements for caches differ from storage devices. A cache must ensure it never returns stale data, but can also return nothing if the data is not present. In contrast, a storage device provides ordering guarantees on when writes become durable.

This paper describes *FlashTier*, a system that explores the opportunities for tightly integrating solid-state caching devices into the storage hierarchy. First, we investigate how small changes to the interface and internal block management of conventional SSDs can result in a much more effective caching device, a *solid-state cache*. Second, we investigate how such a dedicated caching device changes *cache managers*, the software component responsible for migrating data between the flash caching tier and disk storage. This design provides a clean separation between the caching device and its internal structures, the system software managing the cache, and the disks storing data.

FlashTier exploits the three features of caching workloads to improve over SSD-based caches. First, FlashTier provides a *unified address space* that allows data to be written to the SSC at its disk address. This removes the need for a separate table mapping disk addresses to SSD addresses. In addition,


*EuroSys'12,* April 10–13, 2012, Bern, Switzerland.

| Device | Access Latency | | Capacity | Price | Endurance |
|---|---|---|---|---|---|
| | Read | Write | Bytes | $/GB | Erase Cycles |
| DRAM | 50 ns | 50 ns | 8-16 GB | $15 | $\infty$ |
| Flash | 40-100 $\mu$s | 60-200 $\mu$s | TB | $3 | $10^4$ |
| Disk | 500-5000 $\mu$s | 500-5000 $\mu$s | TB | $0.3 | $\infty$ |

**Table 1. Device Attributes**: Price, performance and endurance of DRAM, Flash SSDs and Disk. (GB: gigabyte, TB: terabyte).

an SSC uses internal data structures tuned for large, sparse address spaces to maintain the mapping of block number to physical location in flash.

Second, FlashTier provides *cache consistency guarantees* to ensure correctness following a power failure or system crash. It provides separate guarantees for clean and dirty data to support both write-through and write-back caching. In both cases, it guarantees that stale data will never be returned. Furthermore, FlashTier introduces new operations in the device interface to manage cache contents and direct eviction policies. FlashTier ensures that internal SSC metadata is always persistent and recoverable after a crash, allowing cache contents to be used after a failure.

Finally, FlashTier leverages its status as a cache to reduce the cost of garbage collection. Unlike a storage device, which promises to never lose data, a cache can evict blocks when space is needed. For example, flash must be erased before being written, requiring a garbage collection step to create free blocks. An SSD must copy live data from blocks before erasing them, requiring additional space for live data and time to write the data. In contrast, an SSC may instead *silently evict* the data, freeing more space faster.

We implemented an SSC simulator and a cache manager for Linux and evaluate FlashTier on four different storage traces. We measure the cost and benefits of each of our design techniques. Our results show that:

- FlashTier reduces total memory usage by more than 60% compared to existing systems using an SSD cache.
- FlashTier's free space management improves performance by up to 167% and requires up to 57% fewer erase cycles than an SSD cache.
- After a crash, FlashTier can recover a 100 GB cache in less than 2.4 seconds, much faster than existing systems providing consistency on an SSD cache.

The remainder of the paper is structured as follows. Section 2 describes our caching workload characteristics and motivates FlashTier. Section 3 presents an overview of FlashTier design, followed by a detailed description in Section 4 and 5. We evaluate FlashTier design techniques in Section 6, and finish with related work and conclusions.

## 2. Motivation

Flash is an attractive technology for caching because its price and performance are between DRAM and disk: about five times cheaper than DRAM and an order of magnitude (or more) faster than disk (see Table 1). Furthermore, its persistence enables cache contents to survive crashes or power failures, and hence can improve cold-start performance. As a result, SSD-backed caching is popular in many environments including workstations, virtualized enterprise servers, database backends, and network disk storage [22, 29, 30, 36, 37].

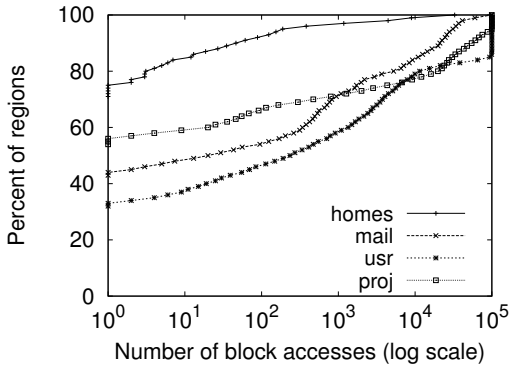


**Figure 1. Logical Block Addresses Distribution:** The distribution of unique block accesses across 100,000 4 KB block regions of the disk address space.

Flash has two characteristics that require special management to achieve high reliability and performance. First, flash does not support *in-place writes*. Instead, a block of flash must be *erased* (a lengthy operation) before it can be written. Second, to support writing a block multiple times, flash devices use *address mapping* to translate block addresses received from a host into physical locations in flash. This mapping allows a block to be written out-of-place to a pre-erased block rather than erasing and rewriting in-place. As a result, SSDs employ *garbage collection* to compact data and provide free, erased blocks for upcoming writes.

The motivation for FlashTier is the observation that caching and storage have different behavior and different requirements. We next study three aspects of caching behavior to distinguish it from general-purpose storage. Our study uses traces from two different sets of production systems downstream of an active page cache over 1-3 week periods [24, 27]. These systems have different I/O workloads that consist of a file server (*homes* workload), an email server (*mail* workload) and file servers from a small enterprise data center hosting user home and project directories (*usr* and *proj*). Table 3 summarizes the workload statistics. Trends observed across all these workloads directly motivate our design for FlashTier.

***Address Space Density.*** A hard disk or SSD exposes an address space of the same size as its capacity. As a result, a mostly full disk will have a dense address space, because there is valid data at most addresses. In contrast, a cache stores only *hot data* that is currently in use. Thus, out of the terabytes of storage, a cache may only contain a few

gigabytes. However, that data may be at addresses that range over the full set of possible disk addresses.

Figure 1 shows the density of requests to 100,000-block regions of the disk address space. To emulate the effect of caching, we use only the top 25% most-accessed blocks from each trace (those likely to be cached). Across all four traces, more than 55% of the regions get less than 1% of their blocks referenced, and only 25% of the regions get more than 10%. These results motivate a change in how mapping information is stored within an SSC as compared to an SSD: while an SSD should optimize for a *dense address space*, where most addresses contain data, an SSC storing only active data should instead optimize for a *sparse address space*.

***Persistence and Cache Consistency.*** Disk caches are most effective when they offload workloads that perform poorly, such as random reads and writes. However, large caches and poor disk performance for such workloads result in exceedingly long cache warming periods. For example, filling a 100 GB cache from a 500 IOPS disk system takes over 14 hours. Thus, caching data persistently across system restarts can greatly improve cache effectiveness.

On an SSD-backed cache, maintaining cached data persistently requires storing cache metadata, such as the state of every cached block and the mapping of disk blocks to flash blocks. On a clean shutdown, this can be written back at low cost. However, to make cached data *durable* so that it can survive crash failure, is much more expensive. Cache metadata must be persisted on every update, for example when updating or invalidating a block in the cache. These writes degrade performance, and hence many caches do not provide crash recovery [5, 14], and discard all cached data after a crash.

A hard disk or SSD provides crash recovery with simple consistency guarantees to the operating system: barriers ensure that preceding requests complete before subsequent ones are initiated. For example, a barrier can ensure that a journal commit record only reaches disk *after* the journal entries [7]. However, barriers provide ordering between requests to a single device, and do not address consistency between data on different devices. For example, a write sent both to a disk and a cache may complete on just one of the two devices, but the combined system must remain consistent.

Thus, the guarantee a cache makes is semantically different than ordering: a cache should never return stale data, and should never lose dirty data. However, within this guarantee, the cache has freedom to relax other guarantees, such as the persistence of clean data.

***Wear Management.*** A major challenge with using SSDs as a disk cache is their limited write endurance: a single MLC flash cell can only be erased 10,000 times. In addition, garbage collection is often a contributor to wear, as live data must be copied to make free blocks available. A recent study showed that more than 70% of the erasures on a full SSD were due to garbage collection [8].

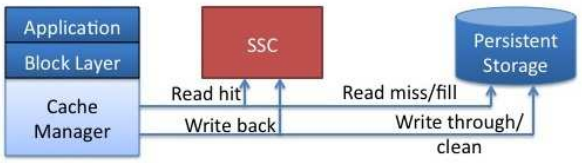


**Figure 2. FlashTier Data Path:** A cache manager forwards block read/write requests to disk and solid-state cache.

Furthermore, caching workloads are often more intensive than regular storage workloads: a cache stores a greater fraction of hot blocks, which are written frequently, as compared to a general storage workload. In looking at the top 25% most frequently referenced blocks in two write-intensive storage traces, we find that the average writes per block is 4 times greater than for the trace as a whole, indicating that caching workloads are likely to place greater durability demands on the device. Second, caches operate at full capacity while storage devices tend to be partially filled. At full capacity, there is more demand for garbage collection. This can hurt reliability by copying data more frequently to make empty blocks [16, 18].

## 3. Design Overview

FlashTier is a block-level caching system, suitable for use below a file system, virtual memory manager, or database. A *cache manager* interposes above the disk device driver in the operating system to send requests to the either the flash device or the disk, while a *solid-state cache (SSC)* stores and assists in managing cached data. Figure 2 shows the flow of read and write requests from the application to SSC and disk-storage tiers from the cache manager.

### 3.1 Cache Management

The cache manager receives requests from the block layer and decides whether to consult the cache on reads, and whether to cache data on writes. On a cache miss, the manager sends the request to the disk tier, and it may optionally store the returned data in the SSC.

FlashTier supports two modes of usage: *write-through* and *write-back*. In write-through mode, the cache manager writes data to the disk and populates the cache either on read requests or at the same time as writing to disk. In this mode, the SSC contains only clean data, and is best for read-heavy workloads, where there is little benefit to caching writes, and when the cache is not considered reliable, as in a client-side cache for networked storage. In this mode, the cache manager consults the SSC on every read request. If the data is not present, the SSC returns an error, and the cache manager fetches the data from disk. On a write, the cache manager must either evict the old data from the SSC or write the new data to it.

In write-back mode, the cache manager may write to the SSC without updating the disk. Thus, the cache may contain dirty data that is later evicted by writing it back to the disk. This complicates cache management, but performs better with write-heavy workloads and local disks. In this mode, the cache manager must actively manage the contents of the cache to ensure there is space for new data. The cache manager maintains a table of dirty cached blocks to track which data is in the cache and ensure there is enough space in the cache for incoming writes. The manager has two options to make free space: it can evict a block, which guarantees that subsequent reads to the block will fail, and allows the manager to direct future reads of the block to disk. Or, the manager notifies the SSC that the block is clean, which then allows the SSC to evict the block in the future. In the latter case, the manager can still consult the cache on reads and must evict/overwrite the block on writes if it still exists in the cache.

### 3.2 Addressing

With an SSD-backed cache, the manager must maintain a *mapping table* to store the block's location on the SSD. The table is indexed by logical block number (LBN), and can be used to quickly test whether block is in the cache. In addition, the manager must track free space and evict data from the SSD when additional space is needed. It does this by removing the old mapping from the mapping table, inserting a mapping for a new LBN with the same SSD address, and then writing the new data to the SSD.

In contrast, an SSC does not have its own set of addresses. Instead, it exposes a *unified address space*: the cache manager can write to an SSC using logical block numbers (or disk addresses), and the SSC internally maps those addresses to physical locations in flash. As flash devices already maintain a mapping table to support garbage collection, this change does not introduce new overheads. Thus the cache manager in FlashTier no longer needs to store the mapping table persistently, because this functionality is provided by the SSC.

The large address space raises the new possibility that cache does not have capacity to store the data, which means the cache manager must ensure not to write too much data or the SSC must evict data to make space.

### 3.3 Space Management

As a cache is much smaller than the disks that it caches, it requires mechanisms and policies to manage its contents. For write-through caching, the data is clean, so the SSC may silently evict data to make space. With write-back caching, though, there may be a mix of clean and dirty data in the SSC. An SSC exposes three mechanisms to cache managers for managing the cached data: *evict*, which forces out a block; *clean*, which indicates the data is clean and can be evicted by the SSC, and *exists*, which tests for the presence of a block and is used during recovery. As described above, for write-through caching all data is clean, whereas with write-back caching, the cache manager must explicitly clean blocks after writing them back to disk.

The ability to evict data can greatly simplify space management within the SSC. Flash drives use garbage collection to compact data and create freshly erased blocks to receive new writes, and may relocate data to perform wear leveling, which ensures that erases are spread evenly across the physical flash cells. This has two costs. First, copying data for garbage collection or for wear leveling reduces performance, as creating a single free block may require reading and writing multiple blocks for compaction. Second, an SSD may copy and compact data that is never referenced again. An SSC, in contrast, can evict data rather than copying it. This speeds garbage collection, which can now erase clean blocks without copying their live data because clean cache blocks are also available in disk. If the data is not later referenced, this has little impact on performance. If the data is referenced later, then it must be re-fetched from disk and cached again.

Finally, a cache does not require overprovisioned blocks to make free space available. Most SSDs reserve 5-20% of their capacity to create free erased blocks to accept writes. However, because an SSC does not promise a fixed capacity, it can flexibly dedicate space either to data, to reduce miss rates, or to the log, to accept writes.

### 3.4 Crash Behavior

Flash storage is persistent, and in many cases it would be beneficial to retain data across system crashes. For large caches in particular, a durable cache can avoid an extended warm-up period where all data must be fetched from disks. However, to be usable after a crash, the cache must retain the metadata mapping disk blocks to flash blocks, and must guarantee correctness by never returning stale data. This can be slow, as it requires synchronous metadata writes when modifying the cache. As a result, many SSD-backed caches, such as Solaris L2ARC and NetApp Mercury, must be reset after a crash [5, 14].

The challenge in surviving crashes in an SSD-backed cache is that the mapping must be persisted along with cached data, and the consistency between the two must also be guaranteed. This can greatly slow cache updates, as replacing a block requires writes to: (i) remove the old block from the mapping, (ii) write the new data, and (iii) add the new data to the mapping.

### 3.5 Guarantees

FlashTier provides consistency and durability guarantees over cached data in order to allow caches to survive a system crash. The system distinguishes *dirty data*, for which the newest copy of the data may only be present in the cache, from *clean data*, for which the underlying disk also has the latest value.

1. A read following a write of dirty data will return that data.
2. A read following a write of clean data will return *either* that data or a not-present error.
3. A read following an eviction will return a not-present error.

The first guarantee ensures that dirty data is durable and will not be lost in a crash. The second guarantee ensures that it is *always* safe for the cache manager to consult the cache for data, as it must either return the newest copy or an error. Finally, the last guarantee ensures that the cache manager can invalidate data in the cache and force subsequent requests to consult the disk. Implementing these guarantees within the SSC is much simpler than providing them in the cache manager, as a flash device can use internal transaction mechanisms to make all three writes at once [33, 34].

## 4. System Design

FlashTier has three design goals to address the limitations of caching on SSDs:

- *Address space management* to unify address space translation and block state between the OS and SSC, and optimize for sparseness of cached blocks.
- *Free space management* to improve cache write performance by silently evicting data rather than copying it within the SSC.
- *Consistent interface* to provide consistent reads after cache writes and eviction, and make both clean and dirty data as well as the address mapping durable across a system crash or reboot.

This section discusses the design of FlashTier's address space management, block interface and consistency guarantees of SSC, and free space management.

### 4.1 Unified Address Space

FlashTier unifies the address space and cache block state split between the cache manager running on host and firmware in SSC. Unlike past work on virtual addressing in SSDs [21], the address space in an SSC may be very sparse because caching occurs at the block level.

***Sparse Mapping.*** The SSC optimizes for sparseness in the blocks it caches with a *sparse hash map* data structure, developed at Google [13]. This structure provides high performance and low space overhead for sparse hash keys. In contrast to the mapping structure used by Facebook's FlashCache, it is fully associative and thus must encode the complete block address for lookups.

The map is a hash table with $t$ buckets divided into $t/M$ groups of $M$ buckets each. Each group is stored sparsely as an array that holds values for allocated block addresses and an occupancy bitmap of size $M$, with one bit for each bucket. A bit at location $i$ is set to 1 if and only if bucket $i$ is non-empty. A lookup for bucket $i$ calculates the value location from the number of 1s in the bitmap before location $i$. We set $M$ to 32 buckets per group, which reduces the overhead of bitmap to just 3.5 bits per key, or approximately 8.4 bytes per occupied entry for 64-bit memory pointers [13]. The runtime of all operations on the hash map is bounded by the constant $M$, and typically there are no more than 4-5 probes per lookup.

The SSC keeps the entire mapping in its memory. However, the SSC maps a fixed portion of the flash blocks at a 4 KB page granularity and the rest at the granularity of an 256 KB erase block, similar to hybrid FTL mapping mechanisms [18, 25]. The mapping data structure supports lookup, insert and remove operations for a given key-value pair. Lookups return the physical flash page number for the logical block address in a request. The physical page number addresses the internal hierarchy of the SSC arranged as flash package, die, plane, block and page. Inserts either add a new entry or overwrite an existing entry in the hash map. For a remove operation, an invalid or unallocated bucket results in reclaiming memory and the occupancy bitmap is updated accordingly. Therefore, the size of the sparse hash map grows with the actual number of entries, unlike a linear table indexed by a logical or physical address.

***Block State.*** In addition to the logical-to-physical map, the SSC maintains additional data for internal operations, such as the state of all flash blocks for garbage collection and usage statistics to guide wear-leveling and eviction policies. This information is accessed by physical address only, and therefore can be stored in the out-of-band (OOB) area of each flash page. This is a small area (64–224 bytes) associated with each page [6] that can be written at the same time as data. To support fast address translation for physical addresses when garbage collecting or evicting data, the SSC also maintains a reverse map, stored in the OOB area of each page and updates it on writes. With each block-level map entry in device memory, the SSC also stores a dirty-block bitmap recording which pages within the erase block contain dirty data.

### 4.2 Consistent Cache Interface

FlashTier provides a consistent cache interface that reflects the needs of a cache to (i) persist cached data across a system reboot or crash, and (ii) never return stale data because of an inconsistent mapping. Most SSDs provide the read-/write interface of disks, augmented with a *trim* command to inform the SSD that data need not be saved during garbage collection. However, the existing interface is insufficient for SSD caches because it leaves undefined what data is returned when reading an address that has been written or evicted [28]. An SSC, in contrast, provides an interface with precise guarantees over consistency of both cached data and mapping information. The SSC interface is a small extension to the standard SATA/SCSI read/write/trim commands.

#### 4.2.1 Interface

FlashTier's interface consists of six operations:

| | |
|---|---|
| **write-dirty** | Insert new block or update existing block with dirty data. |
| **write-clean** | Insert new block or update existing block with clean data. |
| **read** | Read block if present or return error. |
| **evict** | Evict block immediately. |
| **clean** | Allow future eviction of block. |
| **exists** | Test for presence of dirty blocks. |

We next describe these operations and their usage by the cache manager in more detail.

***Writes.*** FlashTier provides two write commands to support write-through and write-back caching. For write-back caching, the *write-dirty* operation guarantees that data is durable before returning. This command is similar to a standard SSD write, and causes the SSC also update the mapping, set the dirty bit on the block and save the mapping to flash using logging. The operation returns only when the data and mapping are durable in order to provide a consistency guarantee.

The *write-clean* command writes data and marks the block as clean, so it can be evicted if space is needed. This operation is intended for write-through caching and when fetching a block into the cache on a miss. The guarantee of *write-clean* is that a subsequent read will return either the new data or a not-present error, and hence the SSC must ensure that data and metadata writes are properly ordered. Unlike *write-dirty*, this operation can be buffered; if the power fails before the write is durable, the effect is the same as if the SSC silently evicted the data. However, if the write replaces previous data at the same address, the mapping change must be durable before the SSC completes the request.

***Reads.*** A read operation looks up the requested block in the device map. If it is present it returns the data, and otherwise returns an error. The ability to return errors from reads serves three purposes. First, it allows the cache manager to request any block, without knowing if it is cached. This means that the manager need not track the state of all cached blocks precisely; approximation structures such as a Bloom Filter can be used safely to prevent reads that miss in the SSC. Second, it allows the SSC to manage space internally by evicting data. Subsequent reads of evicted data return an error. Finally, it simplifies the consistency guarantee: after a block is written with *write-clean*, the cache can still return an error on reads. This may occur if a crash occurred after the write but before the data reached flash.

***Eviction.*** FlashTier also provides a new *evict* interface to provide a well-defined read-after-evict semantics. After issuing this request, the cache manager knows that the cache cannot contain the block, and hence is free to write updated versions to disk. As part of the eviction, the SSC removes the forward and reverse mappings for the logical and physical pages from the hash maps and increments the number of invalid pages in the erase block. The durability guarantee of *evict* is similar to *write-dirty*: the SSC ensures the eviction is durable before completing the request.

Explicit eviction is used to invalidate cached data when writes are sent only to the disk. In addition, it allows the cache manager to precisely control the contents of the SSC. The cache manager can leave data dirty and explicitly evict selected victim blocks. Our implementation, however, does not use this policy.

***Block cleaning.*** A cache manager indicates that a block is clean and may be evicted with the *clean* command. It updates the block metadata to indicate that the contents are clean, but does not touch the data or mapping. The operation is asynchronous, after a crash cleaned blocks may return to their dirty state.

A write-back cache manager can use *clean* to manage the capacity of the cache: the manager can clean blocks that are unlikely to be accessed to make space for new writes. However, until the space is actually needed, the data remains cached and can still be accessed. This is similar to the management of free pages by operating systems, where page contents remain usable until they are rewritten.

***Testing with* exists.** The *exists* operation allows the the cache manager to query the state of a range of cached blocks. The cache manager passes a block range, and the SSC returns the dirty bitmaps from mappings within the range. As this information is stored in the SSC's memory, the operation does not have to scan flash. The returned data includes a single bit for each block in the requested range that, if set, indicates the block is present and dirty. If the block is not present or clean, the bit is cleared. While this version of *exists* returns only dirty blocks, it could be extended to return additional per-block metadata, such as access time or frequency, to help manage cache contents.

This operation is used by the cache manager for recovering the list of dirty blocks after a crash. It scans the entire disk address space to learn which blocks are dirty in the cache so it can later write them back to disk.

#### 4.2.2 Persistence

SSCs rely on a combination of logging, checkpoints, and out-of-band writes to persist its internal data. Logging allows low-latency writes to data distributed throughout memory, while checkpointing provides fast recovery times by keeping the log short. Out-of-band writes provide a low-latency means to write metadata near its associated data.

***Logging.*** An SSC uses an operation log to persist changes to the sparse hash map. A log record consists of a monotonically increasing log sequence number, the logical and physical block addresses, and an identifier indicating whether this is a page-level or block-level mapping.

For operations that may be buffered, such as *clean* and *write-clean*, an SSC uses asynchronous group commit [17] to flush the log records from device memory to flash device periodically. For operations with immediate consistency guarantees, such as *write-dirty* and *evict*, the log is flushed as part of the operation using a synchronous commit. For example, when updating a block with *write-dirty*, the SSC will create a log record invalidating the old mapping of block number to physical flash address and a log record inserting the new mapping for the new block address. These are flushed using an atomic-write primitive [33] to ensure that transient states exposing stale or invalid data are not possible.

***Checkpointing.*** To ensure faster recovery and small log size, SSCs checkpoint the mapping data structure periodically so that the log size is less than a fixed fraction of the size of checkpoint. This limits the cost of checkpoints, while ensuring logs do not grow too long. It only checkpoints the forward mappings because of the high degree of sparseness in the logical address space. The reverse map used for invalidation operations and the free list of blocks are clustered on flash and written in-place using out-of-band updates to individual flash pages. FlashTier maintains two checkpoints on dedicated regions spread across different planes of the SSC that bypass address translation.

***Recovery.*** The recovery operation reconstructs the different mappings in device memory after a power failure or reboot. It first computes the difference between the sequence number of the most recent committed log record and the log sequence number corresponding to the beginning of the most recent checkpoint. It then loads the mapping checkpoint and replays the log records falling in the range of the computed difference. The SSC performs roll-forward recovery for both the page-level and block-level maps, and reconstructs the reverse-mapping table from the forward tables.

### 4.3 Free Space Management

FlashTier provides high write performance by leveraging the semantics of caches for garbage collection. SSDs use garbage collection to compact data and create free erased blocks. Internally, flash is organized as a set of *erase blocks*, which contain a number of pages, typically 64. Garbage collection coalesces the live data from multiple blocks and erases blocks that have no valid data. If garbage collection is performed frequently, it can lead to *write amplification*, where data written once must be copied multiple times, which hurts performance and reduces the lifetime of the drive [16, 18].

The hybrid flash translation layer in modern SSDs separates the drive into data blocks and log blocks. New data is written to the log and then merged into data blocks with garbage collection. The data blocks are managed with block-level translations (256 KB) while the log blocks use finer-grained 4 KB translations. Any update to a data block is performed by first writing to a log block, and later doing a *full merge* that creates a new data block by merging the old data block with the log blocks containing overwrites to the data block.

***Silent eviction.*** SSCs leverage the behavior of caches by evicting data when possible rather than copying it as part of garbage collection. FlashTier implements a *silent eviction* mechanism by integrating cache replacement with garbage collection. The garbage collector selects a flash plane to clean and then selects the top-k victim blocks based on a policy described below. It then removes the mappings for any valid pages within the victim blocks, and erases the victim blocks. Unlike garbage collection, FlashTier does not incur any copy overhead for rewriting the valid pages.

When using silent eviction, an SSC will only consider blocks written with *write-clean* or explicitly cleaned. If there are not enough candidate blocks to provide free space, it reverts to regular garbage collection. Neither *evict* nor *clean* operations trigger silent eviction; they instead update metadata indicating a block is a candidate for eviction during the next collection cycle.

***Policies.*** We have implemented two policies to select victim blocks for eviction. Both policies only apply silent eviction to data blocks. The first policy, *SE-Util* selects the erase bock with the smallest number of valid pages (*i.e.*, lowest utilization). This minimizes the number of valid pages purged, although it may evict recently referenced data. This policy only creates erased *data blocks* and not log blocks, which still must use normal garbage collection.

The second policy, *SE-Merge* uses the same policy for selecting candidate victims (utilization), but allows the erased blocks to be used for either data or logging. This allows the number of log blocks to increase, which reduces garbage collection costs: with more log blocks, garbage collection of them is less frequent, and there may be fewer valid pages in each log block. However, this approach increases memory usage to store fine-grained translations for each block in the log. With *SE-Merge*, new data blocks are created via switch merges, which convert a sequentially written log block into a data block without copying data.

### 4.4 Cache Manager

The cache manager is based on Facebook's FlashCache for Linux [10]. It provides support for both write-back and write-through caching modes and implements a recovery mechanism to enable cache use after a crash.

The write-through policy consults the cache on every read. As read misses require only access to the in-memory mapping, these incur little delay. The cache manager, fetches the data from the disk on a miss and writes it to the SSC with *write-clean*. Similarly, the cache manager sends new data from writes both to the disk and to the SSC with *write-clean*. As all data is clean, the manager never sends any *clean* requests. We optimize the design for memory consumption

assuming a high hit rate: the manager stores no data about cached blocks, and consults the cache on every request. An alternative design would be to store more information about which blocks are cached in order to avoid the SSC on most cache misses.

The write-back mode differs on the write path and in cache management; reads are handled similarly to write-through caching. On a write, the cache manager use *write-dirty* to write the data to the SSC only. The cache manager maintains an in-memory table of cached dirty blocks. Using its table, the manager can detect when the percentage of dirty blocks within the SSC exceeds a set threshold, and if so issues *clean* commands for LRU blocks. Within the set of LRU blocks, the cache manager prioritizes cleaning of contiguous dirty blocks, which can be merged together for writing to disk. The cache manager then removes the state of the clean block from its table.

The dirty-block table is stored as a linear hash table containing metadata about each dirty block. The metadata consists of an 8-byte associated disk block number, an optional 8-byte checksum, two 2-byte indexes to the previous and next blocks in the LRU cache replacement list, and a 2-byte block state, for a total of 14-22 bytes.

After a failure, a write-through cache manager may immediately begin using the SSC. It maintains no transient in-memory state, and the cache-consistency guarantees ensure it is safe to use all data in the SSC. Similarly, a write-back cache manager can also start using the cache immediately, but must eventually repopulate the dirty-block table in order to manage cache space. The cache manager scans the entire disk address space with *exists*. This operation can overlap normal activity and thus does not delay recovery.

## 5. Implementation

The implementation of FlashTier entails three components: the cache manager, an SSC functional emulator, and an SSC timing simulator. The first two are Linux kernel modules (kernel 2.6.33), and the simulator models the time for the completion of each request.

We base the cache manager on Facebook's FlashCache [10]. We modify its code to implement the cache policies described in the previous section. In addition, we added a trace-replay framework invokable from user-space with direct I/O calls to evaluate performance.

We base the SSC simulator on FlashSim [23]. The simulator provides support for an SSD controller, and a hierarchy of NAND-flash packages, planes, dies, blocks and pages. We enhance the simulator to support page-level and hybrid mapping with different mapping data structures for address translation and block state, write-ahead logging with synchronous and asynchronous group commit support for insert and remove operations on mapping, periodic checkpointing from device memory to a dedicated flash region, and a roll-forward recovery logic to reconstruct the mapping and block state. We have two basic configurations of the simulator, targeting the two silent eviction policies. The first configuration (termed *SSC* in the evaluation) uses the *SE-Util* policy and statically reserves a portion of the flash for log blocks and provisions enough memory to map these with page-level mappings. The second configuration, *SSC-R*, uses the *SE-Merge* policy and allows the fraction of log blocks to vary based on workload but must reserve memory capacity for the maximum fraction at page level. In our tests, we fix log blocks at 7% of capacity for SSC and allow the fraction to range from 0-20% for SSC-R.

We implemented our own FTL that is similar to the FAST FTL [25]. We integrate silent eviction with background and foreground garbage collection for data blocks, and with merge operations for *SE-Merge* when recycling log blocks [16]. We also implement inter-plane copy of valid pages for garbage collection (where pages collected from one plane are written to another) to balance the number of free blocks across all planes. The simulator also tracks the utilization of each block for the silent eviction policies.

The SSC emulator is implemented as a block device and uses the same code for SSC logic as the simulator. In order to emulate large caches efficiently (much larger than DRAM), it stores the metadata of all cached blocks in memory but discards data on writes and returns fake data on reads, similar to David [2].

## 6. Evaluation

We compare the cost and benefits of FlashTier's design components against traditional caching on SSDs and focus on three key questions:

- What are the benefits of providing a sparse unified cache address space for FlashTier?
- What is the cost of providing cache consistency and recovery guarantees in FlashTier?
- What are the benefits of silent eviction for free space management and write performance in FlashTier?

We describe our methods and present a summary of our results before answering these questions in detail.

### 6.1 Methods

We emulate an SSC with the parameters in Table 2, which are taken from the latencies of the third generation Intel 300 series SSD [20]. We scale the size of each plane to vary the SSD capacity. On the SSD, we over provision by 7% of the capacity for garbage collection. The SSC does not require over provisioning, because it does not promise a fixed-size address space. The performance numbers are not parameters but rather are the measured output of the SSC timing simulator, and reflect performance on an empty SSD/SSC. Other mainstream SSDs documented to perform better rely on deduplication or compression, which are orthogonal to our design [31].

| | | | |
|---|---|---|---|
| Page read/write | 65/85 $\mu$s | Block erase | 1000 $\mu$s |
| Bus control delay | 2 $\mu$s | Control delay | 10 $\mu$s |
| Flash planes | 10 | Erase block/plane | 256 |
| Pages/erase block | 64 | Page size | 4096 bytes |
| Seq. Read | 585 MB/sec | Rand. Read | 149,700 IOPS |
| Seq. Write | 124 MB/sec | Rand. Write | 15,300 IOPS |

**Table 2. Emulation parameters.**

| Workload | Range | Unique Blocks | Total Ops. | % Writes |
|---|---|---|---|---|
| **homes** | 532 GB | 1,684,407 | 17,836,701 | 95.9 |
| **mail** | 277 GB | 15,136,141 | 462,082,021 | 88.5 |
| **usr** | 530 GB | 99,450,142 | 116,060,427 | 5.9 |
| **proj** | 816 GB | 107,509,907 | 311,253,714 | 14.2 |

**Table 3. Workload Characteristics**: All requests are sector-aligned and 4,096 bytes.

We compare the FlashTier system against the *Native* system, which uses uses the unmodified Facebook FlashCache cache manager and the FlashSim SSD simulator. We experiment with both write-through and write-back modes of caching. The write-back cache manager stores its metadata on the SSD, so it can recover after a crash, while the write-through cache manager cannot.

We use four real-world traces with the characteristics shown in Table 3. These traces were collected on systems with different I/O workloads that consist of a departmental email server (*mail* workload), and file server (*homes* workload) [24]; and a small enterprise data center hosting user home directories (*usr* workload) and project directories (*proj* workload) [27]. Workload duration varies from 1 week (*usr* and *proj*) to 3 weeks (*homes* and *mail*). The range of logical block addresses is large and sparsely accessed, which helps evaluate the memory consumption for address translation. The traces also have different mixes of reads and writes (the first two are write heavy and the latter two are read heavy) to let us analyze the performance impact of the SSC interface and silent eviction mechanisms. To keep replay times short, we use only the first 20 million requests from the *mail* workload, and the first 100 million requests from *usr* and *proj* workloads.

### 6.2 System Comparison

FlashTier improves on caching with an SSD by improving performance, reducing memory consumption, and reducing wear on the device. We begin with a high-level comparison of FlashTier and SSD caching, and in the following sections provide a detailed analysis of FlashTier's behavior.

***Performance.*** Figure 3 shows the performance of the two FlashTier configurations with SSC and SSC-R in write-back and write-through modes relative to the native system with an SSD cache in write-back mode. For the write-intensive *homes* and *mail* workloads, the FlashTier system with SSC outperforms the native system by 59-128% in write-back mode and 38-79% in write-through mode. With SSC-R, the FlashTier system outperforms the native system by 101-167% in write-back mode and by 65-102% in write-through mode. The write-back systems improve the performance of cache writes, and hence perform best on these workloads.

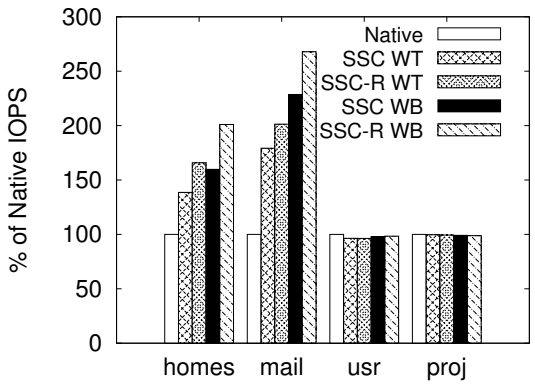


**Figure 3. Application Performance**: The performance of write-through and write-back FlashTier systems normalized to native write-back performance. We do not include native write-through because it does not implement durability.

For the read-intensive *usr* and *proj* workloads, the native system performs almost identical to the FlashTier system. The performance gain comes largely from garbage collection, as we describe in Section 6.5, which is offset by the cost of FlashTier's consistency guarantees, which are described in Section 6.4.

***Memory consumption.*** Table 4 compares the memory usage on the device for the native system and FlashTier. Overall, Flashtier with the SSC consumes 11% more device memory and with SSC-R consumes 160% more. However, both FlashTier configurations consume 89% less host memory. We describe these results more in Section 6.3.

***Wearout.*** For Figure 3, we also compare the number of erases and the wear differential (indicating a skewed write pattern) between the native and FlashTier systems. Overall, on the write-intensive *homes* and *mail* workloads, FlashTier with SSC performs 45% fewer erases, and with SSC-R performs 57% fewer. On the read-intensive *usr* and *proj* workloads, the SSC configuration performs 3% fewer erases and 6% fewer with SSC-R. The silent-eviction policy accounts for much of the difference in wear: in write-heavy workloads it reduces the number of erases but in read-heavy workloads may evict useful data that has to be written again. We describe these results more in Section 6.5.

### 6.3 FlashTier Address Space Management

In this section, we evaluate the device and cache manager memory consumption from using a single sparse address space to maintain mapping information and block state.

***Device memory usage.*** Table 4 compares the memory usage on the device for the native system and FlashTier. The native system SSD stores a dense mapping translating from

| | Size | SSD | SSC | SSC-R | Native | FTCM |
|---|---|---|---|---|---|---|
| **Workload** | **GB** | **Device (MB)** | | | **Host (MB)** | |
| **homes** | 1.6 | 1.13 | 1.33 | 3.07 | 8.83 | 0.96 |
| **mail** | 14.4 | 10.3 | 12.1 | 27.4 | 79.3 | 8.66 |
| **usr** | 94.8 | 66.8 | 71.1 | 174 | 521 | 56.9 |
| **proj** | 102 | 72.1 | 78.2 | 189 | 564 | 61.5 |
| **proj-50** | 205 | 144 | 152 | 374 | 1,128 | 123 |

**Table 4. Memory Consumption:** Total size of cached data, and host and device memory usage for Native and FlashTier systems for different traces. FTCM: write-back FlashTier Cache Manager.

SSD logical block address space to physical flash addresses. The FlashTier system stores a sparse mapping from disk logical block addresses to physical flash addresses using a sparse hash map. Both systems use a hybrid layer mapping (HLM) mixing translations for entire erase blocks with per-page translations. We evaluate both SSC and SSC-R configurations.

For this test, both SSD and SSC map 93% of the cache using 256 KB blocks and the remaining 7% is mapped using 4 KB pages. SSC-R stores page-level mappings for a total of 20% for reserved space. As described earlier in Section 4.3, SSC-R can reduce the garbage collection cost by using the *SE-Merge* policy to increase the percentage of log blocks. In addition to the target physical address, both SSC configurations store an eight-byte dirty-page bitmap with each block-level map entry in device memory. This map encodes which pages within the erase block are dirty.

We measure the device memory usage as we scale the cache size to accommodate the 25% most popular blocks from each of the workloads, and top 50% for *proj-50*. The SSD averages 2.8 bytes/block, while the SSC averages 3.1 and SSC-R averages 7.4 (due to its extra page-level mappings). The *homes* trace has the lowest density, which leads to the highest overhead (3.36 bytes/bock for SSC and 7.7 for SSC-R), while the *proj-50* trace has the highest density, which leads to lower overhead (2.9 and 7.3 bytes/block).

Across all cache sizes from 1.6 GB to 205 GB, the sparse hash map in SSC consumes only 5–17% more memory than SSD. For a cache size as large as 205 GB for *proj-50*, SSC consumes no more than 152 MB of device memory, which is comparable to the memory requirements of an SSD. The performance advantages of the SSC-R configuration comes at the cost of doubling the required device memory, but is still only 374 MB for a 205 GB cache.

The average latencies for remove and lookup operations are less than 0.8 $\mu$s for both SSD and SSC mappings. For inserts, the sparse hash map in SSC is 90% slower than SSD due to the rehashing operations. However, these latencies are much smaller than the bus control and data delays and thus have little impact on the total time to service a request.

***Host memory usage.*** The cache manager requires memory to store information about cached blocks. In write-through mode, the FlashTier cache manager requires no per-block state, so its memory usage is effectively zero, while the native system uses the same amount of memory for both write-back and write-through. Table 4 compares the cache-manager memory usage in write-back mode for native and FlashTier configured with a dirty percentage threshold of 20% of the cache size (above this threshold the cache manager will clean blocks).

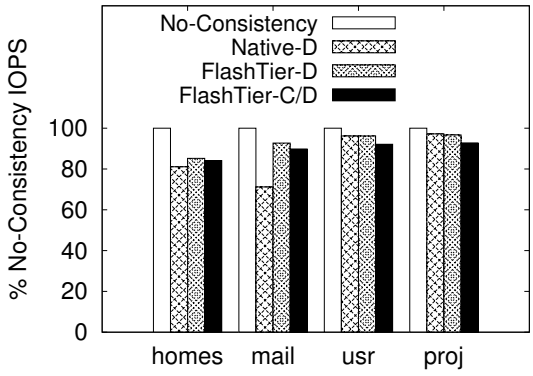


**Figure 4. Consistency Cost**: No-consistency system does not provide any consistency guarantees for cached data or metadata. Native-D and FlashTier-D systems only provide consistency for dirty data. FlashTier-C/D provides consistency for both clean and dirty data.

Overall, the FlashTier cache manager consumes less than 11% of the native cache manager. The native system requires 22 bytes/block for a disk block number, checksum, LRU indexes and block state. The FlashTier system stores a similar amount of data (without the Flash address) for dirty blocks, but nothing for clean blocks. Thus, the FlashTier system consumes only 2.4 bytes/block, an 89% reduction. For a cache size of 205 GB, the savings with FlashTier cache manager are more than 1 GB of host memory.

Overall, the SSC provides a 78% reduction in total memory usage for the device and host combined. These savings come from the unification of address space and metadata across the cache manager and SSC. Even with the additional memory used for the SSC-R device, it reduces total memory use by 60%. For systems that rely on host memory to store mappings, such as FusionIO devices [11], these savings are immediately realizable.

### 6.4 FlashTier Consistency

In this section, we evaluate the cost of crash consistency and recovery by measuring the overhead of logging, checkpointing and the time to recover. On a system with non-volatile memory or that can flush RAM contents to flash on a power failure, consistency imposes no performance cost because there is no need to write logs or checkpoints.

***Consistency cost.*** We first measure the performance cost of FlashTier's consistency guarantees by comparing against a baseline *no-consistency* system that does not make the mapping persistent. Figure 4 compares the throughput of FlashTier with the SSC configuration and the native sys-

tem, which implements consistency guarantees by writing back mapping metadata, normalized to the no-consistency system. For FlashTier, we configure group commit to flush the log buffer every 10,000 write operations or when a synchronous operation occurs. In addition, the SSC writes a checkpoint if the log size exceeds two-thirds of the checkpoint size or after 1 million writes, whichever occurs earlier. This limits both the number of log records flushed on a commit and the log size replayed on recovery. For the native system, we assume that consistency for mapping information is provided by out-of-band (OOB) writes to per-page metadata without any additional cost [16].

As the native system does not provide persistence in write-through mode, we only evaluate write-back caching. For efficiency, the native system (Native-D) only saves metadata for dirty blocks at runtime, and loses clean blocks across unclean shutdowns or crash failures. It only saves metadata for clean blocks at shutdown. For comparison with such a system, we show two FlashTier configurations: FlashTier-D, which relaxes consistency for clean blocks by buffering log records for *write-clean*, and FlashTier-C/D, which persists both clean and dirty blocks using synchronous logging.

For the write-intensive *homes* and *mail* workloads, the extra metadata writes by the native cache manager to persist block state reduce performance by 18-29% compared to the no-consistency system. The *mail* workload has 1.5x more metadata writes per second than *homes*, therefore, incurs more overhead for consistency. The overhead of consistency for persisting clean and dirty blocks in both FlashTier systems is lower than the native system, at 8-15% for FlashTier-D and 11-16% for FlashTier-C/D. This overhead stems mainly from synchronous logging for insert/remove operations from *write-dirty* (inserts) and *write-clean* (removes from overwrite). The *homes* workload has two-thirds fewer *write-clean* operations than *mail*, and hence there is a small performance difference between the two FlashTier configurations.

For read-intensive *usr* and *proj* workloads, the cost of consistency is low for the native system at 2-5%. The native system does not incur any synchronous metadata updates when adding clean pages from a miss and batches sequential metadata updates. The FlashTier-D system performs identical to the native system because the majority of log records can be buffered for *write-clean*. The FlashTier-C/D system's overhead is only slightly higher at 7%, because clean writes following a miss also require synchronous logging.

We also analyze the average request response time for both the systems. For write-intensive workloads *homes* and *mail*, the native system increases response time by 24-37% because of frequent small metadata writes. Both FlashTier configurations increase response time less, by 18-32%, due to logging updates to the map. For read-intensive workloads, the average response time is dominated by the read latencies of the flash medium. The native and FlashTier systems incur a 3-5% increase in average response times for these workloads respectively. Overall, the extra cost of consistency for the request response time is less than 26 $\mu$s for all workloads with FlashTier.

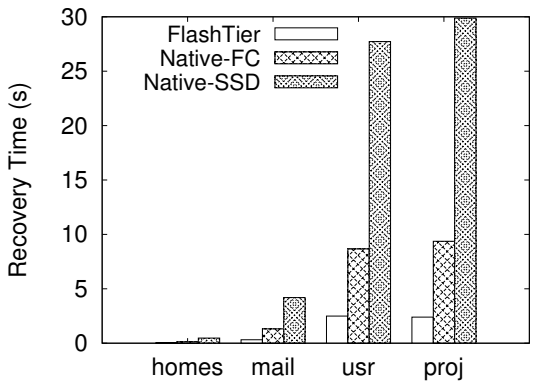


**Figure 5. Recovery Time**: Native-FC accounts for only recovering FlashCache cache manager state. Native-SSD accounts for only recovering the SSD mapping.

***Recovery time.*** Figure 5 compares the time for recovering after a crash. The mapping and cache sizes for each workload are shown in Table 4.

For FlashTier, the only recovery is to reload the mapping and block state into device memory. The cache manager metadata can be read later. FlashTier recovers the mapping by replaying the log on the most recent checkpoint. It recovers the cache manager state in write-back mode using *exists* operations. This is only needed for space management, and thus can be deferred without incurring any start up latency. In contrast, for the native system *both* the cache manager and SSD must reload mapping metadata.

Most SSDs store the logical-to-physical map in the OOB area of a physical page. We assume that writing to the OOB is free, as it can be overlapped with regular writes and hence has little impact on write performance. After a power failure, however, these entries must be read to reconstruct the mapping [16], which requires scanning the whole SSD in the worst case. We estimate the best case performance for recovering using an OOB scan by reading just enough OOB area to equal the size of the mapping table.

The recovery times for FlashTier vary from 34 ms for a small cache (*homes*) to 2.4 seconds for *proj* with a 102 GB cache. In contrast, recovering the cache manager state alone for the native system is much slower than FlashTier and takes from 133 ms for *homes* to 9.4 seconds for *proj*. Recovering the mapping in the native system is slowest because scanning the OOB areas require reading many separate locations on the SSD. It takes from 468 ms for *homes* to 30 seconds for *proj*.

### 6.5 FlashTier Silent Eviction

In this section, we evaluate the impact of silent eviction on caching performance and wear management. We com-

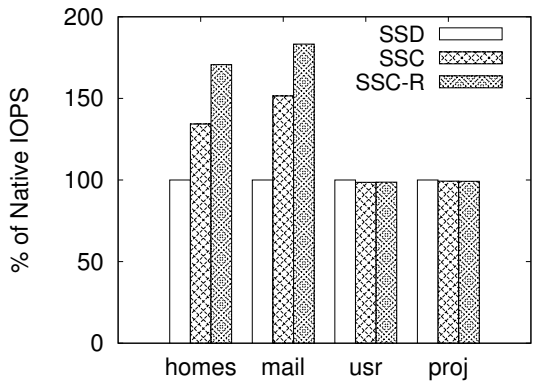


**Figure 6. Garbage Collection Performance**: Comparing the impact of garbage collection on caching performance for different workloads on SSD, SSC and SSC-R devices.

pare the behavior of caching on three devices: SSD, SSC and SSC-R, which use garbage collection and silent eviction with SE-Util and SE-Merge policies. For all traces, we replay the trace on a cache sized according to Table 4. To warm the cache, we replay the first 15% of the trace before gathering statistics, which also ensures there are no available erased blocks. To isolate the performance effects of silent eviction, we disabled logging and checkpointing for these tests and use only write-through caching, in which the SSC is entirely responsible for replacement.

***Garbage Collection.*** Figure 6 shows the performance impact of silent eviction policies on SSC and SSC-R. We focus on the write-intensive *homes* and *mail* workloads, as the other two workloads have few evictions. On these workloads, the Flashtier system with SSC outperforms the native SSD by 34-52%, and SSC-R by 71-83%. This improvement comes from the reduction in time spent for garbage collection because silent eviction avoids reading and rewriting data. This is evidenced by the difference in write amplification, shown in Table 5. For example, on *homes*, the native system writes each block an additional 2.3 times due to garbage collection. In contrast, with SSC the block is written an additional 1.84 times, and with SSC-R, only 1.3 more times. The difference between the two policies comes from the additional availability of log blocks in SSC-R. As described in Section 4.3, having more log blocks improves performance for write-intensive workloads by delaying garbage collection and eviction, and decreasing the number of valid blocks that are discarded by eviction. The performance on *mail* is better than *homes* because the trace has 3 times more overwrites per disk block, and hence more nearly empty erase blocks to evict.

***Cache Misses.*** The time spent satisfying reads is similar in all three configurations across all four traces. As *usr* and *proj* are predominantly reads, the total execution times for these traces is also similar across devices. For these traces, the miss rate, as shown in Table 5, increases negligibly.

On the write-intensive workloads, the FlashTier device has to impose its policy on what to replace when making space for new writes. Hence, there is a larger increase in miss rate, but in the worst case, for *homes*, is less than 2.5 percentage points. This increase occurs because the SSC eviction policy relies on erase-block utilization rather than recency, and thus evicts blocks that were later referenced and caused a miss. For SSC-R, though, the extra log blocks again help performance by reducing the number of valid pages evicted, and the miss rate increases by only 1.5 percentage points on this trace. As described above, this improved performance comes at the cost of more device memory for page-level mappings. Overall, both silent eviction policies keep useful data in the cache and greatly increase the performance for recycling blocks.

***Wear Management.*** In addition to improving performance, silent eviction can also improve reliability by decreasing the number of blocks erased for merge operations. Table 5 shows the total number of erase operations and the maximum wear difference (indicating that some blocks may wear out before others) between any two blocks over the execution of different workloads on SSD, SSC and SSC-R.

For the write-intensive *homes* and *mail* workloads, the total number of erases reduce for SSC and SSC-R. In addition, they are also more uniformly distributed for both SSC and SSC-R. We find that most erases on SSD are during garbage collection of *data blocks* for copying valid pages to free log blocks, and during full merge operations for recycling *log blocks*. While SSC only reduces the number of copy operations by evicting the data instead, SSC-R provides more log blocks. This reduces the total number of full merge operations by replacing them with switch merges, in which a full log block is made into a data block. On these traces, SSC and SSC-R reduce the total number of erases by an average of 26% and 35%, and the overhead of copying valid pages by an average of 32% and 52% respectively, as compared to the SSD.

For the read-intensive *usr* and *proj* workloads, most blocks are read-only, so the total number of erases and wear difference is lower for all three devices. The SSC increases erases by an average of 5%, because it evicts data that must later be brought back in and rewritten. However the low write rate for these traces makes reliability less of a concern. For SSC-R, the number of erases decrease by an average of 2%, again from reducing the number of merge operations.

Both SSC and SSC-R greatly improve performance and on important write-intensive workloads, also decrease the write amplification and the resulting erases. Overall, the SSC-R configuration performs better, has a lower miss rate, and better reliability and wear-leveling achieved through increased memory consumption and a better replacement policy.

| | Erases | | | Wear Diff. | | | Write Amp. | | | Miss Rate | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Workload** | **SSD** | **SSC** | **SSC-R** | **SSD** | **SSC** | **SSC-R** | **SSD** | **SSC** | **SSC-R** | **SSD** | **SSC** | **SSC-R** |
| **homes** | 878,395 | 829,356 | 617,298 | 3,094 | 864 | 431 | 2.30 | 1.84 | 1.30 | 10.4 | 12.8 | 11.9 |
| **mail** | 880,710 | 637,089 | 525,954 | 1,044 | 757 | 181 | 1.96 | 1.08 | 0.77 | 15.6 | 16.9 | 16.5 |
| **usr** | 339,198 | 369,842 | 325,272 | 219 | 237 | 122 | 1.23 | 1.30 | 1.18 | 10.6 | 10.9 | 10.8 |
| **proj** | 164,807 | 166,712 | 164,527 | 41 | 226 | 17 | 1.03 | 1.04 | 1.02 | 9.77 | 9.82 | 9.80 |

**Table 5. Wear Distribution:** For each workload, the total number of erase operations, the maximum wear difference between blocks, the write amplification, and the cache miss rate is shown for SSD, SSC and SSC-R.

## 7. Related Work

The FlashTier design draws on past work investigating the use of solid-state memory for caching and hybrid systems.

**SSD Caches.** Guided by the price, power and performance of flash, cache management on flash SSDs has been proposed for fast access to disk storage. Windows and Solaris have software cache managers that use USB flash drives and solid-state disks as read-optimized disk caches managed by the file system [3, 14]. Oracle has a write-through flash cache for databases [32] and Facebook has started the deployment of their in-house write-back cache manager to expand the OS cache for managing large amounts of data on their Timeline SQL servers [10, 36]. Storage vendors have also proposed the use of local SSDs as write-through caches to centrally-managed storage shared across virtual machine servers [5, 29]. However, all these software-only systems are still limited by the narrow storage interface, multiple levels of address space, and free space management within SSDs designed for persistent storage. In contrast, FlashTier provides a novel consistent interface, unified address space, and silent eviction mechanism within the SSC to match the requirements of a cache, yet maintaining complete portability for applications by operating at block layer.

**Hybrid Systems.** SSD vendors have recently proposed new flash caching products, which cache most-frequently accessed reads and write I/O requests to disk [12, 30]. FlashCache [22] and the flash-based disk cache [35] also propose specialized hardware for caching. Hybrid drives [4] provision small amounts of flash caches within a hard disk for improved performance. Similar to these systems, FlashTier allows custom control of the device over free space and wear management designed for the purpose of caching. In addition, FlashTier also provides a consistent interface to persist both clean and dirty data. Such an interface also cleanly separates the responsibilities of the cache manager, the SSC and disk, unlike hybrid drives, which incorporate all three in a single device. The FlashTier approach provides more flexibility to the OS and applications for informed caching.

**Informed Caching.** Past proposals for multi-level caches have argued for informed and exclusive cache interfaces to provide a single, large unified cache in the context of storage arrays [38, 40]. Recent work on storage tiering and differentiated storage services has further proposed to classify I/O and use different policies for cache allocation and eviction on SSD caches based on the information available to the OS and applications [15, 26]. However, all these systems are still limited by the narrow storage interface of SSDs, which restricts the semantic information about blocks available to the cache. The SSC interface bridges this gap by exposing primitives to the OS for guiding cache allocation on writing clean and dirty data, and an explicit *evict* operation for invalidating cached data.

**Storage Interfaces.** Recent work on a new nameless-write SSD interface and virtualized flash storage for file systems have argued for removing the costs of indirection within SSDs by exposing physical flash addresses to the OS [41], providing caching support [28], and completely delegating block allocation to the SSD [21]. Similar to these systems, FlashTier unifies multiple levels of address space, and provides more control over block management to the SSC. In contrast, FlashTier is the first system to provide internal flash management and a novel device interface to match the requirements of caching. Furthermore, the SSC provides a virtualized address space using disk logical block addresses, and keeps its interface grounded within the SATA read-/write/trim space without requiring migration callbacks from the device into the OS like these systems.

## 8. Conclusions

Flash caching promises an inexpensive boost to storage performance. However, traditional SSDs are designed to be a drop-in disk replacement and do not leverage the unique behavior of caching workloads, such as a large, sparse address space and clean data that can safely be lost. In this paper, we describe FlashTier, a system architecture that provides a new flash device, the SSC, which has an interface designed for caching. FlashTier provides memory-efficient address space management, improved performance and cache consistency to quickly recover cached data following a crash. As new non-volatile memory technologies become available, such as phase-change and storage-class memory, it will be important to revisit the interface and abstraction that best match the requirements of their memory tiers.

## Acknowledgements

This work is supported in part by National Science Foundation (NSF) grant CNS-0834473. We would like to thank our shepherd Andrew Warfield for his valuable feedback. Swift has a significant financial interest in Microsoft.

## References


[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX*, 2008.

[2] N. Agrawal, L. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Emulating Goliath storage systems with David. In *FAST*, 2011.

[3] T. Archer. MSDN Blog: Microsoft ReadyBoost., 2006. `http://blogs.msdn.com/tomarcher/archive/2006/06/02/615199.aspx`.

[4] T. Bisson. Reducing hybrid disk write latency with flash-backed io requests. In *MASCOTS*, 2007.

[5] S. Byan, J. Lentini, L. Pabon, C. Small, and M. W. Storer. Mercury: host-side flash caching for the datacenter. In *FAST Poster*, 2011.

[6] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *FAST*, 2011.

[7] J. Corbet. Barriers and journaling filesystems, May 2008. `http://lwn.net/Articles/283161/`.

[8] S. Doyle and A. Narayan. Enterprise solid state drive endurance. In *Intel IDF*, 2010.

[9] EMC. Fully Automated Storage Tiering (FAST) Cache. `http://www.emc.com/about/glossary/fast-cache.htm`.

[10] Facebook Inc. Facebook FlashCache. `https://github.com/facebook/flashcache`.

[11] FusionIO Inc. ioXtreme PCI-e SSD Datasheet. `http://www.fusionio.com/ioxtreme/PDFs/ioXtremeDS_v.9.pdf`, .

[12] FusionIO Inc. directCache. `http://www.fusionio.com/data-sheets/directcache`, .

[13] Google Inc. Google Sparse Hash. `http://goog-sparsehash.sourceforge.net`.

[14] B. Gregg. Sun Blog: Solaris L2ARC Cache., July 2008. `http://blogs.sun.com/brendan/entry/test`.

[15] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami. Cost effective storage using extent based dynamic tiering. In *FAST*, 2011.

[16] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS*, 2009.

[17] P. Helland, H. Sammer, J. Lyon, R. Carr, and P. Garrett. Group commit timers and high-volume transaction systems. In *Tandem TR 88.1*, 1988.

[18] Intel Corp. Understanding the flash translation layer (ftl) specification, Dec. 1998. Application Note AP-684.

[19] Intel Corp. Intel Smart Response Technology. `http://download.intel.com/design/flash/nand/325554.pdf`, 2011.

[20] Intel Corp. Intel 300 series SSD. `http://ark.intel.com/products/family/56542/Intel-SSD-300-Family`.

[21] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: a file system for virtualized flash storage. In *FAST*, 2010.

[22] T. Kgil and T. N. Mudge. Flashcache: A nand flash memory file cache for low power web servers. In *CASES*, 2006.

[23] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar. FlashSim: A simulator for nand flash-based solid-state drives. *Advances in System Simulation, International Conference on*, 0:125–131, 2009.

[24] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. In *FAST*, 2010.

[25] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst*, 6(3), July 2007.

[26] M. Mesnier, J. B. Akers, F. Chen, and T. Luo. Differentiated storage services. In *SOSP*, 2011.

[27] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-loading: Practical power management for enterprise storage. In *FAST*, 2008.

[28] D. Nellans, M. Zappe, J. Axboe, and D. Flynn. ptrim() + exists(): Exposing new FTL primitives to applications. In *NVMW*, 2011.

[29] NetApp Inc. Flash Cache for Enterprise. `http://www.netapp.com/us/products/storage-systems/flash-cache`.

[30] OCZ Technologies. Synapse Cache SSD. `http://www.ocztechnology.com/ocz-synapse-cache-sata-iii-2-5-ssd.html`.

[31] OCZ Technologies. Vertex 3 SSD. `http://www.ocztechnology.com/ocz-vertex-3-sata-iii-2-5-ssd.html`.

[32] Oracle Corp. Oracle Database Smart Flash Cache. `http://www.oracle.com/technetwork/articles/systems-hardware-architecture/oracle-db-smart-flash-cache-175588.pdf`.

[33] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D.K.Panda. Beyond block i/o: Rethinking traditional storage primitives. In *HPCA*, pages 301 –311, feb. 2011.

[34] V. Prabhakaran, T. Rodeheffer, and L. Zhou. Transactional flash. In *OSDI*, 2008.

[35] D. Roberts, T. Kgil, and T. Mudge. Integrating NAND flash devices onto servers. *CACM*, 52(4):98–106, Apr. 2009.

[36] Ryan Mack. Building Facebook Timeline: Scaling up to hold your life story. `http://www.facebook.com/note.php?note_id=10150468255628920`.

[37] M. Saxena and M. M. Swift. FlashVM: Virtual Memory Management on Flash. In *Usenix ATC*, 2010.

[38] T. M. Wong and J. Wilkes. My cache or yours? Making storage more exclusive. In *Usenix ATC*, 2002.

[39] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *ASPLOS-VI*, 1994.

[40] G. Yadgar, M. Factor, and A. Schuster. Karma: Know-it-all replacement for a multilevel cache. In *FAST*, 2007.

[41] Y. Zhang, L. P. Arulraj, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *FAST*, 2012.