Using Transparent Compression to Improve SSD-based I/O Caches

Thanos Makatos*† Yannis Klonatos*† Manolis Marazakis*
Michail D. Flouris* Angelos Bilas*†

*Foundation for Research and Technology - Hellas (FORTH)
Institute of Computer Science (ICS)
100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece
†Department of Computer Science, University of Crete
P.O. Box 2208, Heraklion, GR 71409, Greece
{mcatos, klonatos, maraz, flouris, bilas}@ics.forth.gr

Abstract

Flash-based solid state drives (SSDs) offer superior performance over hard disks for many workloads. A prominent use of SSDs in modern storage systems is to use these devices as a cache in the I/O path. In this work, we examine how transparent, online I/O compression can be used to increase the capacity of SSD-based caches, thus increasing the cost-effectiveness of the system. We present *FlaZ*, an I/O system that operates at the block-level and is transparent to existing file-systems. To achieve transparent, online compression in the I/O path and maintain high performance, *FlaZ* provides support for variable-size blocks, mapping of logical to physical blocks, block allocation, and cleanup. *FlaZ* mitigates compression and decompression overheads that can have a significant impact on performance by leveraging modern multicore CPUs. We implement *FlaZ* in the Linux kernel and evaluate it on a commodity server with multicore CPUs, using TPC-H, PostMark, and SPECsfs. Our results show that compressed caching trades off CPU cycles for I/O performance and enhances SSD efficiency as a cache by up to 99%, 25%, and 11% for each workload, respectively.

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

General Terms Design, Performance, Experimentation

Keywords Solid state disk caches, I/O performance, Evaluation, Online block-level compression

EuroSys'10, April 13–16, 2010, Paris, France.

1. Introduction

Performance of storage I/O is an important problem in modern systems. The emergence of flash-based solid state drives (SSDs) has the potential to mitigate I/O penalties: SSDs have low read response times and are not affected by seeks. Additionally, recent SSDs provide peak throughput that is significantly superior to magnetic hard disk drives (HDD). Although their capacity continues to increase, there is no indication that the current high cost per GB for SSDs [31] will soon approach that of magnetic disks. As a result, I/O hierarchies will include both HDDs and SSDs, and it is important to examine techniques that can increase SSD cost-efficiency.

One approach to achieve this is the use of multi-level cell (MLC) SSDs that store, e.g. two bits per NAND cell. Another approach, orthogonal to cell density, is to use transparent online data compression in the I/O path. Data compression [25] reduces the space required to store a piece of data, e.g., a file, block, or other data segment, by storing it in compressed form. The original data can then be reconstructed by decompressing the transformed piece of data.

Previously, compression has been mostly applied at the file-level [10, 11, 13]. Although this has the effect of reducing the space required for storing data, it also imposes a number of restrictions. Typically, file-level compression techniques are file-system and OS-specific, which limits where compression is applied in the I/O path. Moreover, file-systems are not usually aware of the properties of physical storage devices, e.g. in a RAID controller. Finally, dealing with compression at the file-level increases management complexity, given that today's storage systems employ multiple file-systems. These concerns can be addressed by moving compression functionality towards lower system layers, such as the block device layer.

In this work we design *FlaZ*, a system that uses SSDs as compressed caches in the I/O path. The address space of SSDs is not visible to applications. *FlaZ* caches data blocks

on dedicated SSD partitions and transparently compresses and uncompresses data as they flow in the I/O path between main memory and SSDs. Data are stored on SSDs only in compressed form and are provided to the file-system in uncompressed form. *FlaZ* does not cache compressed or uncompressed data in DRAM. Although our approach can be extended to use the capacity of SSDs as storage rather than cache, more in the spirit of tiering, or to provide online compression over HDDs as well, we do not explore these directions further. Finally, our design allows for flexibility in placing compression functionality anywhere in the path between the file-system and the storage device, even over popular storage area network configurations. *FlaZ* can run both on a host CPU as well as inside a storage controller plugged in a Windows server. In this work we evaluate the Linux, host-based storage stack.

FlaZ internally consists of two layers, one that achieves transparent compression and one that uses SSDs as an I/O cache. Although these layers are to a large extent independent, in our work we tune their parameters in a combined manner. The caching layer of *FlaZ* is a direct-mapped, write-through cache with one block per cache line, while the compression layer of *FlaZ* is more complex and requires addressing the following issues:

- *Variable block size:* Block-level compression needs to operate on fixed-size input blocks. However, compression generates variable size segments. Therefore, there is a need for per-block placement and size metadata.
- *Logical to physical block mapping:* Block-level compression imposes a many-to-one mapping from logical to physical blocks, as multiple compressed logical blocks are stored in the same physical block. This requires a translation mechanism that imposes low overhead in the common I/O path and scales with the capacity of the underlying devices, as well as a block allocation/deallocation mechanism that affects data layout.
- *Increased number of I/Os:* Using compression increases the number of I/Os required on the critical path during data writes: a write operation will typically require reading another block first, where the compressed data will be placed. This "read-before-write" issue is important for applications that are sensitive to the number or response time of I/Os.
- *Device aging:* Aging of *compressed* SSDs results in fragmentation of data, which may make it harder to allocate new physical blocks and affects locality, making performance of the underlying devices less predictable.

Besides I/O related challenges, compression algorithms themselves introduce significant overheads as well. Although our goal in this work is not to examine alternative compression algorithms and possible optimizations, it is important to quantify their performance impact on the I/O path. Doing so over modern multicore CPUs offers insight about scaling down these overheads in future architectures as the number of cores increases. This understanding can guide further work in three directions: (i) Hiding compression overheads in case of large numbers of outstanding I/Os; (ii) Customizing future CPUs with accelerators for energy and performance purposes; and (iii) Offloading compression overheads from the host to storage controllers.

For our evaluation we use three realistic workloads: TPC-H, PostMark, and SPECsfs. Our results show that compression enhances SSD efficiency as an I/O cache by up to 99%, 25%, and 11% for each workload, respectively. This comes at increased CPU utilization, by up to 450%. We believe that trading CPU with I/O performance is in line with current technology trends for multicore systems [35]. Finally, compressed caching has a negative impact (up to 15%) on response-time bound workloads that use only small I/Os as well as on workloads that fit in the uncompressed cache almost entirely.

The rest of this paper is organized as follows. Section 2 discusses the design of *FlaZ* and how it addresses the associated challenges. Section 3 presents our evaluation methodology. Section 4 presents our experimental results. Section 5 discusses some fundamental considerations for our design. Section 6 discusses previous and related work. Finally, we draw our conclusions in Section 7.

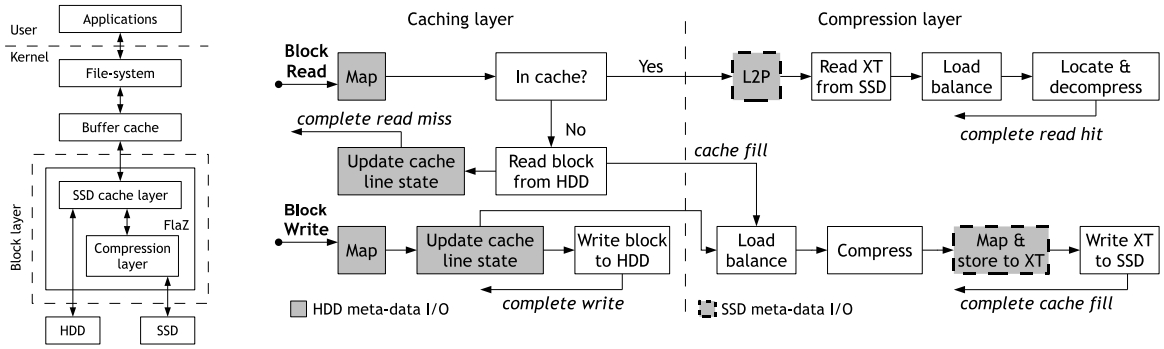
2. System Design

FlaZ internally consists of two layers, one for transparent compression and one that uses dedicated SSD partitions as I/O caches, as shown in Figure 1(a). *FlaZ* intercepts requests in the I/O path and provides a block device abstraction by exporting a contiguous address space of *logical blocks* (LBs) to higher system layers, such as file-systems and databases. Logical blocks are always cached in the SSD cache in compressed form and are always stored on hard disk in uncompressed form. Next, we discuss the issues that arise in each layer in more detail.

2.1 SSD Caching

Although SSD caches bear similarities to DRAM-based caches, there are significant differences as well. First, unlike DRAM caches, SSD caches are persistent and thus, they can avoid warm-up overheads. This is important, as SSD caches can be significantly larger than DRAM caches. Second, the impact of the block mapping policy, e.g. direct-mapped vs. fully-set-associative, is not as clear as in smaller DRAM caches. Traditionally, DRAM caches use a fully-set-associative policy since the small cache size requires reducing capacity conflicts. However, SSD-based caches may be able to use a simpler mapping policy, thus reducing access overhead without increasing capacity conflicts.

Figure 1(b) shows how the caching layer of *FlaZ* handles the intercepted I/O requests. Initially, each logical block is



(a) Architecture (b) Read and write paths

Figure 1. *FlaZ* system architecture and I/O paths.

mapped to the corresponding SSD cache block. For reads, the caching layer checks if the cached block is valid, and if so the initial request is forwarded to the compression layer (*cache hit*). Otherwise, data are fetched from the hard disk (*cache miss*), in cache line units. In the latter case, the initial request is completed when the hard disk read operation is complete and an asynchronous SSD write operation (*cache fill*) is scheduled. For writes (hits or misses), *FlaZ* forwards the operation to both the underlying HDD and SSD devices. This *write-through* policy performs an update in case of a hit and an eviction in case of a miss. In the second case, a metadata update of the valid/dirty bit and tag is required, since the HDD block that previously resided in this cache block is evicted. Future reads for this specific cache block will hit in the SSD cache.

We have implemented a direct-mapped cache because it minimizes metadata requirements and does not impose significant mapping overhead on the critical path. A fully-set-associative cache would require significantly more metadata, especially given the increasing size of the SSDs. Furthermore, we use a write-through policy since it does not require synchronous metadata updates that would be necessary in a write-back policy. In addition, write-back SSD caches will reduce system resilience to SSD failures. A failing SSD with a write-back policy will result in data loss. A write-through policy avoids this issue as well.

FlaZ stores all cache metadata in the beginning of the HDD. This means that metadata writes do not interfere with SSD performance and wear-leveling. To reduce the number of metadata I/Os, metadata are also cached in DRAM. Given that cache metadata are mainly updated in DRAM, the number of additional disk I/Os can be kept small. *FlaZ* requires less than 2.5 MB of metadata per GB of SSD. However, given the increasing size of SSDs, we do present a quantitative evaluation of metadata scalability in Section 4.

2.2 Compression in the I/O path

In general, compression methods operate as follows. They first initialize a workspace, e.g. 256 KB of memory. Then, they compress input into an output buffer, possibly in a piece-wise manner. Finally, the output buffer is finalized and the operation is completed. Decompression follows similar steps. *FlaZ* uses Lempel-Ziff-Welch (LZW) compression [15, 44, 48] though other compression algorithms can also be used. In principle, the compression scheme may change dynamically depending on block contents. In our current implementation we use two alternative LZW implementations on Linux: *zlib* [15] and *lzo* (variant `LZO1X-1`) [34].

Compression, which occurs during writes, is in both libraries a heavy operation and consists of separately compressing each block and placing the result to the corresponding output buffer. This can either be (a) an intermediate buffer, requiring a memory copy to the write-I/O buffer but allowing for higher flexibility, as explained later in this section or (b) the write-I/O buffer itself, resulting in in-place compression. Decompression that occurs during reads, is lighter, and directly places the read data into the read-I/O buffer, without requiring a copy of the uncompressed data.

Block-level compression appears to be deceptively simple: it merely requires intercepting and compressing or decompressing requests as they flow into the system. However, our experience shows that designing a practical and efficient compression system is far from trivial since the above process is complicated by the need for variable-size blocks, mapping of logical to physical blocks, block allocation, extent buffering and cleanup. Next, we discuss each of these issues separately.

2.2.1 Mapping logical blocks to extents

Compressing fixed-size logical blocks results in variable-size segments. These are stored into fixed-size physical units, called *extents*, which are multiples of block size. *FlaZ* uses two mappings to locate the extent and the extent offset, where a compressed block is stored. The first mapping uses a logical to physical translation table, whereas the second one uses a linked list embedded in the extent, as shown in Figure 2.

The logical to physical translation table contains two fields for each logical block: the extent ID followed by a field used to store various flags. This table is stored at the beginning of each compressed block device, indexed by the logical block number.

Compressed blocks stored inside extents have the following structure. A header at the beginning of the extent contains a pointer to the first block as well as a pointer to the free space segment within the extent. All free space in an extent is always contiguous, located at the end of the extent. The first block offset pointer is required for reads to traverse the list of blocks, when searching for a specific logical block. Writes must append new blocks into the extent and thus require a pointer to the free space segment of the extents. Each block is prefixed by another header that contains the logical block number and compressed size along with information about the next block's placement within the extent, forming an extent-wide list of blocks, as shown in Figure 2. The header of each newly appended block is inserted to the beginning of the list. This is essential because two writes for the same block may come close in time and may be stored in the same extent. A read context traversing the list must retrieve the *latest* write of this block, hence it must appear first in the in-extent list. Traversing the list takes time proportional to the number of blocks per extent; however, this is not a problem in practice, as traversal is an in-memory operation. The per-block header (kept inside the extent) along with its mapping and flags (stored at the logical to physical table) are the only metadata required per logical block. Assuming typical space savings of 40% when using compression, a per-block header occupies less than 0.6% of the space of the compressed block.

In case a block cannot be compressed to a smaller size, it is stored uncompressed in the extent. If blocks are not compressed and we use 4-KB extents, this effectively turns *FlaZ* into a log-structured block device. From our experience, less than 2% of blocks fail to compress in the workloads we examine.

2.2.2 Block allocation and immutable physical blocks

Physical blocks are *immutable* in the sense that modifications to logical blocks are always written to a new physical extent on the underlying devices. When a logical block is written for the first time, *FlaZ* compresses the block data and chooses an appropriate extent. In-place update of a logical block is generally complicated, since the block's size may change as a result of the update. For this reason we use immutable physical blocks grouped into larger *extents*, a technique similar to the implementation of log-structured writes [38]. The only difference is that we do not require extents to always be consecutive in the underlying storage, forming a sequential log.

A key benefit of immutable physical blocks is that it does not require reading an extent into memory before modifying it on stable storage. On the other hand, as time progresses a large portion of extents on the underlying device become obsolete. *FlaZ* uses a *cleaner* process which is triggered when the amount of free extents falls below a certain threshold, as described in a later section.

2.2.3 Extent Buffering

To mitigate the impact of additional I/Os on performance due to the "read-modify-write" scheme, the compression layer of *FlaZ* uses a buffer in DRAM for extents. Buffering a small number of extents not only facilitates I/O to the SSD, but also reduces the number of read I/Os; streaming workloads may generate read requests smaller than the extent size, resulting in duplicate I/Os for the same extent by the next read.

The extent buffer is a fully-set-associative data buffer, implemented as a hash table with a collision list and an LRU eviction policy. Key to the hash table is the extent ID. The extent buffer has a somewhat special organization; extents are classified in buckets depending on the amount of free space within the extent. Each bucket is implemented as a LIFO queue of extents. An extent remains in the extent buffer until it becomes "reasonably" full. The extent size, number of buckets, and the total size of the extents buffered are all system parameters. The extent size affects the degree of internal fragmentation and may also affect locality: larger extents have higher probability to contain unrelated data when the application uses random writes, while smaller ones suffer from internal fragmentation.

Finally, the extent buffer design needs to address two more elaborate issues: First, data locality may be affected as extents age by remaining in the extent buffer for long periods. On the other hand, if extents are evicted too quickly, internal fragmentation may be increased. To balance this tradeoff, the extent buffer includes an "aging" timeout that specifies the maximum amount of time an extent can remain in the buffer. Second, when writing compressed blocks from concurrent contexts to the extent buffer, it may make sense from a locality perspective to either write blocks to the same or different extents. As explained later, concurrent writes to an extent involve a tradeoff between lock contention and CPU overhead for memory copies. Our experience has shown that preserving locality offers the best performance, albeit at less efficient space utilization. We determined experimentally that buffering a small number of extents is sufficient for preserving locality. Extents are written back as soon as (a) they become full, i.e. there is not enough space for a write context to append blocks and (b) there is no reference to them, i.e. no active read context uses blocks from this extent.

2.2.4 Extent cleaning

FlaZ maintains two pools of extents with simple semantics: extents can be either completely empty or not empty. Replenishing the free extent pool requires a "cleanup" process, whenever there are few empty extents left. *FlaZ* removes this

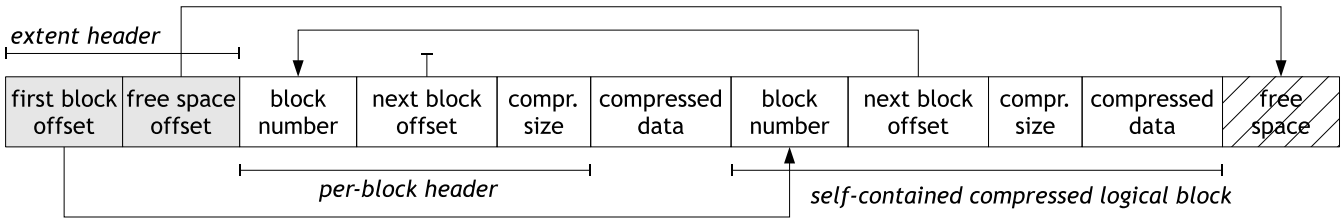


Figure 2. *FlaZ* extent structure. Each compressed block is self-contained, described by a per-block header.

cleanup process from the common path to reduce interference with applications.

The *FlaZ* cleaner runs when the free extent pool drops below a threshold, and has similar goals to the segment cleaner of Sprite LFS [38]. It scans the physical extents on the SSD for full extents using the logical to physical translation mappings and extent headers. It determines which blocks within an extent are "live" by verifying that the mapping of each block (stored inside the block itself) points to that extent. Then, it compacts live blocks into new extents, updates logical to physical translation mappings, and, finally, adds each cleaned extent to the free extent pool. Compaction of live blocks into extents consists of copying these blocks into new extents; no compression/decompression is required. The cleaner is deactivated when the free pool size increases above a threshold. Finally, the only metadata required until the cleaner's next activation is the last scanned extent. This pointer is not required to be persistent; it is merely a hint for the cleaner.

The cleaner generates read I/O traffic proportional to the extents that are scanned and write I/O traffic proportional to the extents resulting from compacting live blocks. To improve the cleaner's efficiency in terms of reclaimed space, we apply a first-fit, decreasing-size packing policy when moving live blocks to new extents. This "greedy" approach minimizes the space wasted when placing variable-size blocks into fixed-size extents: it places larger blocks into as few extents as possible and uses smaller ones to fill extents having little free space. Without this technique, all live blocks would be relocated in the order they were found during the scan phase, thus increasing free space fragmentation in their new extents.

On the other hand, spatial locality suffers, as previously neighboring live blocks may be relocated to different extents. To reduce the impact of this effect, we limit the number of extents the cleaner scans in each iteration to 2 MB. This value is small enough to reduce the negative impact on spatial locality, but large enough to feed the packing algorithm with a range of logical block sizes to be effective.

Placement decisions during cleanup are another important issue. The relative location of logical blocks within an extent is not as important, because extents are read in memory in full. Two issues need to be addressed: (a) which logical blocks should be placed in a specific extent during cleanup; and (b) whether a set of logical blocks that are being compacted will reuse an extent that is currently being compacted or a new extent from the free pool. *FlaZ* tries to maintain the original "proximity" of logical blocks, by combining logical blocks of neighboring extents to a single extent during compaction. As a result, each set of logical blocks is placed in the previously scanned extents rather than new ones, to avoid changing the location of compacted data on the SSD as much as possible.

The log-structured writes of *FlaZ*, together with the cleaner mechanism, practically remove "read-modify-write" sequences from the critical path of write I/O requests, deferring complex space management to a later time. An alternative approach would be to compress *multiple* blocks, e.g. 64 KB chunks, as a single unit and then store the result to a fixed number of blocks. This method would also avoid "read-modify-write" sequences and the impact of delayed space reclamation. However, it fails to support workloads with small-size I/O accesses and poor locality: each random read would require decompressing an *entire* unit of logical blocks in order to retrieve a single logical block.

Overall, we expect that the cleaner will not significantly interfere with application-issued I/O requests, as modern storage subsystems typically exhibit idle device times during a typical day of operation. However, we present indicative results quantifying the impact on performance when the cleaner is active concurrently with application I/O. The rest of our experiments are performed with the cleaner disabled.

2.2.5 I/O concurrency

Modern storage systems usually exhibit a high degree of I/O concurrency, having multiple outstanding I/O requests. Concurrency is very important for transparent compression, as it provides better opportunities for overlapping compression with I/O, effectively hiding the compression latency and associated CPU overhead. Overall, to allow for a high degree of asynchrony, *FlaZ* uses callback handlers to avoid blocking on synchronous kernel calls. *FlaZ* also allows multiple readers/multiple writers in the same extent in order to perform multiple concurrent I/O operations for this extent. Placing a compressed block in an extent is done in two steps. First, free space in the extent is pre-allocated. Second, the block is copied inside the extent. Multiple write contexts can copy blocks into the same extent simultaneously, since the

pre-allocation ensures proper space management in the extent.

However, higher I/O concurrency may have a negative effect on locality for *FlaZ*. In the write path, after the blocks of a request are compressed, they must be mapped to and stored in an extent. To preserve the locality of a single request, blocks belonging to the same request are placed in the *same* or *adjacent* extents, when possible. This requires an atomic mapping operation. All necessary synchronization for inserting compressed blocks into the extent happens while the extent is in the extent buffer in DRAM. Mapping only requires pre-allocating the required space in the extent for the blocks to be stored. Concurrent writes are serialized during the space allocation in extents, but proceed in parallel when processing logical blocks.

Besides highly concurrent I/O streams, *FlaZ* also leverages large I/Os. To hide the impact of compression on large I/Os, *FlaZ* uses multiple CPU cores when processing a single large I/O. Write requests typically come in batches due to the buffer-cache flushing mechanism. Large reads may exhibit low concurrency and decompression will significantly increase their response time. *FlaZ* uses two work queues per thread, one for reads and one for writes, with the read work queue having higher priority. Furthermore, we split large I/Os to units of individual blocks that are compressed or decompressed independently by different CPU cores. This scheduling policy decreases response time for reads and writes and reduces the delay writes may introduce to read processing. Empirically, we find that scheduling work through a global read and a global write work queue is efficient for all *FlaZ* threads and for all I/O sizes.

Finally, decompression is performed after the extent has been read in memory and after the I/O read to the SSD cache has completed. Decompression could also be performed earlier when the read callback for the extent is run in a bottom-half context, reducing the number of context switches. However, bottom-half execution is scheduled in the same CPU as the top-half context, hence restricting parallelism. We address this problem using separate threads for issuing I/Os and performing decompression.

3. Experimental Methodology

Our evaluation platform is an x86-based Linux system with dual-socket, quad-core Intel Xeon 64-bit CPUs at 2 GHz and 256-KB L1 caches, 12-MB L2 caches (2x6 MB shared between two pairs of cores), and 32 GB of RAM. The system is equipped with an Areca (ARC-1680D-IX-12) SAS storage controller, eight 500-GB Western Digital (WD-5001AALS) SATA-II disks and four enterprise-grade 32-GB Intel X25-E (SLC NAND Flash) SSDs. Caching on HDDs is set to write-through mode, as this is a typical setting for enterprise workloads. Table 1 summarizes peak performance numbers for each device.

	Read	Write	Response Time
HDD	100 MB/s	90 MB/s	12.6 ms
SSD	277 MB/s	202 MB/s	0.17 ms

Table 1. HDD and SSD performance metrics.

In our evaluation, we use three popular benchmarks: TPC-H, PostMark, and SPECsfs. For TPC-H we use two system configurations, one with 1 HDD and 1 SSD (1D-1S) and one with 8 HDDs and 2 SSDs (8D-2S). For PostMark we use two configurations, 1D-1S and 8D-4S, and for SPECsfs one configuration, 8D-2S. When multiple HDDs and SSDs are used, they are configured as RAID-0 devices, one for the HDDs and one for the SSDs. We always use a 4-1, 2-1, or 1-1 ratio of HDDs to SSDs as we believe that these ratios are representative of today's setups. All experiments are run on a 250-GB HDD partition. We have not used write-only workloads, as the benefit of our SSD cache implementation is only visible when the read I/O volume comprises a fair portion of the total traffic. We use CentOS release 5.3, with kernel version 2.6.18.8 (x86_64). The RAID configurations in our experiments use the default `md` Linux driver with a chunk size of 64 KB. All experiments run on top of the XFS file-system.

	Compression	Decompression	Space Savings
`lzo`	46μs	14μs	34%
`zlib`	150μs	60μs	54%

Table 2. Compression/decompression cost and space savings of a 4-KB block.

The algorithms and implementations used for compression and decompression of data are the default `zlib` and `lzo` libraries in the Linux kernel. Table 2 summarizes performance metrics of these algorithms.

3.1 TPC-H

TPC-H [43] is a data-warehousing benchmark that consists of a set of queries over a database. We use queries Q3, Q11, and Q14 that are response-time bound, and thus, a more difficult test for *FlaZ*. The queries are executed back to back, in that order. To limit execution time we use a 6.5 GB database (4 GB for data plus 2.5 GB for indices), which has a compression ratio of about 40% using `lzo`. TPC-H does a negligible amount of writes, mostly consisting of updates to file access timestamps and other control operations. We use three cache sizes for the SSD cache: large (7 GB), medium (3.5 GB), and small (1.75 GB) that hold approximately 100%, 50%, and 25% of the workload in the uncompressed cache, respectively. The database server used is MySQL 5.0.45.

3.2 PostMark

PostMark [21] is a synthetic, file-system benchmark that simulates an e-mail server by creating a pool of continually changing files over the file-system, and measures transaction rates and throughput. These transactions consist of (i) a create or delete file operation and (ii) a read or append file operation. Each transaction type and its affected files are chosen randomly. When all transactions complete, the remaining files are deleted. We employ a write dominated configuration with the default 35-65% read-write ratio. We use 150 mailboxes, 600 transactions, 10-100 MB mailbox sizes and 64 KB read/write accesses, resulting in a file-set of 16 GB. By default, PostMark uses random data that cannot be compressed for each written block. In our evaluation we modify PostMark to use real mailbox data as content for the mailbox files. We use two configurations of PostMark, one with a single instance and one with four concurrent instances. We use four cache sizes, being able to hold 100%, 50%, 25%, and 12.5% of the workload in the uncompressed cache, respectively.

3.3 SPECsfs

SPECsfs [3] simulates the operation of an NFSv3/CIFS file server; our experiments use the CIFS protocol. In SPECsfs, a *performance target* is set, expressed in CIFS operation per second; each operation requires 120 MB of disk space. SPECsfs reports the number of operations that can be handled per second, as well as average response time per operation. We select cache sizes to be representative of a scaled-down file-server workload, being able to hold 21%, 10%, 5%, and 2% of the workload in the uncompressed cache, respectively. To eliminate the impact on performance of the network we use a 10 GBit/s Ethernet connection between the CIFS server and the client generating the load. In our experiments we set the number of CIFS operations per second (OPS) to 1280 over a 150 GB file-set and extend execution time from 5 min warmup and 5 min measurement to 20 min warmup and 20 min measurement. As with PostMark, we modify SPECsfs to use real data for each block, rather than randomized content.

In order to be able to obtain a full set of reproducible results within a few days, we have opted to scale down the workload sizes by roughly one order of magnitude. Given this reduction, we limit DRAM from 32 GB to 1 GB, via the kernel boot option `mem=1g`. To study the impact of the effective cache size on performance of each workload, we vary the SSD cache capacity which is within the same down-scale factor. We focus our evaluation on I/O-intensive operating conditions, were the I/O system has to sustain high request rates. Under such conditions, we evaluate the benefit of the SSD caching layer and the trade-offs of using compression to increase the effective cache capacity.

4. Experimental Results

In this section we first examine the impact of compression on SSD cache performance and then we explore the impact of certain parameters on system behavior.

4.1 TPC-H

Figure 7 shows the performance impact of a compressed SSD cache compared to uncompressed caching and to HDD (no SSD caching). Overall, when the workload does not fit in the cache, compression improves performance, whereas performance degrades when the workload fits in the uncompressed cache. As shown in Figure 7(a), in the small and medium (25% and 50%) caches, compression improves execution time by 20% and 99% compared to an uncompressed SSD cache of the same size. Compression effectively increases the cache size resulting in significant performance improvement, despite the additional CPU utilization, shown in Figure 7(c). In the large cache, where 100% of the workload fits in the uncompressed cache, performance degrades by 40% when using compression. In this case, there is no benefit for additional cache hits, as illustrated in Figure 7(d). On the other hand, compression increases CPU utilization for all eight cores, by 29%, 65%, and 101% compared to the uncompressed cache, but it always remains below 25% of the maximum available CPU.

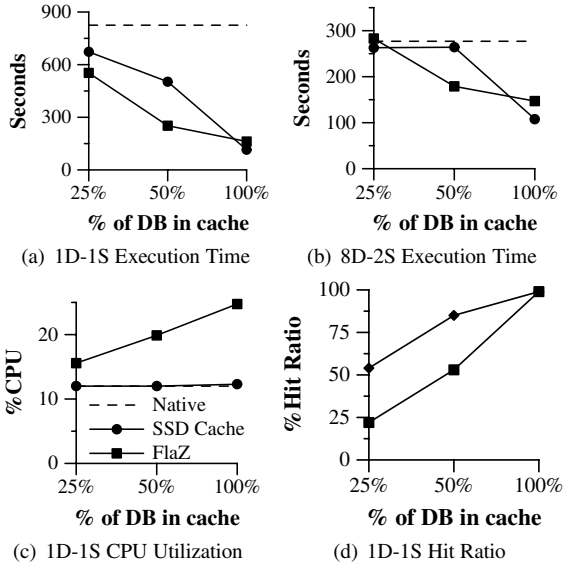
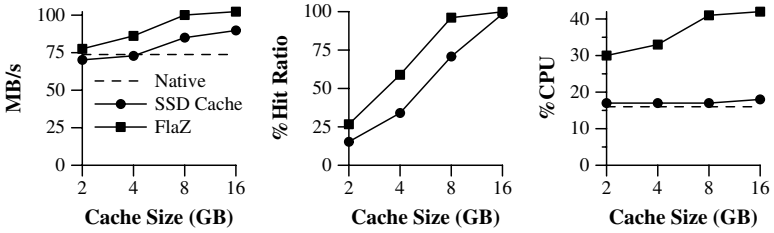


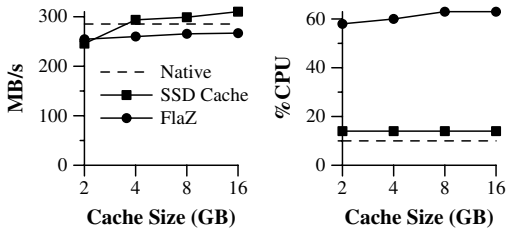
Figure 7. TPC-H results for the 1D-1S and 8D-2S configurations.

For the 8D-2S configuration, the hit ratio and CPU utilization (not shown) follow similar trends as in 1D-1S. Performance, shown in Figure 7(b), improves by 47% when using the medium cache, but degrades at the small and large caches by 7% and 36%, respectively. Given that the medium uncompressed cache is marginally faster than the base case and that the small compressed cache exhibits roughly the



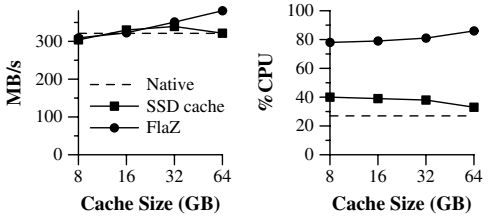
(a) Throughput (b) Cache Hit Ratio (c) CPU Utilization

Figure 3. PostMark results for the 1D-1S configuration.



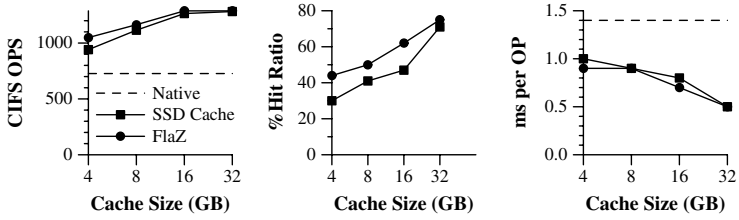
(a) Throughput (b) CPU Utilization

Figure 4. PostMark results for 8D-4S.



(a) Throughput (b) CPU Utilization

Figure 5. PostMark results for 8D-4S (4 instances).



(a) Throughput (b) %Hit Ratio (c) Latency

Figure 6. SPECsfs results for the 8D-2S configuration.

same hit ratio as the former, *FlaZ* suffers a performance slowdown in the small cache from higher response time due to decompression. For the large compressed cache, performance degrades for the same reason it does in the 1D-1S configuration.

4.2 PostMark

Figure 3 shows our results for PostMark in the 1D-1S configuration. Compression improves SSD caching throughput by up to 20% over the uncompressed SSD cache and by 44% compared to the native configuration, as shown in Figure 3(a). Performance improves because, apart from the increased hit ratio shown in Figure 3(b), PostMark is write-dominated and the log-structured writes of the compressed cache improve SSD write performance. Figure 3(c) shows that CPU utilization increases from 20% with uncompressed caching up to 40% with compression.

Figure 4 shows our results for the 8D-4S configuration. As with TPC-H, hit ratio (not shown) follows similar trends as in 1D-1S. Figure 4(a) shows that in this configuration the uncompressed SSD cache exhibits 16% less throughput for a small, 2-GB cache and improves throughput up to 8% for a 16-GB cache. Compressed caching suffers a 6%-15% performance penalty due to small number of outstanding I/Os in this configuration. Moreover, CPU overhead for decompression cannot be effectively overlapped with I/O. As read I/O response time is low with SSDs, decompression latency becomes comparable to I/O response time and affects PostMark performance. CPU utilization increases by up to 350%, as shown in Figure 4(b).

To examine what happens with PostMark at higher I/O rates we use four concurrent instances of PostMark. Figure 5(a) shows that *FlaZ* achieves better performance than uncompressed caching by 1%-18%, except for the 32-GB cache size where it is slower by 3%. At the 64-GB cache size, *FlaZ* almost exhausts available CPU (86% utilization) and compressed SSD caching eventually becomes CPU-bound, as shown in Figure 5(b).

4.3 SPECsfs

Figure 6 shows our results for SPECsfs in the 8D-2S configuration. The benefit of cache compression is best shown when using the smallest cache: *FlaZ* achieves 11% performance improvement due to a 46% better hit ratio, as illustrated in Figures 6(a) and 6(b), respectively. As the cache size increases, the hit ratio of the uncompressed cache approaches that of *FlaZ*, mitigating compression benefits.

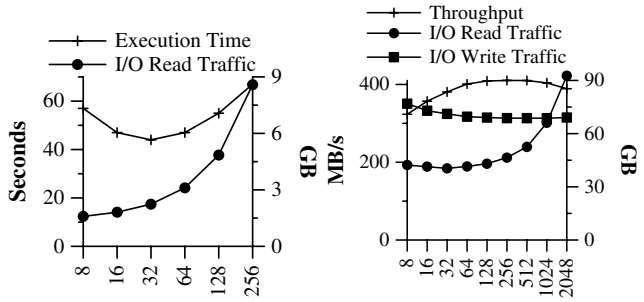
Figure 6(c) shows that response time per operation improves in *FlaZ* at the 4 and 16-GB cache size by 11% and 14% compared to uncompressed SSD caching, while it remains the same for the other two sizes. In all cases, CPU utilization (not shown) is increased by 25%, yet remaining below 10% of maximum available: the small random writes that SPECsfs exhibits makes the HDDs the performance bottleneck, underutilizing both the CPU and the SSDs.

Next, we explore the effect on system performance of extent size, cleaning overhead, compression efficiency, metadata cache size, and compression granularity.

4.4 Extent Size

In all the experiments presented so far we use an extent size of 32 KB. Extent size in *FlaZ* determines I/O read volume, fragmentation, and placement. In general, a small extent size is preferable for applications with little spatial

locality, whereas application with good spatial locality can benefit from larger extent sizes. Figure 8(a) shows TPC-H performance when we vary extent size. To isolate the impact of extent size in these experiments, the working set fits in the SSD cache (100% hit ratio). Also, we use only Q14, as Q3 and Q11 are less sensitive to the size of the extent. We see that best execution time is achieved in TPC-H with 32 KB extents. Although I/O traffic increases when using larger extents, performance increases by up to 29% compared to smaller extents. Further increasing extent size results in lower performance as I/O traffic increases disproportionally, up to 4x with 256 KB extents.



(a) TPC-H, 1D-1S (b) PM, 4 instances, 8D-2S

Figure 8. Impact of varying extent sizes for PostMark and TPC-H

The four-instance PostMark exhibits the highest throughput with extents between 128-512 KB, with 64 KB extents being only slightly (2%) worse than the best throughput, as illustrated in Figure 8(b). Larger extent sizes lead to less fragmentation, resulting in lower write I/O traffic. The smallest read I/O traffic is achieved when using 32-KB extents. Very small extents ($\leq$ 16 KB) lead to poor extent space management resulting in consecutive blocks "spreading" to neighboring extents, while large extents ($\geq$ 128 KB) lead to extents containing unrelated blocks. SPECsfs presents similar behavior with PostMark with respect to the extent size. Given these results, an extent size in the range of 32-64 KB seems to be appropriate as a base parameter.

4.5 Effect of cleanup on performance

Figure 9 illustrates the impact on performance when the cleaning mechanism is active during benchmark execution. In this configuration we use PostMark on a 2D-1S configuration and the workload marginally fitting in the compressed cache, thus activating the cleaner to reclaim free space. To better visualize the impact of cleaner in throughput, we use a lower threshold of 10% of free extents below which the cleaner is activated and an upper threshold of 20% of free extents above which the cleaner stops. In Figure 9, the impact of the cleaner on performance is seen as two "valleys" in the throughput graph during time periods from 200s to 280s and from 315s to 390s. The succession of "plateaus" and "valleys" indicates that the cleaner regularly starts and stops the cleaning process, as the amount of available extents is depleted and refilled. Performance degrades by up to 100% when the cleaner is running. Average performance, including all periods, drops to 123.5 MB/s, resulting in a 20% performance slowdown compared to the same configuration without the cleaner. The rate the cleaner reclaims space is practically limited by the SSD peak throughput, since the cleaner does very large sequential reads during scan and fewer sequential writes to write back live blocks. The CPU cost of the compaction algorithm along with the copying of live blocks is overlapped with the reading of extents.

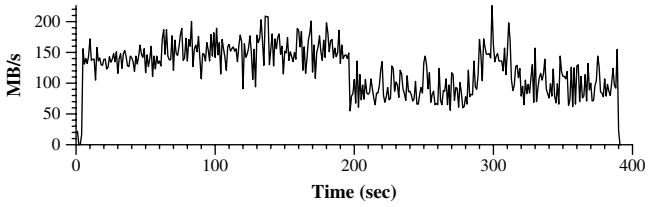


Figure 9. Impact of cleaner in PostMark throughput. "Valleys" depict time periods where the cleaner is active.

Typical storage systems exhibit idle I/O periods over large intervals of operation. In such systems, our cleaner would not impact performance as it would run during these idle periods. However, the cleaner would require further work in order to be suitable for systems that do not exhibit idle periods for very long intervals, i.e. a system that runs at peak I/O rates for days.

4.6 Compression Efficiency

An issue when employing block-level compression is the achieved compression efficiency when compared to larger compression units, such as files. Also, the layer at which compression is performed affects coverage of the compression scheme; block-level compression schemes will typically compress both data and file-system metadata, whereas file-level approaches compress only file data. Table 3 shows the compression ratio obtained for various types of data using three different levels: per archive where all files are placed in a single archive, per file, and per block.

In all cases, compressing data as a single archive will generally yield the best compression ratio. We see that in most cases, block-level compression with *FlaZ* is slightly inferior to file-level compression, when using `lzo` and slightly superior, when using `zlib`.

Next, we examine the impact of different compression libraries on performance and in particular CPU utilization that affects both hit time and hit ratio. Table 2 shows that `lzo` is about 3x and 5x faster than `zlib` for compression and decompression respectively and results in up to 37% worse compression ratio. We use PostMark on 1D-1S configuration, start with a cache size that marginally fits the workload when using `zlib`, and we use the same cache size for

Files	Orig. MB	gzip -r	gzip .tar	NTFS	ZFS	*FlaZ* (`zlib`)	*FlaZ* (`lzo`)
Mailbox 1 (single file folder)	125	N/A	71%	93%	96%	83%	89%
Mailbox 2 (single file folder)	63	N/A	22%	61%	69%	46%	66%
MS word collection	1100	50%	49%	63%	65%	56%	67%
MS excel collection	756	33%	33%	53%	59%	45%	53%
pdf collection	1400	78%	78%	86%	85%	85%	88%
Linux kernel source	277	45%	24%	73%	57%	31%	54%
Linux kernel compiled	1400	37%	29%	53%	48%	33%	42%

Table 3. Compression ratios for various level of compression and file types. Ratios shown are the size of the compressed data divided by the original data volume (lower is better). *gzip* is used with *-6* (default) in all cases.

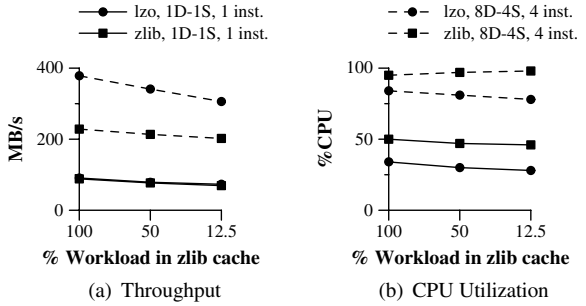


Figure 10. `lzo` vs. `zlib` performance in PostMark.

`lzo`. We decrease cache size to only fit 50% and 12.5% of the workload. Figure 10(a) shows that although `zlib` achieves 30-50% better hit ratio, its CPU cost, shown in Figure 10(b), is significantly higher (up to 50%), resulting in only marginal improvement in performance. Seen from another angle, `zlib` can achieve the same performance as `lzo` using a 30% smaller SSD cache. With four PostMark instances on 8D-4S, `zlib` achieves up to 40% lower throughput than `lzo`, as it becomes CPU limited.

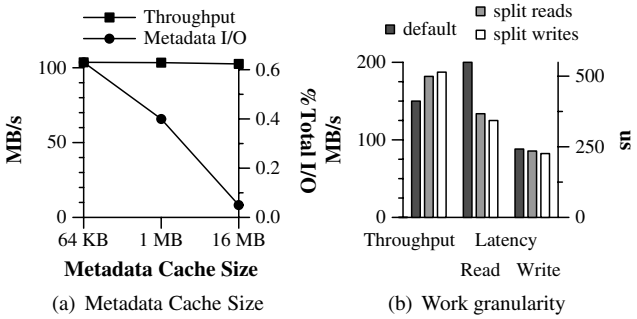


Figure 11. Impact of metadata cache size on 1D-1S PostMark and work granularity on 4D-2S PostMark.

4.7 Metadata Cache Size

The current capacity ratio of SSD-DRAM allows *FlaZ* to keep all required metadata in DRAM. However, given the increasing size of SSDs, it would be meaningful to examine more scalable solutions. For this reason, we implement a *metadata cache* for *FlaZ*; a general-purpose, fully-set-associative cache implemented as a hash table with a collision list and an LRU eviction policy. The size of the metadata cache in *FlaZ* affects the amount of additional I/O that will be incurred due to metadata accesses. Figure 11(a) illustrates the impact of cache size on performance using 1D-1S PostMark. Performance remains unchanged with the number of misses, as the amount of extra I/Os is less than 1% of application traffic. This shows that a small metadata cache of a few hundred KBs is adequate for *FlaZ*.

4.8 Compression Granularity

Figure 11(b) shows the impact of job granularity and the priority of read vs. write on PostMark. In particular, we see that PostMark is sensitive to read latency. Increasing concurrency for large read I/Os results in almost 50% improvement in latency and 20% in overall throughput. Introducing split writes, further improves latency by about 7%.

5. Discussion

There are five remaining considerations in the design of *FlaZ*: (a) the amount of DRAM required for metadata purposes; (b) how the cache remains persistent after a normal shutdown or system failure; (c) how the capacity of a compressed device whose size varies over time is presented to higher layers; (d) FTL and wear-leveling issues; and (e) power efficiency.

5.1 Metadata memory footprint

FlaZ requires 2.31 MB of cache metadata per GB of SSD: 880 KB for the caching layer and 1.25 MB for the compression layer. This metadata footprint scales with the size of the SSD cache size in the system, not with the capacity of the underlying HDDs. Therefore, DRAM space requirements for metadata are moderate. Additionally, a larger cache-line size further reduces metadata footprint. Finally, although DRAM caching can be used to reduce metadata footprint at the cost of additional I/Os (*FlaZ* does support caching for metadata),

this is not necessary for today's DRAM and SSD capacities in server systems.

5.2 SSD cache persistence

Traditional block-level systems do not update metadata in the common I/O path but only perform configuration updates that result in little additional synchronous I/O. Metadata consistency after a failure is not an issue in such systems. *FlaZ* makes extensive use of metadata to keep track of block placement. A key observation is that SSD cache metadata are not required to be consistent after a system failure. After an unclean shutdown, *FlaZ* assumes that a failure occurred and starts with a cold cache; all data have their latest copy in the hard disk. If the SSD cache has to survive failures, this would require an additional synchronous I/O for metadata to guarantee consistency, resulting in lower performance in the common path. In *FlaZ*, we choose to optimize the common path at the expense of starting with a cold cache after a failure.

5.3 Variable compressed device size

Space-saving techniques, such as compression result in devices whose effective size is data-dependent and varies over time. When a device that supports transparent, online compression is created, most today's operating systems and file-systems need to attach a specific size attribute to it before applications can access it. Although some operating systems may allow resizing a device dynamically, not many file-systems and applications support such functionality. When a device is created in *FlaZ*, its nominal size provided to higher layers is a configurable multiple of its actual size. This over-booking approach has the disadvantage that if compression is eventually less or more effective than estimated, the application will either see write errors or device space will remain unused.

An alternative approach is to be conservative at the beginning and declare the nominal device size to be its actual size. Then, when the logical device fills up, the remaining space in the physical device can be presented as a new device in the system, allowing applications to use the space that has been saved by *FlaZ*. This approach has the disadvantage that further use of a device may result in different compression ratios and thus, the need to allocate more free space to the device. This leads to the need for a mechanism that supports multiple fixed-size logical block devices on top of a pool of free storage, similar to thin provisioning techniques in use today. We believe that such a mechanism will be an essential part of future block-level storage systems; However, this is beyond the scope of our work and we do not consider it any further.

5.4 FTL and wear-leveling

In this work we focus on how a compression layer could be used to extend SSD capacity in a manner that is orthogonal to SSD characteristics. Given that SSD controllers currently do not expose any block state information, we rely on the flash translation layer (FTL) implementation within the SSD for wear-leveling. Furthermore, we cannot directly influence the FTL's block allocation and re-mapping policies. Currently, our implementation avoids issuing small random writes as much as possible. Designing block-level drivers and file-systems in a manner cooperative to SSD FTLs which improves wear-leveling and reduces FTL overhead is an important direction, especially while raw access to SSDs is not provided by vendors to system software.

5.5 Power efficiency

Trading of CPU cycles for increased SSD capacity has power implications as well. On the one hand, by consuming more CPU cycles for compression and decompression, we increase power consumption. On the other hand, a higher SSD cache hit rate improves I/O power consumption. This creates an additional parameter that should be taken into account when trading CPU cycles for I/O performance. However, we believe that it is important to examine this tradeoff alongside offloading compression, e.g. to specialized hardware, and we leave this for future work.

6. Related Work

6.1 SSD Caching

Current SSD technology exhibits interesting performance and cost characteristics. For this reason, there has been a lot of work on how to improve I/O performance using SSDs either by replacing HDDs or inserting SSDs between HDDs and DRAM. For instance, in [22] the authors propose the use of flash memory as a secondary file cache for web servers. They show that the main disadvantages of flash, long write cycle and wear leveling, can be overcome with both energy and performance benefits. The authors in [24] study whether SSDs can benefit transaction processing performance. They find that despite the poor performance of flash memory for small-to-moderate sized random writes, transaction processing performance can improve up to one order of magnitude by migrating to SSDs the transaction log, rollback segments, and temporary table spaces. The authors in [31] examine whether SSDs can fully replace traditional disks in data-center environments. They find that SSDs are not a cost-effective technology for this purpose, yet. Thus, given current tradeoffs, mixed SSD and HDD environments are more attractive. Along similar lines, currently, there exist a number of storage products that use flash-based memory for performance purposes: FusionIO [2], HotZone [33], MaxIQ [4], and ZFS's L2ARC [26], all using SSDs as disk caches. Our work builds on these observations and further improves the effectiveness of SSD caching by employing transparent, online compression in the I/O path.

ReadyBoost [30] uses flash-based devices as a disk cache to improve I/O performance. In addition, ReadyBoost compresses and encrypts the data cache. It requires a Windows-

specific file-system on top of the cache device. Hence, it can take advantage of the file-systems ability, e.g. to write variable size segments as well as to manage metadata. This, for instance, implies that applications requiring raw access to the storage medium, such as databases, cannot take advantage of ReadyBoost. In contrast, *FlaZ* employs all required metadata and techniques for achieving transparent compression and can be layered below any file-system or other application.

6.2 Use of Compression to Improve I/O Performance

The argument that trends in processor and interconnect speeds will increasingly favor the use of compression for optimizing throughput in various distributed and networked systems, even if compression is performed in software, has been discussed in [17]. We believe that the same trend holds today with increasingly powerful multicore CPUs that offer both processing cycles and require large amounts of effective data throughput.

Improving I/O performance with compression has been examined in the past, mainly in the context of database systems. The authors in [35] examine how a compression algorithm specifically designed for relational data can further boost Oracle's performance. They show that compression decreases full table scan execution time by 50% while increasing total CPU utilization only slightly (5%). They state that the actual overhead of decompressing blocks is small and that the performance gain of compression can be large.

In [14] the authors encompass compression to IBM's Information Management Systems (IMS) by using a method based on modified Huffman codes. They find that this method achieves 42.1% saving of space in student-record databases and less on employee-record databases, where custom compression routines were more effective. Their approach reduces I/O traffic necessary for loading the database by 32.7% and increases CPU utilization by 17.2%, showing that online compression can be beneficial not only for space savings, but for performance reasons as well.

The authors in [32] discuss compression techniques for large statistical databases. They find that these techniques can reduce the size of real census databases by up to 76.2% and improve query I/O time by up to 41.3%.

Similar to these techniques, our work shows that online compression can be beneficial for improving I/O performance. However, these approaches target specifically database environments or specific structures within databases. In contrast, our work shows how I/O compression can be achieved transparently to all applications. In addition, we quantify the benefits when employing compression on top of SSD-based caches. To achieve this, *FlaZ* employs extensive metadata at the block-layer and multiple techniques to reduce all induced metadata-and I/O-related overheads.

Also, previous research [7, 18, 37, 45] has argued that online compression of memory (fixed-size) pages can improve memory performance. More recently, Linux Compcache [20] is a system that transparently compresses memory pages to increase the effective capacity of system memory and to reduce swapping requirements. All these systems maintain compressed pages only in memory and always store to disk uncompressed data. Furthermore, these approaches require only in memory metadata and the memory subsystem in the OS offers the required structures for managing additional metadata. Unlike disk blocks, the address of each virtual page in memory and on disk is explicitly recorded in a page table. Our work requires *introducing* metadata to the block layer and employing a number of techniques to mitigate associated performance issues.

6.3 File-system Compression

Compression has been employed by file-systems to improve effective storage capacity: Overall, [13] categorizes and discusses various file-level compression approaches, while a survey of data compression algorithms can be found in [25, 41].

The authors in [11] describe how LFS can be extended with compression. They use a 16-KB compression unit and a mechanism similar to our extents in order to store compressed data on disks. They find that compression in their storage system has cost benefits and that in certain cases there is a 1.6 performance degradation for file-system intensive operations. They use the Sprite file-system for managing variable size metadata.

In [12], the authors gather data from various systems and they show that compression can double the amount of data stored in a system. To mitigate the performance impact of compression, they propose a tiered, two-level file-system architecture; Tier 1 caches *uncompressed*, frequently accessed files and Tier 2 stores the infrequently accessed files in a compressed form.

More recently, both Sun's ZFS [10] and Microsoft's NTFS [29] support compression over any storage device. `e2compr` [8] is an extension of the *ext2* file-system, where a user explicitly selects the files and directories to be compressed. LogFS [19] is specifically designed for flash devices and supports compression. FuseCompress [1] offers a mountable Linux file-system that transparently compresses its contents. Compression is also supported in read-only file-systems, such as SquashFS [27] and CramFS [42].

Our work differs in two ways from these efforts: First, we explore how compression can improve I/O performance which requires dealing with the cost of compression itself. Second, we perform compression below the file-system, which requires introducing several techniques for metadata management. The benefit of our approach is both improved I/O performance and transparency to all applications, regardless of whether they require a specific file-system or they run directly on top of block storage.

6.4 Other Capacity and I/O Optimization Techniques

Deduplication [9, 28, 47] is an alternative, space-saving approach that has recently attracted significant attention. Deduplication tries to identify and eliminate *identical*, variable-size segments in files. Compression is orthogonal to deduplication and is typically applied at some stage of the deduplication process. The authors in [47] show how they are able to achieve over 210 MB/s for four write data streams and over 140 MB/s for four read data streams on a storage server with two dual-core CPUs at 3 GHz, 8 GB of main memory, and a 15-drive disk subsystem (software RAID6 with one spare drive). Deduplication has so far been used in archival storage systems due to its high overhead.

6.5 Understanding and Improving SSDs

Recent efforts have explored SSD tradeoffs [5] and performance characteristics [16] as well as improving various aspects of SSDs, such as block management [36], random write performance [23], and file-system behavior [6, 19, 46]. Our work is orthogonal to these efforts, as *FlaZ* can use any type of flash-based device as a compressed cache.

6.6 Block-level Compression

Finally, the only systems that have considered block-level compression are cloop [39] and CBD [40]. However, these systems offer read-only access to a compressed block device and offer limited functionality. Building a read-only block device image requires compressing the input blocks, storing them in compressed form and finally, storing the translation table on the disk. *FlaZ* uses a similar translation table to support reads. However, this mechanism alone cannot support writes after the block device image is created, as the compressed footprint of a block re-write is generally different from the one already stored. *FlaZ* is a fully functional block device and supports compressed writes. To achieve this, *FlaZ* uses an out-of-place update scheme that requires additional metadata and deals with the associated challenges.

7. Conclusions

In this work we examine how transparent compression can increase the effectiveness of SSD caching in the I/O path. We present the design and implementation of *FlaZ*, a system that uses SSDs as a cache and compresses data as they flow between main memory and the cache. Compression is performed online, in the common path, and increases the effective cache size. Our work addresses a number of challenges related to dealing with variable size (compressed) blocks, logical to physical block mapping and allocation, compression and I/O overheads. We evaluate our approach using realistic workloads, TPC-H, PostMark, and SPECsfs, on a system with eight SATA-II disks and four SSDs.

We find that even modest increases in the amount of available SSD capacity improve I/O performance, for a variety of server workloads. Transparent block-level compression allows such improvements by increasing the effective SSD capacity, at the cost of increased CPU utilization. Overall, although compression at the block-level introduces significant complexity, our work shows that online data compression is a promising technique for improving the performance of I/O subsystems.

Acknowledgments

We would like to thank the anonymous reviewers for their comments that have helped us improve our work. We thankfully acknowledge the support of the European Commission under the 6th and 7th Framework Programs through the STREAM (FP7-ICT-216181), HiPEAC (NoE-004408), and HiPEAC2 (FP7-ICT-217068) projects.

References

[1] FuseCompress, a Linux file-system that transparently compresses its contents. http://miio.net/wordpress/projects/fusecompress.

[2] Fusion-io's Solid State Storage – A New Standard for Enterprise-Class Reliability. http://www.dpie.com/manuals/storage/fusionio/Whitepaper_Solidstatestorage2.pdf.

[3] SPECsfs2008: SPEC's benchmark designed to evaluate the speed and request-handling capabilities of file servers utilizing the NFSv3 and CIFS protocols. http://www.spec.org/sfs2008/.

[4] ADAPTEC, INC. Adaptec MaxIQ SSD Cache Performance Kit. www.adaptec.com/en-US/products/CloudComputing/-MaxIQ/SSD-Cache-Performance/index.htm.

[5] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference* (2008), pp. 57–70.

[6] ALEPH ONE LTD, EMBEDDED DEBIAN. Yaffs: A NAND-Flash Filesystem. www.yaffs.net, 2002.

[7] APPEL, A. W., AND LI, K. Virtual memory primitives for user programs. *SIGPLAN Not. 26*, 4 (1991), 96–107.

[8] AYERS, L. E2compr: Transparent File Compression for Linux . http://e2compr.sourceforge.net/, June 1997.

[9] BOBBARJUNG, D. R., JAGANNATHAN, S., AND DUBNICKI, C. Improving duplicate elimination in storage systems. *Trans. Storage 2*, 4 (2006), 424–448.

[10] BONWICK, J., AND MOORE, B. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/.

[11] BURROWS, M., JERIAN, C., LAMPSON, B., AND MANN, T. On-line data compression in a log-structured file system. In *Proc. of ASPLOS-V* (1992), ACM, pp. 2–9.

[12] CATE, V., AND GROSS, T. Combining the concepts of compression and caching for a two-level filesystem. In *Proc. of ASPLOS-IV* (1991), ACM, pp. 200–211.

[13] COFFING, C., AND BROWN, J. H. A survey of modern file compression techniques, Oct. 1997.

[14] CORMACK, G. V. Data compression on a database system. *Commun. ACM 28*, 12 (1985), 1336–1342.

[15] DEUTSCH, L. P., AND GAILLY, J. ZLIB Compressed Data Format Specification version 3.3. Internet RFC 1950, May 1996.

[16] DIRIK, C., AND JACOB, B. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proc. of ISCA '09* (2009), ACM, pp. 279–289.

[17] DOUGLIS, F. On the role of compression in distributed systems. In *Proc. of ACM SIGOPS, EW 5* (1992), pp. 1–6.

[18] DOUGLIS, F. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *Proc. of 1993 Winter USENIX Conference* (1993), pp. 519–529.

[19] ENGEL, J., AND MERTENS, R. LogFS - finally a scalable flash file system. http://logfs.org/logfs/.

[20] GUPTA, N. Compcache: Compressed in-memory swap device for Linux. http://code.google.com/p/compcache.

[21] KATCHER, J. PostMark: A New File System Benchmark. *NetAPP TR3022* (1997). http://www.netapp.com/tech_library/3022.html.

[22] KGIL, T., AND TREVOR, M. FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers. In *Proc. of CASES '06* (2006), ACM, pp. 103–112.

[23] KIM, H., AND AHN, S. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proc. of FAST'08* (2008), USENIX Association, pp. 1–14.

[24] LEE, S., MOON, B., PARK, C., KIM, J., AND KIM, S. A Case for Flash Memory SSD in Enterprise Database Applications. In *Proc. of SIGMOD '08* (2008), ACM, pp. 1075–1086.

[25] LELEWER, D. A., AND HIRSCHBERG, D. S. Data compression. *ACM Comput. Surv. 19*, 3 (1987), 261–296.

[26] LEVENTHAL, A. Flash storage memory. *Commun. ACM 51*, 7 (2008), 47–51.

[27] LOUGHER, P., AND LOUGHER, R. SquashFS: a compressed, read-only file-system for Linux. squashfs.sourceforge.net.

[28] MANBER, U. Finding similar files in a large file system. In *WTEC'94: Proc. of the USENIX Winter 1994 Technical Conference* (1994), USENIX Association, pp. 2–2.

[29] MICROSOFT CORP. Best practices for NTFS compression in Windows. support.microsoft.com/default.aspx?scid=kb;en-us;Q251186, September 2009.

[30] MICROSOFT CORPORATION. Explore the features: Windows ReadyBoost. www.microsoft.com/windows/windows-vista/features/readyboost.aspx.

[31] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating server storage to SSDs: analysis of tradeoffs. In *Proc. of EuroSys '09* (2009), ACM, pp. 145–158.

[32] NG, W. K., AND RAVISHANKAR, C. V. Block-Oriented Compression Techniques for Large Statistical Databases. *IEEE Trans. on Knowl. and Data Eng. 9*, 2 (1997), 314–328.

[33] NORTH AMERICAN SYSTEMS INTERNATIONAL, INC. FalconStor HotZone - Maximize the performance of your SAN. http://www.nasi.com/hotZone.php.

[34] OBERHUMER, M. F. X. J. LZO – a real-time data compression library. http://www.oberhumer.com/opensource/lzo/.

[35] POESS, M., AND POTAPOV, D. Data Compression in Oracle. In *Proc. 29th VLDB Conference* (2003).

[36] RAJIMWALE, A., PRABHAKARAN, V., AND DAVIS, J. D. Block management in solid-state devices. *Proceedings of the USENIX Annual Technical Conference (USENIX'09)* (June 2009).

[37] RIZZO, L. A very fast algorithm for RAM compression. *SIGOPS Oper. Syst. Rev. 31*, 2 (1997), 36–45.

[38] ROSENBLUM, M., AND OUSTERHOUT, J. K. The Design and Implementation of a Log-Structured File System. *ACM TOCS 10*, 1 (Feb. 1992), 26–52.

[39] RUSSEL, P. Cloop: the compressed loopback block device. http://www.knoppix.net/wiki/Cloop.

[40] SAVAGE, S. CBD Compressed Block Device, New embedded block device. http://lwn.net/Articles/168725, January 2006.

[41] SMITH, M. E. G., AND STORER, J. A. Parallel algorithms for data compression. *Journal of ACM 32*, 2 (1985), 344–373.

[42] TORVALDS, L., AND QUINLAN, D. CramFS Tools. http://sourceforge.net/projects/cramfs.

[43] TRANSACTION PROCESSING PERFORMANCE COUNCIL. TPC-H: an ad-hoc, decision support benchmark. http://www.tpc.org/tpch/.

[44] WELCH, T. A. A technique for high-performance data compression. *IEEE Computer Society Press 17*, 6 (1984), 8–19.

[45] WILSON, P. R., KAPLAN, S. F., AND SMARAGDAKIS, Y. The case for compressed caching in virtual memory systems. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference* (1999), USENIX Association, pp. 8–8.

[46] WOODHOUSE, D. JFFS: The Journalling Flash File System. http://sourceware.org/jffs2/jffs2-html, 2001.

[47] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. of FAST'08* (1998), USENIX Association, pp. 1–14.

[48] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23* (1977), 337–343.