Compression and SSD: Where and How?

Aviad Zuck*, Sivan Toledo, Dmitry Sotnikov, Danny Harnik
Tel Aviv University *IBM Research - Haifa*

Abstract

Compression is widely used in storage systems to reduce the amount of data that is written to physical storage devices, in order to improve both bandwidth and price per GB. In SSDs, which use NAND flash devices, compression also helps to improve endurance, which is limited to a fixed number of raw bytes written to the media, and to reduce garbage-collection overheads.

Compression is typically implemented in one of three layers: the application, the file system or the firmware of the storage device. Our main findings are that compression embedded within the SSD outperforms the built-in host-side compression engines of a well-known database and file systems. Therefore we focus on intra-SSD compression schemes. We investigate the effects of compression granularity and the arrangement of compressed data in NAND flash pages on data reduction and the lifetime of the SSD. We compare several schemes in this design space, some taken from the literature and some new.

1 Introduction

Compression within SSDs is poised to become ubiquitous because of two technological trends. One is the physical characteristics of NAND flash devices. Over time, endurance limits deteriorate and latencies remain essentially constant. The other trend is the continuing improvement in logic gate sizes (Moore's law), which makes computation cheaper overtime. Together these trends make compression a sensible choice for SSDs, potentially bringing several positive effects. First, it decreases the storage footprint of user data. This can increase the exposed capacity of the drive, thereby improving the $/GB ratio and additionally it significantly reduces garbage collection overheads. Second, it decreases the wear of the NAND flash cells that make up the SSD. This not only increases the lifetime of the SSD, but also delays the performance degradation that results from the aging of cells. Finally, it can also reduce IO latency. On the other hand, compression requires additional resources and its benefits may vary substantially depending on the data at hand. Thus employing compression should be done carefully in order to avoid potential pitfalls.

Where to Compress? To fully exploit the potential benefits of compression, we need to decide whether to embed compression functionality within the SSD, as implemented in [1, 2, 3], or in higher layers of the storage hierarchy (in data-storage application like databases [4, 5] or the file system [6, 7, 8]).

One of the main findings of our research is that placing compression low in the storage hierarchy, and in particular placing it in the SSD's firmware (possibly with a hardware accelerator) leads to better results than using the compression at existing file systems or applications. This result is somewhat counter-intuitive, but backed by several consistent experiments on realistic data and systems.

In general, there is a tradeoff at hand, where higher layers in the storage hierarchy have more information about the data but less control over where it is eventually stored. The additional information that higher layers have about data can be used to improve the compression ratios and overall system performance. For example by compressing together related data (e.g., rows from a single database table) or avoiding compression of uncompressible or temporary data.

On the other hand, modern day interfaces between the different layers of the storage hierarchy are based on storing and mapping fixed-sized chunks. Databases and other data-intensive applications pack data into aligned fixed-size pages (e.g., 4KB) that they transfer to/from file systems and block devices. File systems move fixed-size blocks to/from block devices. Block device drivers move fixed-size blocks to/from SSDs and HDDs. However, the introduction of compression breaks this abstraction since compression of fixed-size chunks yields variable sized outputs. The disparity between the fixed-size interfaces and the variable sizes of compressed data can create all sorts of inefficiencies, for example, adding another level of indirection. These problems are mostly avoided once compression is shifted down to the lowest level, i.e. by embedding compression into the SSD, and this is consistent with our observed results.

A different approach is to change the file system or application layers both by modifying their standard data access behaviors and by removing their fixed-sized interface constraints to better suit those of the underlying SSD. However this requires tailoring the file system or application to the intimate characteristics of the underlying storage, for example, to the specific SSD's page size.

How to Embed Compression? Compressing data inside an SSD requires a mapping of uncompressed data to its physical compressed layout. There are sev-

*Supported by the Israeli Ministry of Science, Technology and Space

eral techniques to implement such a mapping that vary mainly in the size of the compressed data chunks and their ordering. In our study we evaluate a number of such techniques, some taken from the literature and some new. The first is a `chunk-based` approach [9] that compresses fixed-size groups of LBAs into a single chunk. Another scheme, which we call `binpacking`, is based on the work of [10]. This scheme accumulates several compressed LBAs and tries to pack them into a single flash page. We extend this approach by intelligently `re-ordering` the compressed LBAs to improve compression effectiveness. The last scheme, called `compaction`, writes compressed LBAs sequentially to flash, leaving no space between them, but allowing a compressed LBA to span two physical pages.

We analyzed these methods not only with respect to their compression benefits, but also by the amount of hardware and system resources required to implement the compression scheme. Our investigation shows that the chunk-based scheme performs the worst, wearing out the device 30-109% faster than the alternative schemes and increasing the storage footprint by 33-54%. The compaction scheme achieves the best storage utilization. The re-ordering scheme is only slightly worse in terms of storage utilization, yet uses up to 30% less RAM; it appears to be the best scheme among the ones we tested. The binpacking scheme is not competitive with the re-ordering and compaction schemes.

Method of evaluation. In our study we focus on an Online Transaction Processing (OLTP) workload for databases. OLTP workloads are ideal for SSDs, because they are characterized by small random accesses and high IO rates. A workload of this type that can be serviced by a single SSD may require using tens or even hundreds of HDDs. In addition, different data types can have very different compressibility and therefore will not achieve the same degree of benefits associated with compression. Database contents have been established as one of the most interesting "compression friendly" data types [3, 11, 12]. Moreover, the integration of compression in a system is especially challenging for small random access IO patterns, making this workload a particularly interesting study case for compression.

Typical benchmarking tools generate either incompressible data (e.g., totally random) or highly compressible data, neither of which are typical of real data. For our testing, we extended the TPC-C workload of OLTP-bench [13], a standard OLTP benchmarking tool, to generate data with variable compression ratios. Our evaluation consists of running the OLTP-bench workloads on MySQL, a popular database management system, which in turn runs on standard file systems. Data was stored to a high-precision SSD emulator [14] that we modified to support the compression schemes described above. We evaluated compression in the SSD emulator, as well as compression at the MySQL level, or compressing at file systems with built-in compression support.

Summary of contributions.

- We built an evaluation environment for exploring the effect of compression on various OLTP workloads with SSDs. For this end we extended the OLTP benchmark and the SSD emulation.
- We explored and evaluated the design space of using compression with SSDs and showed that it is best to embed compression within the SSD, not in the host's file system or in the application layer.
- We expanded the solutions for intra-SSD compression and compared the new methods with existing schemes. In particular, we show that for random access workloads previously suggested schemes fall short in terms of either wear and resource requirements, or compression effectiveness.

Paper organization. Section 2 provides background and related work on compression in SSDs and at the host. Section 3 describes the intra-SSD compression schemes covered in this paper and their implementation. Section 4 describes the methodology and results of our evaluation. Conclusions appear in Section 5.

2 Background

2.1 Compression in SSDs

Several papers have explored the use of compression to improve the lifetime of small single-chip flash devices. zFTL [15] checks whether data is compressible. If not, it writes it as is to flash. Otherwise, it tries to compress as much data as possible into a 4KB buffer. The file system described in [16] also uses a similar approach. Yim et. al [10] have suggested the Internal Packing Scheme (IPS) to improve the lifetime of SmartMedia Cards. IPS maintains several write buffers in battery-backed SRAM. Every write request is compressed, and the resulting data is added to one of the write buffers. The authors describe two main buffering policies, first-fit which puts the data in the first write buffer with enough free space, and best-fit which tries to minimize internal fragmentation in the write buffers.

Others have explored compression in multi-chip SSDs. LSI uses a proprietary technology to support compression in some of its flash controllers [1], but does not provide any details on the compression algorithms, hardware, and data structures being used. Boboila et. al [17] suggested using the on-board cores in SSDs to assist the host CPU in high-performance compression computations. Delta-FTL [18] achieves data reduction through extensive write buffering coupled with selective storing of compressed deltas for a small portion of the

data. We have not implemented this scheme because the results in [18] show that this approach is fairly ineffective for OLTP workloads. Chen et al. also suggested in [19] that intra-SSD compression might be beneficial but did not specify how it should be implemented. Lastly, CaFTL [9] suggested a technique that attempts to improve not only the lifetime but also the performance of SSDs. CaFTL employs a hardware accelerated compression module called BlueZIP. This module includes a 4-page write buffer, which caches recent writes from the host. When the buffer is full, the data is compressed and written as a single chunk to flash. The IPS and CaFTL schemes are described in further detail in Section 3.

2.2 Host-side Compression

Many file systems and databases can compress data on the host. We point to some popular examples in this section. In this paper we evaluate the behavior of compression under a typical application workload, using two well-known file systems and one database, all of which support built-in compression schemes.

Btrfs [6] is a Linux-based file system. Btrfs is considered experimental, but has already been integrated into the mainline kernel. It stores files by dividing them to chunks indexed by B-trees. By default, Btrfs tries to compress the first extent of every file to determine its compressibility. Every extent then serves as a compression block. Since Btrfs writes data to disk in 4K-aligned offsets, padding is used to re-align the chunks. The size of an extent is not fixed and varies according to file access patterns.

ZFS [7] is an established open-source file system, known for its data protection capabilities. It divides files into fixed-size records, and tries to compress them as much as it can to fit into block sizes.

For evaluating compression in databases we have chosen the MySQL relational database, probably the most popular freely-available open-source database system [20]. MySQL's InnoDB engine compresses both data and index chunks using the LZ77 compression algorithm, which eliminates repeated sequences of data. Since data is arranged in B-trees, MySQL stores changes to each page within the tree to a modification log. When the modification log is full, the data is recompressed and possibly split. The size of uncompressed and maximum compressed B-tree pages is configurable. Obviously, to retain a decent level of compression, the B-tree uncompressed page size must be larger than the compressed page size. Typical sizes are 16K for the uncompressed page, and 1K-8K for a compressed page.

Other options for host side compression include: NTFS [8] is a proprietary Windows file system with a compression solution reportedly close to that of ZFS. Oracle's Database also uses compression at the block-level, by saving a symbol table stored within each block, as a dictionary for compressed data [5]. ZBD [21] is a block-layer driver that includes compression to reduce writes to the device. ZBD requires an additional redundant translation layer, and several tens of MB of NV-RAM on the host for buffering.

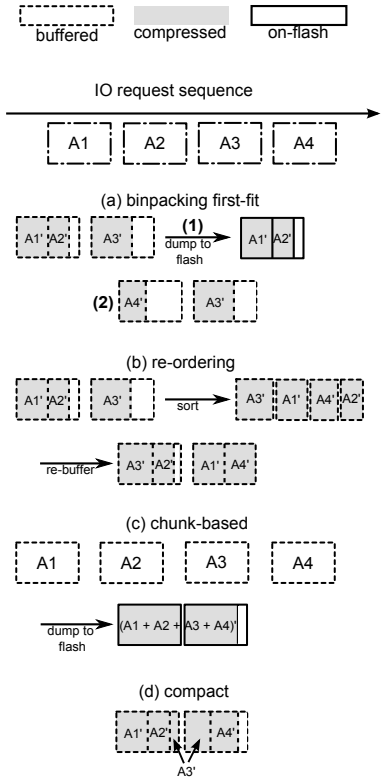

Figure 1: Handling an IO request sequence with four 4KB LBAs A1-A2-A3-A4 under each type of compression scheme. Each LBA compresses to a fraction of its original size 0.6, 0.3, 0.65, 0.4 respectively.

3 Intra-SSD Compression Schemes

To evaluate the effectiveness of hardware-based compression, we compare four possible schemes illustrated in Figure 1.

Chunk-Based Scheme. The `chunk-based` scheme is based on CaFTL [9] described earlier. It optimizes compression by clustering together every n consecutive 4KB blocks of written data from the host into an NV-RAM buffer. CaFTL uses $n = 4$, we also tested $n = 8$. When the buffer is full, its data is compressed to a large chunk, whose size varies between 1 and n flash pages. The chunk is written as a unit to consecutive pages. A write request is acknowledged to the host as soon as the block is written to the NV-RAM buffer, thereby keeping the write latency bounded even if it takes a long time to collect the n blocks that are compressed together. The use of an NV-RAM buffer has additional benefits that are unrelated to compression.

Binpacking. The `binpacking` (or `bp` in short) scheme is based on the IPS method [10]. This scheme also uses a large NV-RAM buffer partitioned into into m page-sized bins (4KB). Whenever the host issues a new write request for a logical block of data, the FTL compresses it. It then tries to fit the newly compressed block of data into one of the bins, using either a first-fit or a best-fit policy. When none of the bins contain enough free space, the fullest bin (i.e. one which contains the least free space) is committed to flash, and the new block of data is inserted to the newly freed bin.

Re-ordering Scheme. We developed the new `reordering` scheme (or `re-bp` in short) as an improvement to binpacking. This scheme uses a much more aggressive binpacking heuristic in order to reduce even further the padding at the end of each bin. The scheme uses the same NV-RAM data structure as binpacking, but when there is no room in the bins for a new compressed block, it first tries to re-bin the blocks. If re-binning creates enough space in some bin for the new block, we avoid flushing a bin at that point. If there is still not enough room, we flush the fullest bin. Re-binning is done in two stages: (1) sort all the compressed blocks in the bins by size, and (2) re-insert blocks to bins by size using the first-fit policy.

Compaction Scheme. The `compaction` scheme is also new. It uses a small page-sized NV-RAM buffer (4KB). It compresses each host-written block separately as soon as it arrives from the host. It copies the compressed block to the end of the NV-RAM buffer. If the buffer overflows during copying, it writes the exactly-full buffer to a flash page, clears the buffer, and copies the end of the compressed block into the beginning of the buffer. This scheme requires a more complex mapping table in the FTL, because a single host-written block may reside on two physical flash pages (which are often but not always consecutive).

3.1 Comparison

The chunk-based, bin-packing, and re-ordering schemes usually leave some unused space at the end of pages containing compressed blocks. We expect the chunk-based scheme to perform most poorly in this respect and the re-ordering scheme to perform the best. The compaction scheme leaves no unused space. Apart from the padding issue, the chunk-based scheme has the best potential for effective compression, because it compresses n blocks at a time whereas the other three compress each block separately. In the chunk-based scheme, reading an LBA (host block) may require reading and decompressing up to n compressed LBAs, possibly spanning more than one physical page. In the bin-packing and re-ordering schemes reading a block requires reading exactly one physical page and decompressing exactly one LBA. In the compaction scheme, reading an LBA requires reading one or two physical pages and decompressing exactly one LBA. All the schemes require an NV-RAM buffer. Compaction requires the smallest buffer (one page). The more complex mapping table of the compaction scheme requires more RAM on the SSD.

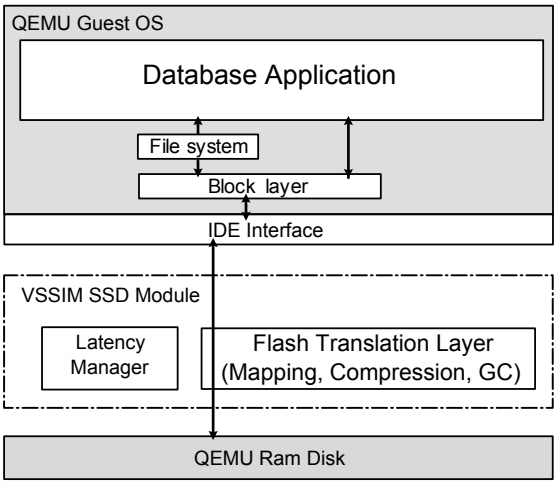

Figure 2: Evaluating a database over an SSD FTL with compression, as part of the VSSIM evaluaition platform

3.2 Implementation

We have implemented all intra-SSD compression schemes using the VSSIM [14] SSD emulator. VSSIM is unique in that it emulates not only an SSD, but also its entire eco-system. VSSIM works as a plugin to a `qemu` [22] virtual machine. It emulates flash response times by storing its virtual disks on faster media (i.e. RAM). A disk that is designated as an SSD uses a software module that implements a flash controller logic (i.e. FTL). The data is saved on RAM, but the software module inserts carefully calculated emulated delays to mimic the response times of flash operations. Figure 2 illustrates the VSSIM architecture used in this paper.

To minimize FTL overhead, we implemented each of our schemes by extending a simple page-mapping FTL, included in the VSSIM distribution. The page mapping FTL was also evaluated without intra-SSD compression as a baseline in our experiments. We also extended the victim block selection policy of the FTL's garbage collection module, to improve wear leveling, by maintaining a threshold of no more than 20% difference between the amount of free blocks in every chip.

4 Evaluation

4.1 Methodology

All experiments are carried out on a machine with an Intel quad-core i7-3770 3.4GHz processor and 16 GB of RAM, running Linux kernel 3.13.0. The qemu guest

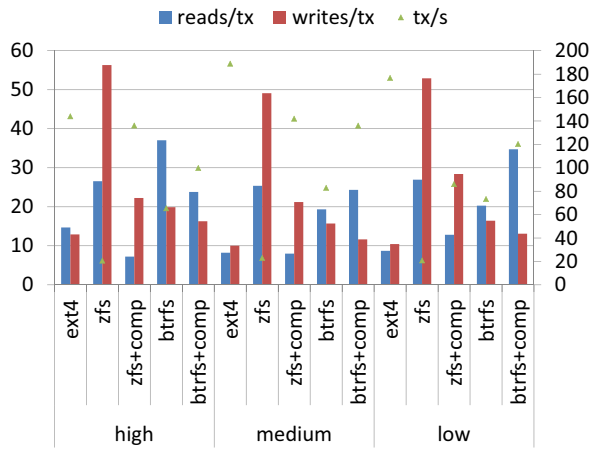

Figure 3: The average flash read/write accesses per transaction during an OLTP benchmark (bars), with varying levels of compressibility under three file systems. The y-axis on the right (triangles) shows the number of transactions/second

machine used had the default setting of VSSIM of Core 2 Duo processor. The guest machine uses 2 virtual disks (stored in RAM), one 6 GB disk is used as the main disk for storing the operating system and application code and data, and another 4 GB disk is used for SSD emulation. The guest is equipped with 1 GB of RAM, and runs Linux kernel 3.8.0. The SSD emulator is configured with 8 flash chips, each residing on a separate channel. The chips are divided into 4 KB pages and 64-page erase blocks. Flash access times for reads and writes are configured to 25 μs and 250 μs respectively. The SSD uses a standard 25% overprovisioning factor, so it exposes a 3 GB capacity to the guest machine. Measurements using the fio [23] benchmark suite show that the device is capable of delivering 7K random 4K IOPs.

We performed the evaluation using TPC-C, a well-known benchmark for OLTP, for several reasons:

- OLTP workloads are dominated by small random accesses. This makes SSDs an ideal candidate for storage media of OLTP data. In fact in recent years a plethora of flash based OLTP oriented solutions [24, 25, 26] have emerged. This development stems from the fact Solid-State Drives (SSDs) provide superior performance for OLTP workloads [27, 28], versus the much slower Hard-Disk Drives (HDDs) which were the mainstay for database storage until recently.
- The benefits of compression only materialize if user data is compressible. Database content is typically highly compressible [3]. We also verified this observation on a 10GB real-life database, which contains tables created by Computer Science university students for a variety of projects. Standard compression libraries achieve saving of 76% for this database (it compressed to 24% of its original size).

All experiments were run using the TPC-C benchmark of the OLTP-bench framework [13]. However, this benchmarking tool, and others as well [29] store random data in most fields. Our initial experiments showed that a database created using the default data generator compresses by only 13%, which leaves little room for optimizations, and does not match known compression properties of OLTP databases [3]. We therefore made a small modification to decrease the randomness of data fields generated by the workload. By default most values are randomly generated strings and numbers. We varied the fraction of truly random characters and digits in every string/number. This resulted in 3 different "optimal" levels of compressibility for the entire database data, of 82%, 70% and 63% which we refer to as HighComp, MedComp and LowComp respectively. These levels span the range of compressibility results on many databases, including on our own 10 GB real-life database.

We used three file systems in our evaluation: ext4 (which does not support compression), ZFS and Btrfs (both of which do support compression). All file systems were configured with a 4KB block size to match the SSD's write-unit size. We configured ZFS to use a smaller 16KB record size, which is more suitable for small random access workloads, and proved the most stable in our tests.

All file systems were tested under MySQL 5.6.16 as the database for a TPC-C workload. To improve performance we made two modifications to MySQL's default configuration. First we configured the InnoDB storage engine to use 4 KB disk page size, instead of the default 16 KB page size. The larger disk page size is an optimization intended for HDDs, and is not recommended for OLTP workloads on SSDs, where random access is much faster [30]. Our tests proved that using the default page size indeed deteriorated performance.

For evaluating intra-SSD compression schemes we also configured the storage engine to use the SSD as a raw partition, instead of using data and index files over a mounted file system. This removes any file system overhead, and allows us to stress the device and minimize the effects of the buffer cache. To minimize the effect of compression on our evaluation, all schemes employ a fast LZ4 compression algorithm for compressing data.

Each test begins with an initial priming phase of creating a clean database over a fresh SSD, and loading of 20 warehouses to the database. We then run a TPC-C workload with 32 users. We configure the workloads to a maximum rate of 500 transactions per second, and run the test for 30 minutes. The throughputs we measured were always lower than this maximum.

Our main evaluation metric is the amount of flash accesses per workload. We average these results per trans-

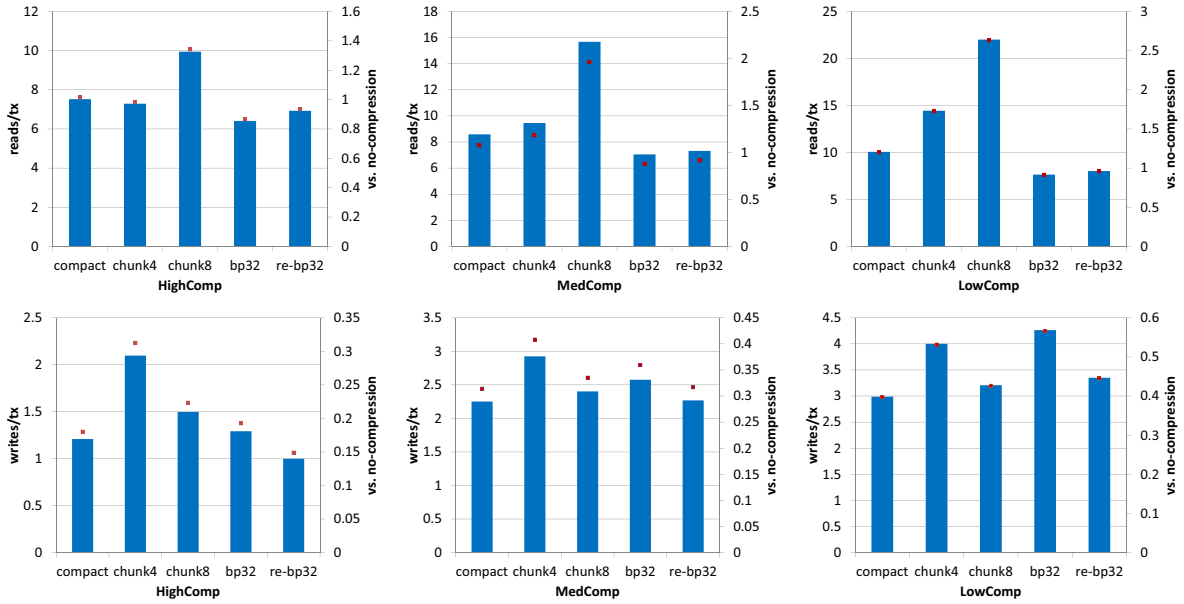

Figure 4: The average flash read/write accesses per transaction during an OLTP benchmark (bars), with varying compressibility levels. The y-axis on the right (dots) shows the relative number of accesses versus a simple page-mapping FTL with no compression

action to compare the effectiveness of compression.

4.2 Results

In this section we present the results of our experiments. We first present the results of flash accesses generated on average by each transaction, under the three evaluated file systems, with and without built-in compression. Next we evaluate every intra-SSD compression scheme and compressibility level. We then continue to present the average decompression work which resulted from every transaction, and analyze its implications for compressor hardware in SSDs.

We also measured the effectiveness of MySQL's built-in compression module (of the InnoDB storage engine). We configured it with a 16K page size, and 4K compression chunk size (key_block_size=4 configuration). Versus a configuration with no compression, the results show a 30-50% decrease in transactions per second. Furthermore, even under the HighComp workload it performed twice as more writes to flash than the simple page-mapping FTL with no compression under ext4, and also worse than both ZFS and Btrfs. The results became significantly worse as we reduced the compressibility of data. This surprising result stems from the fact MySQL uses in-page logs to store compressed data. This causes a read-modify-write behavior, and also means that a significant portion of a B-tree page contains only padding. These results were disappointing, so due to lack of space we decided not to include them in this paper.

4.2.1 Reads and Writes

Figure 3 shows the flash accesses generated on average by an OLTP transaction under three file systems. In this experiment compression was turned off in the SSD and in MySQL. For each file system we repeated the experiment with built-in compression turned on, whenever available. In all experiments a simple page-mapping FTL was used. The results show that using file-system compression can significantly improve performance. It yields up to 6x and 64% more transactions per second for ZFS and Btrfs respectively. This can be explained by the fact compression significantly reduces the amount of used space in the device. Therefore the task of free space management, usually done in the background, can greatly affect performance and response times. Compression is clearly much more effective for ZFS than Btrfs. However, even with compression ZFS and Btrfs perform more poorly than ext4 which does not perform compression. This can be explained by the fact that file system compression engines aim to compress large chunks of data. However, compressing in large units makes them susceptible to small random accesses (which is the common workload for SSDs).

Figure 4 shows the flash accesses generated on average by an OLTP transaction, for every type of compression scheme: compaction, chunk-based with 4 or 8 pages per chunk, binpacking with and without reordering, and simple page mapping without compression. In this experiment compression in MySQL was turned off, and MySQL stored data on the raw device. For the binpacking and re-ordering schemes we present the best-performing configuration, with 32 bins

(128 KB). All workloads resulted in 190-230 Tps (transactions per second). The major factor affecting this figure was the compressibility of data being used. For comparison, moving the database backend storage to an HDD instead of an SSD emulator based on a RAM disk, resulted in a rate of about 10 Tps, meaning that even for our limited setup reaching the same level of Tps would require using 20 HDDs.

For read performance, the chunk-based scheme delivers the worst performance. With a 4-page chunk it requires 13-88% more read accesses than the best performing scheme on all workloads, which is binpacking with 32 bins. Using an 8-page chunk almost doubles the amount of reads, as expected. This validates the significance of small random read accesses in OLTP workloads. Under such workloads, the chunk-based scheme suffers from high read amplification, due to reading large chunks which span multiple pages, in-order to extract data for a single LBA.

For reads, the binpacking scheme performs best, since random accesses require at most a single page read. The compaction scheme on the other hand may require reading 2 pages to serve a single logical read, which is manifested in 16% more reads. For LowComp data this is exacerbated, since the chances of data of a compressed logical page to overflow to the next physical page are increased, and the compaction scheme now performs 31% worse. The reordering scheme performs worse by 5%. This is due to the loss of read locality when reordering all bins before deciding whether to dump a victim bin to flash. The page mapping scheme delivers on-par read performance, since it also does not require redundant accesses to serve small random reads, but this comes of course at the cost of significant write amplification.

For writes the compaction scheme demonstrates superior performance for both MedComp and LowComp workloads, though the re-ordering scheme does not lag far behind. For the compaction scheme this comes at the cost of using 30% more RAM, for storing extra mapping entries in the translation map. In the HighComp workload, the compaction scheme is out-performed by the reordering scheme by 16%. This is not only because of improved space utilization, but also because the bins in effect function as a small write buffer; For highly-compressible data they can cache more LBAs. For larger SSDs the write buffering effect due to such a small buffer in NV-RAM would probably be much smaller.

Not surprisingly, the chunk-based scheme performed better with larger chunks, but still not as good as the compaction scheme. When using a 4-page chunk it performed 30-109% worse than the compaction scheme. For the re-ordering binpacking scheme, reducing the number of bins has a negative effect on writes, as expected. In the worst performing configuration using only

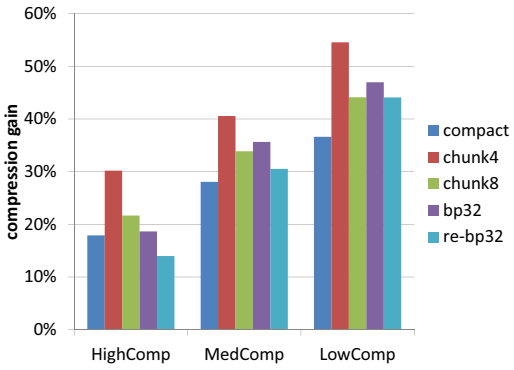

Figure 5: Compression gain achieved for every intra-SSD compression scheme, under the OLTP benchmark with varying levels of compressibility

4 bins, it now performs 3%, 7% and 23% more writes to flash than the 32-bin configuration (for the LowComp, MedComp and HighComp workloads respectively).

The results for the simple page-mapping FTL are displayed on the secondary Y axis. They validated that it performed more writes than all compression schemes in proportion to the compressibility level of each workload.

We note that the benchmark does not fully utilize the high IOPs rate the device is capable of. This is consistent with measurements of OLTP benchmarks on real-life SSDs [27, 28] capable of 100Ks of IOPs; the OLTP benchmarks only reached rates of thousands of transactions per second on these devices.

The results described so far indicate that **intra-SSD compression is superior to host-side compression**. It performs significantly fewer read and write accesses for the same workload. This improves the life span of the device, and also results in better rates of transactions per second in almost all workloads. Therefore we proceed to evaluate the storage footprint of each intra-SSD compression scheme.

4.2.2 Intra-SSD Storage Footprint

To further evaluate the effective compression gained by each intra-SSD compression scheme, under each workload, we measure the ratio between the number of bytes written to flash and the number of bytes written to the SSD by the host. The results are displayed in Figure 5. Again, for highly-compressible data, the reordering scheme achieves the best results, because of its larger write buffer. The chunk-based scheme always performs the worst when using 4 pages per chunk, using 33-54% more space than the other schemes. The numbers improve, as expected, as we increase the amount of data fed to the chunk-based compressor, but as we've seen this comes at a heavy cost of performing more reads (and as we see next, also decompression).

The potential benefit of the binpacking scheme (with

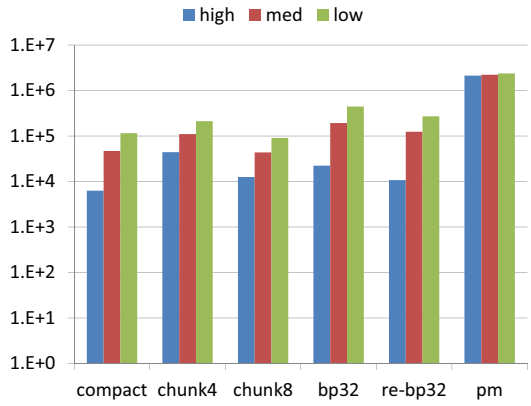

Figure 6: GC-related writes as a result of one hour long TPC-C workload for each scheme

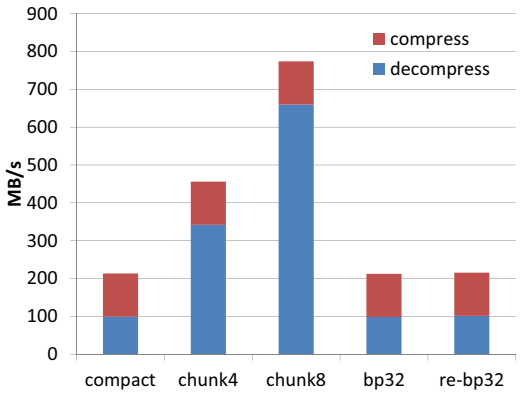

Figure 7: Compression/decompression rates required by each scheme, based on Tps rate extracted from real-life SSDs

and without re-ordering) dramatically decreases when it has to pack fewer and larger chunks of data to the same number of bins. This also explains why the compaction scheme performs better than the re-ordering scheme, as the data becomes less compressible. When using only 4 bins, the re-ordering scheme deteriorates by 22% in compression gain versus the 32-bin configuration in the HighComp workload, but much less dramatically for the rest of the workloads. We have also verified the results using an offline analysis of each intra-SSD compression scheme on a real-life 10GB database. The results are omitted due to lack of space.

Lastly, we note that each compression level results in different levels of valid data stored on the device, and therefore different levels of stress on the garbage collection process. For compressible data, embedding compression within the device increases the effective over-proviosining factor; as a result the reclamation process requires reading and writing very little data. For example when the data is 82% compressible, much less data is effectively written to flash. Figure 6 demonstrates this effect for one-hour long runs of the TPC-C benchmark.

4.2.3 Hardware Requirements

In our evaluation so far we assumed that the compression and decompression resources for intra-SSD schemes are unlimited. However, in real-life SSDs, the hardware resources dedicated for compression are of course limited in terms of cost, power, and size. Requiring less from the compression hardware used by the SSD would allow vendors to use weaker, cheaper and more energy-efficient hardware without creating a bottleneck.

Most of our OLTP workloads using intra-SSD compression delivered a rate of about 200 transactions per second. Even in the least-performing scheme, i.e. the chunk-based scheme with the 8-page configuration, this translates to throughput of 7 MB/s compressed and 40 MB/sec decompressed data. These numbers are low, and can be supported even with relatively weak compression hardware. On several high-end SSDs, transaction rates of up to 3K per second have been reported [27, 28]. Extrapolating our results to such SSDs would require the compression/decompression rates shown in Figure 7. Supporting these rates would require relatively powerful compression hardware.

We can see that for the chunk-based schemes, the decompression requirements are 342-660 MB/s, 3x-6x higher than the other schemes. The requirements for compression throughput are the same for all techniques, but these figures do not take into account garbage collection writes. These would add additional penalty in the case of the chunk-based scheme, which re-compresses data during garbage collection. High-end compression hardware can probably meet even these relatively high requirements; low-end compressors appear to be too weak [31, 18, 32]. Therefore, using the compaction or re-ordering scheme would be preferable, allowing SSD vendors to use cheaper compression hardware.

5 Conclusions

Our paper explores mechanisms and policies for compressing data over SSDs. Using a representative workload, we show that intra-SSD compression is superior to using existing application or file system level compression. Flash does not allow straightforward write in-place and therefore almost all SSD firmwares keep some form of large translation maps in RAM on the device. Embedding compression within the SSD avoids the need for an additional level of indirection by utilizing the SSD's existing translation maps. In addition having the compression at this low level sidesteps most of the problems related to variable and fixed chunk sizes.

We evaluate several methods for embedding compression in SSDs and suggest improvements over the existing methods.

References

[1] LSI durawrite data reduction. http://www.lsi.com/company/technology/duraclass/pages/durawrite.aspx.

[2] Data compression – intel. www.intel.com/support/ssdc/hpssd/sb/CS-034293.htm.

[3] Jon Tate, Bosmat Tuv-El, Jorge Quintal, Eyal Traitel, Barry Whyte, et al. *Real-time Compression in SAN Volume Controller and Storwize*, volume 7000. IBM Redbooks, 2012.

[4] MySQL InnoDB compressed tables. https://dev.mysql.com/doc/refman/5.6/en/innodb-compression.html.

[5] Oracle advanced compression with Oracle database 12c. http://www.oracle.com/technetwork/database/options/compression/advanced-compression-wp-12c-1896128.pdf.

[6] Btrfs compression. https://btrfs.wiki.kernel.org/index.php/Compression.

[7] OpenZFS project homepage. http://open-zfs.org/wiki/Main_Page.

[8] NTFS file compression and decompression. http://msdn.microsoft.com/en-us/library/windows/desktop/aa364219

[9] Sungjin Lee, Jihoon Park, K. Fleming, Arvind, and Jihong Kim. Improving performance and lifetime of solid-state drives using hardware-accelerated compression. *Consumer Electronics, IEEE Transactions on*, 57(4):1732–1739, November 2011.

[10] Keun Soo Yim, Hyokyung Bahn, and Kern Koh. A flash compression layer for smartmedia card systems. *Consumer Electronics, IEEE Transactions on*, 50(1):192–197, Feb 2004.

[11] Cornel Constantinescu, Joseph S. Glider, and David D. Chambliss. Mixing deduplication and compression on active data sets. In *Data Compression Conference (DCC 2011)*, pages 393–402, 2011.

[12] D. Harnik, E. Khaitzin, D. Sotnikov, and S. Taharlev. A Fast Implementation of Deflate. In *DCC*. IEEE Computer Society, 2014.

[13] OLTP-bench benchmarking tool. http://oltpbenchmark.com.

[14] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choi, Sungroh Yoon, and Jaehyuk Cha. VSSIM: Virtual machine based ssd simulator. In *MSST*, 2013.

[15] Youngjo Park and Jin-Soo Kim. zftl: power-efficient data compression support for nand flash-based consumer electronics devices. *Consumer Electronics, IEEE Transactions on*, 57(3):1148–1156, August 2011.

[16] Yangwook Kang and Ethan L. Miller. Adding aggressive error correction to a high-performance compressing flash file system. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, 2009.

[17] S. Boboila, Youngjae Kim, S.S. Vazhkudai, P. Desnoyers, and G.M. Shipman. Active flash: Out-of-core data analytics on flash storage. In *Mass Storage Systems and Technologies (MSST)*, 2012.

[18] Guanying Wu and Xubin He. Delta-ftl: Improving ssd lifetime via exploiting content locality. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, 2012.

[19] Feng Chen, Tian Luo, and Xiaodong Zhang. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[20] DB-engines database management systems popularity ranking. http://db-engines.com/en/ranking.

[21] T. Makatos, Y. Klonatos, M. Marazakis, M.D. Flouris, and A. Bilas. Zbd: Using transparent compression at the block level to increase storage space efficiency. In *Storage Network Architecture and Parallel I/Os (SNAPI), 2010 International Workshop on*, pages 61–70, May 2010.

[22] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.

[23] fio bencmarking tool. https://github.com/axboe/fio.

[24] ZD-XL SQL accelerator. http://ocz.com/enterprise/zd-xl-sql-accelerator-pcie-ssd.

[25] IBM Netezza data warehouse appliances. http://www-01.ibm.com/software/data/netezza/.

[26] A. De, M. Gokhale, R. Gupta, and S. Swanson. Minerva: Accelerating data analysis in next-generation ssds. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 9–16, April 2013.

[27] Sysbench OLTP benchmark. http://www.storagereview.com.

[28] Virident flashmax m1400 mysql - tpcc-mysql report. http://www.percona.com/files/white-papers/virident-mlc-tpcc.pdf.

[29] tpcc-mysql bencchmarking tool. https://code.launchpad.net/ percona-dev/perconatools/tpcc-mysql.

[30] Ilia Petrov, Robert Gottstein, Todor Ivanov, Daniel Bausch, and Alejandro Buchmann. Page size selection for oltp databases on ssd storage. *Journal of Information and Data Management*, 2(1):11, 2011.

[31] J.L. Nunez and S. Jones. Gbit/s lossless data compression hardware. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11(3):499–510, June 2003.

[32] S. Navqi, R. Naqvi, R.A. Riaz, and F Siddiqui. Optimized rtl design and implementation of lzw algorithm for high bandwidth applications. *Electrical Review*, 2011(4):279âĂŞ285, April 2011.