Computer Science Department
University of Crete

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

Master's Thesis

Thanos Makatos

February 2010

Heraklion, Greece

University of Crete
Computer Science Department

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

Thesis submitted by
Thanos Makatos
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author:

Thanos Makatos

Committee approvals:

Angelos Bilas
Associate Professor, Thesis Supervisor

Kostas Magoutis
Researcher

Dimitris Nikolopoulos
Professor

Departmental approval:

Panagiotis Trahanias
Professor, Director of Graduate Studies

Heraklion, February 2010

Abstract

In this work we propose a method for transparent compression in the I/O path. 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. Our results show that employing on-line 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.

Supervisor Professor: Angelos Bilas

Περίληψη

Σε αυτή την εργασία προτείνουμε μία μέθοδο για τη διαφανή συμπίεση δεδομένων στο μονοπάτι Ε/Ε. Επεκτείνουμε το στρώμα του μπλοκ του λειτουργικού συστήματος με την δυνατότητα να συμπιέζει και να αποσυμπιέζει δεδομένα καθώς αυτά ρέουν μεταξύ του συστήματος διαχείρισης αρχείων και τον μαγνητικό δίσκο. Η επίτευξη διαφανούς συμπίεσης απαιτεί εκτεταμένη διαχείρηση μετα-πληροφορίας για την διαχέιριση μπλοκ μεταβλητού μεγέθους, τη δυναμική αντιστοίχηση μπλοκ, την δέσμευση μπλοκ, τον ρητό χρονοπρογραμματισμό εργασιών και βελτιστοποιήσεις του συστήματος Ε/Ε για την άμβλυνση των επιπλέων αιτήσεων Ε/Ε και του επιπρόσθετου κόστους της συμπίεσης. Τα αρχικά αποτελέσματα δείχνουν πως η χρήση διαφανούς συμπίεσης σε πραγματικό χρόνο είναι μια εφικτή επιλογή για την βελτίωση του πραγματικού αποθηκευτικού χώρου, μπορεί να επιτύχει βελτιώση της επίδοσης της λειτουργίας του συστήματος Ε/Ε λόγω της μείωσης του όγκου Ε/Ε και της μείωσης της απόστασης αναζήτησης του δίσκου, και έχει αρνητική επίπτωση στην επίδοση μόνον όταν είναι σημαντικός ο χρόνος απόκρισης ενός νήματος.

Επόπτης καθηγητής: Άγγελος Μπίλας

Acknowledgements

I would like to thank my supervisor, Angelos Bilas and my colleauges, Manolis Marazakis, Michail D. Flouris, Stavros Passas and my co-author, Yannis Klonatos. Also, I would like to thank Dimitris Tsaliagos for encouraging me to follow the field of computer science I liked the most. I would like to thank my parents, Vasilis and Ksanthi and my sister, Krysta, for their support and guidance in life. Finally, I would like to thank my friends, Kapravelos Alexandros, Papavasileiou Vicky, Myros Papadakis, Katerina "Koukou" Boutsika and most of all, I would like to thank Evi Dagalaki, for her unreserved support and understanding in all these years. Last but not least, I would like to thank Zira.

Thanos Makatos
Heraklion, February 2009

Contents

List of Figures

List of Tables

Chapter 1

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 [32]. In this work we examine the potential of transparent data compression [20] for improving space efficiency in *on-line* storage systems.

Previously, compression has been mostly applied at the file level [13]. 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. We design, implement, and evaluate 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 *on-line* 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. More-

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

In this work we design, implement, and evaluate a transparent compression system at the block layer for on-line storage. We examine the performance and tradeoffs associated with I/O volume, CPU utilization and metadata I/Os. In contrast to previous approaches, our system is implemented at the block-level, making it possible to use *any* file-system. Moreover, we address a number of issues that arise and are not typical to systems that operate at the block-level.

For our evaluation we use four benchmarks: PostMark, SPEC SFS, TPC-C and TPC-H. Our results show that compression degrades performance by 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 increased CPU utilization, up to 311%. 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 has the potential to increase I/O performance for workloads that exhibit enough I/O concurrency.

The rest of this thesis is organized as follows. Chapter 2 discusses previous and related work. Chapter 3 discusses the design of *ZBD* and how it addresses the associated challenges. Chapter 4 presents our evaluation methodology and your experimental results. Finally, we draw our conclusions in Chapter 5.

Chapter 2

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. 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 this paper, we have exposed the performance-related tradeoffs in the implementation of on-line block-level compression, under a variety of I/O intensive workloads.

Previous research [9, 27, 25, 18] has also argued that online compression of memory (fixed-size) pages can improve memory performance, while the authors in [17] argue that trends in processor and interconnect speeds will favor the use of compression in various distributed and networked systems, even if compression is performed in software. A survey of data compression algorithms appears in [20].

The authors in [13] 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 [10] (an extension for the *ext2* file-system), ZFS [11], 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 [2], 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 [12] 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 [14] 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 [23] 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 [26] 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 [21, 32, 15] 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 [32] 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.

Finally, the only systems that have considered block-level compression are cloop [29] and CBD [30]. 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.

Chapter 3

System Design

ZBD intercepts requests in the I/O path and provides a block device abstraction by exporting a contiguous address space of *logical blocks* to higher system layers, such as file-systems, databases and even virtual block devices, shown in Figure 3.2

In general, compression methods operate as follows. They use a workspace, e.g 256 KB, and compress input into an output buffer. After the necessary workspace initialization, the library proceeds compressing the input data, possibly in successive write operations. Then, the output buffer is finalized and the operation is completed. Decompression follows similar steps.

ZBD uses Lempel-Ziff-Welch (LZW) compression [33, 31, 16] though other compression algorithms can also be used. In principle, the compression scheme may change dynamically depending on block contents. In the current implementation we use two alternative LZW implementations on Linux: *zlib* [16] and *lzo* (variant `LZO1X-1`) [24].

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. Decompression, occurring during reads, is lighter, and directly places the data

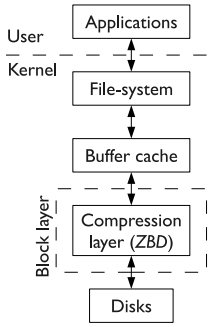


FIGURE 3.1: System Architecture

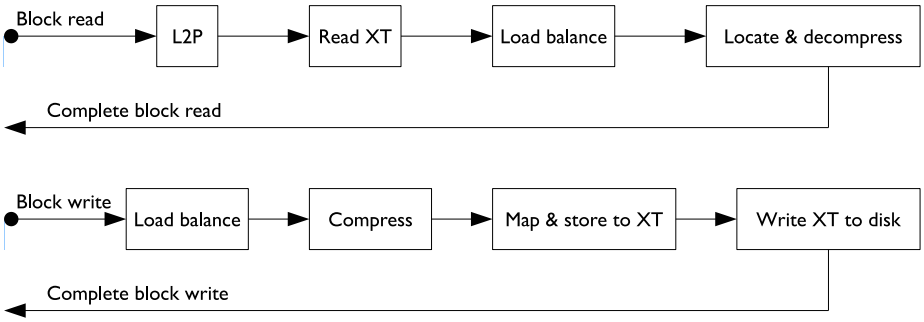


FIGURE 3.2: Read and Write I/O Paths

read into the read-I/O buffer, without requiring a copy of the uncompressed data.

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

3.1 Mapping Logical Blocks to Extents

Compressing fixed-size logical blocks results in variable-size segments. These are stored into fixed-size physical units, called *extents*, which are multiples of block size. *ZBD* uses two mappings to locate both 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 3.3.

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

Compressed blocks stored inside extents have the following structure. A header at the beginning of the extent contains a pointer to the first block as well as a pointer to the free space segment within the extent. All free space in an extent is always contiguous, located at the end of 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 extent. 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 3.3. 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

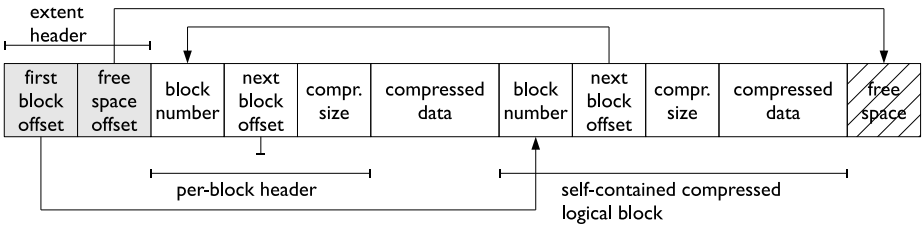


FIGURE 3.3: *ZBD* extent structure. Each compressed block is self-contained, described by a per-block header.

the list takes time proportional to the number of blocks per extent; however, this is not be a problem in practice, as traversal is an in-memory operation. The per-block header (kept inside the extent), along with its mapping and flags (stored at the logical to physical table), are the only metadata required per logical block. Assuming typical space savings of 40% when using compression, a per-block header occupies less than 0.6% of the space the compressed block itself does.

It may occur however, that a block cannot be compressed to a smaller size. In this case, 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.

3.2 Block Allocation and Immutable Updates

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 updates. For this reason we use immutable physical blocks grouped into larger extents, a technique similar to the implementation of log-structured writes [28].

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.

3.3 Extent Buffering

To mitigate the impact on performance of additional I/Os due to the "read-modify-writeback" 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 disk, but also reduces the number of read I/Os, as 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 manager, implemented as a hash table with a collision list and uses 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, the total size of the extents buffered, and when an extent is deemed full are all system 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 the application uses random writes, while smaller ones suffer from internal space 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 it. Second, when writing compressed blocks from concurrent contexts to the extent buffer, it may make sense from a locality perspective to either write blocks to the same or different extents. As explained later, concurrent writes to an extent involve a tradeoff between lock contention and CPU overhead for memory copies. Our experience has shown that preserving locality offers the best performance, albeit at less efficient space utilization. We determined experimentally that buffering a small number of extents is sufficient for preserving locality. Extents are written back as soon as (a) they become full, i.e. there is not enough space for a write context to append blocks and (b) there is no reference to them, i.e. no active read context uses blocks from this extent.

3.4 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 [28]. It scans the physical extents on the disk for full extents using the logical to physical translation mappings and extent headers. It determines which blocks within an extent are "live" by verifying that the mapping of each block (stored inside the block itself) points to that extent. Then, it compacts live blocks into new extents, updates logical to physical translation mappings, and, finally, adds each cleaned extent to the free extent pool. Compaction of live blocks into extents consists of copying these blocks into new extents; no compression/decompression is required. The cleaner is deactivated when the free pool size increases above a threshold. Finally, the only metadata required until the cleaner's next activation is the last scanned extent. This pointer is not required to be persistent; it is merely a hint for the cleaner.

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

space. Without this technique, all live blocks would be relocated in the order they were found during the scan phase, thus increasing free space fragmentation in their new extents. On the other hand, spatial locality suffers, as previously neighboring live blocks may be relocated to different extents. To reduce the impact of this effect, we limit the number of extents the cleaner scans in each iteration to 2 MB. This value is small enough to reduce the negative impact on spatial locality, but large enough to feed the packing algorithm with a range of logical block sizes to be effective.

Placement decisions during cleanup are another important issue. The relative location of logical blocks within an extent is not as important, because extents are read in memory in full. Two issues need to be addressed: (a) which logical blocks should be placed in a specific extent during cleanup; and (b) whether a set of logical blocks that are being compacted will reuse an extent that is currently being compacted or a new extent from the free pool. *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 disk as much as possible.

The log-structured writes 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, e.g. 64 KB chunks, as a single unit and then store the result to a fixed number of blocks. This method would also avoid "read-modify-write" sequences and the impact of delayed space reclamation. However, it fails to support workloads with small-size I/O accesses and poor locality: each random read would require decompressing an *entire* unit of logical blocks in order to retrieve a single logical block.

Overall, we expect that the cleaner will not significantly interfere with application I/Os, as modern storage subsystems typically exhibit idle device times during a typical day of operation. However, we do 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, unless otherwise stated.

3.5 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 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 on the extent. Callback handlers accessing meta-data for a given logical block are synchronized via a per-block lock based on a single bit. Access to the extent buffer when issuing reads/writes for extents are synchronized by a spinlock In addition, callbacks are serialized by a spinlock during the mapping assignment phase, as mapping has to be done atomically in order to preserve locality. Finally, synchronization is required to place or retrieve compressed logical blocks in extents, as we

want to allow multiple readers/multiple writers. Placing a compressed logical block inside an extent is done in three steps using two atomic operations: First, free space in the extent is pre-allocated by updating the free space offset pointer, i.e. an offset inside the extent after which free space begins, with *fetch_and_increment*. Then, the compressed logical block is copied and finally, the last step is to add the block to the list of the other blocks, done using a *compare_and_swap*. 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 should be 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 mapping for space allocation in extents, but proceed in parallel when processing logical blocks.

Besides highly concurrent I/O streams, *ZBD* also leverages large I/Os. In our first *ZBD* design the unit of work is an *entire* I/O request for writes and an *entire* extent for reads. Each unit is dispatched to a worker thread in a round-robin manner and each thread processes its load in FIFO order. To hide the impact of compression on large I/Os, *ZBD* uses multiple 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 cores. This decreases response time for reads and writes and reduces the delay writes may introduce to read processing. Empirically, we find that a global read and a global write work queue is adequate for all *ZBD* threads.

Finally, decompression is performed after the extent has been read in memory and after the I/O read to the disk 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. Using separate threads for issuing I/Os and performing decompression, addresses this problem.

3.5.1 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.

Chapter 4

Experimental Evaluation

4.1 Experimental Platform

We present our evaluation results using a commodity server built using the following components: 8 500-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 4-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, 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.

The algorithms and implementations used for compression and decompression of data are the default `zlib` [33] and `lzo` [31] 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 4.1 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 4.7. 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.

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

Table 4.1: Compression/decompression cost of a 4-KB block.

PostMark PostMark [19] 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 [3], and, moreover, the I/O workload is write-dominated. For these reasons, we use 1 GB of DRAM for PostMark.

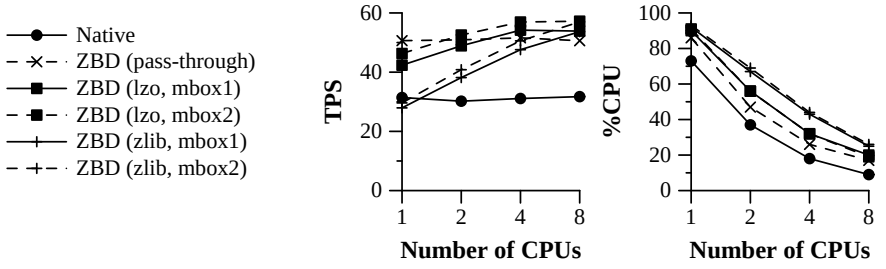
SPECsfs2008 SPEC SFS [4] simulates the operation of an NFSv3/CIFS file-server; our experiments use the CIFS protocol. In SPEC SFS, a *performance target* is set, expressed in operations-per-second. Operations, both read/writes of data-blocks and metadata-related accesses to the file-system, are executed over a file-set generated at benchmark-initialization time. The size of this file-set is proportional to the performance target (≈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 8 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 [5].

TPC-C (DBT-2) DBT-2 [1] 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 [7]. 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.

TPC-H TPC-H [8] 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 an additional 2.5 GB for 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 [6].

Next, we examine the impact of compression on performance and the impact of certain parameters on system behavior.

4.2 PostMark

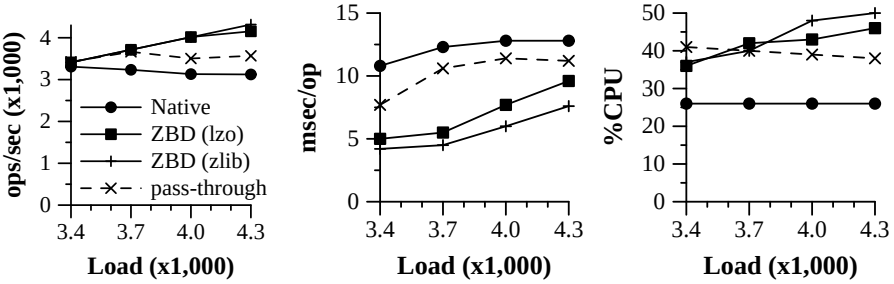


(a) Trans. per sec (TPS) (b) CPU Utilization

FIGURE 4.1: Results for PostMark with variable number of CPUs.

Figure 4.1 shows results for PostMark during the transaction execution phase, with 1, 2, 4 and 8 CPUs, two compression libraries, `lzo` and `zlib`, and two mailboxes, `mbox1` and `mbox2`, where the second one has higher compression ratio than the first one. 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 4.1(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 `mbox2` is used as data generated by Postmark, performance increases by 2% over `mbox1`, 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 improves I/O performance.

4.3 SPEC SFS

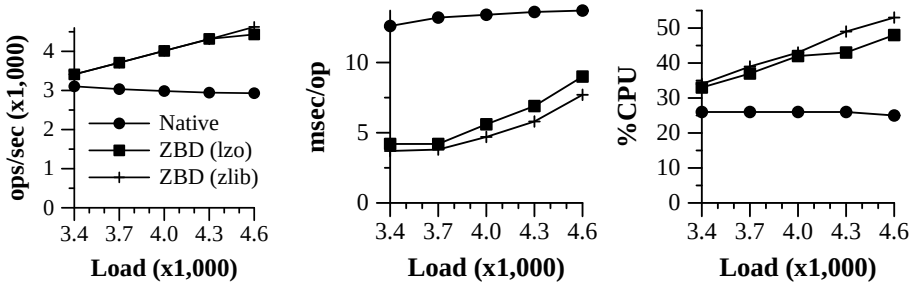


(a) Throughput (b) Latency (c) CPU Utilization

FIGURE 4.2: Results for SPEC SFS.

Figure 4.2 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` but increases CPU utilization by 80% and 90% for `lzo` and `zlib`, respectively.



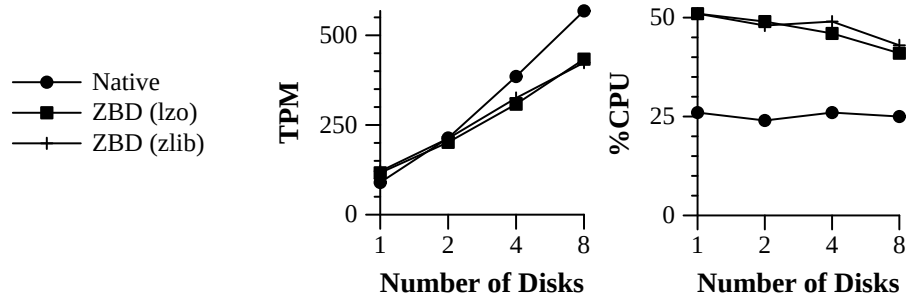
(a) Throughput (b) Latency (c) CPU Utilization

FIGURE 4.3: Results for SPEC SFS using more compressible data.

Figure 4.3 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. Native is unaffected by the change of the file contents.

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 exhibited by compression compensates for the CPU cost with important performance benefits.

4.4 TPC-C (DBT2)

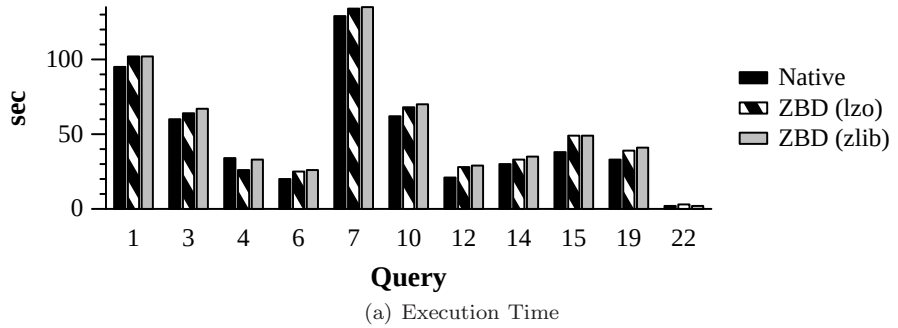


(a) New Order Transactions per min (b) CPU Utilization

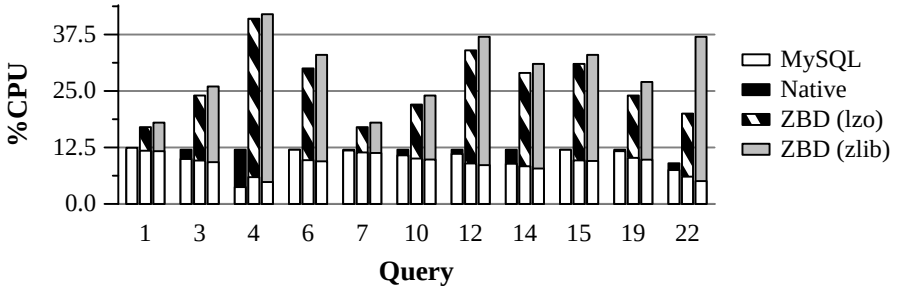
FIGURE 4.4: Results for TPC-C.

Figure 4.4 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 very few outstanding reads, that are small (usually 4 KB) and random. As these reads exhibit poor concurrency, decompression cost is directly exposed to the DBMS. In addition, the fact that they are small and random leads to disproportionally high I/O read volume, as each 4 KB read practically translates to reading a full extent (32 KB in this configuration). As the number of disks decreases, I/O latency significantly increases making decompression latency less important. When using only one disk, performance improves by 29% and 34% for `lzo` and `zlib`, respectively.

4.5 TPC-H



(a) Execution Time

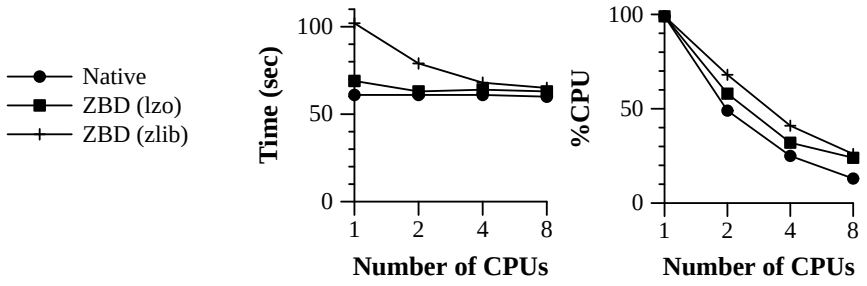


(b) CPU utilization. 100% is the maximum CPU utilization for all eight cores.

FIGURE 4.5: Results for TPC-H.

Figure 4.5 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 4.6 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.



(a) Execution Time (b) CPU Utilization

FIGURE 4.6: Results for TPC-H (Q3) with variable number of CPUs.

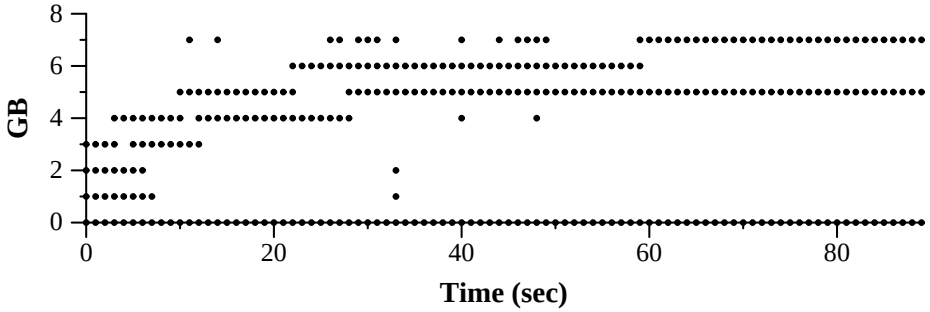
Next, we explore the effect on system performance of data locality, extent size, compression efficiency, cleaning overhead and metadata I/Os.

4.6 Effect of Compression on Spatial Locality

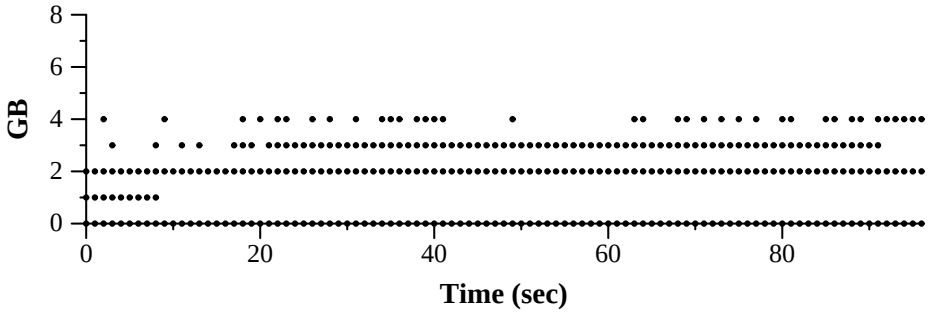
When using transparent compression, there are some less obvious factors that affect performance. As data are kept compressed, disk transfer time is reduced by a factor governed by the compression ratio achieved. Furthermore, the average seek distance is reduced by roughly the same ratio, as data are "compacted" to a smaller area on the disk platter. Figure 4.7 illustrates the access pattern for TPC-H (Q3); *ZBD* exhibits disk accesses that are within a 4 GB zone on the disk, whereas native's accesses are laid on a 6.5 GB zone. This practically 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.

4.7 Extent Size

Figure 4.8 illustrates the impact of the size of the extent on performance. For PostMark, shown in Figure 4.8(a), performance increases with extent size, but starts to decline after 512-KB 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 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 4.8(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 4.8(c)) performs roughly the same for extents between 8 KB and 32 KB, but performance drops for larger extents. TPC-H, shown in Figure 4.8(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.



(a) Native



(b) *ZBD* (`lzo`)

FIGURE 4.7: Disk access pattern for TPC-H (Q3).

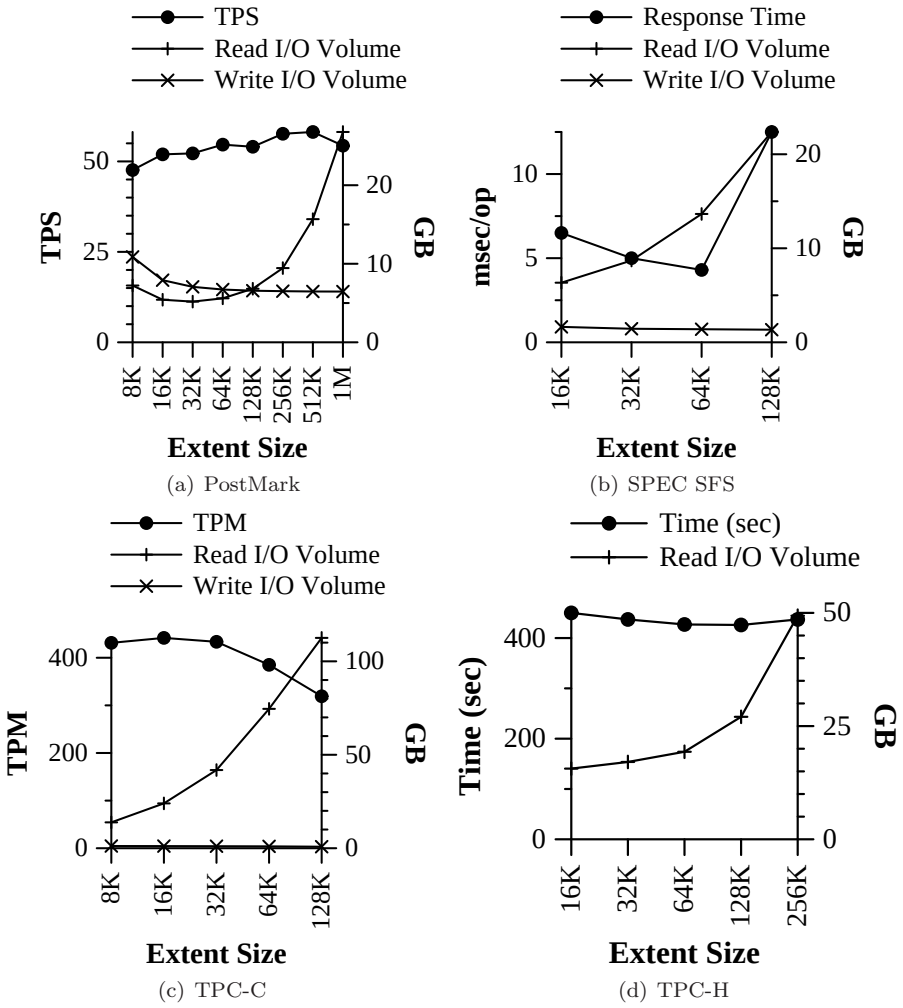
4.8 Compression Efficiency

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 4.2 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`.

4.9 Effect of Cleanup on Performance

Figure 4.9 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



(a) PostMark (b) SPEC SFS

(c) TPC-C (d) TPC-H

FIGURE 4.8: Impact on performance of the extent size.

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.

4.10 Metadata I/Os

Figure 4.10(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

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 4.2: Space savings for various compression methods and file types. *gzip* is used with *-6* (default) in all cases.

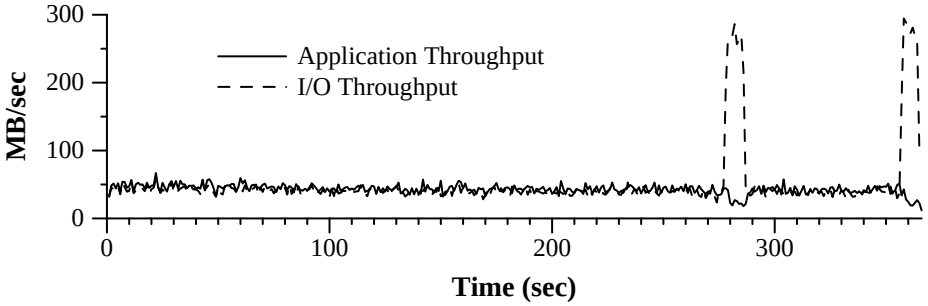
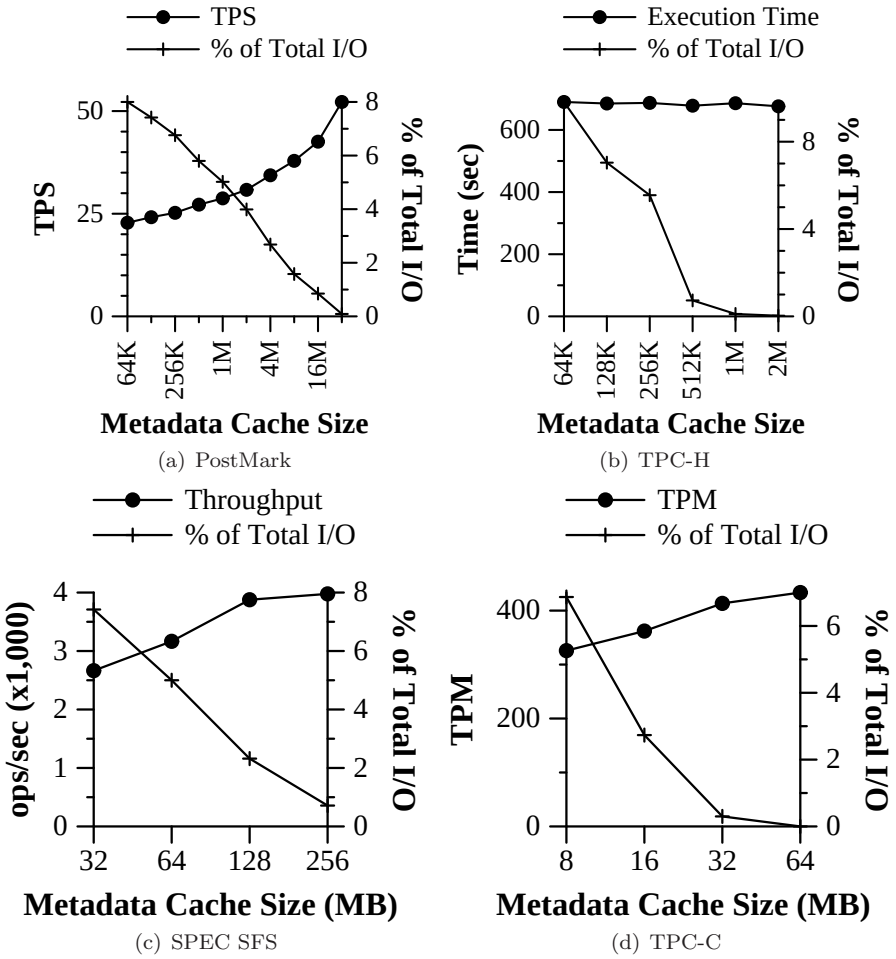


FIGURE 4.9: Impact of cleaner on performance.

I/Os are synchronous, random and interfere with application I/Os. Given the fact 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 4.10(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 4.10(c)) suffers less performance penalty compared to SPEC SFS, up to 33%. Finally, TPC-H performance degrades only slightly, shown in Figure 4.10(d), as the bottleneck is the single CPU the queries can consume.

4.11 Compressed SSD Caching

Performance of storage I/O is an important problem in modern systems. The emergence of flash-based solid state drives (SSDs) has the potential to mitigate I/O penalties: SSDs have low read response times and are not affected by seeks. Additionally, recent SSD drives provide peak throughput that is significantly superior to magnetic hard disk drives. Given the current high cost per gigabyte for SSDs [22], it is important to examine techniques that can increase their cost-efficiency. One approach to achieve this is the use of multi-level cell (MLC)



(a) PostMark (b) TPC-H

(c) SPEC SFS (d) TPC-C

FIGURE 4.10: Impact on performance of metadata cache size.

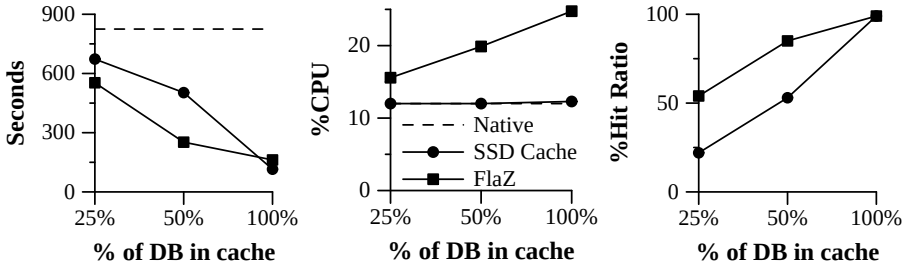
SSDs that store, e.g. two bits per NAND cell. Another approach is to use SSDs as a block cache in the I/O path, reducing read response time for "hot" data.

In this work we employ *ZBD* in a larger system that uses SSDs as compressed cached in the I/O path, called *Flaz*. *Flaz* internally consists of two layers, one that achieves transparent compression (*ZBD*) and one that uses SSDs as an I/O cache. Although these layers are to a large extent independent, in our work we tune their parameters in a combined manner. The caching layer of *Flaz* is a direct-mapped, write-through cache with one block per cache line.

We have implemented a direct-mapped cache because it minimizes metadata requirements and does not impose significant mapping overheads 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. Our *write-through* policy avoids this issue.

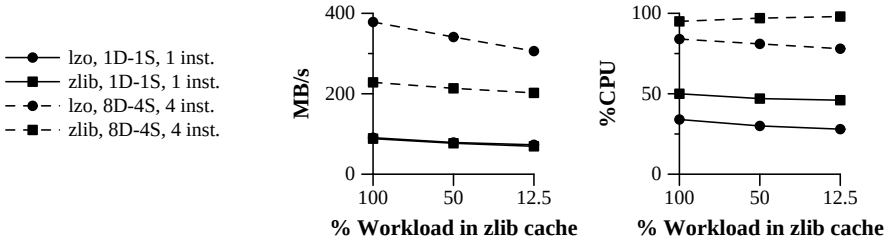
Figure 4.11 shows the performance impact of a compressed SSD cache compared to uncompressed caching and to disk (no SSD caching), using one disk to store the data-base and one SSD as a block-level cache. We use three cache sizes for the SSD cache: large (7 GB), medium (3.5 GB), and small (1.75 GB) that hold approximately 100%, 50%, and 25% of the workload, respectively. Overall, when the workload does not fit in the cache, compression improves performance, whereas performance degrades when the workload entirely fits in the uncompressed cache. As shown in Figure 4.11(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 4.11(b). 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 hits, as illustrated in Figure 4.11(c). On the other hand, compression increases CPU utilization by 29%, 65%, and 101% compared to the uncompressed cache, but it always remains below 25% of the maximum available CPU. With compressed caching, performance increased as a result of increased hit ratio. The average hit time and miss penalty also increases due to decompression and compression, respectively.



(a) Execution Time (b) CPU Utilization (c) Hit Ratio

FIGURE 4.11: Results for TPC-H (Q3) with compressed caching.

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 4.1 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 one disk for storage and one SSDs as a cache, and start with a cache size that marginally fits the workload when using `zlib`. We use the same cache size for `lzo`. We decrease cache size to only fit 50% and 12.5% percent of the workload. In Figure 4.12(a), we see that although `zlib` achieves 30-50% better hit ratio, its CPU cost, shown in Figure 4.12(b), is significantly higher (up to 50%) resulting in only marginal improvement in performance. However, seen from another angle, `zlib` can achieve the same performance as `lzo` using a 30% smaller SSD cache. With four concurrent PostMark instances on eight disks and four SSDs, shown in Figure 4.12(a), `zlib` achieves up to 40% lower throughput than `lzo`, as it becomes CPU limited.



(a) Throughput (b) CPU Utilization

FIGURE 4.12: `lzo` vs. `zlib` performance in PostMark.

4.12 Summary of Results

In this section we present our experimental evaluation, using a commodity storage server. 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 decreased I/O volume, reduced average seek distance and reduced transfer time. 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.

Chapter 5

Conclusions

5.1 Discussion and Future Work

There are two remaining considerations in the design of *ZBD*: (a) how the capacity of a compressed device whose size varies over time is presented to higher layers; and (b) power efficiency.

5.1.1 Variable compressed device size

Space-saving techniques such as compression result in devices whose effective size is data-dependent and varies over time. When a device that supports transparent, online compression is created, most today's operating systems and file-systems need to attach a specific size attribute to it before applications can access it. Although some operating systems may allow resizing a device dynamically, not many file-systems and applications support such functionality. When a device is created in *ZBD*, its nominal size provided to higher layers is a configurable multiple of its actual size. 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. More elaborate policies can be designed in conjunction with capacity planning systems.

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

5.1.2 Power efficiency

Trading CPU cycles for increased storage capacity has power implications as well. By consuming more CPU cycles for compression and decompression, we increase power consumption. On the other hand, compression translates to

smaller I/Os and reduces the amount of devices, improving power consumption. This creates an additional parameter that can be taken into account when trading CPU cycles for I/O performance. However, we believe that it is important to examine this tradeoff alongside offloading compression, e.g. to specialized hardware, and we leave this for future work.

Next, we examine possible extensions to our work.

5.1.3 Differential Compression

Data compression and deduplication is practically the same method for reducing the size a piece of data occupies, however they operate on a radically different data-set size. Compression eliminates redundancy *w*ithin a single block whereas deduplication eliminates redundancy *a*cross blocks. An important observation is that if a set of blocks that exhibits a high degree of similarity is compressed as a single entity, the compression ratio achieved can be significantly high. Based on this observation, a system that first detects similar blocks and then compress them together can be an alternative approach to deduplication. A key benefit of such a system is that it does not rely on hashing, which involves the hazard of collisions, providing higher data protection guarantees.

5.1.4 Compressed Versioning

ZBD uses copy-on-write in order to avoid costly read-modify-write sequences. Data Versioning is a very common feature in copy-on-writes systems, since new data are written elsewhere making older data, usually the most recent, available. Such systems typically require a "pointer" or a mechanism to locate older versions of a piece of data. An important observation of *ZBD* is that since a block is written to an extent in a compressed form, a pointer to the extent containing the previous write of this block can be embedded in the block header of the new write. Retrieving older versions of a block would require reading the latest version of the block (as in the common case of serving a read request) and the traverse the across-extents version list of this block. This mechanism offers implicit snapshots, with *e*ach block write resulting in a new snapshot with no performance cost and minimal capacity cost, about 0.2%.

5.1.5 Data Protection

Similar to versioning, a portion of the extra space saved in the extents by compression can be trivially used to store checksums, thus providing protection against corruption. Again, no performance overhead is introduced by this mechanism, only a 0.1% decrement of space savings.

5.1.6 Improving cost per GB ratio for SSDs

Modern SSDs have a high cost per GB ratio, making them cost ineffective for storing large amounts of data. MLC SSDs store more bits per NAND cell thus increasing their capacity but hurting both performance and NAND flash wear. Transparent compression could be used to reduce the amount of data written on the SSD, thus reducing the negative impact of increased NAND cell density. In this manner, compression could either (a) aid in furtherly increasing NAND cell density in order to improve SSD capacity but without significantly hurting performance, or (b) to maintain the same NAND cell density and extend the longevity of the NAND flash. It is important to note that *ZBD*'s operations are very similar to typical FTL operations, such as garbage collection, out-of-place

updates etc. Moving *ZBD*'s functionality within the SSD firmware is a simple task, assuming hardware accelaration for compression.

In this work we design, implement, and evaluate transparent compression for on-line storage. We examine the performance and tradeoffs associated with I/O volume, CPU utilization and metadata I/Os. Our results show that online transparent compression in 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. In addition, our results indicate that compression has a potential in increasing I/O performance, by up to 80% for PostMark and 35% for SPEC SFS, provided that the workload exhibits enough I/O concurrency to effectively overlap compression with I/O and that the CPU power is enough to accommodate compression.

Bibliography

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

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

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

[4] 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/.

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

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

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

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

[9] A. W. Appel and K. Li. Virtual memory primitives for user programs. *SIGPLAN Not.*, 26(4):96–107, 1991.

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

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

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

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

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

[15] Deepak R. B. et al. Improving duplicate elimination in storage systems. *Trans. Storage*, 2(4):424–448, 2006.

[16] L. P. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification version 3.3. Internet RFC 1950, May 1996.

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

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

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

[20] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Comput. Surv.*, 19(3):261–296, 1987.

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

[22] Narayanan, D. et al. Migrating server storage to SSDs: analysis of tradeoffs. In *Proc. of EuroSys '09*, pages 145–158. ACM, 2009.

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

[24] M. F. X. J. Oberhumer. Lzo – a real-time data compression library. http://www.oberhumer.com/opensource/lzo/.

[25] P. R. Wilson et al. The case for compressed caching in virtual memory systems. In *Proc. of ATEC '99*, pages 8–8. USENIX Association.

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

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

[28] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM TOCS*, 10(1):26–52, Feb. 1992.

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

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

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

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

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