Transparent Online Storage Compression at the Block-Level

YANNIS KLONATOS, Foundation for Research and Technology – Hellas and University of Crete
THANOS MAKATOS, MANOLIS MARAZAKIS, and MICHAIL D. FLOURIS,
Foundation for Research and Technology – Hellas
ANGELOS BILAS, Foundation for Research and Technology – Hellas and University of Crete

In this work, we examine how transparent block-level compression in the I/O path can improve both the space efficiency and performance of *online* storage. We present *ZBD*, a block-layer driver that transparently compresses and decompresses data as they flow between the file-system and storage devices. Our system provides support for variable-size blocks, metadata caching, and persistence, as well as block allocation and cleanup. *ZBD* targets maintaining high performance, by mitigating compression and decompression overheads that can have a significant impact on performance by leveraging modern multicore CPUs through explicit work scheduling. We present two case-studies for compression. First, we examine how our approach can be used to increase the capacity of SSD-based caches, thus increasing their cost-effectiveness. Then, we examine how *ZBD* can improve the efficiency of online disk-based storage systems.

We evaluate our approach in the Linux kernel on a commodity server with multicore CPUs, using PostMark, SPECsfs2008, TPC-C, and TPC-H. Preliminary results show that transparent online block-level compression is a viable option for improving effective storage capacity, it can improve I/O performance up to 80% by reducing I/O traffic and seek distance, and has a negative impact on performance, up to 34%, only when single-thread I/O latency is critical. In particular, for SSD-based caching, our results indicate that, in line with current technology trends, compressed caching trades off CPU utilization for performance and enhances SSD efficiency as a storage cache up to 99%.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: *Design studies*; D.4.2 [**Operating Systems**]: Storage Management—*Storage hierarchies*; D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms: Design, Performance, Experimentation, Measurement

Additional Key Words and Phrases: Block-level compression, SSD-based I/O cache

ACM Reference Format:
Klonatos, Y., Makatos, T., Marazakis, M., Flouris, M. D., and Bilas, A. 2012. Transparent online storage compression at the block-level. ACM Trans. Storage 8, 2, Article 5 (May 2012), 33 pages.
DOI = 10.1145/2180905.2180906 http://doi.acm.org/10.1145/2180905.2180906

Parts of this work were presented at the 5th European Conference on Computer Systems (Eurosys 2010) [Makatos et al. 2010a] and at the 6th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI 2010) [Makatos et al. 2010b].
This work was partially supported by the European Commission under the 7th Framework Programs through the STREAM (FP7-ICT-216181), HiPEAC (NoE-004408), HiPEAC2 (FP7-ICT-217068), and IOLanes (FP7-STREP-248615) projects.
Authors' address: Y. Klonatos, T. Makatos, M. Marazakis, M. D. Flouris, and A. Bilas, Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), 100 N. Plastira Avenue, Vassilika Vouton, Heraklion, GR-70013, Greece; email: yannis.klonatos@epfl.ch.
Y. Klonatos and A. Bilas are also associated with Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR-71409, Greece.

DOI 10.1145/2180905.2180906 http://doi.acm.org/10.1145/2180905.2180906

1. INTRODUCTION

Although the primary storage cost per GB has been steadily declining, the demand for additional capacity has been growing faster. For this reason, various techniques for improving the effective capacity of disk storage have gained significant attention [Zhu et al. 2008].

One such technique is employing data compression to improve the efficiency of storage space and data transfers [Lelewer and Hirschberg 1987]. Previously, compression has been mostly applied at the file level [Cate and Gross 1991]. This is because it allows using larger objects which compress better, as well as access to knowledge of data semantics and statistics maintained by the file-system itself. However, this approach 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. Then, this increases management complexity, given that today's storage systems may choose to employ from a variety of different file-systems. Finally, file-systems are not usually aware of the properties of block storage devices, for example, in a RAID controller. This can be important since it prohibits offloading specific compression operations to the controller itself. These concerns can be addressed by moving compression functionality towards lower system layers, such as the block device layer.

In this work, we examine the potential of transparent, block-level compression for *online* storage systems. We design *ZBD*, a system that transparently compresses and decompresses data as they flow in the system between main memory and hard disk drives (HDDs). While traditional compression systems, such as NTFS, mainly target improving only the space efficiency of the underlying storage, we additionally aim to significantly improve its performance whenever possible. We do so by thoroughly and systematically analyzing the performance and tradeoffs associated with I/O volume, CPU utilization, and metadata I/Os in the Linux kernel. Our design, and as we also show in our evaluation, significantly reduces disk seeks and effectively allows hiding the compression overhead when there is a sufficient amount of outstanding I/Os.

Block-level compression appears to be deceptively simple. Conceptually, it merely requires intercepting requests in the I/O path, compressing data before writes, and decompressing them after reads. However, our experience shows that designing an efficient system for online storage is far from trivial and requires addressing a number of challenges.

— *Variable Block Size.* Block-level compression needs to operate on fixed-size input and output blocks. However, compression itself 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 must be stored in the same physical block. This requires using 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 placement.
— *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 of I/Os or to I/O latency. Moreover, reducing metadata footprint and achieving metadata persistence can result in significant number of additional I/Os in the common path.

— *Device Aging*. Aging of compressed block devices 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. Although in this work we do not examine alternative compression algorithms and possible optimizations, we quantify their performance impact on the I/O path of modern storage systems. 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 from the host to storage controllers.

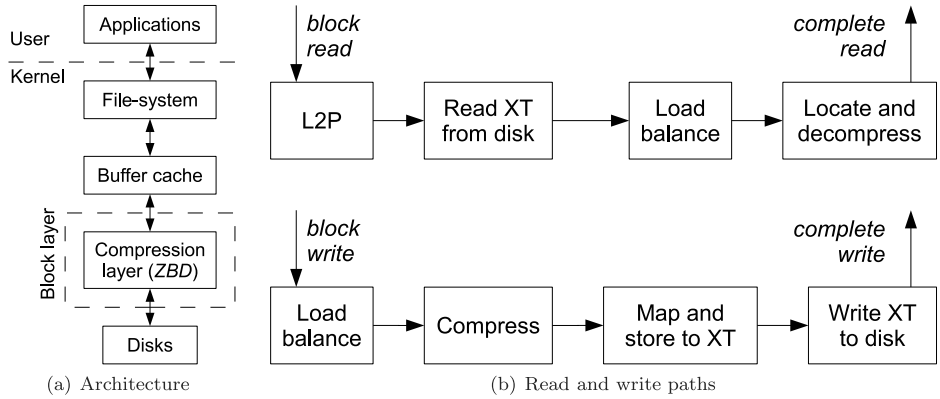
Then, we explore how transparent online compression can improve flash-based solid-state disk (SSDs) based I/O caches. The emergence of these devices has the potential to mitigate I/O penalties, by offering low read response times and peak throughput significantly superior to HDDs. Additionally, SSDs are not affected by seeks. However, and although SSD capacity continues to increase, there is no indication that their current high cost per GB will soon approach that of HDDs [Narayanan et al. 2009]. Thus, it is important to examine techniques that can increase SSD cost-efficiency. One such technique is to utilize these devices, instead of primary storage, as a *cache* in the I/O path, between the main memory and the HDDs, where the cost of the SSDs is expected to be amortized over increased I/O performance, both in terms of throughput (MB/sec) and access rate (IOPS).

For this purpose, we develop a system, called *FlaZ*, which caches data blocks on dedicated SSD partitions and transparently compresses and decompresses data as they flow in the I/O path between main memory and SSDs, with the address space of the latter not being visible to applications. We examine how transparent compression can increase the capacity of an SSD cache, thus improving I/O performance. *FlaZ* internally consists of two layers, one that achieves transparent compression (*ZBD* itself) 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.

For our evaluation we use four realistic workloads: PostMark, SPECsfs2008, TPC-C, and TPC-H. For *FlaZ*, compression increases performance when used to improve effective SSD cache capacity, up to 99%, 25%, and 11% for TPC-H, PostMark, and SPECsfs2008, respectively. This improvement comes at increased CPU utilization, by up to 450%. 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.

Furthermore, for disk-based compression, our results show that *ZBD* degrades performance by up to 15% and 34% for TPC-H and TPC-C respectively, but improves by 80% and 35% for PostMark and SPECsfs2008, respectively. These results indicate that compression can increase I/O performance for workloads that exhibit enough I/O concurrency. This comes at the cost of increased CPU utilization, up to 311% on an 8-core system. We believe that trading CPU with storage capacity for online storage is in-line with current technology trends. However, compression in the I/O path has a negative impact on performance (up to 34%) for latency sensitive workloads that use only small I/Os.

Finally, we provide an experimental comparison to native file system compression, by comparing *ZBD* and NTFS. Our results show that block-level compression seems to be a more viable option than NTFS, when both space-efficiency and performance is concerned. For instance, *ZBD* achieves 55% better performance in PostMark than



(a) Architecture (b) Read and write paths

Fig. 1. *ZBD* system architecture and I/O paths.

NTSF. In addition, we compare the space efficiency of *ZBD* with that achieved by other widely used compression methods, like ZFS or gzip.

The rest of this article is organized as follows. Section 2 discusses the design of *ZBD* and *FlaZ* and how they address the associated challenges. Section 3 presents our evaluation methodology. Section 4 presents our experimental results. Section 5 discusses some further considerations, while Section 6 discusses previous and related work. Finally, we draw our conclusions in Section 7.

2. SYSTEM DESIGN

In our systems application-generated file-level I/O operations pass through the *ZBD* and *FlaZ* virtual block-device layers, where compression/decompression and caching takes place correspondingly, for write and read accesses. The block-level interface allows us to be file-system agnostic. In handling writes, *ZBD* compresses the data supplied by the request originator, and fit the resulting variable-size segments into fixed-size blocks, which are then mapped onto block storage devices. In handling reads, *ZBD* locates the corresponding fixed-size blocks, and by decompression obtains the data to be returned to the request originator. Figure 1 illustrates the read and write paths for ZBD. In this section, we describe these procedures in detail, while in the next section we discuss the issues that arise from using *ZBD* in SSD-based I/O caches.

2.1. Compression in the I/O Path

In general, compression methods operate as follows. They first initialize a workspace, for example, 256KB 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.

ZBD uses Lempel-Ziff-Welch (LZW) compression [Deutsch and Gailly 1996; Welch 1984; Ziv and Lempel 1977] 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* [Deutsch and Gailly 1996] and *lzo* (variant `LZO1X-1`) [Oberhumer 2008].

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.

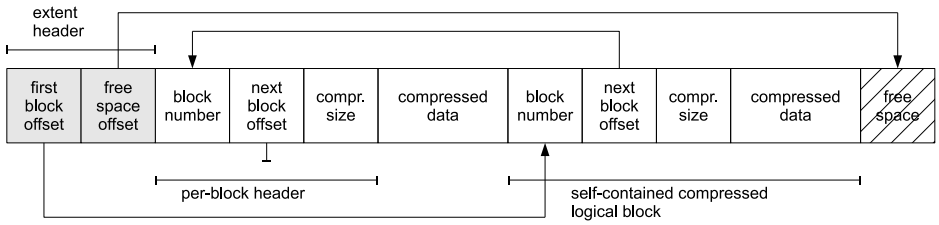


Fig. 2. *ZBD* extent structure. Each compressed block is self-contained, described by a per-block header.

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. In both operations, our system equally utilizes all available CPU cores, by load balancing the requests to the cores as they reach *ZBD*.

2.1.1. 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* (XTs), which are multiples of block size. *ZBD* 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. A portion of this table is cached in main memory by a metadata cache, as will be explained in the next section.

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. The size of the extent header is 8 bytes, containing two 4-byte offset pointers. The per-block header is 14 bytes, 8 bytes for the block number, 2 bytes for size and 4 bytes for the next block offset inside the extent. 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 *ZBD* into a log-structured block device. From our experience, less than 2% of blocks fail to compress in the workloads we examine.

2.1.2. 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, *ZBD* 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 [Rosenblum and Ousterhout 1992]. 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. *ZBD* uses a *cleaner* process which is triggered when the amount of free extents falls below a certain threshold, as described in a later section.

2.1.3. Extent Buffering. To mitigate the impact of additional I/Os on performance due to the "read-modify-write" scheme, the compression layer of *ZBD* uses a buffer in DRAM for extents. Buffering a small number of extents not only facilitates I/O to the HDDs, 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, that is there is not enough space for a write context to append blocks and (b) there is no reference to them, that is no active read context uses blocks from this extent.

2.1.4. Extent Cleaning. ZBD 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. *ZBD* removes this cleanup process from the common path to reduce interference with applications.

The *ZBD* cleaner runs when the free extent pool drops below a threshold, and has similar goals to the segment cleaner of Sprite LFS [Rosenblum and Ousterhout 1992].

In our design, we scan the HDD for extents that have no free space ("full" extents) using the logical to physical translation mappings and extent headers. This is a sequential read operation. Then, we determine which blocks within an extent are "live" by verifying that the mapping of each block in the logical to physical translation table points to the extent under examination. Otherwise, the block is "old" and can be reclaimed by the cleaner during compaction. The cleaner 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. The only metadata required until the cleaner's next activation is the last scanned extent. This pointer does not need to be persistent; it is merely a hint for the cleaner. Finally, it is worth mentioning that there will be always completely full extents available for the cleaner to examine, since an extent is filled up before moving to the next one during compression of logical blocks.

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 per iteration to 2MB. 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. *ZBD* 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 out-of-place updates of *ZBD*, 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, for example, 64KB 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, disproportionally increasing CPU overheads.

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 and low device utilization over large periods of time, for example, a day. CPU-wise, the average server utilization of data centers is low, often below 30% [Meisner et al. 2009].

However, we believe that data centers are reluctant to go to standby mode in order to save power through idle periods, since this increases response time. In this work, 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 deactivated.

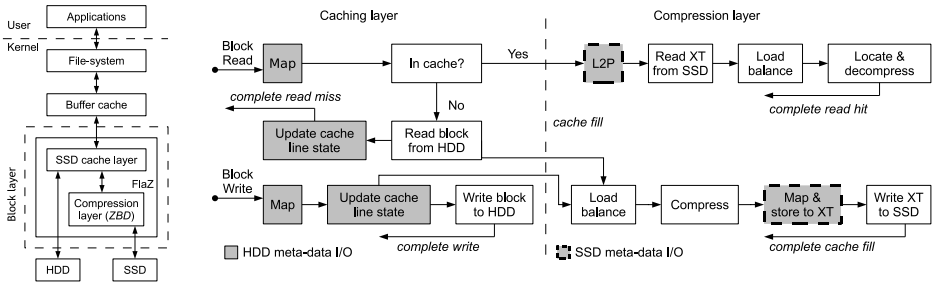
2.1.5. I/O Concurrency. Modern storage systems usually exhibit a high degree of I/O concurrency, having multiple outstanding I/O requests. Concurrency is very important for transparent compression, as it provides better opportunities for overlapping compression with I/O, effectively hiding the compression latency and associated CPU overhead. Overall, to allow for a high degree of asynchrony, *ZBD* uses callback handlers to avoid blocking on synchronous kernel calls. *ZBD* 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 *ZBD*. 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, *ZBD* also leverages large I/Os. To hide the impact of compression on large I/Os, *ZBD* 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. *ZBD* 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 *ZBD* 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 HDD 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.

2.1.6. Metadata and Data Consistency. By our design, we have managed to keep metadata memory footprint quite low. This means that a very small NVRAM suffices to keep all modified metadata, without having to introduce two synchronous I/Os (one for the logical to physical translation table and one the free extents pool) for each application write. We also keep extents used by writes through the extent buffer in NVRAM in order to avoid a full extent flush for each small write. However, our design does not necessitate the use of NVRAM. A similar failure recovery mechanism to the one of Sprite LFS would be enough to ensure consistency after failure, albeit at a performance cost.



(a) Architecture (b) Read and write paths

Fig. 3. *FlaZ* system architecture and I/O paths.

2.2 *FlaZ*: Compressed SSD Caching with *ZBD*

In this section, we explain how we use *ZBD* in a larger system, called *FlaZ*, in order to implement a compressed SSD cache in the I/O path. *FlaZ* internally consists of two layers, one for transparent compression (*ZBD* itself) and one that uses dedicated SSD partitions as I/O caches, as shown in Figure 3(a).

2.2.1. SSD Caching in the I/O Path. 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, for example, 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.

FlaZ exports 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, through *ZBD*, and are always stored on hard disk in uncompressed form. 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. The address space of SSDs is not visible to applications. The caching layer of *FlaZ* is a direct-mapped, write-through cache with one block per cache line.

Figure 3(b) shows how the caching layer of *FlaZ* handles the intercepted I/O requests. Initially, each logical block is 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 *ZBD* 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 small metadata block 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 2.5MB of metadata per GB of SSD. However, given the increasing size of SSDs, we do present a quantitative evaluation of metadata scalability.

There are four 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) FTL and wear-leveling issues; and (d) power efficiency. We discuss these considerations next.

2.2.2. Metadata Memory Footprint. *FlaZ* requires 2.31MB of cache metadata per GB of SSD: 880KB for the caching layer and 1.25MB 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.

2.2.3. 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.

2.2.4. 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.

2.2.5. 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

Table I. HDD and SSD Performance Metrics

	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 II. Compression/Decompression Cost of a 4-KB Block

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

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, for example, to specialized hardware, and we leave this for future work.

3. EXPERIMENTAL PLATFORM

We present our evaluation results using a commodity server built using the following components: eight 500-GB Western Digital WD800JD-00MSA1 SATA-II disks connected on an Areca ARC-1680D-IX-12 SAS/SATA storage controller, a Tyan S5397 motherboard with two quad-core Intel Xeon 5400 processors running at 2GHz, and 32GB of DDR-II DRAM. The OS installed on this host is CentOS 5.3 (kernel version 2.6.18-128.1.6.el5, 64-bit). Caching on HDDs is set to write-through mode, as this is a typical setting for enterprise workloads. When multiple devices are used, they are configured as RAID-0 devices (one for the HDDs and one for the SSDs), using the `MD` software-RAID with the chunk-size set to 64KB. Table I summarizes peak performance numbers for each device. 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 size when necessary, via the kernel boot option `mem=1g`.

For *ZBD*, the algorithms and implementations used for compression and decompression of data are the default `zlib` [Ziv and Lempel 1977] and `lzo` [Welch 1984] libraries in the Linux kernel without any modifications, except for the preallocation of workspace buffers. `zlib` supports nine compression levels, with the lowest favoring speed over compression efficiency and the highest vice-versa. We set the compression level to one, since for 4KB blocks higher compression levels disproportionally increase the compression overhead with a minimal improvement in space-consumption (30% additional compression cost for only 2% additional space savings). The implementation of `lzo` we use does not support compression levels. Table II compares the performance characteristics of the `zlib` and `lzo` implementations. The extent size used is 32KB in all of our experiments but we evaluate the impact of extent size separately in Section 4.2.7. We use four popular benchmarks, running over an XFS file-system with block-size set to 4KB: PostMark, SPECsfs2008, TPC-C, and TPC-H. For TPC-C and TPC-H, we use MySQL (v.5.1) with the default configuration and the MyISAM storage engine.

In the evaluation of *FlaZ*, we use two system configurations for TPC-H, 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 SPECsfs2008 one configuration, 8D-2S. 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. 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. To study the impact of the effective cache size on performance of each workload, we vary the SSD cache capacity that is within

the same down-scale factor. We focus our evaluation on I/O-intensive operating conditions, where the I/O system has to sustain high request rates. Under such conditions, we evaluate the benefit of compression on disk-based systems and on SSD caches as a means of increasing storage space efficiency and effective cache capacity, respectively.

3.1. PostMark

PostMark [Katcher 1997] is a file-system benchmark that simulates a mail server that uses the `maildir` file organization. It creates a pool of continually changing files and measures transaction rates and I/O throughput. In our evaluation we have modified PostMark to use real mailbox data as the contents of the mailbox files. In general, mail servers benefit little from data caching in DRAM, since it is common for the size of the mail-store to exceed that of the server's DRAM by at least one order of magnitude [SPEC 2009], and, moreover, the I/O workload is write-dominated. For these reasons, we use 1GB of DRAM for PostMark.

For disk compression, we present results from executing 50,000 transactions for a 35:65% read-write ratio, with 16KB read/write operations, over 100 mailboxes where each mailbox is a directory containing 500 messages, and the message size ranging from 4KB to 1MB. For SSD cache compression, we use 150 mailboxes, 600 transactions, 10-100MB mailbox sizes and 64KB read/write accesses, resulting in a file-set of 16GB. 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.

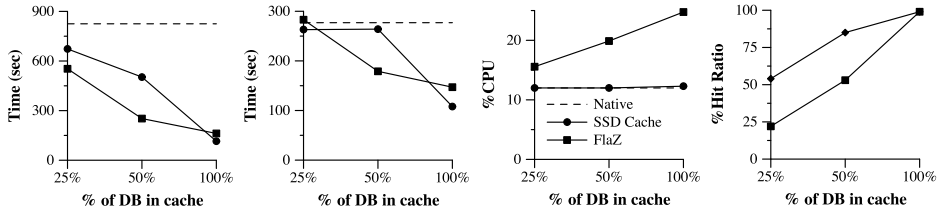
3.2. SPECsfs2008

SPECsfs2008 [SPEC 2008a] simulates the operation of an NFSv3/CIFS file-server; our experiments use the CIFS protocol. In SPECsfs2008, a *performance target* is set, expressed in operations-per-second. Operations, both read/writes of data-blocks and metadata-related accesses to the file-system, are executed over a file-set generated at benchmark-initialization time. The size of this file-set is proportional to the performance target ($\approx$120MB per operation/sec). SPECsfs2008 reports the number of operations-per-second actually achieved, and the average response time per operation. For the SPECsfs2008 results, the DRAM size is set to 2GB, under the assumption that this is close to the common file-set size to DRAM-size ratios in audited SPECsfs2008 results [SPEC 2008b]. As with PostMark, we modify SPECsfs2008 to use compressible contents for each block. To eliminate the impact on performance of the network we use a 10GBit/s Ethernet connection between the CIFS server and the client generating the load.

For disk compression, we set the performance target at 3,400 CIFS ops/sec, a load that the eight disks can sustain, and then increase the load up to 4,600 CIFS ops/sec. For SSD cache compression, 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. In these experiments, we set the number of CIFS ops/sec to 1280 over a 150GB file-set and extend execution time from 5-minute warmup and 5-minute measurement to 20-minute warmup and 20-minute measurement.

3.3. TPC-C (DBT-2)

DBT-2 [Thomas and Wong 2007] is an OLTP transactional performance test, simulating a wholesale parts supplier where several workers access a database, update customer information, and check on parts inventories. DBT-2 is a fair usage implementation of the Transaction Processing Council TPC-C benchmark specification [TPC 1997]. We use a workload of 300 warehouses, which corresponds to a 28GB database,



(a) 1D-1S Execution Time (b) 8D-2S Execution Time (c) 1D-1S %CPU (d) 1D-1S Hit Ratio

Fig. 4. TPC-H results for the 1D-1S and 8D-2S configurations.

with 3,000 connections, 10 terminals per warehouse and benchmark execution time limited to 30 min. The database is compressed by 34% and 46%, when using `lzo` and `zlib`, respectively. For TPC-C, we limit system memory to 1GB. This amount of DRAM is large enough to avoid swapping, but small enough to create more pressure on the I/O system.

3.4. TPC-H

TPC-H [TPC 2009b] is data-warehouse benchmark that issues data-analysis queries to a database of sales data. For our evaluation, we have generated a scale-4 TPC-H database (4GB of data, plus 2.5GB of indices). TPC-H does a negligible amount of writes, mostly consisting of updates to file-access timestamps. For this workload, we have set the DRAM size to 1GB, under the assumption that this is close to the common database-size to DRAM-size ratios in audited TPC-H results [TPC 2009a]. The compression ratio for this dataset is 39% using `lzo` and 48% using `zlib`.

For disk compression, we use queries Q1, Q3, Q4, Q6, Q7, Q10, Q12, Q14, Q15, Q19, and Q22 that keep execution time to reasonable levels. For SSD cache compression, we use three cache sizes: large (7GB), medium (3.5GB), and small (1.75GB) that hold approximately 100%, 50%, and 25% of the workload in the uncompressed cache, respectively, and use queries Q3, Q11, and Q14.

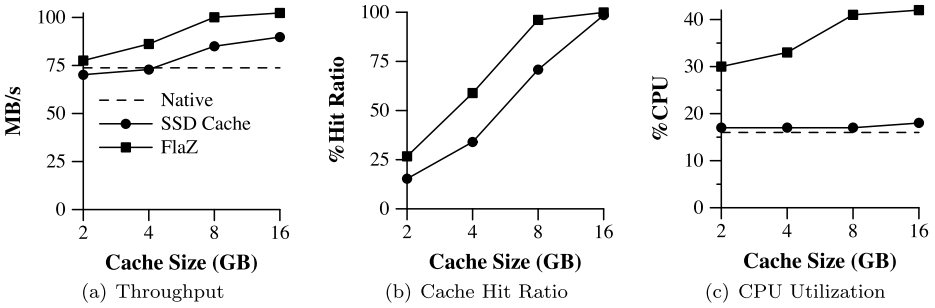
4. EXPERIMENTAL RESULTS

4.1. Compressed Caching (*FlaZ*)

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.

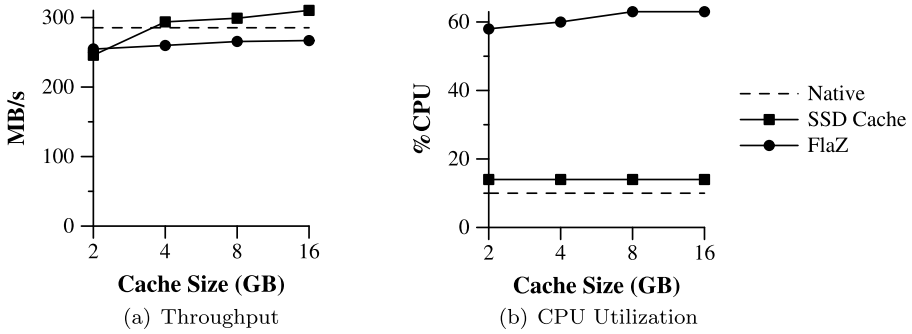
4.1.1. TPC-H. Figure 4 shows the performance impact of a compressed SSD cache compared to uncompressed caching and to HDD (no SSD caching). Overall, when the workload does not fit in the cache, compression improves performance, whereas performance degrades when the workload fits in the uncompressed cache. As shown in Figure 4(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 4(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 4(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.

For the 8D-2S configuration, the hit ratio and CPU utilization (not shown) follow similar trends as in 1D-1S. Performance, shown in Figure 4(b), improves by 47% when using the medium cache, but degrades at the small and large caches by 7% and 36%,



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

Fig. 5. PostMark on 1D-1S configuration.



(a) Throughput (b) CPU Utilization

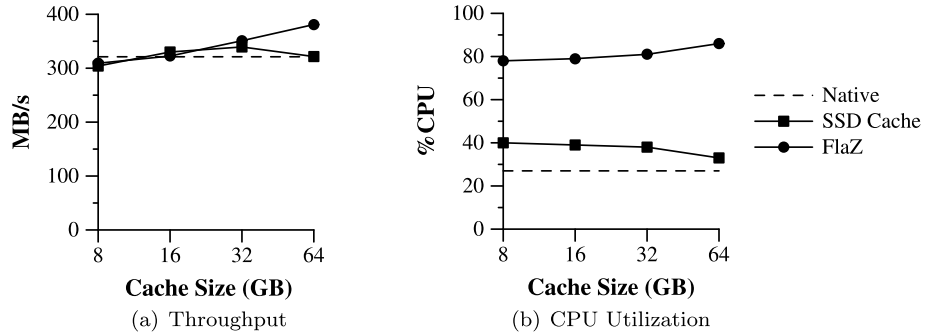
Fig. 6. PostMark on 8D-4S configuration.

respectively. Given that the medium uncompressed cache is marginally faster than the base case and that the small compressed cache exhibits roughly the 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.

4.1.2. PostMark. Figure 5 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 5(a). Performance improves because, apart from the increased hit ratio shown in Figure 5(b), PostMark is write-dominated and the log-structured writes of the compressed cache improve SSD write performance. Figure 5(c) shows that CPU utilization increases from 20% with uncompressed caching up to 40% with compression.

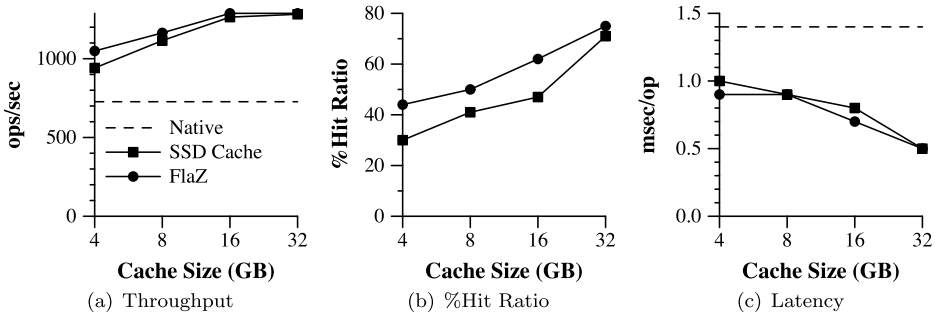
Figure 6 shows our results for the 8D-4S configuration. As with TPC-H, hit ratio (not shown) follows similar trends as in 1D-1S. Figure 6(a) shows that in this configuration the uncompressed SSD cache exhibits 16% less throughput for a small, 2-GB cache and improves throughput up to 8% for a 16-GB cache. 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 6(b).

To examine what happens with PostMark at higher I/O rates we use four concurrent instances of PostMark. Figure 7(a) shows that *FlaZ* achieves better performance than uncompressed caching by 1%–18%, except for the 32-GB cache size where it is slower



(a) Throughput (b) CPU Utilization

Fig. 7. PostMark using four instances.



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

Fig. 8. SPECsfs2008 throughput on 8D-2S configuration.

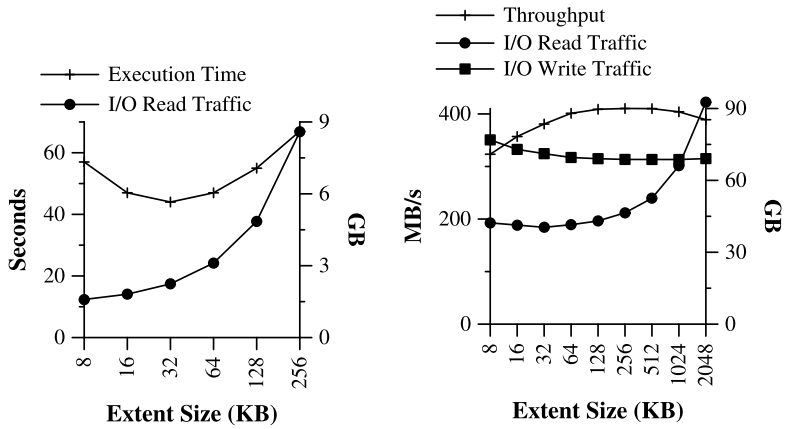
by 3%. At the 64GB cache size, *FlaZ* almost exhausts available CPU (86% utilization) and compressed SSD caching eventually becomes CPU-bound, as shown in Figure 7(b).

4.1.3. SPECsfs2008. Figure 8 shows our results for SPECsfs2008 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 8(a) and 8(b), respectively. As the cache size increases, the hit ratio of the uncompressed cache approaches that of *FlaZ*, mitigating compression benefits.

Figure 8(c) shows that response time per operation improves in *FlaZ* at the 4 and 16GB 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 SPECsfs2008 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.

4.1.4. Extent Size. In all the experiments of compressed caching presented so far we use an extent size of 32KB. Similar to *ZBD*, 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 9(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



(a) TPC-H, 1D-1S

(b) PostMark on 8D-2S with four concurrent instances

Fig. 9. TPC-H and PostMark with varying extent sizes.

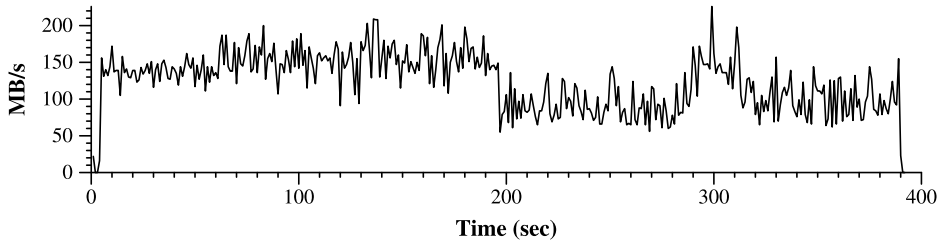
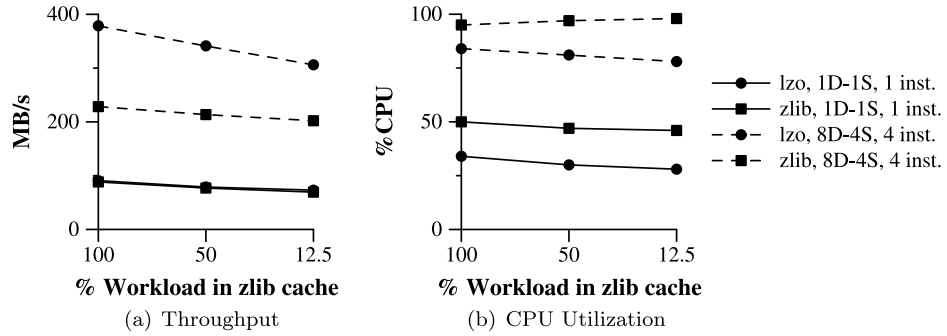


Fig. 10. Impact of cleaner in PostMark throughput. "Valleys" depict time periods where the cleaner is active.

less sensitive to the size of the extent. We see that best execution time is achieved in TPC-H with 32KB 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 256KB extents.

The four-instance PostMark exhibits the highest throughput with extents between 128-512KB, with 64KB extents being only slightly (2%) worse than the best throughput, as illustrated in Figure 9(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 32KB extents. Very small extents ($\leq$ 16KB) lead to poor extent space management resulting in consecutive blocks "spreading" to neighboring extents, while large extents ($\geq$ 128KB) lead to extents containing unrelated blocks. SPECsfs2008 presents similar behavior with PostMark with respect to the extent size. Given these results, an extent size in the range of 32-64KB seems to be appropriate as a base parameter.

4.1.5. Effect of Cleanup on Performance. Figure 10 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. Similar to *ZBD*, 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. Performance degrades by up to 100% when the cleaner is running. Average performance, including



(a) Throughput (b) CPU Utilization

Fig. 11. `lzo` vs. `zlib` performance in PostMark.

all periods, drops to 123.5MB/s, resulting in a 20% performance slowdown compared to the same configuration without the cleaner.

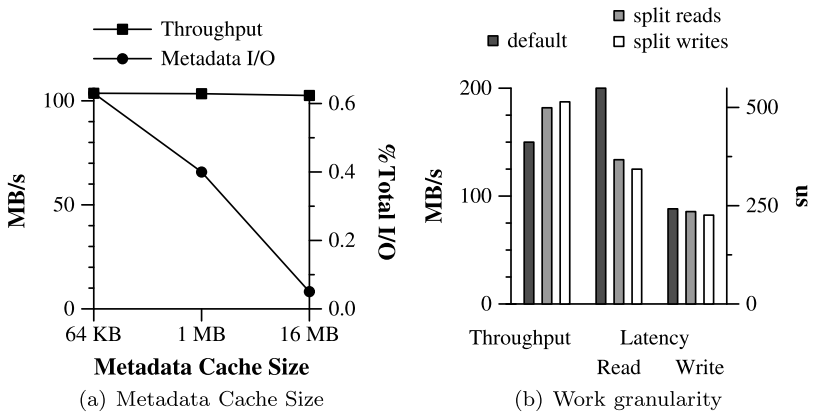
4.1.6. Compression Efficiency. Next, we examine the impact of different compression libraries on performance and in particular CPU utilization that affects both hit time and hit ratio. Table II shows that `lzo` is about 3x and 5x faster than `zlib` for compression and decompression respectively and results 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 `lzo`. We decrease cache size to only fit 50% and 12.5% of the workload. Figure 11(a) shows that although `zlib` achieves 30–50% better hit ratio, its CPU cost, shown in Figure 11(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.

4.1.7. 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 12(a) illustrates the impact of cache size on performance using 1D-1S PostMark. In contrast to *ZBD*, *FlaZ*'s performance remains unchanged with the number of misses, as a few percent of additional random I/Os pose no burden on the SSDs.

4.1.8. Compression Granularity. Figure 12(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%. *ZBD* is much less sensitive to compression granularity, where application response time is dominated by I/O latency, as opposed to SSDs, where it becomes *comparable* to compression/decompression latency.

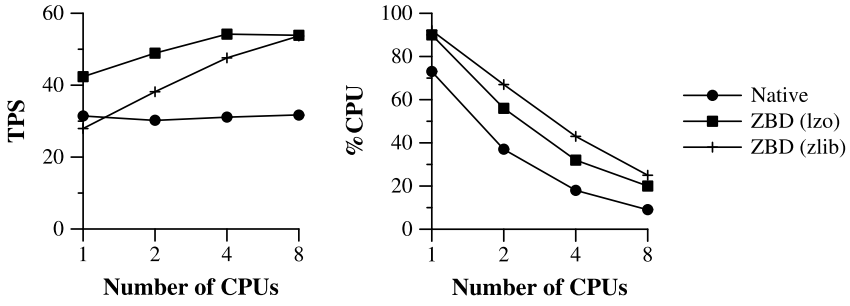
4.2. Compression on Magnetic Disks

Next, we first examine the impact of compression on performance of disk-based storage using *ZBD* and then we explore the impact of certain parameters on system behavior.



(a) Metadata Cache Size (b) Work granularity

Fig. 12. Impact of metadata cache size on 1D-1S PostMark and work granularity on 4D-2S PostMark.



(a) Trans. per sec (b) CPU Utilization

Fig. 13. Results for PostMark with variable number of CPUs.

4.2.1 PostMark. Figure 13 shows results for PostMark during the transaction execution phase, with 1, 2, 4, and 8 CPUs, and two compression libraries, `lzo` and `zlib`. Native performance is unaffected by the number of CPUs, as PostMark's needs in CPU cycles are minor. *ZBD* achieves higher performance than native by up to 69%, mainly due to the log-structured writes, as indicated in Figure 13(a). As the number of CPUs decreases, *ZBD* performance drops by up to 12%, especially when using `zlib`, as it is much more demanding in CPU cycles.

When more compressible contents used as data generated by Postmark, shown in Figure 14, performance increases by 2% over the less compressible data, as the former requires less CPU and its higher compression ratio results to lower I/O volume.

4.2.2. SPECsfs2008. Figure 15 shows our results for SPECsfs2008. Native sustains the initial load of 3,400 CIFS ops/sec, but fails to do so for higher loads. Data compression further improves performance; *ZBD* (`lzo`) sustains the load of 4,000 CIFS ops/sec. By using `zlib`, the higher compression ratio can sustain the highest load point, but fails to do so for loads beyond 4,300 CIFS ops/sec. Compression also improves latency, by up to 150% for `zlib`. The impact of higher compression ratio is also reflected by the reduced latency for `zlib` compared to `lzo`, up to 23%. Finally, compression increases CPU utilization by 80% and 90% for `lzo` and `zlib`, respectively.

Figure 16 shows our results for SPECsfs2008 when using more compressible data for file contents. Higher compression ratio results in lower I/O volume, hence higher

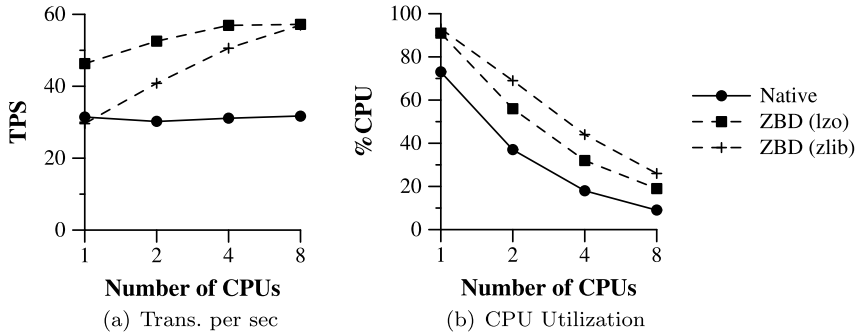


Fig. 14. Results for PostMark using more compressible data.

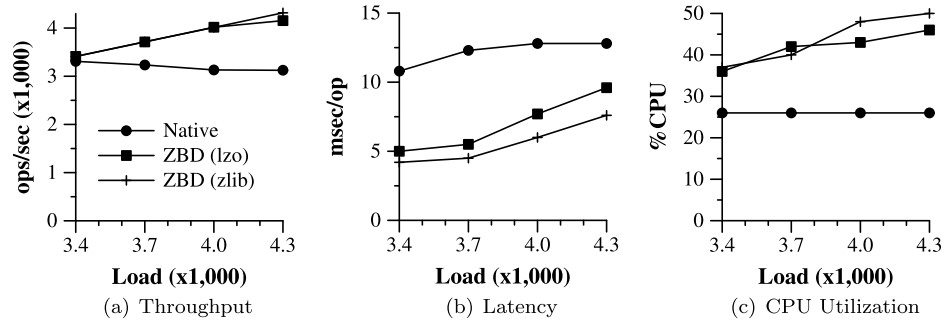


Fig. 15. Results for SPECsfs2008.

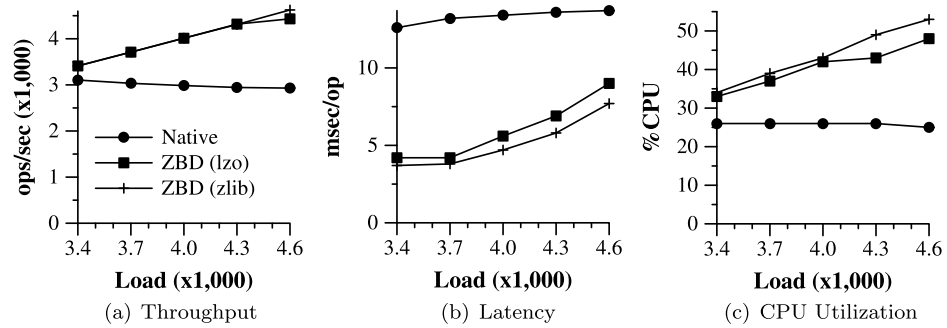
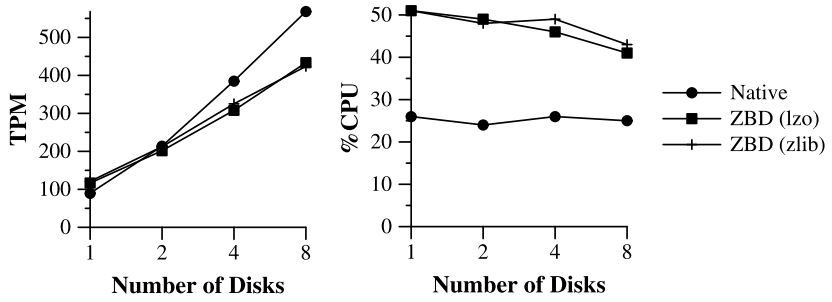


Fig. 16. Results for SPECsfs2008 using more compressible data.

throughput. *ZBD* (`lzo`) now sustains the load of 4,300 CIFS ops/sec but not the one of 4,600 ops/sec. *ZBD* (`zlib`) sustains the highest load point due to higher compression ratio.

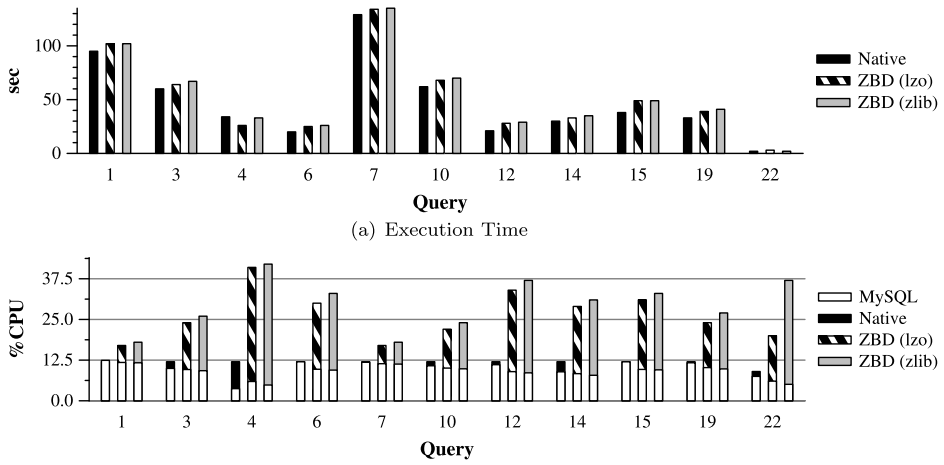
SPECsfs2008 has an abundance of outstanding I/Os, hence overall performance is not affected by compression, as it is overlapped with I/O. Further, the additional overhead introduced in the I/O path by compression does not hurt latency, since the more efficient I/O compensates for the increased CPU overhead with significant performance benefits.

4.2.3. TPC-C (DBT2). Figure 17 shows our results for TPC-C. Performance degrades for *ZBD* by 31% for `lzo` and by 34% for `zlib`, whereas CPU utilization increases by



(a) Trans. per min (b) CPU Utilization

Fig. 17. Results for TPC-C with variable number of CPUs.



(a) Execution Time

(b) CPU utilization. 100% is the maximum CPU utilization for all eight cores.

Fig. 18. Results for TPC-H.

64% and 72%. TPC-C exhibits reads that are small (usually 4KB) and random, leading to disproportionally high I/O read volume. Each read request practically translates to reading a full extent (32KB in this configuration). In addition, decompression can not be parallelized effectively for small reads, leading to increased read response. As the number of disks decreases, I/O latency significantly increases making the decompression overhead less important. When using only one disk, performance improves by 29% and 34% for `lzo` and `zlib`, respectively.

4.2.4. TPC-H. Figure 18 shows per-query results for a subset of the TPC-H queries, executed back-to-back. Most queries suffer performance penalty when using *ZBD* by up to 33% and 38% for `lzo` and `zlib`, respectively. Overall, performance decreases by 11% and 15% and CPU utilization increases by up to 242% and 311%. TPC-H has very few outstanding I/Os, and decompression cannot be effectively overlapped with I/O, thus degrading performance.

In Figure 19, we execute Q3, varying the number of CPUs. Native is unaffected by the reduction in CPU power, as the query can consume at most one CPU. *ZBD* (`lzo`) suffers performance penalty when running at only one CPU by 13%, whereas *ZBD* (`zlib`) intensively contends with MySQL for CPU cycles, resulting in 67% performance degradation.

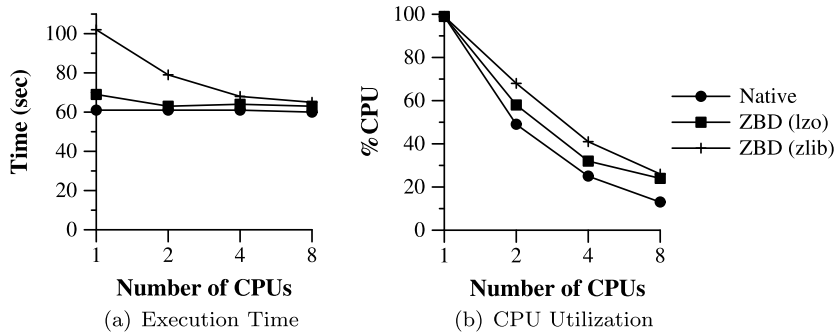


Fig. 19. Results for TPC-H (Q3) with variable number of CPUs.

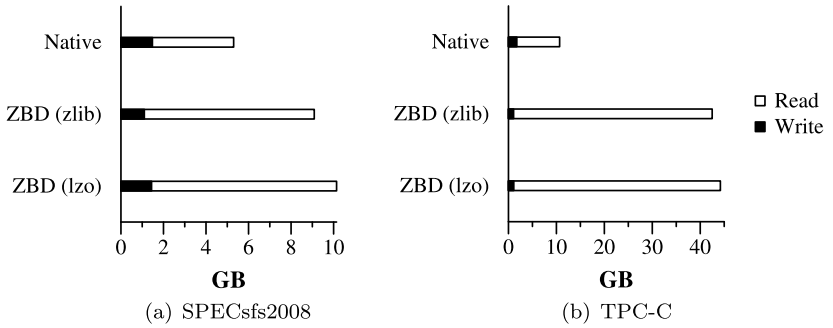


Fig. 20. I/O volume for SPECsfs2008 at 3,400 CIFS ops/sec load, and for TPC-C using eight disks.

Next, we explore the effect on system performance of log-structured writes, data locality, extent size, cleaning overhead and metadata I/Os.

4.2.5. Effect of Log-Structured Writes on Performance. The log-structured writes of *ZBD* increase performance for workloads that consist of a fair amount of writes and are not sensitive to latency. Figure 20(a) shows a breakdown of the I/O volume for SPECsfs2008 at the 3,400 CIFS ops/sec load point. Total I/O volume increases by 91% and 71% for *ZBD* (`lzo`) and *ZBD* (`zlib`), respectively. This increase is the result of reading *entire* extents for each 4KB read issued by the SPECsfs2008 load generators. With *ZBD* (`lzo`, `zlib`) write I/O volume is reduced by 3% and 26% due to compression. Although total I/O volume increases for all *ZBD* configurations, performance increases, as shown in Figure 15(b). In contrast to Native's writes, *ZBD*'s writes are *log-structured*, resulting in a more sequential workload. The large increment of read volume affects performance negatively but it is offset by the improved write pattern. This additional read volume translates to increased I/O transfer time, but no additional seeks are introduced. On the other hand, log-structured writes *reduce* the number of seeks compared to Native.

In contrast to SPECsfs2008, TPC-C has a much higher read-write ratio. The poor read locality TPC-C exhibits results in more than 4x additional read I/O volume, shown in Figure 20(b). This unnecessary I/O activity, in conjunction with TPC-C being sensitive to latency, results in a 26% performance degradation compared to Native, as shown in Figure 17(a). *ZBD* improves TPC-C writes. However, this improvement alone

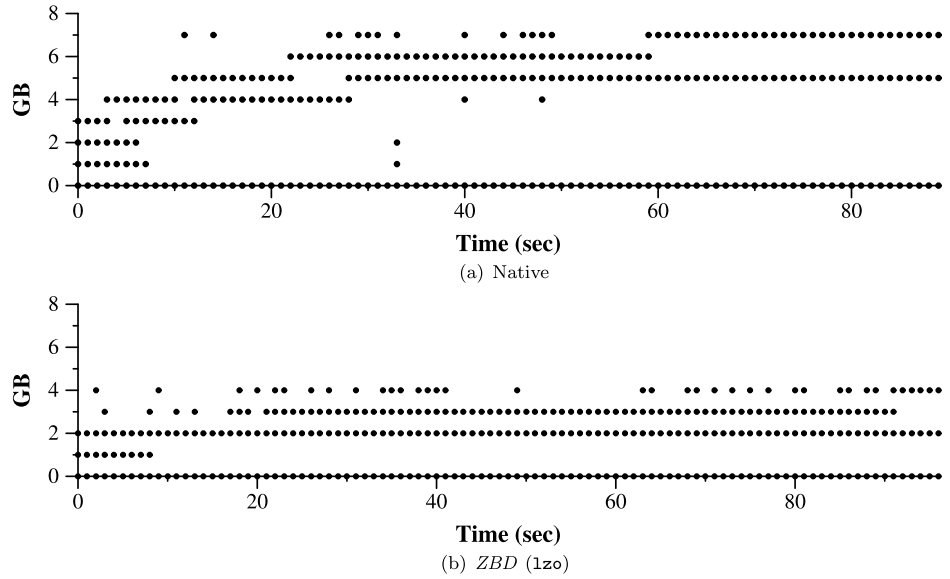


Fig. 21. Disk access pattern for TPC-H (Q3).

cannot offset the negative impact of the additional read volume, as writes only make up 14% of TPC-C's I/O volume.

4.2.6. Effect of Compression on Data Locality. When using transparent compression, there are some less obvious factors that affect performance. As data are kept compressed, disk transfer time is reduced by a factor governed by the compression ratio achieved. Furthermore, the average seek distance is reduced by roughly the same ratio, as data are "compacted" to a smaller area on the disk platter. Figure 21 illustrates the access pattern for TPC-H (Q3); *ZBD* generates disk accesses that are within a 4-GB zone on the disk, whereas native's accesses are within a 6.5GB zone. This means that the average seek distance for *ZBD* is smaller than native's. Finally, compacting data to a smaller area keeps it to the outer zone of the disk platter, making ZCAV effects more vivid. Despite these considerations, performance is still lower than native, as decompression cost dominates response time.

4.2.7. Extent Size. Figure 22 illustrates the impact of the size of the extent on performance. For PostMark, shown in Figure 22(a), performance increases with extent size, but starts to decline after 512KB extents. Larger extents favor performance as larger sequential writes are exhibited, in conjunction with PostMark being write-dominated. Write I/O volume always decreases, as larger extents have less free space left unutilized. Read I/O volume is high when using 8KB extents, as the placement of compressed block is inefficient and more extents must be used to store the same amount of compressed data. Read volume remains the same for extents between 16KB and 64KB, but increases after 128KB, as the degree of locality exhibited by PostMark is smaller than the extent size. For SPECsfs2008, shown in Figure 22(b), we use a 3,400 ops/sec load. SPECsfs2008 performs best when using 64KB extents and degrades at larger extent sizes as the read I/O volume significantly increases. Similarly, TPC-C (Figure 22(c)) performs roughly the same for extents between 8KB and 32KB, but performance drops for larger extents. TPC-H, shown in Figure 22(d), is less sensitive to

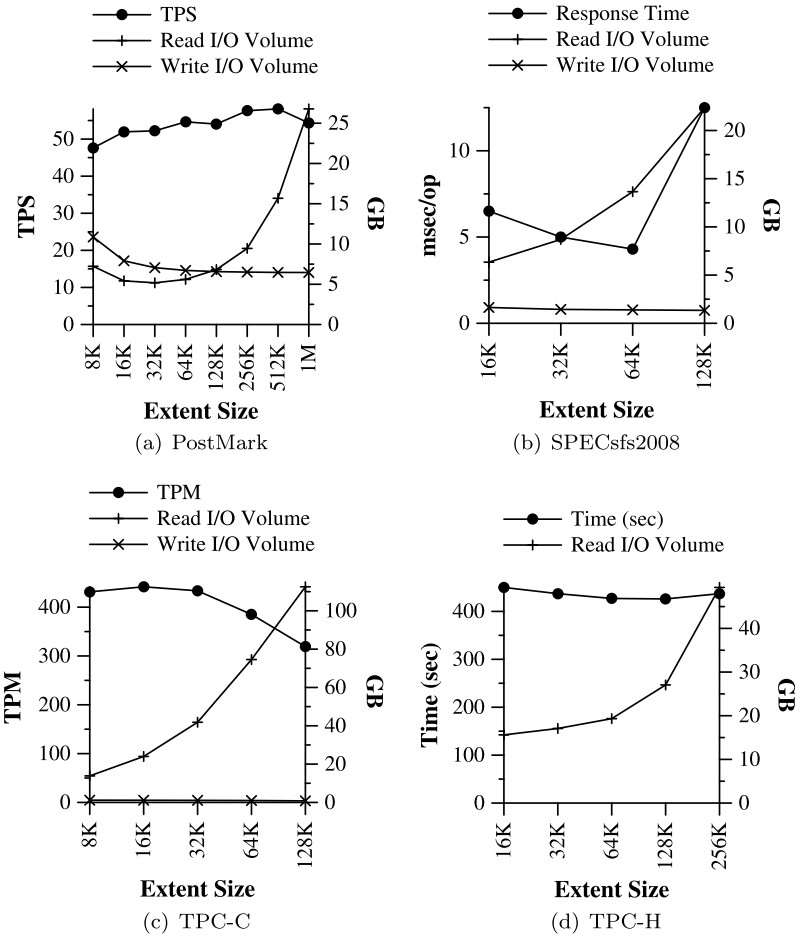


Fig. 22. Impact on performance of the extent size.

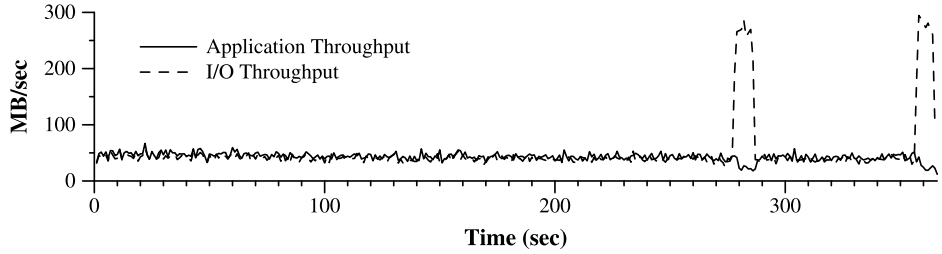


Fig. 23. Impact of cleaner on performance.

the extent size, as most queries are CPU bound. Overall, extent sizes between 16KB and 64KB seem to be a reasonable choice for such workloads.

4.2.8. Effect of Cleanup on Performance. Figure 23 illustrates the impact on PostMark performance when the cleaning mechanism is activated during PostMark execution. In this configuration, we use a disk partition that cannot hold the entire write volume

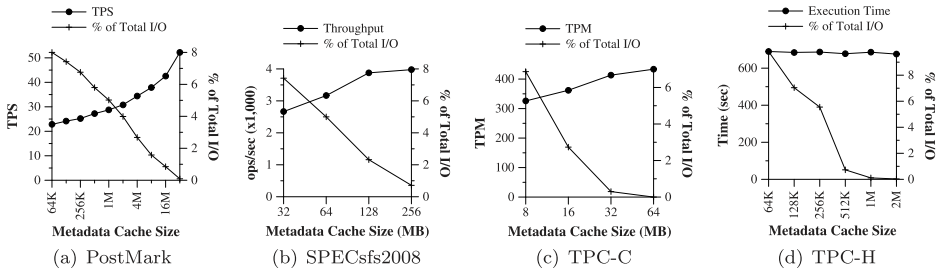


Fig. 24. Impact on performance of metadata cache size.

generated by PostMark, 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 25% of free extents above which the cleaner stops. The impact of the cleaner on performance is seen as two "valleys" in the throughput graph, between time periods from 280 to 290 and from 355 to 370. 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. When the cleaner is running, PostMark performance degrades by up to 150% but I/O throughput increases as a result of the large reads the cleaner exhibits during the extent scan phase. In these two time periods, the cleaner reclaims 15% of the disk capacity in 10 and 15 seconds, corresponding to 1.4GB of free space.

The rate the cleaner reclaims space is practically limited by the disk 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. 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.

4.2.9. Metadata I/Os. Figure 24(a) illustrates the impact of metadata I/Os on PostMark performance. Although the metadata I/O volume only accounts for a small fraction of the application's total one, it's impact on performance is substantial. Metadata I/Os are synchronous, random and interfere with application I/Os. Given that PostMark has only one outstanding operation, single-thread latency is significantly affected. As the size of the metadata cache increases, performance significantly improves, by up to 100%. For SPECsfs2008, shown in Figure 24(b), we use the `lzo` compression library at a target load of 4,000 CIFS ops/sec. Similarly to PostMark, SPECsfs2008 also suffers performance penalty due to metadata I/Os by up to 49%, although at a much smaller scale, mainly due to the abundance of outstanding I/Os. TPC-C (Figure 24(c)) suffers less performance penalty compared to SPECsfs2008, up to 33%. Finally, TPC-H performance degrades only slightly, shown in Figure 24(d), as the bottleneck is the single CPU that MySQL uses for the queries.

4.2.10. Summary of Results. Overall, we find that transparent compression degrades performance by up to 34% and 15% for TPC-C and TPC-H, respectively, as they are sensitive to latency. For PostMark and SPECsfs2008, compression actually improves performance by up to 80% and 35%, respectively, as a result of the log-structured writes. CPU utilization increases due to compression, by up to 311%. These results

Table III. Space Savings for Various Compression Methods and File Types. *gzip* is Always with *-6* (default)

Files	Orig. MB	gzip -r	gzip .tar	NTFS	ZFS	*ZBD* (`zlib`)	*ZBD* (`lzo`)
mbox 1	125	N/A	29%	7%	4%	17%	11%
mbox 2	63	N/A	68%	39%	31%	54%	34%
MS word	1100	50%	51%	37%	35%	44%	33%
MS excel	756	67%	67%	47%	41%	55%	47%
PDF	1400	22%	22%	14%	15%	15%	12%
Linux kernel							
source	277	55%	76%	27%	33%	69%	46%
compiled	1400	63%	71%	47%	52%	67%	58%

show that transparent compression is a viable option for increasing effective storage capacity when single-thread latency is not critical. Moreover, compression is beneficial for I/O performance when there is an abundance of outstanding I/O operations, as compression cost is effectively overlapped with I/O.

5. DISCUSSION

There are three remaining considerations for our proposed compression scheme: (1) the space and performance efficiency of *ZBD*, compared to using other alternatives; (2) the percentage of performance improvement that comes as a result of modifying the underlying disk layout; and (3) how the capacity of a compressed device whose size varies over time is presented to higher layers. We discuss these issues next.

5.1. Compression Efficiency

In this section, we compare the efficiency of our approach with other alternatives in terms of: (1) Space, and (2) Performance.

5.1.1. Compression Space Efficiency. An issue when employing block-level compression is the achieved compression efficiency when compared to larger compression units, such as files. The layer at which compression is performed affects coverage of the compression scheme. For instance, block-level compression schemes will typically compress both data and file-system metadata, whereas file-level approaches compress only file data. Table III 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 *ZBD* is slightly superior to file-level compression when using `zlib`, and slightly inferior, when using `lzo`. Finally, we must point out that although NTFS and ZLIB use the same basic algorithm, the latter is more space efficient than the former. We argue that this is due to two reasons. First, the compression variant used is different between the two systems (*LZNT1* for NTFS, *DEFLATE* for ZLIB). This accounts for the up to 15% difference in the first five cases. Then, the significant difference, up to 42%, observed between ZLIB and NTFS in the Linux kernel case, is because compressing files less than 4KB using NTFS may make these files bigger. Given that about 80% of the files of the Linux Kernel is less than 4KB, the compression space efficiency is decreased significantly when using NTFS.

5.1.2. Compression Performance Efficiency. In this section, we compare the performance of our approach with that achieved by NTFS. We run all measurements using all eight

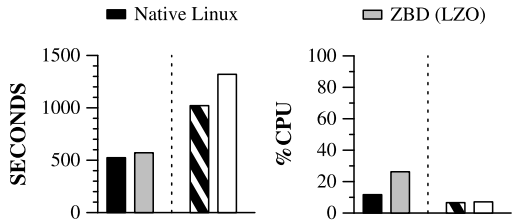


Fig. 25. *ZBD*/NTFS Comparison for TPCH.

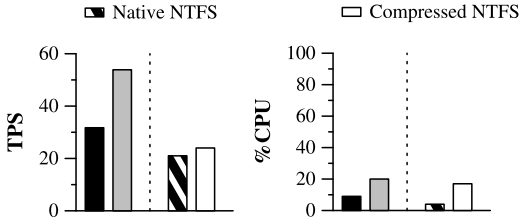


Fig. 26. *ZBD*/NTFS Comparison in PostMark.

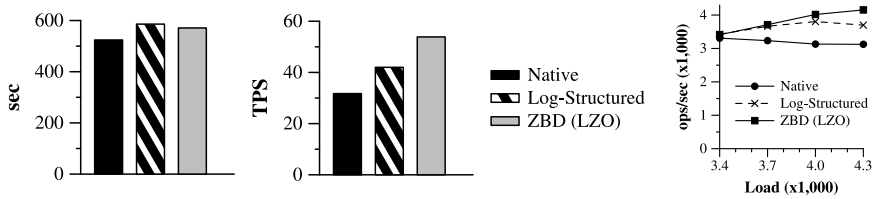
available CPUs in the system, as we want to compare the best case scenario in both environments. In all Windows experiments, we use Microsoft Windows Server 2003. Our results for TPC-H and PostMark are shown in Figures 25 and 26, respectively.

Overall, we see that for TPC-H both *ZBD* and NTFS cause performance degradation compared to the native Linux and Windows (without compression) experiments, by 8.9% and 29% respectively. This is because TPC-H has very few outstanding I/Os and decompression cannot be effectively overlapped with I/O, thus degrading performance in both cases. Still, the performance degradation in *ZBD* is significantly less than that caused by NTFS with compression enabled. Contrary to TPC-H, for PostMark, both *ZBD* and NTFS improve performance compared to the native Linux and Windows experiments, by 69% and 14%, respectively. We observe that, although both *ZBD* and NTFS significantly increase CPU utilization, *ZBD* achieves better performance.

5.2. Decoupling Compression and Log-Structure Layout

An important aspect of our compression design is that it modifies the on-disk layout underneath the file system, by using log-structure layout for write operations. We have already pointed out that PostMark and SPECsfs2008 benefit significantly from this log-structured layout. To quantify the impact of log-structure layout on performance, we develop a block device driver which performs log-structure writes, and efficiently executes the read operations by keeping all the required metadata in DRAM. This is equivalent to *ZBD* if we turn the cleaner off, perform no compression/decompression, and using a small metadata cache since, as we have shown in Section 4.1.7, a cache of 64KB suffices to sustain I/O performance in the workloads we examine. Figure 27 shows our results for TPC-H, PostMark, and SPECsfs2008. In all cases, we use all of the eight available CPU cores of our test system.

Overall, we observe that the log-structure writes accounts for 32% out of 69%for PostMark and for the 3,700 CIFS ops/sec out of 4,300 CIFS ops/sec for SPECsfs2008. When using all eight CPUs, *ZBD* achieves substantially higher performance than the log-structured writes, as compression further reduces the I/O volume as well. Thus, 50% improves comes from log-structured writes, and another 50% from using compression. However, unlike in the SPECSsfs2008 and PostMark experiments, the



(a) TPC-H (b) PostMark (c) SPECsfs2008

Fig. 27. Performance comparison between (i) the native HDD setup, (ii) a custom-build driver that performs log-structure writes, and (iii) *ZBD*.

performance for TPC-H with log-structured writes is worse than with *ZBD*, by an additional 3% degradation to the 9% degradation of *ZBD*.

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 filesystems need to attach a specific size attribute to it, before applications can access it. To avoid resizing, *ZBD* presents to higher layer a larger size for newly created devices. This overbooking 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.

Therefore, we employ an alternative approach to unused space: *ZBD* is conservative at the beginning and declares 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 *ZBD*. 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, for example, using thin provisioning.

6. RELATED WORK

This work has previously been presented in Makatos et al. [2010a, 2010b]. The purpose of this article is to provide an overall view of our work in the area. Next, we discuss related work highlighting the main differences from this work.

Using 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 Douglis [1992]. 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. Poess and Potapov [2003] examined 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.

Cormack [1985] 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.

Ng and Ravishankar [1997] 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, *ZBD* employs extensive metadata at the block-layer and multiple techniques to reduce all induced metadata-and I/O-related overheads.

Also, previous research [Appel and Li 1991; Douglis 1993; Rizzo 1997; Wilson et al. 1999] has argued that online compression of memory (fixed-size) pages can improve memory performance. More recently, Linux Compcache [Gupta 2010] 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.

File-System Compression. Compression has been employed by file-systems to improve effective storage capacity: Overall, Coffing and Brown [1997] categorizes and discusses various file-level compression approaches, while a survey of data compression algorithms can be found in Lelewer and Hirschberg [1987] and Smith and Storer [1985].

Burrows et al. [1992] describe how LFS can be extended with compression. They use a 16KB 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 1.6 performance degradation for file-system intensive operations. They use the Sprite file-system for managing variable size metadata.

Cate and Gross [1991] 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.

Sun's ZFS [Oracle Corporation and Sun Microsystems, Inc. 2009] and Microsoft's NTFS [Microsoft Corporation 2009] are the widely-used file-systems supporting compression over any storage device. NTFS compresses *clusters* of data (64KB by default) and uses a finite number of blocks to store the compressed chunk [Microsoft Corporation 2008], that is, eight 4KB-blocks for a 64-KB compressed chunk and a typical 2:1 compression ratio. This technique greatly simplifies block allocation/deallocation, as the block allocation process is not affected by the fact that data are compressed. The only difference between storing compressed and uncompressed clusters is that the block allocator has to allocate (or deallocate) a fewer number of blocks, and not variable-size segments within a block. However, such coarse-grained compression can lead to devastating performance if the application exhibits poor locality; random reads

result in decompressing 64KB instead of 4KB, and small random writes in decompressing the chunk, replacing the new block, recompressing the entire chunk, and writing the chunk back to disk. Our approach uses fine-grained unit of compression, making it suitable for any kind of workload.

`e2compr` [Ayers 1997] is an extension of the *ext2* file-system, where a user explicitly selects the files and directories to be compressed. LogFS [Engel and Mertens 2006] is specifically designed for flash devices and supports compression. FuseCompress [Svoboda 2010] offers a mountable Linux file-system that transparently compresses its contents. Compression is also supported in read-only file-systems, such as SquashFS [Lougher and Lougher 2008] and CramFS [Yang et al. 2005].

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.

Other Capacity and I/O Optimization Techniques. Deduplication [Manber 1994; Zhu et al. 2008; Bobbarjung et al. 2006] 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. Zhu et al. [2008] show how they are able to achieve over 210MB/s for four write data streams and over 140MB/s for four read data streams on a storage server with two dual-core CPUs at 3GHz, 8GB 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.

Understanding and Improving SSDs. Recent efforts have explored SSD tradeoffs [Agrawal et al. 2008] and performance characteristics [Dirik and Jacob 2009] as well as improving various aspects of SSDs, such as block management [Rajimwale et al. 2009], random write performance [Kim and Ahn 2008], and file-system behavior [Aleph One Ltd, Embedded Debian 2002; Engel and Mertens 2006; Woodhouse 2001]. Our work is orthogonal to these efforts, as *FlaZ* can use any type of flash-based device as a compressed cache.

Block-Level Compression. The only systems that have considered block-level compression are cloop [Russel 2002] and CBD [Savage 2006]. 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. *ZBD* 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. *ZBD* is a fully functional block device and supports compressed writes. To achieve this, *ZBD* uses an out-of-place update scheme that requires additional metadata and deals with the associated challenges.

SSD Caching. Finally, 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, Kgil and Trevor [2006] 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. Lee et al. [2008] 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. Narayanan et al. [2009] examine whether SSDs can fully replace traditional disks in datacenter 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 [Fusion-io 2007], HotZone [North American Systems International, Inc.], MaxIQ [Adaptec Inc. 2009], and ZFS's L2ARC [Leventhal 2008], 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 [Microsoft Corporation 2010] 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, for example, 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, *ZBD* employs all required metadata and techniques for achieving transparent compression and can be layered below any file-system or other application.

7. CONCLUSIONS

In this work, we use transparent compression to improve the space-efficiency and performance of online disk-based storage systems. We design and implement *ZBD* and examine the tradeoffs associated with I/O volume, CPU utilization and metadata I/Os. Our results show that online transparent compression is a viable option for increasing storage capacity. Performance improves for write intensive workloads, 80% for PostMark and 35% for SPECsfs2008, since the log-structure writes greatly improve the access pattern, reducing seeks via tighter disk placement. Performance degradation occurs only when single-thread latency is critical, by up to 34% for TPC-C and 15% for TPC-H. Our results indicate that compression has the potential to increase I/O performance, provided that the workload exhibits enough I/O concurrency to effectively overlap compression with I/O and that the CPUs can accommodate compression overheads.

Then, we examine how 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 uses *ZBD* to compress data as they flow between main memory and the SSD cache. Compression is performed online, in the common path, and increases the effective cache size. We evaluate our approach using realistic workloads, TPC-H, PostMark, and SPECsfs2008, 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 and cost-efficiency of I/O subsystems.

REFERENCES

ADAPTEC, INC. 2009. MaxIQ SSD cache performance. White paper. www.adaptec.com/en-US/products/CloudComputing/-MaxIQ/SSD-Cache-Performance/index.htm.

AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 57–70.

ALEPH ONE LTD, EMBEDDED DEBIAN. 2002. Yaffs: A NAND-Flash Filesystem.

APPEL, A. W. AND LI, K. 1991. Virtual memory primitives for user programs. *SIGPLAN Notes 26,* 4, 96–107.

AYERS, L. 1997. E2compr: Transparent file compression for Linux. http://e2compr.sourceforge.net/.

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

BURROWS, M., JERIAN, C., LAMPSON, B., AND MANN, T. 1992. On-line data compression in a log-structured file system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*. ACM, New York, 2–9.

CATE, V. AND GROSS, T. 1991. Combining the concepts of compression and Caching for two-level filesystem. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*. ACM, New York, 200–211.

COFFING, C. AND BROWN, J. H. 1997. A survey of modern file compression techniques. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.9847.

CORMACK, G. V. 1985. Data compression on a database system. *Comm. ACM 28,* 12, 1336–1342.

DEUTSCH, L. P. AND GAILLY, J.-L. 1996. ZLIB Compressed Data Format Specification version 3.3. Internet RFC 1950.

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

DOUGLIS, F. 1992. On the role of compression in distributed systems. In *Proceedings of the ACM SIGOPS, EW 5*. 1–6.

DOUGLIS, F. 1993. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of the Winter USENIX Conference*. 519–529.

ENGEL, J. AND MERTENS, R. 2006. LogFS - finally a scalable flash file system. http://lazybastard.org/ joern/logfs1.pdf.

FUSION-IO. 2007. Fusion-IO's solid state storage: A new standard for enterprise-class reliability. http://www.fusionio.com.

GUPTA, N. 2010. Compcache: Compressed in-memory swap device for Linux. http://code.google.com/p/compcache.

KATCHER, J. 1997. PostMark: A new file system benchmark. http:// www.netapp.com/ tech_library/3022.html.

KGIL, T. AND TREVOR, M. 2006. Flashcache: A NAND flash memory file cache for low power web servers. In *Proceedings of the CASES'06*. ACM, 103–112.

KIM, H. AND AHN, S. 2008. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, Berkeley, CA, 1–14.

LEE, S.-W., MOON, B., PARK, C., KIM, J.-M., AND KIM, S.-W. 2008. A case for flash memory SSD in enterprise database applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. ACM, New York, 1075–1086.

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

LEVENTHAL, A. 2008. Flash storage memory. *Comm. ACM 51,* 7, 47–51.

LOUGHER, P. AND LOUGHER, R. 2008. SquashFS. http://squashfs.sourceforge.net.

MAKATOS, T., KLONATOS, Y., MARAZAKIS, M., FLOURIS, M. D., AND BILAS, A. 2010a. Using transparent compression to improve SSD-based I/O caches. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, NY, 1–14.

MAKATOS, T., KLONATOS, Y., MARAZAKIS, M., FLOURIS, M. D., AND BILAS, A. 2010b. ZBD: Using transparent compression at the block level to increase storage space efficiency. In *Proceedings of the IEEE International Workshop on Storage Network Architecture and Parallel I/Os*. 61–70.

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

MEISNER, D., GOLD, B. T., AND WENISCH, T. F. 2009. POWERNAP: Eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. ACM, New York, 205–216.

MICROSOFT CORPORATION. 2008. Understanding NTFS Compression. http://blogs.msdn.com/ntdebugging/archive/2008/05/20/-understanding-ntfs-compression.aspx.

MICROSOFT CORPORATION. 2009. Best practices for NTFS compression in Windows. support.microsoft.com/default.aspx?scid=kb;en-us;Q251186.

MICROSOFT CORPORATION. 2010. Explore the features: Windows ReadyBoost. www.microsoft.com/windows/windows-vista/features/readyboost.aspx.

NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. 2009. Migrating server storage to SSDS: Analysis of tradeoffs. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*. ACM, New York, 145–158.

NG, W. K. AND RAVISHANKAR, C. V. 1997. Block-oriented compression techniques for large statistical databases. *IEEE Trans. Knowl. Data Eng. 9,* 2, 314–328.

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

OBERHUMER, M. F. X. J. 2008. LZO–A real-time data compression library. http://www.oberhumer.com/opensource/lzo/.

ORACLE CORPORATION AND SUN MICROSYSTEMS, INC. 2009. Oracle Solaris ZFS. http://www.oracle.com/us/products/servers-storage/storage/storage-software/031857.htm.

POESS, M. AND POTAPOV, D. 2003. Data compression in oracle. In *Proceedings of the 29th VLDB Conference*.

RAJIMWALE, A., PRABHAKARAN, V., AND DAVIS J. D. 2009. Block management in solid-state devices. In *Proceedings of the USENIX Annual Technical Conference*.

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

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10,* 1, 26–52.

RUSSEL, P. 2002. The compressed loopback device. http://www.knoppix.net/wiki/Cloop.

SAVAGE, S. 2006. CBD compressed block device, new embedded block device. http://lwn.net/Articles/168725.

SMITH, M. E. G. AND STORER, J. A. 1985. Parallel algorithms for data compression. *J. ACM 32,* 2, 344–373.

SPEC. 2008a. 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/.

SPEC. 2008b. SPECsfs2008_cifs published results, as of Nov-10-2009. http://www.spec.org/sfs2008/results/-sfs2008.html.

SPEC. 2009. SPECmail2009 published results, as of Nov-06-2009. http://www.spec.org/mail2009/results/-specmail_ent2009.html.

SVOBODA, M. 2010. FuseCompress, a mountable Linux file system which transparently compress its content. http://miio.net/wordpress/projects/fusecompress/.

THOMAS, C. AND WONG, M. 2007. Database Test 2 (DBT-2), an OLTP transactional performance test. http://osdldbt.sourceforge.net/.

TPC. 1997. Overview of the TPC benchmark C: The order-entry benchmark. http://www.tpc.org/tpcc/default.asp.

TPC. 2009a. Top ten non-clustered TPC-H published results by performance. http://tpc.org/tpch/results/tpch_perf_results.asp?resulttype=noncluster.

TPC. 2009b. TPC-H: An ad-hoc, decision support benchmark. www.tpc.org/tpch.

WELCH, T. A. 1984. A technique for high-performance data compression. *IEEE Computer 17,* 6, 8–19.

WILSON, P. R., KAPLAN, S. F., AND SMARAGDAKIS, Y. 1999. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 101–116.

WOODHOUSE, D. 2001. JFFS: The Journalling Flash File System. http://www.csie.nctu.edu.tw/~ijsung/documents/jffs2.pdf.

YANG, L., DICK, R. P., LEKATSAS, H., AND CHAKRADHAR, S. 2005. Crames: Compressed ram for embedded systems. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*. ACM, New York, 93–98.

ZHU, B., LI, K., AND PATTERSON, H. 2008. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, Berkeley, CA, 1–14.

ZIV, J. AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23*, 337–343.

Received March 2011; revised July 2011; accepted October 2011