ZBD: Using Transparent Compression at the Block Level to Increase Storage Space Efficiency

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

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

Abstract—In this work we examine how transparent compression in the I/O path can improve space efficiency for online storage. We extend the block layer with the ability to compress and decompress data as they flow between the file-system and the disk. Achieving transparent compression requires extensive metadata management for dealing with variable block sizes, dynamic block mapping, block allocation, explicit work scheduling and I/O optimizations to mitigate the impact of additional I/Os and compression overheads. Preliminary results show that online transparent compression is a viable option for improving effective storage capacity, it can improve I/O performance by reducing I/O traffic and seek distance, and has a negative impact on performance only when single-thread I/O latency is critical.

Keywords-online block-level compression; I/O performance; log-structured block device

I. INTRODUCTION

Although disk storage cost per GB has been steadily declining, the demand for additional capacity has been growing faster. For this reason, various techniques for improving effective capacity have gained significant attention [1]. In this work we examine the potential of transparent data compression [2] for improving space efficiency in *online* disk-based storage systems.

Previously, compression has been mostly applied at the file level [3]. Although this approach has the effect of reducing the space required for storing data, it imposes restrictions, such as the use of specific file-systems (i.e. NTFS or ZFS). In our work we explore data compression at the block-level by using a block-storage layer, *ZBD*, in the Linux kernel that transparently compresses and decompresses data as they flow in the system.

Block-level compression appears to be deceptively simple. Conceptually, it merely requires intercepting requests in the I/O path and compressing (decompressing) data before (after) writes (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 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 when using disk-based 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.

In this work we use a transparent compression system at the block layer for improving the efficiency of online storage. We examine the performance and tradeoffs associated with I/O volume, CPU utilization, and metadata I/Os.

For our evaluation we use four realistic workloads: PostMark, SPECsfs2008, TPC-C and TPC-H. Our results show that compression degrades performance by up to 15% and 34%

for TPC-H and TPC-C respectively, but improves by 80% and 35% for PostMark and SPEC SFS respectively. 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 is in-line with current technology trends. 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. However, our results indicate that compression can increase I/O performance for workloads that exhibit enough I/O concurrency.

The rest of this paper is organized as follows. Section II discusses the design of *ZBD* and how it addresses the associated challenges. Section III presents our evaluation methodology. Section IV presents our experimental results. Section V discusses previous and related work. Finally, we draw our conclusions in Section VI.

Fig. 1. *ZBD* read and write paths.

II. DESIGN AND IMPLEMENTATION

The design of the *ZBD* system has been presented in [4]. For completeness we summarize here the main design issues. In *ZBD*, application-generated file-level I/O operations pass through the *ZBD* virtual block-device layer, where compression and decompression takes place, for write and read accesses, respectively. The block-level interface allows us to be *file-system agnostic*, but raises the complication of variable-size segments. 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.

A. Space Management

Compressing fixed-size blocks results in variable-size segments, that require special handling for storing them. Compressed blocks are stored in fixed-size *physical extents*. The extent size, a multiple of the block size of the underlying storage, defines the unit of I/O operations in our system. A logical-to-physical mapping, via a translation table, determines the corresponding physical extent for each logical block referenced in an I/O operation. This translation map is stored persistently within a reserved region of the underlying storage. Since an extent may host at any point in time multiple logical blocks, an additional mapping is needed to determine which region within the extent corresponds to a referenced logical block. This extent-level mapping is achieved by traversing a linked-list embedded within the extent. The traversal takes place in system memory, once the extent has been fetched from the underlying storage. With space savings from data compression typically in the order of 40%, we have found that the linked-list within an extent consumes around 0.6% of the space needed for the actual data contents. All free space within an extent is contiguous, located at the end of the extent.

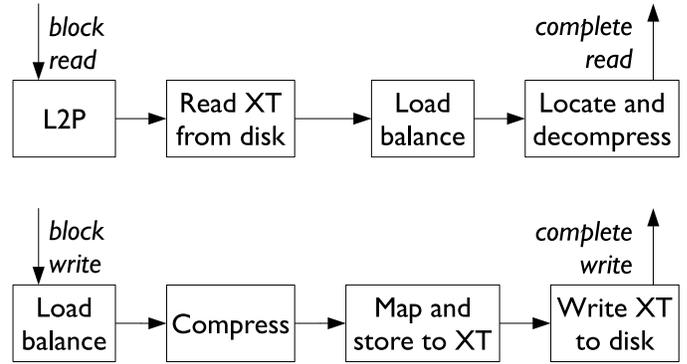Block re-writes generally result in a different compressed footprint, making in-place updates complicated. Rather than handling the complications of updating the space representation within the affected extent, we have opted to always write modifications to new extents, similar to log-structured file-systems [5]. A key benefit is that we do not have to read the affected extent in memory before modifying it. On the other hand, as time progresses the contents of a large fraction of the extents on the underlying storage may become invalid, while still consuming space. *ZBD* incorporates a *cleaner* thread, which is triggered when the amount of free extents falls below a threshold and tries to reclaim extents, similarly to the cleaner of Sprite LFS [5]. The cleaner needs to determine which of the blocks within an extent are valid, or "live", and then to compact "live" blocks into new extents, updating the logical-to-physical translation map in the process. Extents with no "live" blocks are marked as members of a *free-extents* pool. Moving a "live" block to a new extent only requires a memory copy but no compression/decompression.

The cleaner generates read I/O volume proportional to the extents that are scanned and write I/O volume proportional to the extents resulting from compacting live blocks. To improve cleaner efficiency in terms of reclaimed space, we use a first-fit, decreasing-size policy when moving live blocks to new extents. This approach minimizes the space wasted when placing variable-size blocks into fixed size extents, as it places larger blocks into as few extents as possible and uses smaller ones to fill extents having little free space.

Placement issues during cleanup are important. Note that the relative location of logical blocks within an extent is not as important, because extents are read in memory in full. There are two issues that arise as a consequence: (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 an 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 overall location on the disk.

Production servers usually exhibit periods of low activity, so

we expect the cleaner to not directly interfere with application-generated I/O operations. The impact of the cleaner is explicitly evaluated in Section IV-I.

B. Extent Buffer

To mitigate the impact of additional I/Os due to read-modify-writeback sequences, the compression layer of *ZBD* uses a buffer in DRAM for extents. This a fully-set-associative buffer with LRU eviction. 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 tunable parameters.

The extent size affects the degree of internal fragmentation, and may have an effect on locality: larger extents have higher probability to contain unrelated data when applications generate random writes, while smaller extents suffer from internal space fragmentation. Furthermore, 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 early, internal fragmentation may be increased. To balance this tradeoff, the extent buffer uses an *aging timeout* that specifies the maximum amount of time an extent can remain in it.

When writing compressed blocks from concurrently executing contexts, a decision must be made as whether to write blocks in the same or different extents. Concurrent writes to an extent involve a tradeoff between lock contention and CPU overhead for memory copies. Our experience has shown that it is best to preserve locality of reference as much as possible, albeit at the cost of less efficient space utilization. We have determined that buffering a small number of extents in memory suffices for preserving locality. Extents are written back (if modified) as soon as (a) they become full, i.e. there is not enough space for a write to append more blocks, and (b) there is no reference to them, i.e. no ongoing read I/O operation references this extent.

The size of the extent buffer is set to 16 MB: this size can help performance for workloads with small but sequential accesses. A larger extent buffer does not affect performance, as the more DRAM dedicated to the extent buffer, the less DRAM is left for the buffer cache.

C. Work Scheduling

Adding compression and decompression overheads in the I/O path increases the latency of single I/O operations; yet, the common case is that a server needs to handle multiple outstanding I/O operations. By explicitly managing concurrency, we have a chance to effectively hide the compression latency, as the result of overlapping these computations with other concurrent operations.

To allow for a high degree of concurrent I/O operations progressing asynchronously, *ZBD* uses callback handlers to avoid blocking on synchronous kernel calls. To hide the impact of compression on large I/Os, *ZBD* uses multiple cores when processing a single large I/O. We split large I/O requests to units of individual blocks that are compressed or decompressed independently by different cores. This decreases response time for reads and writes, and also reduces the delay writes may introduce to read processing.

With *ZBD* active beneath the file-system layer and the buffer-cache, write requests typically come in batches as the result of buffer-cache flushing. However, it is less common to have multiple concurrent large reads, therefore decompression can significantly increase their response time. *ZBD* manages two work-queues for all CPU-cores, one for reads and one for writes, with the read work-queue having higher priority.

Decompression is performed after the I/O read to the disk has completed and the extent has been read in memory. Decompression could 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 context execution is scheduled on the same CPU the top-half occurred, hence restricting parallelism. With separate threads for issuing I/Os and performing decompression, this problem is mitigated.

Finally, *ZBD* takes special care to allow multiple readers and writers to access non-overlapping regions within the same extent. Placing a compressed block in an extent is done in two steps: First, free space in the extent is reserved. Second, block contents are copied inside the extent. Multiple write contexts can copy blocks into the same extent simultaneously, since the pre-allocation step ensures proper space management.

D. Metadata and Data Consistency

The logical-to-physical translation map along with the free extents pool are all the metadata *ZBD* requires. In this work we focus on the performance aspect of transparent compression and assume that metadata consistency in case of a failure is guaranteed by the use of NVRAM. The use of NVRAM is essential in our design to avoid synchronous metadata updates. Moreover, the extent buffer also needs the persistence guarantees of NVRAM, otherwise a write request would require a full extent flush before it is completed, which would result in significantly higher write I/O volume, and consequently, lower performance. The amount of NVRAM required is small, typically in the order of a few tens of MB, as it only requires to store *pending* extent writes and dirty metadata blocks.

E. Power Efficiency

The trading of CPU cycles for increased storage 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, compression translates to reduced I/O volume and possibly less devices, improving power consumption. This is an additional parameter to be taken into account when trading CPU cycles for I/O performance, storage capacity or improved manageability. However, we believe that it is important to examine this tradeoff alongside offloading compression, e.g. to specialized hardware, and we leave this for future work.

III. 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 2 GHz, and 32 GB 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). The peak disk throughput is 100 MB/sec for reads, and 90 MB/sec for writes, respectively, while the average seek time is 12.6 milliseconds. Disk caching is set to write-through mode. The disks are configured as RAID-0 devices, using the MD software-RAID with the chunk-size set to 64 KB.

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

TABLE I
COMPRESSION/DECOMPRESSION COST OF A 4-KB BLOCK.

The algorithms and implementations used for compression and decompression of data are the default zlib [6] and lzo [7] libraries in the Linux kernel without any modifications, except for the pre-allocation 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 4 KB 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 I compares the performance characteristics of the zlib and lzo implementations. The extent size used is 32 KB in all of our experiments but we evaluate the impact of extent size separately in Section IV-F. We use four popular benchmarks, running over an XFS file-system with block-size set to 4 KB: 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.

A. PostMark

PostMark [8] 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. We present results from executing 50,000 transactions for a 35:65% read-write ratio, with 16 KB read/write operations, over 100 mailboxes where each mailbox is a directory containing 500 messages, and the message size ranging from 4 KB to 1 MB. By default, PostMark generates random (therefore, uncompressible) contents for each written block. 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 [9], and, moreover, the I/O workload is write-dominated. For these reasons, we use 1 GB of DRAM for PostMark.

B. SPECsfs2008

SPEC SFS [10] simulates the operation of an NFSv3/CIFS file-server; our experiments use the CIFS protocol. In SPEC SFS, 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$120 MB per operation/sec). SPEC SFS reports the number of operations-per-second actually achieved, and the average response time per operation. 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. As with PostMark, we modify SPEC SFS to use compressible contents for each block. For the SPEC SFS results, the DRAM size is set to 2 GB, under the assumption that this is close to the common file-set size to DRAM-size ratios in audited SPEC SFS results [11].

C. TPC-C (DBT-2)

DBT-2 [12] 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 TPC's TPC-C Benchmark specification [13]. We use a workload of 300 warehouses, which corresponds to a 28 GB database, 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 1 GB. This amount of DRAM is large enough to avoid swapping, but small enough to create more pressure on the I/O system.

D. TPC-H

TPC-H [14] 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 (4 GB of data, plus 2.5 GB of indices). We use queries Q1, Q3, Q4, Q6, Q7, Q10, Q12, Q14, Q15, Q19, and Q22, that keep execution time to reasonable levels. The compression ratio for this dataset is 39% using lzo and 48% using zlib. 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 1 GB, under the assumption that this is close to the common database-size to DRAM-size ratios in audited TPC-H results [15].

IV. Experimental Evaluation

In this section we first examine the impact of compression on performance and then we explore the impact of certain parameters on system behavior.
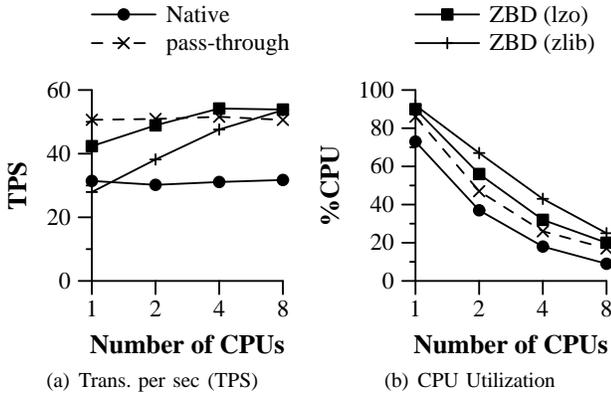
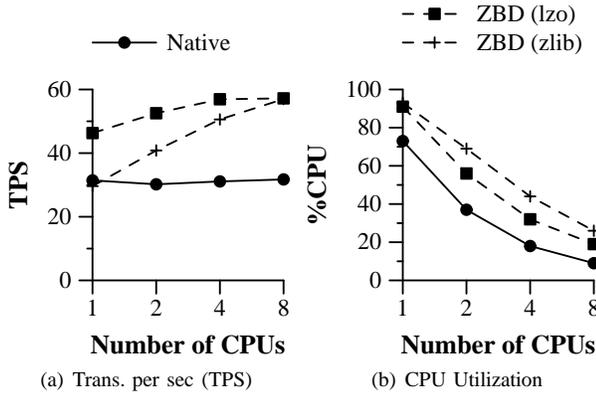Fig. 2. Results for PostMark with variable number of CPUs.

Fig. 3. Results for PostMark using more compressible data.

(a) Throughput

(b) Latency

(c) CPU Utilization

Fig. 4. Results for SPEC SFS.

A. PostMark

Figure 2 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 2(a). In pass-through mode, *ZBD* processes each I/O as if compression fails, no actual compression/decompression is performed. 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 3, 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. When using all eight CPUs, *ZBD* offers substantially higher performance than *ZBD* in pass-through mode, as compression reduces the I/O volume of the log-structured writes.

B. SPEC SFS

Figure 4 shows our results for SPEC SFS. Native sustains the initial load of 3,400 CIFS ops/sec, but fails to do so for higher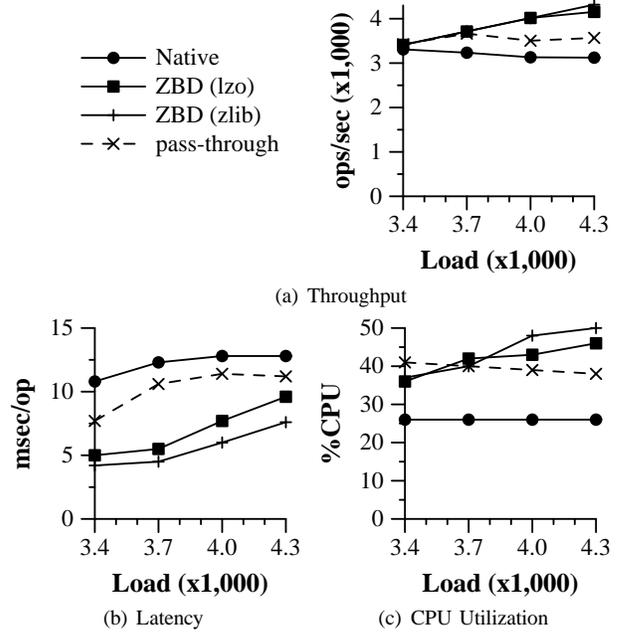 loads. Log-structured writes help *ZBD* to sustain higher loads, up to 3,700 CIFS ops/sec, as indicated by *ZBD* in pass-through mode. Data compression further improves performance; *ZBD* (`lzo`) sustains the load of 4,000 CIFS ops/sec. By using `zlib`, the higher compression ratio can sustain the highest 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 5 shows our results for SPEC SFS when using more compressible data for file contents. Higher compression ratio results in lower I/O volume, hence higher 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.

SPEC SFS 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.

C. TPC-C (DBT2)

Figure 6 shows our results for TPC-C. Performance degrades for *ZBD* by 31% for `lzo` and by 34% for `zlib`, whereas CPU utilization increases by 64% and 72%. TPC-C exhibits reads that are small (usually 4 KB) and random, leading to disproportionally high I/O read volume. Each read request practically translates to reading a full extent (32 KB in this configuration). In addition, decompression can not be parallelized effectively for small reads, leading to increased
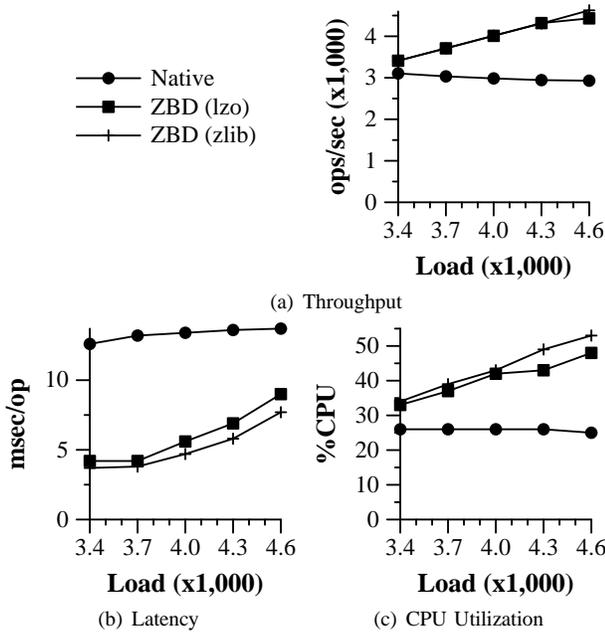
(a) Throughput

(b) Latency

(c) CPU Utilization

Fig. 5.   Results for SPEC SFS using more compressible data.

(a) New Order Transactions per min

(b) CPU Utilization

Fig. 6.   Results for TPC-C.

(a) Execution Time

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

Fig. 7.   Results for TPC-H.

(a) Execution Time

(b) CPU Utilization

Fig. 8.   Results for TPC-H (Q3) with variable number of CPUs.
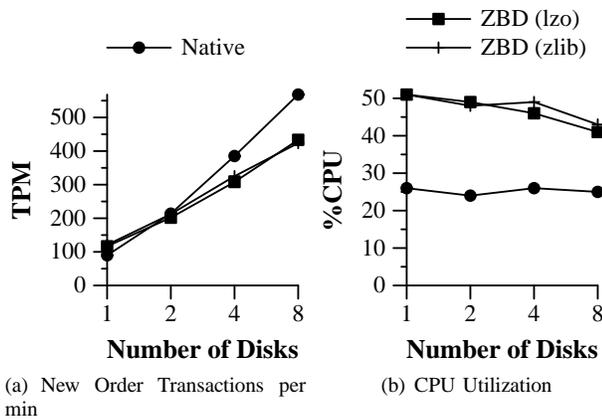
read response. As the number of disks decreases, I/O latency significantly increases making the decompression overhead less important. When using only one disk, performance improves by 29% and 34% for `lzo` and `zlib`, respectively.

D. TPC-H

Figure 7 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.

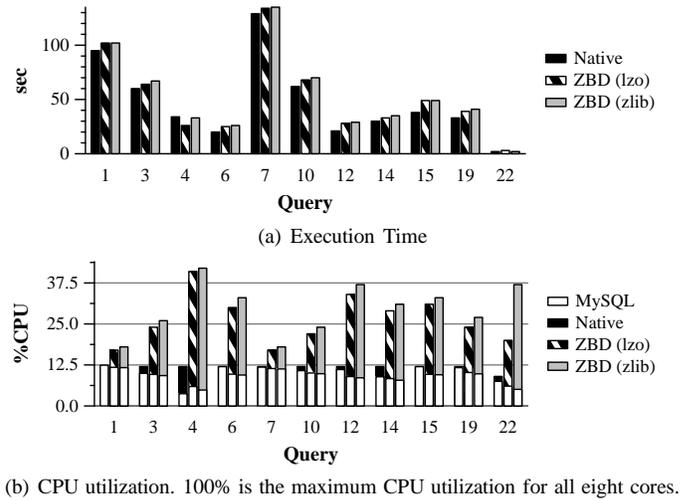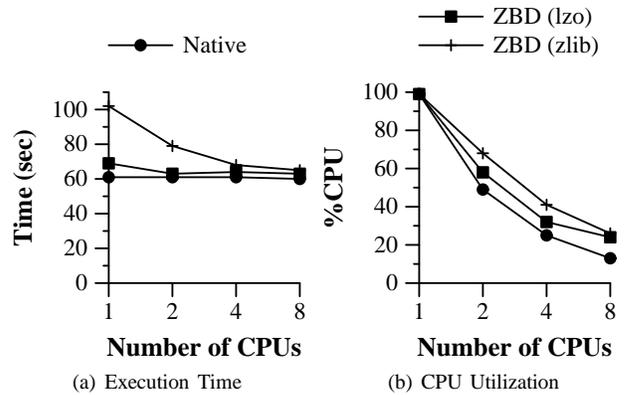In Figure 8 we execute Q3, varying the number of CPUs. Native is unaffected by the reduction in CPU power, as the query can consume at most one CPU. *ZBD* (`lzo`) suffers performance penalty when running at only one CPU by 13%, whereas *ZBD* (`zlib`) intensively contends with MySQL for CPU cycles, resulting in 67% performance degradation.

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

E. 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 9 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 6.5 GB 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

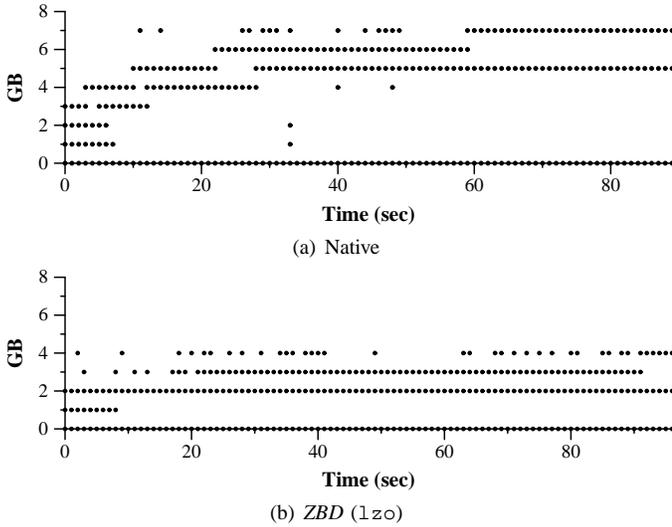(a) Native

(b) *ZBD* (lzo)

Fig. 9.   Disk access pattern for TPC-H (Q3).

these considerations, performance is still lower than native, as decompression cost dominates response time.
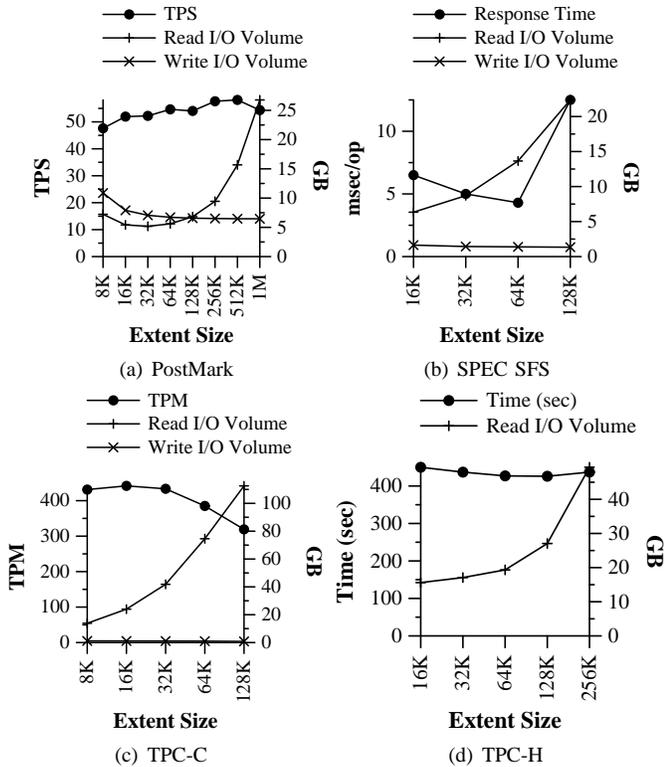

(a) PostMark

(b) SPEC SFS

(c) TPC-C

(d) TPC-H

Fig. 10.   Impact on performance of the extent size.

F. Extent Size

Figure 10 illustrates the impact of the size of the extent on performance. For PostMark, shown in Figure 10(a), performance increases with extent size, but starts to decline after 512-KB extents. Larger extents favor performance as larger
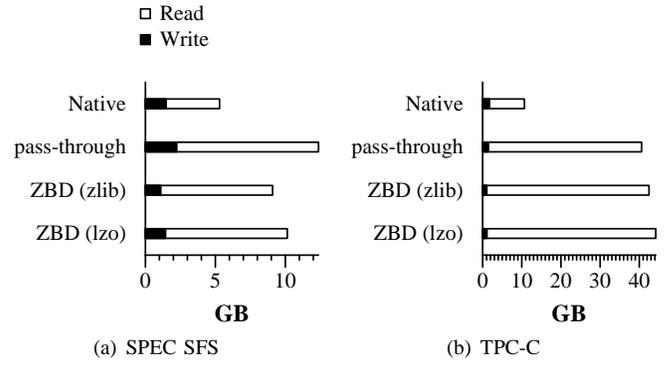

(a) SPEC SFS

(b) TPC-C

Fig. 11.   I/O volume for SPEC SFS at 3,400 CIFS ops/sec load, and for TPC-C using eight disks.

sequential writes are exhibited, in conjunction with PostMark being write-dominated. Write I/O volume always decreases, as larger extents have fewer free space left unutilized. Read I/O volume is high when using 8-KB 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 16 KB and 64 KB, but increases after 128 KB, as the degree of locality exhibited by PostMark is smaller than the extent size. For SPEC SFS, shown in Figure 10(b), we use a 3,400 ops/sec load. SPEC SFS performs best when using 64-KB extents and degrades at larger extent sizes as the read I/O volume significantly increases. Similarly, TPC-C (Figure 10(c)) performs roughly the same for extents between 8 KB and 32 KB, but performance drops for larger extents. TPC-H, shown in Figure 10(d), is less sensitive to the extent size, as most queries are CPU bound. Overall, extent sizes between 16 KB and 64 KB seem to be a reasonable choice for such workloads.

G. 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 11(a) shows a breakdown of the I/O volume for SPEC SFS at the 3,400 CIFS ops/sec load point. Total I/O volume increases by 91%, 71%, and 133% for *ZBD* (lzo), *ZBD* (zlib), and *ZBD* (pass-through), respectively. This increase is the result of reading *entire* extents for each 4 KB read issued by the SPEC SFS load generators. With *ZBD* (lzo, zlib) write I/O volume is reduced by 3% and 26% due to compression and increases for *ZBD* (pass-through) by 51% due to the space overheads of extent and per-block headers. Although total I/O volume increases for all *ZBD* configurations, performance increases, even for *ZBD* (pass-through), as shown in Figure 4(b). The only difference of *ZBD* (pass-through) and Native, apart from *ZBD* reading 185% more data due to extent reads, is the fact that Native's writes are random, whereas *ZBD*'s 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

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 source	277	55%	76%	27%	33%	69%	46%
compiled	1400	63%	71%	47%	52%	67%	58%

TABLE II

SPACE SAVINGS FOR VARIOUS COMPRESSION METHODS AND FILE TYPES. *gzip* IS USED WITH *-6* (DEFAULT) IN ALL CASES.

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 SPEC SFS, 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 11(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 6(a). *ZBD* improves TPC-C writes. However, this improvement alone cannot offset the negative impact of the additional read volume, as writes only make up 14% of TPC-C's I/O volume.

H. Compression 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 II 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.
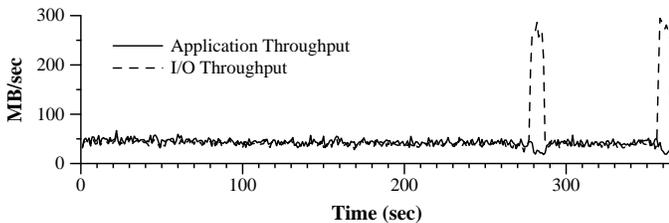

Fig. 12. Impact of cleaner on performance.

I. Effect of Cleanup on Performance

Figure 12 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 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 1.4 GB of free space.
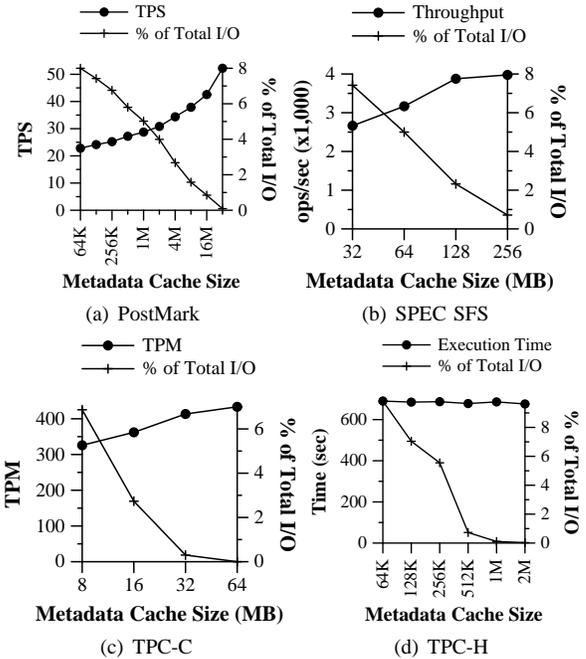

Fig. 13. Impact on performance of metadata cache size.

J. Metadata I/Os

Figure 13(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 SPEC SFS, shown in Figure 13(b),

we use the `lzo` compression library at a target load of 4,000 CIFS ops/sec. Similarly to PostMark, SPEC SFS 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 13(c)) suffers less performance penalty compared to SPEC SFS, up to 33%. Finally, TPC-H performance degrades only slightly, shown in Figure 13(d), as the bottleneck is the single CPU that MySQL uses for the queries.

K. 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 SPEC SFS, 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 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.

V. RELATED WORK

To the best of our knowledge, this is the first work that examines online data compression below the file-system, in the common block I/O path to improve the space-efficiency of disk-based storage systems. By being independent of the file-system implementation, it becomes possible to achieve high compression ratios, but also to assist other storage system optimizations. In a previous work [4], we have demonstrated how online compression can be used to increase the effective capacity of an SSD-based storage cache. In this paper we expose performance-related tradeoffs in the implementation of online block-level compression for hard disks.

The only systems that have considered block-level compression are cloop [16] and CBD [17]. 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.

Previous research [18], [19], [20], [21], [22], [23] has also argued that online compression of memory (fixed-size) pages can improve memory performance, while the authors in [24] argue that trends in processor and interconnect speeds will favor the use of compression in various distributed and net-worked systems, even if compression is performed in software. A survey of data compression algorithms appears in [2].

The authors in [3] gather data from various systems and show that compression can double the amount of data stored in a system. They also propose the architecture of a two-level, tiered file-system, where the first level is responsible for caching (uncompressed) files that are used frequently and the second for storing compressed files that are used less frequently. Finally, today exist a number of storage systems that encompass or take advantage of compression at the file-level: e2compr [25] (an extension for the *ext2* file-system), ZFS [26], Windows NTFS, and LogFS (designed specifically for flash devices, supporting compression). Unlike *ZBD*, these approaches are limited to a single file-system, they are typically not used with online storage. An exception to the aforementioned systems is FuseCompress [27], a pseudo file-system in FUSE that can be used atop of any file-system. However, FuseCompress compresses/decompresses entire files, making it suitable only for archival storage. With large files, as in the case of files representing the tables and indices of a relational database system, this would be prohibitively expensive.

The authors in [28] describe how LFS can be extended with compression. They use a 16-KB compression unit and a similar mechanism to our extents to store compressed data on disk. However, they rely on the Sprite file-system metadata for managing variable size metadata. They find that compression in the storage system has cost benefits and that in certain cases there is a 1.6 performance degradation for file-system intensive operations. In our work we use compression to improve storage space efficiency independent of the file-system at the cost of significant metadata complexity. We show how this complexity can be mitigated and evaluate our approach on modern architectures with realistic workloads.

The authors in [29] 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 routines for compression were more effective. Their approach reduces I/O traffic necessary for loading the data-base 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. Similarly, the authors in [30] 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 I/O performance.

Compression has been integrated in the Oracle database system [31] with the twin goals to not only reduce storage requirements for the database tables and indices in large-scale warehouse, but also to improve the execution time of certain classes of queries that access a large portion of the dataset. The implementation of the compression algorithm is specific to the database system, based on eliminating all duplicate values in a database block. In our work, we show that is feasible to use

a transparent block-level compression layer to achieve these benefits for a broader range of workloads.

Deduplication [32], [1], [33] is an alternative, space-savings approach that recently has attracted a lot of interest. 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 to the remaining data segments. The authors in [1] show how they are able to achieve over 210 MB/sec for 4 multiple write data streams and over 140 MB/sec for 4 read data streams on a storage server with two dual-core CPUs at 3 GHz, 8 GB of main memory, and a 15-drive disk subsystem (software RAID6 with one spare drive). Deduplication has so far been used in archival storage systems due to its high overhead.

VI. Conclusions

In this work we use transparent compression for online disk-based storage systems. We examine the performance and 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, and performance degradation is visible only when single-thread latency is critical, by up to 34% for TPC-C and 15% for TPC-H. Performance improves for write intensive workloads (80% for PostMark and 35% for SPEC SFS), mainly due to the log-structured writes. Our results indicate that compression has a 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.

Acknowledgements

References

[1] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. of FAST'08*, pp. 1–14, USENIX Association.

[2] D. A. Lelewer and D. S. Hirschberg, "Data compression," *ACM Comput. Surv.*, vol. 19, no. 3, pp. 261–296, 1987.

[3] V. Cate and T. Gross, "Combining the concepts of compression and caching for a two-level filesystem," in *Proc. of ASPLOS-IV*, pp. 200–211, ACM, 1991.

[4] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas, "Using Transparent Compression to Improve SSD-based I/O Caches." to appear in the ACM/SIGOPS European Conference on Computer Systems (EuroSys 2010).

[5] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM TOCS*, vol. 10, pp. 26–52, Feb. 1992.

[6] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, pp. 337–343, 1977.

[7] T. A. Welch, "A technique for high-performance data compression," *IEEE Computer Society Press*, vol. 17, no. 6, pp. 8–19, 1984.

[8] J. Katcher, "PostMark: A New File System Benchmark." http://www.netapp.com/ tech_library/3022.html.

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

[10] "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/.

[11] "SPECsfs2008_cifs published results, as of Nov-10-2009." http://www.spec.org/sfs2008/results/sfs2008.html.

[12] "Database Test 2 (DBT-2), an OLTP transactional performance test." http://osdldbt.sourceforge.net/.

[13] "TPC-C is an on-line transaction processing benchmark.." http://www.tpc.org/tpcc/default.asp.

[14] "TPC-H: an ad-hoc, decision support benchmark." www.tpc.org/tpch.

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

[16] P. Russel, "The compressed loopback device." http://www.knoppix.net/wiki/Cloop.

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

[18] A. W. Appel and K. Li, "Virtual memory primitives for user programs," *SIGPLAN Not.*, vol. 26, no. 4, pp. 96–107, 1991.

[19] L. Rizzo, "A very fast algorithm for RAM compression," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 2, pp. 36–45, 1997.

[20] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The case for compressed caching in virtual memory systems," in *In Proceedings of the 1999 USENIX Annual Technical Conference*, pp. 101–116, USENIX Association, 1999.

[21] F. Douglis, "The Compression Cache: Using On-line Compression to Extend Physical Memory," in *Proc. of 1993 Winter USENIX Conference*, pp. 519–529, 1993.

[22] T. Cortes, Y. Becerra, R. Cervera, and Ra, "Swap compression: Resurrecting old ideas," *SoftwarePractice and Experience (SPE*, vol. 30, p. 2000, 2000.

[23] L. Yang, R. P. Dick, H. Lekatsas, and S. Chakradhar, "Crames: compressed ram for embedded systems," in *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 93–98, ACM, 2005.

[24] F. Douglis, "On the role of compression in distributed systems," in *Proc. of ACM SIGOPS, EW 5*, pp. 1–6, 1992.

[25] L. Ayers, "E2compr: Transparent File Compression for Linux ." http://e2compr.sourceforge.net/, June 1997.

[26] J. Bonwick and B. Moore, "ZFS: The Last Word in File Systems." http://opensolaris.org/os/community/zfs/.

[27] "FuseCompress, a mountable Linux file system which transparently compress its content." http://miio.net/wordpress/projects/fusecompress/.

[28] Burrows, M. et al, "On-line data compression in a log-structured file system," in *Proc. of ASPLOS-V*, pp. 2–9, ACM, 1992.

[29] G. V. Cormack, "Data compression on a database system," *Commun. ACM*, vol. 28, no. 12, pp. 1336–1342, 1985.

[30] W. K. Ng and C. V. Ravishankar, "Block-Oriented Compression Techniques for Large Statistical Databases," *IEEE Trans. on Knowl. and Data Eng.*, vol. 9, no. 2, pp. 314–328, 1997.

[31] M. Poess and D. Potapov, "Data Compression in Oracle," in *Proc. 29th VLDB Conference*, 2003.

[32] U. Manber, "Finding similar files in a large file system," in *WTEC'94: Proc. of the USENIX Winter 1994 Technical Conference*, pp. 2–2, USENIX Association, 1994.

[33] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki, "Improving duplicate elimination in storage systems," *Trans. Storage*, vol. 2, no. 4, pp. 424–448, 2006.