Migratory Compression: Coarse-grained Data Reordering to Improve Compressibility

Xing Lin, *University of Utah;* Guanlin Lu, Fred Douglis, Philip Shilane, and Grant Wallace, *EMC Corporation— Data Protection and Availability Division*

https://www.usenix.org/conference/fast14/technical-sessions/presentation/lin

This paper is included in the Proceedings of the
12th USENIX Conference on File and Storage Technologies (FAST '14).
February 17–20, 2014 • Santa Clara, CA USA
ISBN 978-1-931971-08-9

Migratory Compression: Coarse-grained Data Reordering to Improve Compressibility

Xing Lin[1], Guanlin Lu[2], Fred Douglis[2], Philip Shilane[2], Grant Wallace[2]
[1]*University of Utah*, [2]*EMC Corporation – Data Protection and Availability Division*

Abstract

We propose *Migratory Compression* (MC), a coarse-grained data transformation, to improve the effectiveness of traditional compressors in modern storage systems. In MC, similar data chunks are re-located together, to improve compression factors. After decompression, migrated chunks return to their previous locations. We evaluate the compression effectiveness and overhead of MC, explore reorganization approaches on a variety of datasets, and present a prototype implementation of MC in a commercial deduplicating file system. We also compare MC to the more established technique of delta compression, which is significantly more complex to implement within file systems.

We find that Migratory Compression improves compression effectiveness compared to traditional compressors, by 11% to 105%, with relatively low impact on runtime performance. Frequently, adding MC to a relatively fast compressor like gzip results in compression that is more effective in both space and runtime than slower alternatives. In archival migration, MC improves gzip compression by 44–157%. Most importantly, MC can be implemented in broadly used, modern file systems.

1 Introduction

Compression is a class of data transformation techniques to represent information with fewer bits than its original form, by exploiting statistical redundancy. It is widely used in the storage hierarchy, such as compressed memory [24], compressed SSD caches [15], file systems [4] and backup storage systems [28]. Generally, there is a tradeoff between computation and compressibility: often much of the available compression in a dataset can be achieved with a small amount of computation, but more extensive computation (and memory) can result in better data reduction [7].

There are various methods to improve compressibility, which largely can be categorized as increasing the *look-back* window and *reordering data*. Most compression techniques find redundant strings within a window of data; the larger the window size, the greater the opportunity to find redundant strings, leading to better compression. However, to limit the overhead in finding redundancy, most real-world implementations use small window sizes. For example, DEFLATE, used by `gzip`, has a 64 KB sliding window [6] and the maximum window for `bzip2` is 900 KB [8]. The only compression algorithm we are aware of that uses larger window sizes is LZMA in `7z` [1], which supports history up to 1 GB.[1] It usually compresses better than `gzip` and `bzip2` but takes significantly longer. Some other compression tools such as `rzip` [23] find identical sequences over a long distance by computing hashes over fixed-sized blocks and then rolling hashes over blocks of that size throughout the file; this effectively does intra-file deduplication but cannot take advantage of small interspersed changes. Delta compression (DC) [9] can find small differences in similar locations between two highly similar files. While this enables highly efficient compression between similar files, it cannot delta-encode widely dispersed regions in a large file or set of files without targeted pair-wise matching of similar content [12].

Data reordering is another way to improve compression: since compression algorithms work by identifying repeated strings, one can improve compression by grouping similar characters together. The Burrows-Wheeler Transform (BWT) [5] is one such example that works on relatively small blocks of data: it permutes the order of the characters in a block, and if there are substrings that appear often, the transformed string will have single characters repeat in a row. BWT is interesting because the operation to invert the transformed block to obtain the original data requires only that an index be stored with the transformed data and that the transformed data be sorted lexicographically to use the index to identify the original contents. `bzip2` uses BWT as the second layer in its compression stack.

[1]The specification of LZMA supports windows up to 4 GB, but we have not found a practical implementation for Linux that supports more than 1 GB and use that number henceforth. One alternative compressor, xz [27], supports a window of 1.5 GB, but we found its decrease in throughput highly disproportionate to its increase in compression.

What we propose is, in a sense, a coarse-grained BWT over a large range (typically tens of GBs or more). We call it *Migratory Compression* (MC) because it tries to rearrange data to make it more compressible, while providing a mechanism to reverse the transformation after decompression. Unlike BWT, however, the unit of movement is kilobytes rather than characters and the scope of movement is an entire file or group of files. Also, the recipe to reconstruct the original data is a nontrivial size, though still only ~0.2% of the original file.

With MC, data is first partitioned into chunks. Then we 'sort' chunks so that similar chunks are grouped and located together. Duplicated chunks are removed and only the first appearance of that copy is stored. Standard compressors are then able to find repeated strings across adjacent chunks.[2] Thus MC is a *preprocessor* that can be combined with arbitrary adaptive lossless compressors such as `gzip`, `bzip2`, or `7z`; if someone invented a better compressor, MC could be integrated with it via simple scripting. We find that MC improves `gzip` by up to a factor of two on datasets with high rates of similarity (including duplicate content), usually with better performance. Frequently `gzip` with MC compresses both better and faster than other off-the-shelf compressors like `bzip2` and `7z` at their default levels.

We consider two principal use cases of MC:

mzip is a term for using MC to compress a single file. With `mzip`, we extract the resemblance information, cluster similar data, reorder data in the file, and compress the reordered file using an off-the-shelf compressor. The compressed file contains the recipe needed to restore the original contents after decompression. The bulk of our evaluation is in the context of stand-alone file compression; henceforth `mzip` refers to integrating MC with traditional compressors (gzip by default unless stated otherwise).

Archival involves data migration from backup storage systems to archive tiers, or data stored directly in an archive system such as Amazon Glacier [25]. Such data are cold and rarely read, so the penalty resulting from distributing a file across a storage system may be acceptable. We have prototyped MC in the context of the archival tier of the Data Domain File System (DDFS) [28].

There are two runtime overheads for MC. One is to detect similar chunks: this requires a preprocessing stage to compute *similarity features* for each chunk, followed by clustering chunks that share these features. The other overhead comes from the large number of I/Os necessary to reorganize the original data, first when performing compression and later to transform the uncompressed output back to its original contents. We quantify the effectiveness of using fixed-size or variable-size chunks, three chunk sizes (2 KB, 8 KB and 32 KB), and different numbers of features, which trade compression against runtime overhead. For the data movement overhead, we evaluate several approaches as well as the relative performance of hard disks and solid state storage.

In summary, our work makes the following contributions. First, we propose *Migratory Compression*, a new data transformation algorithm. Second, we evaluate its effectiveness with real-world datasets, quantify the overheads introduced and evaluate three data reorganization approaches with both HDDs and SSDs. Third, we compare `mzip` with DC and show that these two techniques are comparable, though with different implementation characteristics. Last, we demonstrate its effectiveness with an extensive evaluation of Migratory Compression during archival within a deduplicating storage system, DDFS.

2 Alternatives

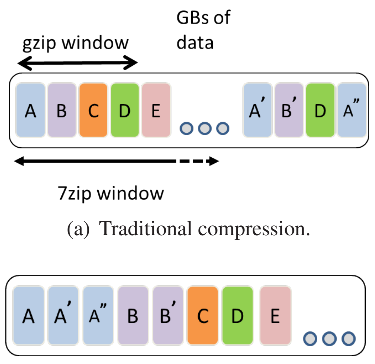
One goal of any compressor is to distill data into a minimal representation. Another is to perform this transformation with minimal resources (computation, memory, and I/O). These two goals are largely conflicting, in that additional resources typically result in better compression, though frequently with diminishing returns [7]. Here we consider two alternatives to spending extra resources for better compression: moving similar data together and delta-compressing similar data in place.

2.1 Migratory v Traditional Compression

Figure 1 compares traditional compression and Migratory Compression. The blue chunks at the end of the file (*A*' and *A*") are similar to the blue chunk at the start (*A*), but they have small changes that keep them from being entirely identical. With (a) traditional compression, there is a limited window over which the compressor will look for similar content, so *A*' and *A*" later in the file don't get compressed relative to *A*. With (b) MC, we move these chunks to be together, followed by two more similar chunks *B* and *B*'. Note that the two green chunks labeled *D* are identical rather than merely similar, so the second is replaced by a reference to the first.

One question is whether we could simply obtain extra compression by increasing the window size of a standard compressor. We see later (Section 5.4.4) that the "maximal" setting for `7z`, which uses a 1 GB lookback

[2]It is possible for an adaptive compressor's history to be smaller than size of two chunks, in which case it will not be able to take advantage of these adjacent chunks. For instance, if the chunks were 64 KB, gzip would not match the start of one chunk against the start of the next chunk. By making the chunk size small relative to the compressor's window size, we avoid such issues.



(a) Traditional compression.

(b) Migratory Compression.

Figure 1: Compression alternatives. With MC similar data moves close enough together to be identified as redundant, using the same compression window.

window (and a memory footprint over 10 GB) and substantial computation, often results in worse compression with poorer throughput than the default `7z` setting integrated with MC.

2.2 Migratory v Delta Compression

Another obvious question is how MC compares to a similar technology, *delta compression* (DC) [9]. The premise of DC is to encode an object A' relative to a similar object A, and it is effectively the same as compressing A, discarding the output of that compression, and using the compressor state to continue compressing A'. Anything in A' that repeats content in A is replaced by a reference to its location in A, and content within A' that repeats previous content in A' can also be replaced with a reference.

When comparing MC and DC, there are striking similarities because both can use features to identify similar chunks. These features are compact (64 bytes per 8 KB chunk by default), allowing GBs or even TBs of data to be efficiently searched for similar chunks. Both techniques improve compression by taking advantage of redundancy between similar chunks: MC reads the chunks and writes them consecutively to aid standard compressors, while DC reads two similar chunks and encodes one relative to the other. We see in Section 5.3 that MC generally improves compression and has faster performance than intra-file DC, but these differences are rather small and could be related to internal implementation details.

One area where MC is clearly superior to DC is in its simplicity, which makes it compatible with numerous compressors and eases integration with storage systems. Within a storage system, MC is a nearly seamless addition since all of the content still exists after migration—it is simply at a different offset than before migration. For storage systems that support indirection, such as deduplicated storage [28], MC causes few architectural changes, though it likely increases fragmentation. On the other hand, DC introduces dependencies between data chunks that span the storage system: storage functionality has to be modified to handle indirections between delta compressed chunks and the *base* chunks against which they have been encoded [20]. Such modifications affect such system features as garbage collection, replication, and integrity checks.

3 Approach

Much of the focus of our work on Migratory Compression is in the context of reorganizing and compressing a single file (`mzip`), described in Section 3.1. In addition, we compare `mzip` to in-place delta-encoding of similar data (Section 3.2) and reorganization during migration to an archival tier within DDFS (Section 3.3).

3.1 Single-File Migratory Compression

The general idea of MC is to partition data into chunks and reorder them to store similar chunks sequentially, increasing compressors' opportunity to detect redundant strings and leading to better compression. For standalone file compression, this can be added as a pre-processing stage, which we term `mzip`. A reconstruction process is needed as a post-processing stage in order to restore the original file after decompression.

3.1.1 Similarity Detection with Super-features

The first step in MC is to partition the data into chunks. These chunks can be fixed size or variable size "content-defined chunks." Prior work suggests that in general variable-sized chunks provide a better opportunity to identify duplicate and similar data [12]; however, virtual machines use fixed-sized blocks, and deduplicating VM images potentially benefits from fixed-sized blocks [21]. We default to variable-sized chunks based on the comparison of fixed-sized and variable-sized units below.

One big challenge to doing MC is to identify similar chunks efficiently and scalably. A common practice is to generate *similarity features* for each chunk; two chunks are likely to be similar if they share many features. While it is possible to enumerate the closest matches by comparing all features, a useful approximation is to group sets of features into *super-features* (SFs): two data objects that have a single SF in common are likely to be fairly similar [3]. This approach has been used numerous times to successfully identify similar web pages, files, and/or chunks within files [10, 12, 20].

Because it is now well understood, we omit a detailed explanation of the use of SFs here. We adopt the "First-Fit" approach of Kulkarni, et al. [12], which we will term the *greedy* matching algorithm. Each time a chunk is processed, its N SFs are looked up in N hash tables, one per SF. If any SF matches, the chunk is associated with the other chunks sharing that SF (i.e., it is added to a list and the search for matches terminates). If no SF matches, the chunk is inserted into each of the N hash tables so that future matches can be identified.

We explored other options, such as sorting all chunks on each of the SFs to look for chunks that match several SFs rather than just one. Across the datasets we analyzed, this sort marginally improved compression but the computational overhead was disproportionate. Note, however, that applying MC to a file that is so large that its metadata (fingerprints and SFs) is too large to process in memory would require some out-of-core method such as sorting.

3.1.2 Data Migration and Reconstruction

Given information about which chunks in a file are similar, our `mzip` preprocessor outputs two recipes: *migrate* and *restore*. The migrate recipe contains the chunk order of the reorganized file: chunks identified to be similar are located together, ordered by their offset within the original file. (That is, a later chunk is moved to be adjacent to the first chunk it is similar to.) The restore recipe contains the order of chunks in the reorganized file and is used to reconstruct the original file. Generally, the overhead of generating these recipes is orders of magnitude less than the the overhead of physically migrating the data stored in disk.

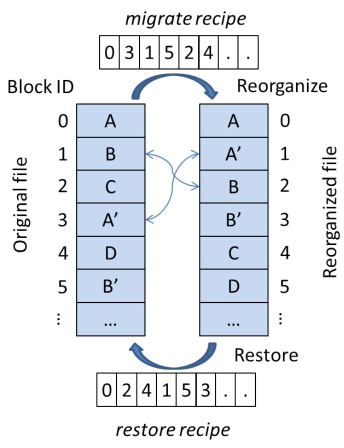


Figure 2: An example of the reorganization and restore procedures.

Figure 2 presents a simplified example of these two procedures, assuming fixed chunk sizes. We show a file with a sequence of chunks A through D, and including A' and B' to indicate chunks that are similar to A and B respectively. The reorganized file places A' after A and B' after B, so the *migrate* recipe specifies that the reorganized file consists of chunk 0 (A), chunk 3 (A'), chunk 1 (B), chunk 5 (B'), and so on. The *restore* recipe shows that to obtain the original order, we output chunk 0 (A), chunk 2 (B), chunk 4 (C), chunk 1 (A'), etc. from the reorganized file. (For variable-length chunks, the recipes contain byte offsets and lengths rather than block offsets.)

Once we have the migrate recipe and the restore recipe, we can create the reorganized file. Reorganization (migration) and reconstruction are complements of each other, each moving data from a specific location in an input file to a desired location in the output file. (There is a slight asymmetry resulting from deduplication, as completely identical chunks can be omitted completely in the reorganized file, then copied 1-to-N when reconstructing the original file.)

There are several methods for moving chunks.

In-Memory. When the original file can fit in memory, we can read in the whole file into memory and output chunks in the reorganized order sequentially. We call this the 'in-mem' approach.

Chunk-level. When we cannot fit the original file in memory, the simplest way to reorganize a file is to scan the chunk order in the migrate recipe: for every chunk needed, seek to the offset of that chunk in the original file, read it, and output it to the reorganized file. When using HDDs, this could become very inefficient because of the number of random I/Os involved.

Multi-pass. We also designed a 'multi-pass' algorithm, which scans the original file repeatedly from start to finish; during each pass, chunks in a particular *reorg range* of the reorganized file are saved in a memory buffer while others are discarded. At the end of each pass, chunks in the memory buffer are output to the reorganized file and the reorg range is moved forward. This approach replaces random I/Os with multiple scans of the original file. (Note that if the file fits in memory, the in-memory approach is the multi-pass approach with a single pass.)

Our experiments in Section 5.2 show that the in-memory approach is best, but when memory is insufficient, the multi-pass approach is more efficient than chunk-level. We can model the relative costs of the two approaches as follows. Let T be elapsed time, where T_{mp} is the time for multipass and T_c is the time when using individual chunks. Focusing only on I/O costs, T_{mp} is the time to read the entire file sequentially N times, where N is the number of passes over the data. If disk throughput is D and the file size is S, $T_{mp} = S * N/D$. For a size of 15GB, 3 passes, and 100MB/s throughput, this works out to 7.7 minutes for I/O. If CS represents the chunk size, the number of chunk-level I/O operations is S/CS and the elapsed time is $T_c = \frac{S/CS}{IOPS}$. For a disk with 100 IOPS and an 8KB chunk size, this equals 5.4 hours. Of course there is some locality, so what fraction of I/Os must be sequential or cached for the chunk approach to break even with the multi-pass one? If we assume that the total cost for chunk-level is the cost of reading the file once sequentially[3] plus the cost of random I/Os, then we solve for the cost of the random fraction (RF) of I/Os equaling the cost of $N-1$ sequential reads of the file:

$$S*(N-1)/D = \frac{S*RF/CS}{IOPS}$$

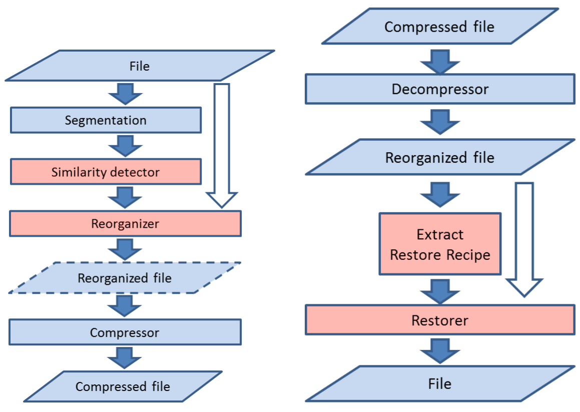
giving

$$RF = \frac{(N-1)*IOPS*CS}{D}.$$

In the example above, this works out to $\frac{16}{1024} = 1.6\%$; *i.e.*, if more than 1.6% of the data has dispersed similarity matches, then the multi-pass method should be preferred.

Solid-state disks, however, offer a good compromise. Using SSDs to avoid the penalty of random I/Os on HDDs causes the chunk approach to come closer to the in-memory performance.

3.1.3 `mzip` Workflow



(a) Compression (b) Decompression

Figure 3: Migratory Compression workflow.

Figure 3 presents the compression and decompression workflows in `mzip`. Compression/decompression and segmentation are adopted from existing tools, while similarity detection and reorganization/restoration are specially developed and highlighted in red. The original file is read once by the segmenter, computing cryptographically secure fingerprints (for deduplication) and resemblance features, then it is read again by the reorganizer to produce a file for compression. (This file may exist only as a pipeline between the reorganizer and the compressor, not separately stored, something we did for all compressors but `rzip` as it requires the ability to *seek*.) To restore the file, the compressed file is decompressed and its restore recipe is extracted from the beginning of the resulting file. Then the rest of that file is processed by the restorer, in conjunction with the recipe, to produce the original content.

[3]Note that if there are so many random I/Os that we do not read large sequential blocks, T_{mp} is reduced by a factor of $1-RF$.

3.2 Intra-file Delta Compression

When applied in the context of a single file, we hypothesized that `mzip` would be slightly better than delta compression (DC) because its compression state at the time the similar chunk is compressed includes content from many KBs-MBs of data, depending on the compressor. To evaluate how `mzip` compares with DC within a file, we implemented a version of DC that uses the same workflows as `mzip`, except the 'reorganizer' and the 'restorer' in `mzip` are replaced with a 'delta-encoder' and a 'delta-decoder.' The delta-encoder encodes each similar chunk as a delta against a base chunk while the delta-decoder reconstructs a chunk, by patching the delta to its base chunk. (In our implementation, the chunk earliest in the file is selected as the base for each group of similar chunks. We use `xdelta` [14] for encoding, relying on the later compression pass to compress anything that has not been removed as redundant.)

3.3 Migratory Compression in an Archival Storage System

In addition to reorganizing the content of individual files, MC is well suited for reducing data requirements within an entire file system. However, this impacts read locality, which is already an issue for deduplicating storage systems [13]. This performance penalty therefore makes it a good fit for systems with minimal requirements for read performance. An archival system, such as Amazon Glacier [25] is a prime use case, as much of its data will not be reread; when it is, significant delays can be expected. When the archival system is a tier within a backup environment, such that data moves in bulk at regular intervals, the data migration is an opportune time to migrate similar chunks together.

To validate the MC approach in a real storage system, we've implemented a prototype using the existing deduplicating Data Domain Filesystem (DDFS) [28]. After deduplication, chunks in DDFS are aggregated into *compression regions* (CRs), which in turn are aggregated into *containers*. DDFS can support two storage tiers: an *active* tier for backups and a long-term retention tier for *archival*; while the former stores the most recent data within a time period (e.g., 90 days), the latter stores the relatively ‘cold’ data that needs to be retained for an extended time period (e.g., 5 years) before being deleted. Data migration is important for customers who weigh the dollar-per-GB cost over the migrate/retrieval performance for long-term data.

A daemon called *data migration* is used to migrate selected data periodically from the active tier to the archive tier. For performance reasons, data in the active tier is compressed with a simple LZ algorithm while we use `gzip` in the archive tier for better compression. Thus, for each file to be migrated in the namespace, DDFS reads out the corresponding compression regions from the active tier, uncompresses each, and recompresses with `gzip`.

The MC technique would offer customers a further tradeoff between the compression ratio and migrate/retrieval throughput. It works as follows:

Similarity Range. Similarity detection is limited to files migrated in one iteration, for instance all files written in a span of two weeks or 90 days.

Super-features. We use 12 similarity features, combined as 3 SFs. For each container to be migrated, we read out its metadata region, extract the SFs associated with each chunk, and write these to a file along with the chunk’s fingerprint.

Clustering. Chunks are grouped in a similar fashion to the greedy single SF matching algorithm described in Section 3.1.1, but via sorting rather than a hash table.

Data reorganization. Similar chunks are written together by collecting them from the container set in multiple passes, similar to the single-file multi-pass approach described in Section 3.1.2 but without a strict ordering. Instead, the passes are selected by choosing the largest clusters of similar chunks in the first one-third, then smaller clusters, and finally dissimilar chunks. Since chunks are grouped by any of 3 SFs, we use 3 Bloom filters, respectively, to identify which chunks are desired in a pass. We then copy the chunks needed for a given pass into the CR designated for a given chunk’s SF; the CR is flushed to disk if it reaches its maximum capacity.

Note that DDFS already has the notion of a mapping of a file identifier to a tree of chunk identifiers, and relocating a chunk does not affect the chunk tree associated with a file. Only the low-level index mapping a chunk fingerprint to a location in the storage system need be updated when a chunk is moved. Thus, there is no notion of a *restore recipe* in the DDFS case, only a recipe specifying which chunks to co-locate.

In theory, MC could be used in the backup tier as well as for archival: the same mechanism for grouping similar data could be used as a background task. However, the impact on data locality would not only impact read performance [13], it could degrade ingest performance during backups by breaking the assumptions underlying data locality: DDFS expects an access to the fingerprint index on disk to bring nearby entries into memory [28].

4 Methodology

We discuss evaluation metrics in Section 4.1, tunable parameters in Section 4.2, and datasets in Section 4.3.

4.1 Metrics

The high-level metrics by which to evaluate a compressor are the **compression factor** (CF) and the **resource usage** of the compressor. CF is the ratio of an original size to its compressed size, i.e higher CFs correspond to more data eliminated through compression; deduplication ratios are analogous.

In general, resource usage equates to processing time per unit of data, which can be thought of as the **throughput** of the compressor. There are other resources to consider, such as the required memory: in some systems memory is plentiful and even the roughly 10 GB of DRAM used by `7z` with its maximum 1 GB dictionary is fine; in some cases the amount of memory available or the amount of compression being done in parallel results in a smaller limit.

Evaluating the performance of a compressor is further complicated by the question of parallelization. Some compressors are inherently single-threaded while others support parallel threads. Generally, however, the fastest compression is also single-threaded (e.g., `gzip`), while a slower but more effective compressor such as `7z` is slower despite its multiple threads. We consider end-to-end time, not CPU time.

Most of our experiments were run inside a virtual machine, hosted by an ESX server with 2x6 Intel 2.67GHz Xeon X5650 cores, 96 GB memory, and 1-TB 3G SATA 7.2k 2.5in drives. The VM is allocated 90 GB memory except in cases when memory is explicitly limited, as well as 8 cores and a virtual disk with a 100 GB ext4 partition on a two-disk RAID-1 array. For 8 KB random accesses, we have measured 134 IOPS for reads (as well as 385 IOPS for writes, but we are not evaluating random writes), using a 70 GB file and an I/O queue depth of 1. For 128 KB sequential accesses, we measured 108 MB/s for reads and 80 MB/s for writes. The SSD used is a Samsung Pro 840, with 22K IOPS for random 8 KB reads and 264 MB/s for 128 KB sequential reads (write

Dataset		Size (GB)	Dedupe (X)	Compression Factor of Standalone Compressors (X)			
Type	Name			`gzip`	`bzip2`	`7z`	`rzip`
Workstation Backup	WS1	17.36	1.69	2.70	3.22	4.44	4.46
	WS2	15.73	1.77	2.32	2.61	3.16	3.12
Email Server Backup	EXCHANGE1	13.93	1.06	1.83	1.92	3.35	3.99
	EXCHANGE2	17.32	1.02	2.78	3.13	4.75	4.79
VM Image	Ubuntu-VM	6.98	1.51	3.90	4.26	6.71	6.69
	Fedora-VM	27.95	1.19	3.21	3.49	4.22	3.97

Table 1: Dataset summary: size, deduplication factor of 8 KB variable chunking and compression ratios of standalone compressors.

throughputs become very low because of no TRIM support in the hypervisor: 20 MB/s for 128 KB sequential writes). To minimize performance variation, all other virtual machines were shut down except those providing system services. Each experiment was repeated three times; we report averages. We don't plot error bars because the vast majority of experiments have a relative standard error under 5%; in a couple of cases, decompression timings vary with 10–15% relative error.

To compare the complexity of MC with other compression algorithms, we ran most experiments in-memory. In order to evaluate the extra I/O necessary when files do not fit in memory, some experiments limit memory size to 8 GB and use either an SSD or hard drive for I/O.

The tool that computes chunk fingerprints and features is written in C, while the tools that analyze that data to cluster similar chunks and reorganize the files are written in Perl. The various compressors are off-the-shelf Linux tools installed from repositories.

4.2 Parameters Explored

In addition to varying the workload by evaluating different datasets, we consider the effect of a number of parameters. Defaults are shown in **bold**.

Compressor. We consider **gzip**, `bzip2`, `7z`, and `rzip`, with or without MC.

Compression tuning. Each compressor can be run with a parameter that trades off performance against compressibility. We use the **default parameters** unless specified otherwise.

MC **chunking.** Are chunks fixed or **variable** sized?

MC **chunk size.** How large are chunks? We consider 2, **8**, and 32 KB; for variable-sized chunks, these represent target averages.

MC **resemblance computation.** How are super-features matched? (Default: **Four SFs**, matched greedily, one SF at a time.)

MC **data source.** When reorganizing an input file or reconstructing the original file after decompression, where is the input stored? We consider an **in-memory file system**, SSD, and hard disk.

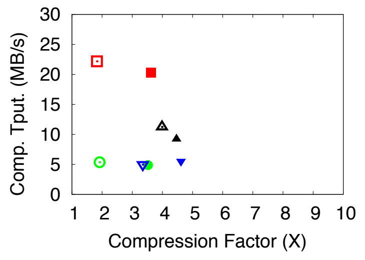
4.3 Datasets

Table 1 summarizes salient characteristics of the input datasets used to test `mzip`, two types of backups and a pair of virtual machine images. Each entry shows the total size of the file processed, its deduplication ratio (half of them can significant boost their CF using MC simply through deduplication), and the CF of the four off-the-shelf compressors. We find that `7z` and `rzip` both compress significantly better than the others and are similar to each other.

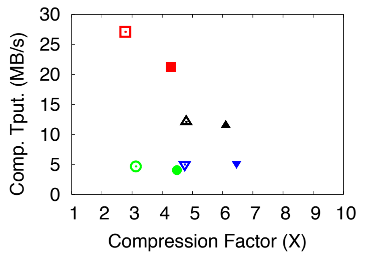
- We use four single backup image files taken from production deduplication backup appliances. Two are backups of workstations while the other two are backups of Exchange email servers.
- We use two virtual machine disk images consisting of VMware VMDK files. One has Ubuntu 12.04.01 LTS installed while the other uses Fedora Core release 4 (a dated but stable build environment).

5 `mzip` Evaluation

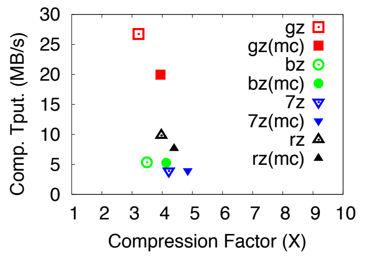
The most important consideration in evaluating MC is whether the added effort to find and relocate similar content is justified by the improvement in compression. Section 5.1 compares CF and throughput across the six datasets. Section 5.2 looks specifically at the throughput when memory limitations force repeated accesses to disk and finds that SSDs would compensate for random I/O penalties. Section 5.3 compares `mzip` to a similar intra-file DC tool. Finally, Section 5.4 considers additional sensitivity to various parameters and configurations.



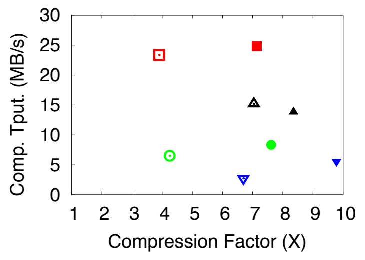
(a) EXCHANGE1



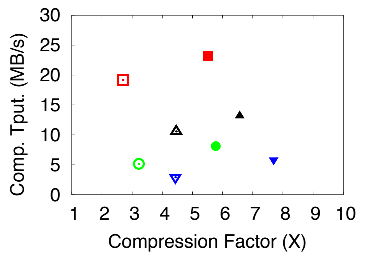
(b) EXCHANGE2



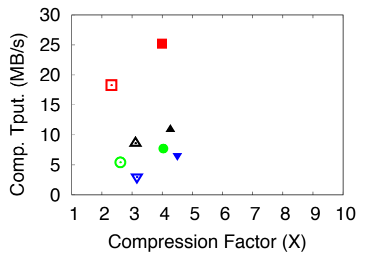
(c) Fedora-VM



(d) Ubuntu-VM



(e) WS1



(f) WS2

Figure 4: Compression throughput vs. Compression Factor for all datasets, using unmodified compression or MC, for four compressors. The legend for all plots appears in (c).

5.1 Compression Effectiveness and Performance Tradeoff

Figure 4 plots compression throughput versus compression factor, using the six datasets. All I/O was done using an in-memory file system. Each plot shows eight points, four for the off-the-shelf compressors (`gzip`, `bzip2`, `7z`, and `rzip`) using default settings and four for these compressors using MC.

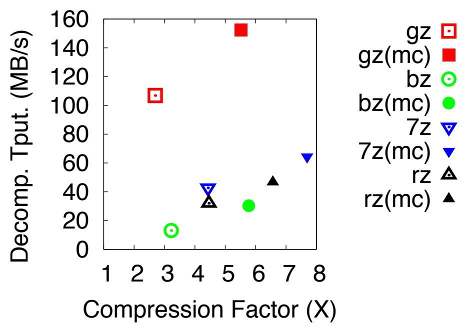
Generally, adding MC to a compressor significantly improves the CF (23–105% for `gzip`, 18–84% for `bzip2`, 15–74% for `7z` and 11–47% for `rzip`). It is unsurprising that `rzip` has the least improvement, since it already finds duplicate chunks across a range of a file, but MC further increases that range. Depending on the compressor and dataset, throughput may decrease moderately or it may actually improve as a result of the compressor getting (a) deduplicated and (b) more compressible input. We find that `7z` with MC always gets the highest CF, but often another compressor gets nearly the same compression with better throughput. We also note that in general, for these datasets, off-the-shelf `rzip` compresses just about as well as off-the-shelf `7z` but with much higher throughput. Better, though, the combination of `gzip` and MC has a comparable CF to any of the other compressors without MC, and with still higher throughput, making it a good choice for general use.

Decompression performance may be more important than compression performance for use cases where something is compressed once but uncompressed many times. Figure 5 shows decompression throughput versus CF for two representative datasets. For WS1, we see that adding MC to existing compressors tends to improve CF while significantly improving decompression throughput. It is likely because deduplication leads to less data to decompress. For EXCHANGE1, CF improves substantially as well, with throughput not greatly affected. Only for Fedora-VM (not shown) does `gzip` decompression throughput decrease in any significant fashion (from about 140 MB/s to 120).

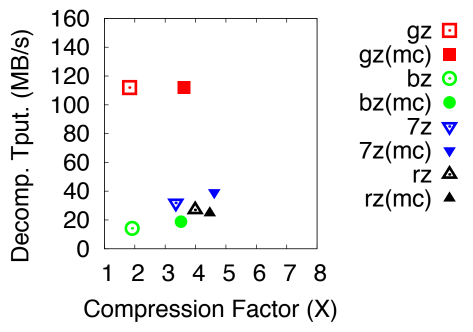
5.2 Data Reorganization Throughput

To evaluate how `mzip` may work when a file does not fit in memory, we experimented with a limit of 8 GB RAM when the input data is stored in either a solid state disk (SSD) or hard disk drive (HDD). The output file is stored in the HDD. When reading from HDD, we evaluated two approaches: chunk-level and multi-pass. Since SSD has no random-access penalty, we use only chunk-level and compare SSD to in-mem.

Figure 6 shows the compression throughputs for SSD-based and HDD-based `mzip`. (Henceforth `mzip` refers to `gzip`+MC.) We can see that SSD approaches in-memory performance, but as expected there is a significant reduction in throughput using the HDD. This reduction can be mitigated by the multipass approach. For instance, using a reorg range of 60% of memory, 4.81 GB, if the file does not fit in memory, the throughput can be improved significantly for HDD-based `mzip` by compar-



(a) WS1



(b) EXCHANGE1

Figure 5: Decompression throughput vs. Compression Factor for two representative datasets (WS1 and EXCHANGE1), using unmodified compression or MC.

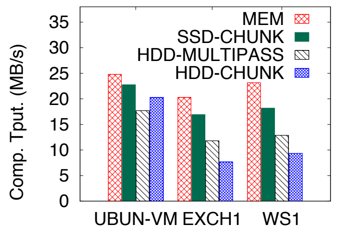


Figure 6: Compression throughput comparison for HDD-based or SSD-based gzip (MC).

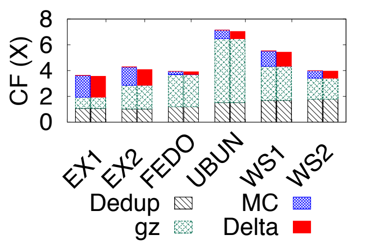
ison to accessing each chunk in the order it appears in the reorganized file (and paying the corresponding costs of random I/Os).

Note that Ubuntu-VM can approximately fit in available memory, so the chunk-level approach performs better than multi-pass: multi-pass reads the file sequentially twice, while chunk-level can use OS-level caching.

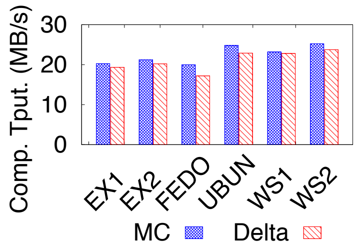
5.3 Delta Compression

Figure 7 compares the compression and performance achieved by `mzip` to compression using in-place delta-encoding,[4] as described in Section 3.2. Both use `gzip` as the final compressor. Figure 7(a) shows the CF for each dataset, broken out by the contribution of each technique. The bottom of each stacked bar shows the impact of deduplication (usually quite small but up to a factor of 1.8). The next part of each bar shows the *additional* contribution of `gzip` after deduplication has been applied, but with no reordering or delta-encoding. Note that these two components will be the same for each pair of bars. The top component is the additional benefit of either `mzip` or delta-encoding. `mzip` is always slightly better (from 0.81% to 4.89%) than deltas, but with either technique we can get additional compression beyond the gain from deduplication and traditional compression: $>80\%$ more for EXCHANGE1, $>40\%$ more for EXCHANGE2 and $>25\%$ more for WS1.

Figure 7(b) plots the compression throughput for `mzip` and DC, using an in-memory file system (we omit decompression due to space limitations). `mzip` is consistently faster than DC. For compression, `mzip` averages 7.21% higher throughput for these datasets. while for decompression `mzip` averages 29.35% higher throughput.



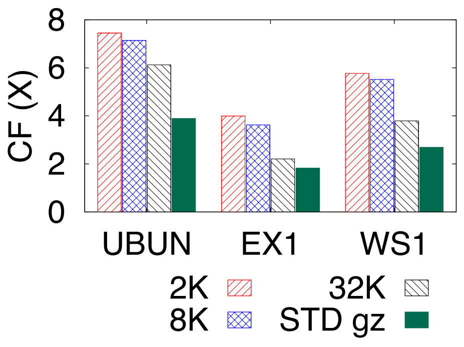
(a) Compression Factor, by contributing technique



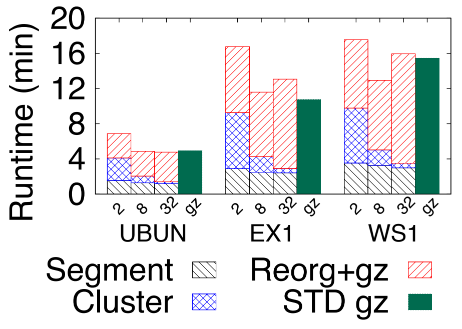
(b) Compression Throughput

Figure 7: Comparison between `mzip` and `gzip` (delta compression) in terms of compression factor and compression throughput. CFs are broken down by dedup and `gzip` (same for both), plus the additional benefit of either MC or DC.

[4]Delta-encoding plus compression is delta compression. Some tools such as `vcdiff` [11] do both simultaneously, while our tool delta-encodes chunks and then compresses the entire file.

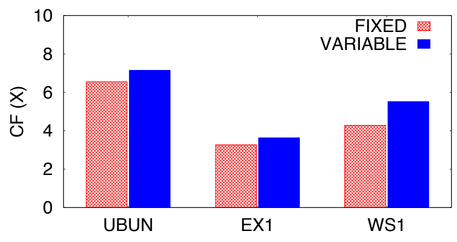


(a) Compression Factor

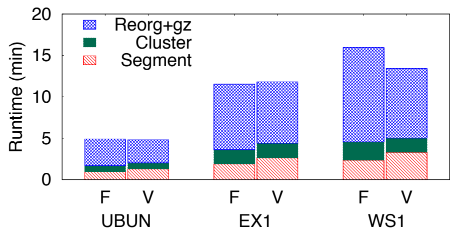


(b) Runtime, by component cost

Figure 8: Compression factor and runtime for `mzip`, varying chunk size.



(a) Compression Factor



(b) Runtime

Figure 9: Compression factor and runtime for `mzip`, when either fixed-size or variable-size chunking is used.

5.4 Sensitivity to Environment

The effectiveness and performance of MC depend on how it is used. We looked into various chunk sizes, compared fixed-size with variable-size chunking, evaluated the number of SFs to use in clustering and studied different compression levels and window sizes.

5.4.1 Chunk Size

Figure 8 plots `gzip`-MC (a) CF and (b) runtime as a function of chunk size (we show runtime to break down individual components by their contribution to the overall delay). We shrink and increase the default 8 KB chunk size by a factor of 4. Compression increases slightly in shrinking from 8 KB to 2 KB but decreases dramatically moving up to 32 KB. The improvement from the smaller chunksize is much less than seen when only deduplication is performed [26], because MC eliminates redundancy among similar chunks as well as identical ones. The reduction when increasing to 32 KB is due to a combination of fewer chunks to be detected as identical and similar and the small `gzip` lookback window: similar content in one chunk may not match content from the preceding chunk.

Figure 8(b) shows the runtime overhead, broken down by processing phase. The right bar for each dataset corresponds to standalone `gzip` without MC, and the remaining bars show the additive costs of segmentation, clustering, and the pipelined reorganization and compression. Generally performance is decreased by moving to a smaller chunk size, but interestingly in two of the three cases it is also worse when moving to a larger chunk size. We attribute the lower throughput to the poorer deduplication and compression achieved, which pushes more data through the system.

5.4.2 Chunking Algorithm

Data can be divided into fixed-sized or variable-sized blocks. For MC, supporting variable-sized chunks requires tracking individual byte offsets and sizes rather than simply block offsets. This increases the recipe sizes by about a factor of two, but because the recipes are small relative to the original file, the effect of this increase is limited. In addition, variable chunks result in better deduplication and matching than fixed, so CFs from using variable chunks are 14.5% higher than those using fixed chunks.

Figure 9 plots `mzip` compression for three datasets, when fixed-size or variable-size chunking is used. From Figure 9(a), we can see that variable-size chunking gives consistently better compression. Figure 9(b) shows that the overall performance of both approaches is comparable and sometimes variable-size chunking has better performance. Though variable-size chunking spends more time in the segmentation stage, the time to do compression can be reduced considerably when more chunks are duplicated or grouped together.

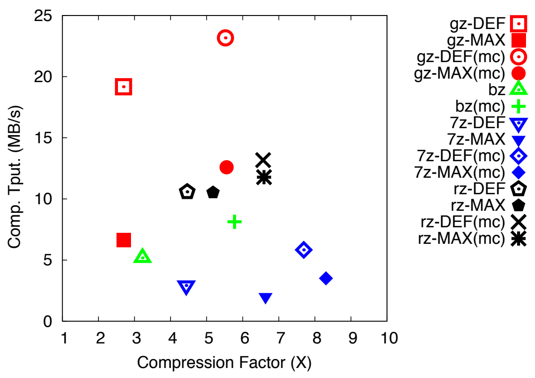


Figure 10: Comparison between the default and the maximum compression level, for standard compressors with and without MC, on the WS1 dataset.

5.4.3 Resemblance Computation

By default we use sixteen features, combined into four SFs, and a match on any SF is sufficient to indicate a match between two chunks. In fact most similar chunks are detected by using a single SF; however, considering three more SFs has little change in compression throughputs and sometimes improves compression factors greatly (e.g., a 13.6% improvement for EXCHANGE1). We therefore default to using 4 SFs.

5.4.4 Compression Window

For most of this paper we have focused on the default behavior of the three compressors we have been considering. For `gzip`, the "maximal" level makes only a small improvement in CF but with a significant drop in throughput, compared to the default. In the case of `bzip2`, the default is equivalent to the level that does the best compression, but overall execution time is still manageable, and lower levels do not change the results significantly. In the case of `7z`, there is an enormous difference between its default level and its maximal level: the maximal level generally gives a much higher CF with only a moderate drop in throughput. For `rzip`, we use an undocumented parameter "-L20"to increase the window to 2 GB; increasing the window beyond that had diminishing returns because of the increasingly coarse granularity of duplicate matching.

Figure 10 shows the compression throughput and CF for WS1 when the default or maximum level is used, for different compressors with and without MC. (The results are similar for other datasets.) From this figure, we can tell that maximal `gzip` reduces throughput without discernible effect on CF; `7z` without MC improves CF disproportionately to its impact on performance; and maximal `7z` (MC) moderately improves CF and reduces performance. More importantly, with MC and standard compressors, we can achieve higher CFs with much higher compression throughput than compressors' standard maximal level. For example, the open diamond marking `7z`-DEF(MC) is above and to the right of the close inverted triangle marking `7z`-MAX. Without MC, `rzip`'s maximal level improves performance with comparable throughput; with MC, `rzip` gets the same compression as `7z`-MAX with much better throughput, and `rzip`-MAX decreases that throughput without improving CF. The best compression comes from `7z`-MAX with MC, which also has better throughput than `7z`-MAX without MC.

6 Archival Migration in DDFS

In addition to using MC in the context of a single file, we can implement it in the file system layer. As an example, we evaluated MC in DDFS, running on a Linux-based backup appliance equipped with 8x2 Intel 2.53GHz Xeon E5540 cores and 72 GB memory. In our experiment, either the active tier or archive tier is backed by a disk array of 14 1-TB SATA disks. To minimize performance variation, no other workloads ran during the experiment.

6.1 Datasets

DDFS compresses each compression region using either LZ or `gzip`. Table 2 shows the characteristics of a few backup datasets using either form of compression. (Note that the WORKSTATIONS dataset is the union of several workstation backups, including WS1 and WS2, and all datasets are many backups rather than a single file as before.) The logical size refers to pre-deduplication data, and most datasets deduplicate substantially.

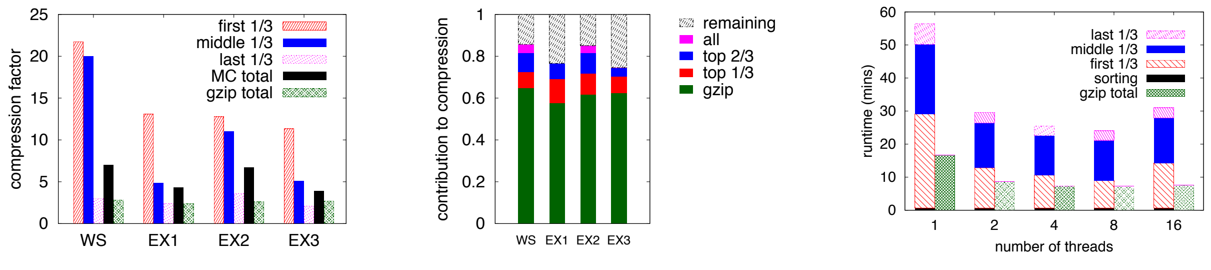
The table shows that `gzip` compression is 25–44% better than LZ on these datasets, hence DDFS uses `gzip` by default for archival. We therefore compare base `gzip` with `gzip` after MC preprocessing. For these datasets, we reorganize all backups together, which is comparable to an archive migration policy that migrates a few months at a time; if archival happened more frequently, the benefits would be reduced.

6.2 Results

Figure 11(a) depicts the compressibility of each dataset, including separate phases of data reorganization. As described in Section 3.3, we migrate data in thirds. The top third contains the biggest clusters and achieves the greatest compression. The middle third contain smaller clusters and may not compress quite as well, and the bottom third contains the smallest clusters, including clusters of a single chunk (nothing similar to combine it with). The next bar for each dataset shows the aggregate CF

Type	Name	Logical Size (GB)	Dedup. Size (GB)	Dedup. + LZ Size (GB)	LZ CF	Dedup. + gzip (GB)	gzip CF
Workstation	WORKSTATIONS	2471	454	230	1.97	160	2.84
Email Server	EXCHANGE1	570	51	27	1.89	22	2.37
	EXCHANGE2	718	630	305	2.07	241	2.61
	EXCHANGE3	596	216	103	2.10	81	2.67

Table 2: Datasets used for archival migration evaluation.



(a) CFs as a function of migration phase (b) Fraction of data saved in each migration phase (c) Durations, as a function of threads, for EXCHANGE1

Figure 11: Breakdown of the effect of migrating data, using just gzip or using MC in 3 phases.

using MC, while the right-most bar shows the compression achieved with gzip and no reorganization. Collectively, MC achieves 1.44–2.57× better compression than the gzip baseline. Specifically, MC outperforms gzip most (by 2.57×) on the workstations dataset, while it improves the least (by 1.44×) on EXCHANGE3.

Figure 11(b) provides a different view into the same data. Here, the cumulative fraction of data saved for each dataset is depicted, from bottom to top, normalized by the post-deduplicated dataset size. The greatest savings (about 60% of each dataset) come from simply doing gzip, shown in green. If we reorganize the top third of the clusters, we additionally save the fraction shown in red. By reorganizing the top two-thirds we include the fraction in blue; interestingly, in the case of WORKSTATIONS, the reduction achieved by MC in the middle third relative to gzip is higher than that of the top third, because gzip alone does not compress the middle third as well as it compresses the top. If we reorganize everything that matches other data, we may further improve compression, but only two datasets have a noticeable impact from the bottom third. Finally, the portion in gray at the top of each bar represents the data that remains after MC.

There are some costs to the increased compression. First, MC has a considerably higher memory footprint than the baseline: compared to gzip, the extra memory usage for reorganization buffers is 6 GB (128 KB compression regions * 48 K regions filled simultaneously). Second, there is run-time overhead to identify clusters of similar chunks and to copy and group the similar data. To understand what factors dominate the run-time overhead of MC, Figure 11(c) reports the elapsed time to copy the post-deduplication 51 GB EXCHANGE1 dataset to the archive tier, with and without MC, as a function of the number of threads (using a log scale). We see that multithreading significantly improves the processing time of each pass. We divide the container range into multiple subranges and copy the data chunks from each subrange into in-memory data reorganization buffers with multiple worker threads. As the threads increase from 1 to 16, the baseline (gzip) duration drops monotonically and is uniformly less than the MC execution time. On the other hand, MC achieves the minimum execution time with 8 worker threads; further increasing the thread count does not reduce execution time, an issue we attribute to intra-bucket serialization within hash table operations and increased I/O burstiness.

Reading the entire EXCHANGE1 dataset, there is a 30% performance degradation after MC compared to simply copying in the original containers. Such a read penalty would be unacceptable for primary storage, problematic for backup [13], but reasonable for archival data given lower performance expectations. But reading back just the final backup within the dataset is 7× slower than without reorganization, if all chunks are relocated whenever possible. Fortunately, there are potentially significant benefits to *partial* reorganization. The greatest compression gains are obtained by grouping the biggest clusters, so migrating only the top-third of clusters can provide high benefits at moderate cost. Interestingly, if just the top third of clusters are reorganized, there is only a 24% degradation reading the final backup.

7 Related Work

Compression is a well-trodden area of research. Adaptive compression, in which strings are matched against patterns found earlier in a data stream, dates back to the variants of Lempel-Ziv encoding [29, 30]. Much of the early work in compression was done in a resource-poor environment, with limited memory and computation, so the size of the adaptive dictionary was severely limited. Since then, there have been advances in both encoding algorithms and dictionary sizes, so for instance Pavlov's `7z` uses a "Lempel-Ziv-Markov-Chain" (LZMA) algorithm with a dictionary up to 1 GB [1]. With `rzip`, standard compression is combined with rolling block hashes to find large duplicate content, and larger lookahead windows decrease the granularity of duplicate detection [23].

The Burrows-Wheeler Transform (BWT), incorporated into `bzip2`, rearranges data—within a relatively small window—to make it more compressible [5]. This transform is reasonably efficient and easily reversed, but it is limited in what improvements it can effect.

Delta compression, described in Section 2.2, refers to compressing a data stream relative to some other known data [9]. With this technique, large files must normally be compared piecemeal, using subfiles that are identified on the fly using a heuristic to match data from the old and new files [11]. MC is similar to that sort of heuristic, except it permits deltas to be computed at the granularity of small chunks (such as 8 KB) rather than a sizable fraction of a file. It has been used for network transfers, such as updating changing Web pages over HTTP [16]. One can also deduplicate identical chunks in network transfers at various granularities [10, 17].

DC has also been used in the context of deduplicating systems. Deltas can be done at the level of individual chunks [20] or large units of MBs or more [2]. Fine-grained comparisons have a greater chance to identify similar chunks but require more state.

These techniques have limitations in the range of data over which compression will identify repeated sequences; even the 1 GB dictionary used by `7-zip` is small compared to many of today's files. There are other ways to find redundancy spread across large corpa. As one example, REBL performed fixed-sized or content-defined chunking and then used resemblance detection to decide which blocks or chunks should be delta-encoded [12]. Of the approaches described here, MC is logically the most similar to REBL, in that it breaks content into variable sized chunks and identifies similar chunks to compress together. The work on REBL only reported the savings of pair-wise DC on any chunks found to be similar, not the end-to-end algorithm and overhead to perform standalone compression and later reconstruct the original data. From the standpoint of *rearranging* data to make it more compressible, MC is most similar to BWT.

8 Future Work

We briefly mention two avenues of future work, application domains and performance tuning.

Compression is commonly used with networking when the cost of compression is offset by the bandwidth savings. Such compression can take the form of simple in-line coding, such as that built into modems many years ago, or it can be more sophisticated traffic shaping that incorporates delta-encoding against past data transmitted [19, 22]. Another point along the compression spectrum would be to use `mzip` to compress files prior to network transfer, either statically (done once and saved) or dynamically (when the cost of compression must be included in addition to network transfer and decompression). We conducted some initial experiments using `rpm` files for software distribution, finding that a small fraction of these files gained a significant benefit from `mzip`, but expanding the scope of this analysis to a wider range of data would be useful. Finally, it may be useful to combine `mzip` with other redundancy elimination protocols, such as content-based naming [18].

With regard to performance tuning, we have been gaining experience with MC in the context of the archival system. The tradeoffs between compression factors and performance, both during archival and upon later reads to an archived file, bear further analysis. In addition, it may be beneficial to perform small-scale MC in the context of the backup tier (rather than the archive tier), recognizing that the impact to read performance must be minimized. `mzip` also has potential performance improvements, such as multi-threading and reimplementing in a more efficient programming language.

9 Conclusions

Storage systems must optimize space consumption while remaining simple enough to implement. Migratory Compression reorders content, improving traditional compression by up to 2× with little impact on throughput and limited complexity. When compressing individual files, MC paired with a typical compressor (e.g., `gzip` or `7z`) provides a clear improvement. More importantly, MC delivers slightly better compression than delta-encoding without the added complexities of tracking dependencies (for decoding) between non-adjacent chunks. Migratory Compression can deliver significant additional consumption for broadly used file systems.

Acknowledgments

We acknowledge Nitin Garg for his initial suggestion of improving data compression by collocating similar content in the Data Domain File System. We thank Remzi Arpaci-Dusseau, Scott Auchmoody, Windsor Hsu, Stephen Manley, Harshad Parekh, Hugo Patterson, Robert Ricci, Hyong Shim, Stephen Smaldone, Andrew Tridgell, and Teng Xu for comments and feedback on earlier versions and/or the system. We especially thank our shepherd, Zheng Zhang, and the anonymous reviewers; their feedback and guidance have been especially helpful.

References

[1] 7-zip. `http://www.7-zip.org/`. Retrieved Sep. 7, 2013.

[2] ARONOVICH, L., ASHER, R., BACHMAT, E., BITNER, H., HIRSCH, M., AND KLEIN, S. T. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009).

[3] BRODER, A. Z. On the resemblance and containment of documents. In *In Compression and Complexity of Sequences (SEQUENCES97)* (1997), IEEE Computer Society.

[4] BURROWS, M., JERIAN, C., LAMPSON, B., AND MANN, T. On-line data compression in a log-structured file system. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems* (1992), ASPLOS V.

[5] BURROWS, M., AND WHEELER, D. J. A block-sorting lossless data compression algorithm. Tech. Rep. SRC-RR-124, Digital Equipment Corporation, 1994.

[6] DEUTSCH, P. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.

[7] FIALA, E. R., AND GREENE, D. H. Data compression with finite windows. *Communications of the ACM 32*, 4 (Apr. 1989), 490–505.

[8] GILCHRIST, J. Parallel data compression with bzip2. In *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems* (2004), vol. 16, pp. 559–564.

[9] HUNT, J. J., VO, K.-P., AND TICHY, W. F. Delta algorithms: an empirical analysis. *ACM Trans. Softw. Eng. Methodol. 7* (April 1998), 192–214.

[10] JAIN, N., DAHLIN, M., AND TEWARI, R. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *4th USENIX Conference on File and Storage Technologies* (2005).

[11] KORN, D. G., AND VO, K.-P. Engineering a differencing and compression data format. In *USENIX Annual Technical Conference* (2002).

[12] KULKARNI, P., DOUGLIS, F., LAVOIE, J., AND TRACEY, J. M. Redundancy elimination within large collections of files. In *USENIX 2004 Annual Technical Conference* (June 2004).

[13] LILLIBRIDGE, M., ESHGHI, K., AND BHAGWAT, D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *11th USENIX Conference on File and Storage Technologies* (Feb 2013).

[14] MACDONALD, J. File system support for delta compression. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[15] MAKATOS, T., KLONATOS, Y., MARAZAKIS, M., FLOURIS, M. D., AND BILAS, A. Using transparent compression to improve SSD-based I/O caches. In *Proceedings of the 5th European Conference on Computer Systems* (2010), EuroSys '10.

[16] MOGUL, J. C., DOUGLIS, F., FELDMANN, A., AND KRISHNAMURTHY, B. Potential benefits of delta encoding and data compression for http. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication* (1997), SIGCOMM '97.

[17] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001), SOSP '01.

[18] PARK, K., IHM, S., BOWMAN, M., AND PAI, V. S. Supporting practical content-addressable caching with CZIP compression. In *USENIX ATC* (2007).

[19] RIVERBED TECHNOLOGY. Wan Optimization (Steelhead). http://www.riverbed.com/products-solutions/products/wan-optimization-steelhead/, 2014. Retrieved Jan. 13, 2014.

[20] SHILANE, P., WALLACE, G., HUANG, M., AND HSU, W. Delta compressed and deduplicated storage using stream-informed locality. In *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems* (June 2012), USENIX Association.

[21] SMALDONE, S., WALLACE, G., AND HSU, W. Efficiently storing virtual machine backups. In *Proceedings of the 5th USENIX conference on Hot Topics in Storage and File Systems* (June 2013), USENIX Association.

[22] SPRING, N. T., AND WETHERALL, D. A protocol-independent technique for eliminating redundant network traffic. In *ACM SIGCOMM* (2000).

[23] TRIDGELL, A. *Efficient algorithms for sorting and synchronization*. PhD thesis, Australian National University Canberra, 1999.

[24] TUDUCE, I. C., AND GROSS, T. Adaptive main memory compression. In *USENIX 2005 Annual Technical Conference* (April 2005).

[25] VARIA, J., AND MATHEW, S. Overview of amazon web services, 2012.

[26] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *FAST'12: Proceedings of the 10th Conference on File and Storage Technologies* (2012).

[27] XZ. `http://tukaani.org/xz/`. Retrieved Sep. 25, 2013.

[28] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies* (Feb 2008).

[29] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (May 1977), 337–343.

[30] ZIV, J., AND LEMPEL, A. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on 24*, 5 (1978), 530–536.