RJ10533 (ALM1511-003) November 11, 2015
Computer Science

IBM Research Report

Quick Access to Compressed Data in Storage Systems

Cornel Constantinescu, David Chambliss

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Quick Access to Compressed Data in Storage Systems

Cornel Constantinescu and David Chambliss
IBM Almaden Research Center
San Jose, California, USA

Abstract

Primary storage systems that compress data in real time, use some form of on disk metadata to perform the *virtualization* needed in storing compressed data. Usually this metadata is in the form of B-trees (eventually compressed) and stored on disk. For random accesses to compressed data, where the metadata is not in cache, this additional layer significantly slows down random reads & writes. Our solution is to use much less metadata that only provides an approximation of the location of compressed data on disk and can be easily stored in the memory of the storage system. Read operations are extended to compensate for the imprecise position information in the metadata, and index marks embedded in the data are used to locate the required data within the expanded read. The data placement of written data is constrained to be described by the reduced metadata. The placement uses a piecewise linear scheme based on the locality in compressibility of data and we support this assumption with experiments.

1 Introduction

Although processor speeds and the available MIPS in storage systems increased significantly lately, adding compression to primary storage systems still can impact negatively the overall performance of the systems for some types of workloads. One reason for this is the need for additional metadata to map between raw (uncompressed) address domain and physical disk space where the compressed data is stored. This affects performance especially for random reads / writes where the metadata is not in cache. With some significant architectural changes this metadata could be stored on more expensive flash storage. However this block level mapping metadata is in addition and would compete for flash with other types of metadata (thin provisioning or file system metadata).

Our solution is to use much less metadata that only provides an approximation of the location of compressed data on disk and can be easily stored in the memory of the storage system. Read operations are extended to compensate for the imprecise position information in the metadata, and index marks embedded in the data are used to locate the required data within the expanded read. The data placement of written data is constrained to be described by the reduced metadata. The placement uses a piecewise linear scheme based on the locality in compressibility of data and we support this assumption with experiments.

In Section 2 we introduce some terminology and assumptions generally used in storage virtualization. Section 3 presents the main ideas in our method at a higher level. In

Section 4 we discuss the scenario of filling a new storage volume progressively by sequential writes and how we can accurately and efficiently approximate their placement by a piecewise linear interpolation scheme using little metadata. Section 5 shows how we can accurately read the compressed data (partitions) knowing that the partitions were placed satisfying an error window. Section 6 describes the cases where partitions are written in arbitrary order (not sequential) by using the first few partitions that happen to be written in the new subsegment to predict the overall compression ratio of the data (say 100 partitions) to be written in the subsegment. Section 7 summarises practical mechanisms that can be employed when one or a subset of partitions do not fit in the corresponding area in the extent, as required by the linear mapping plus the error window. Finally in Section 8 we conclude with few thoughts and further work.

2 Assumptions and Terminology

In our description, we use the term “virtual” to mean uncompressed, for example, virtual address means address in raw (uncompressed) data space. This paper proposes basically a storage virtualization system that manages the mapping from virtual addresses to the physical addresses which correspond to the allocation of physical space. The virtual address space in a block storage device is divided into coarse-grain ”segments” which may be of fixed size (say 10 GB each). The segments are divided into subsegments which are the basic unit for our piecewise linear mapping (see Section 4). The physical address space is divided into “extents” which may be of fixed size. Normally the virtualization system maps each subsegment into a contiguous range within an extent. That range may be called a subextent. There are cases however when this map crosses few extents.

2.1 Coarse-level Storage Virtualization

A coarse-grain virtualization layer enables subextents to be allocated for use by subsegments, and freed for reuse. Full extents are claimed as needed from the environment to fulfill these requests.

The coarse virtualization is structured to reduce the metadata size. Each segment is dynamically assigned a limited number of extents into which the segment’s data may be stored. For example, there might be a maximum of 32 extents per segment. The extent size is large enough so that the maximum number of extents will be enough to hold the maximum size the segment’s data may need to occupy after compression (e.g., 10 GB/32 = 320 MB). In the simplest case, an extent is owned by only one segment, but it is possible that a small number of segments might use different parts of the same extent. Since the possibilities for placement of a subextent are restricted, the number of bits needed to express its actual placement is reduced, as compared with an unrestricted placement.

The constrained placement may cause some space to go unused which might have been used with unconstrained placement. The amount depends on the particular formulation. In the simplest case where an extent is used only by one segment, then storage utilization for each segment is rounded up to a multiple of the ratio (extent size / segment size) because parts of extents might be unused.

3 Overview of Our Method

We consider data compressed in relatively small independent compression units named partitions (usually of 8 KB to 64KB). These compressed partitions are placed on storage in a constrained way that enables to be accessed with very little metadata which fits in memory. The large majority of partitions are written in accordance with a piecewise linear approximate mapping. That is, the data for a given range of virtual addresses (say 10 MB wide), called a subsegment, is written so that the nominal location in physical storage is a linear function of virtual address. Generally, the mapping is approximate in that the compressed partitions may be placed within a known margin (say 64 KB) of the nominal location. A logical read operation (details in Section 5) consults the piecewise linear function, extends the range of nominal locations by the margin amount, performs a physical read of the data in the extended physical range, locates the relevant partitions within the region read, performs their decompression, and returns the result. The expanded read size carries a very small incremental performance cost for a disk device, and it offers the latitude for data units of different compressibilities to be placed according to one linear slope.

Data write operations involve complexity because of the need to adhere to the approximate piecewise linear mapping. The mapping is created and adjusted in response to the actual compressibility of data partitions and therefore the physical device space they need to occupy. In some cases, an updated version of data at a given address may fit in the same location, possibly with the insertion or consumption of padding there. In other cases a group of neighboring partitions must be rewritten in shifted locations to make space available. In other cases an entire range of data is written to a new location, and the piecewise mapping is changed accordingly.

A very small number of partitions are allowed to be placed out of conformance with the mapping. One mechanism is that a new partition may be written in a location away from the linear mapping, and a redirection record may be written in the location designated by the mapping. Redirection records are used only very infrequently, since a read for data represented with an indirection record requires a second physical read. Another mechanism is an in-memory exceptions table. These out-of-conformance placements allow the postponement of relatively costly rewrite actions that are sometimes required for conformant placements, to a time when more information is available to select the best rewrite action.

The “knots” are the points in our piecewise linear interpolation where the slope changes. By extension, we name knot the in memory data structure holding the information needed in our interpolation: the extent ID and in extent offset for the current subsegment (say 20 bits), slope (7 + 2*4 bits), error_windows (2 bits), empty subsegment flag (1 bit). We think that about 5 bytes are enough. We choose to use equidistant knots (i.e. fixed size subsegments) to speed-up the search for the subsegment containing a given address and reduce knot memory size. However, for subsegments being filled or actively modified we can hold temporarily more metadata than for the cold subsegments (the large majority of subsegments). The additional metadata consists of ”lighter weight” knots containing only changes to main knot info (like different slope or error window). If these subsegments don’t become ”smoother” we may add their ID in the exceptions table, and continue using few different slopes and/or error windows inside them.

The main fact that enables our relatively simple mapping between raw and compressed data representation is the locality in compressibility of data. We have found that we can use a linear map (with constant slope equal with the average compression ratio) to map between raw addresses (raw LBA) and compressed addresses. This linear map can span hundreds of partitions (of 32 KB each) and the overall error relatively small that we can keep bounded (details and experimental results in Section 4).

An additional fact is the possibility of reliably predicting the compression ratio, (the slope of our linear mapping), using just one or few partitions in the subsegment (as shown for example in [1] and [2]). This fact enables placing non sequential compressed partitions at the right location in the extent (see details in Section 6).

Copy of a sequence of disk sectors from one location to another is a useful mechanism to maintain the linear mapping when we encounter a subset of partitions with significantly different CR than the others in the segment (for example when a large JPEG image is in the middle of a mostly text segment) or when rewriting data with widely different compression ratio than the previous data. Although copying can be minimized (using indirection pointers on disk, or leaving slack space between partitions, as shown in Section 6.1) it probably can't be avoided. We think that hardware support for the copy operation with no CPU involvement (that seems to be in the SCSI protocol, but not generally implemented) would be very useful to support our linear mapping.

4 Progressive Sequential Writes into New Volume - and Their Approximation

To evaluate if interpolation is effective when mapping from raw domain to compressed domain, we performed interpolation experiments on data stored on the hard drives of a number of machines. We extracted compression traces of hard drive data, each trace containing compression sizes for 307,200 partitions of 32 KB each, totaling about 10 GB raw data. The traces were taken on the first 10 GB of the hard drives of three servers: Trace1 on a SuSE 10 Linux running for more than 10 years, Trace2 and Trace3 on more recent Fedora and RedHat Linux.

A linear time online algorithm to construct a piecewise linear (pwl) interpolation of a set of adjacent partitions for a given maximum interpolation error (or error-window) was implemented; it maintains a min and max slope for the subsegment as new partitions are written (padded). When the approximation error becomes larger than the error window we generate a new "knot"- end points of the linear interpolation interval. So, we incrementally build the linear approximation using only the information on the current partition and two slopes (numbers) to find the next knot. The two slopes are the minimum and maximum slopes seen so far in the interval (subsegment). This algorithm is very simple and it runs very fast because there is no backtracking when checking the error window.

Figure 1 illustrates the workings of our linear interpolation algorithm. On X axis are addresses in raw (uncompressed) domain; to simplify, the raw addresses are multiples of 32K blocks (as if logical blocks on disk are 32KB). On Y axis are (real) addresses in bytes of compressed partitions. The compression ratio for each 32K partition is basically the slope of the segment mapping between raw and compressed domain. The linear interpolation algorithms tries to find the longest linear mapping that satisfies a given error window.

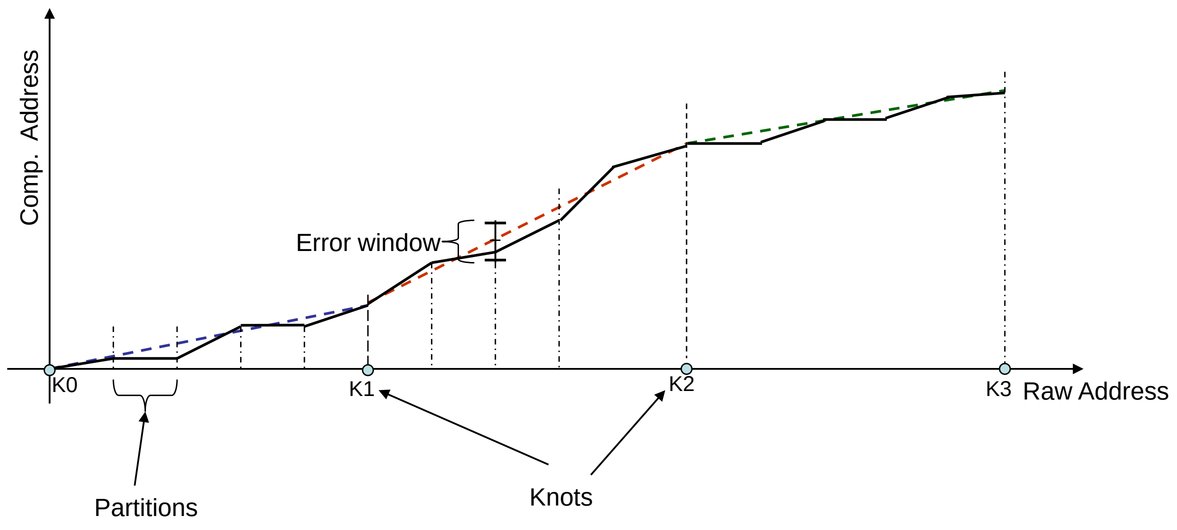

Figure 1: Progressive sequential writes into a new volume - and their approximation. Data is compressed in small independent units named *partitions*. Partitions have a *header* containing the virtual (raw) address of the partition. *Knots* are the places in the linear interpolation where the slope changes.

The average size of the subsegments for each trace when using few error windows are shown in Table 1. As can be seen, using an error window of 64K, the average number of partitions (of 32KB each) in an interpolation interval is well over 100.

Average size of the interpolation intervals for various interpolation thresholds (in #partitions)			
	(+/-)128	**(+/-)64**	**(+/-)32**
Trace1	447	246	120
Trace2	270	154	81
Trace3	222	124	72

Table 1: Experimental results on three traces of 10GB (307,200 partitions of 32KB) each.

Based on these results, we choose to predefine knots at fixed intervals of say 100 partitions (as in Figure 2). This simplifies knot representation and speeds-up the search to locate the knot interval for a given virtual address. In the following we will refer to such a fixed knot interval (in the virtual space) as a subsegment, and assume that the physical storage is allocated to such subsegments starting from address 0 (this is enabled by the coarse virtualization mechanism in Section 2.1). As we said above, as result of the linear interpolation, all partitions in a subsegment share the same slope (CR) and the starting knot tells where the compressed subsegment (i.e. the *subextent*)is stored on disk storage. By extension, we name knot the in memory data structure holding the information needed in interpolation, (total less than 10 bytes per knot) as follows:

- extent ID 5 bits, (if a segment has assigned 32 extents)
- offset in extent of a 4KB (block): 17 bits (extent size = 10GB/32 = 320MB = $2^{17} * 4KB$)

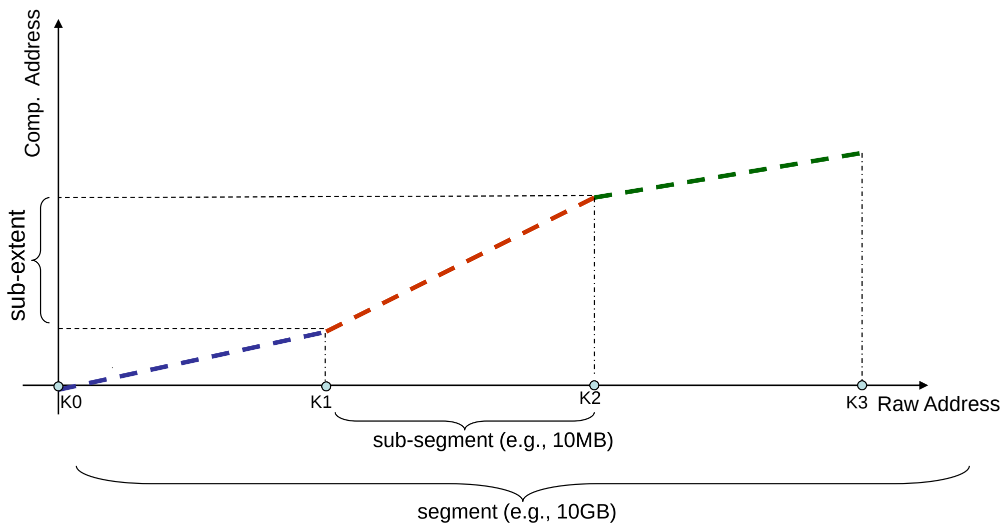

Figure 2: Mapping virtual space segments / subsegments to physical space extents / sub-extents.

- sub-segment slope (i.e., CR): 7 bits + 2*4 bits (slope, min_slope, max_slope)
- one of say 4 error windows: 2 bits
- empty sub-segment: 1 bit

If we use say 10 bytes for each subsegment of 10MB, for a 100TB system we need 100MB memory. Therefore linear interpolation is effective in reducing the amount of metadata needed to map from raw domain to compressed (physical) domain.

5 Read Operation

The fast online algorithm above, is a good solution for the case of sequential initial writes into a segment. By guaranteeing an error window it allows random reads of any amount l bytes as long as the amount read is larger by one error window (w) at each end (so we read $l+2*w$ bytes) as shown in Figure 3. Compressed partitions also have a header with additional information that includes the (virtual) address of the (uncompressed) data in the partition, and compressed partitions are placed in the extents in increasing order of their virtual address and the free space between partitions is zeroed to ease retrieval; this way it is easy to find the relevant compressed partition between the cached partitions after read.

6 Quasi-Sequential and Random Writes in New Sub-segments

For the case when some of the partitions are not written in sequential order, we can still apply a slightly modified version of the online algorithm. Basically, if we have a small sequence of partitions that were written in sequential order, we can estimate their

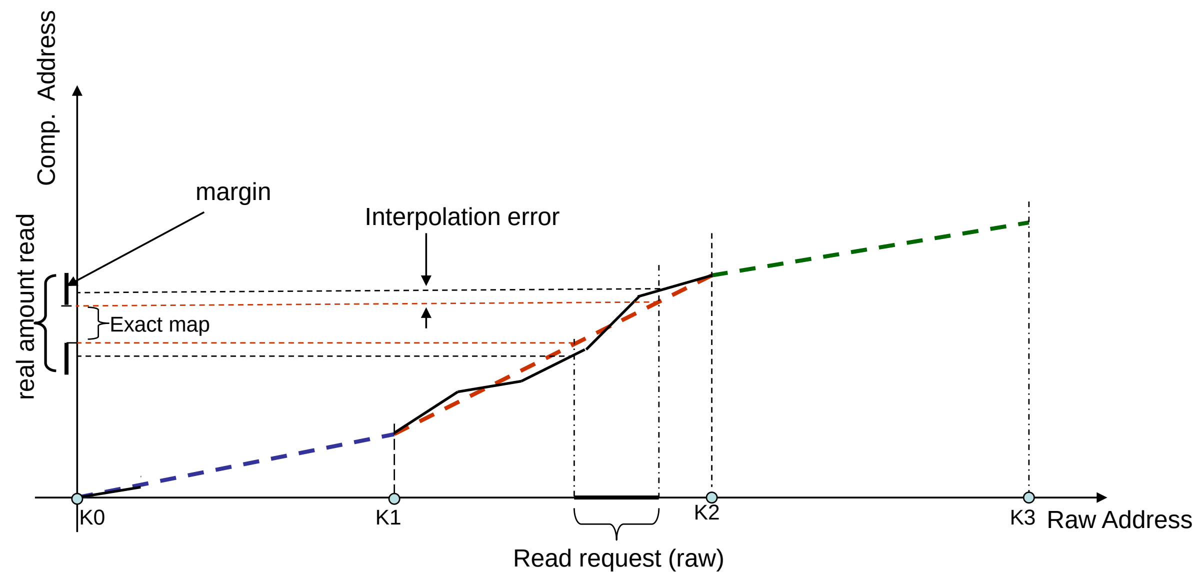

Figure 3: Read Operation

aggregate slope (CR) as well as the minimum and maximum slopes, as we did in the online algorithm above. Then, to write the compressed partitions that are not adjacent with the ones already placed, we use the value of the aggregate slope to map their virtual space offset into an offset in physical extent (compressed domain): offset_in_compressed_extent = offset_in_raw_segment * CR.

We can quite reliably extend the slope computed for the first few partitions to the entire subsegment (of 100 partitions) based on our experimental results in estimating the compression ratio for the subsegments using just one or few partitions. The following chart in Figure 4 is a scatter plot that illustrate the goodness of estimating the compression ratios for subsegments (knot intervals) of 100 partitions using only the first 4 partitions, for Trace1. Each trace has 3,072 subsegments (of 100 partitions each).

The standard error of the estimation of CR (that is the standard deviation of the residuals after linear interpolation) and the R^2 (the square of the correlation coefficient, indicating the strength of the linear relationship) are summarized in Table 2.

	Standard error (s)	**R_square**
Trace1	0.09	95%
Trace2	0.1	75%
Trace3	0.11	88%

Table 2: Experimental results on three traces of 10GB (307,200 partitions of 32KB) each.

If some subsegments contain partitions with somewhat wider variation in CR we can use just a couple bits of metadata (in the knots) to specify a larger or smaller error window. For example: 0,1,2,and 3 can indicate the use of an error window of 32 KB, 64 KB, 128 KB or 512 KB.

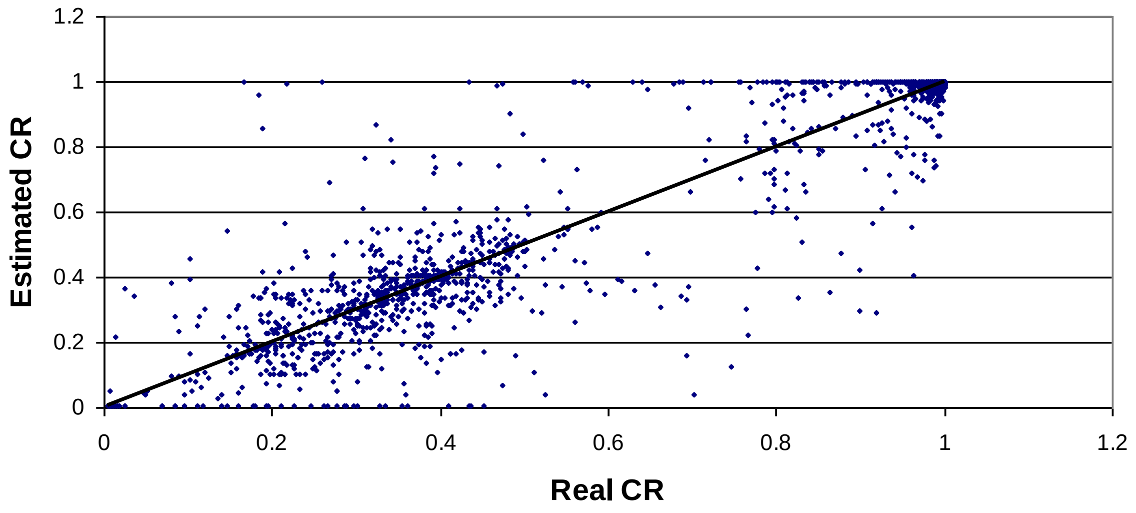

Figure 4: Precision of CR estimation for subsegments of 100 partitions using only the first 4 partitions for Trace1 ($y = 0.9962x + 0.0025$ and $R^2 = 0.9548$)

6.1 Random writes of isolated partitions

If the workload consists mostly in random writes of isolated partitions into empty subsegments, we can still estimate the compression ratio (slope) for the subsegment as above, place tentatively the compressed partition(s) in extent at the mapped offset(s), compute the lower and upper bounds for the slope given the interpolation error window and eventually adjust the placement of the partition(s) to meet the upper and lower slope bounds, partition(s) being still in cache. One interesting experiment we performed for this random placement case was to place the compressed partitions exactly at the mapped offset in extent, but to relax (increase) a bit the estimated slope (CR): offset_in_compressed_extent = offset_in_raw_segment * (CR + slack), where slack is 2% - 8%. Note that in this case the error_window = 0, but by relaxing the CR we basically leave some slack (space) between compressed partitions. In the experiment we shuffled the order of compressed partitions placement in subsegments and mapped them (individually) using a slack of 5% added to the estimated CR.

	Real_CR	Estim_CR+slack	Perfect_fit_subsegments	Total_subsegments
trace1	0.57	0.64	2204	3072
trace2	0.2	0.27	1413	3072
trace3	0.46	0.54	1276	3072

Table 3: Random write of isolated partitions into new subsegments. Each trace (of approx. 10 GB) was partitioned into about 3072 subsegments of 100 partitions each (partition size is 32KB). The compression ratio estimated on the first 4 partitions (while still in cache) to be placed in each subsegment (estim_CR) was extrapolated to entire subsegment. A slack of 5% was added to the estim_CR and used to write (map) each of the 100 partitions, one partition at a time, in random order. A "perfect fit" subsegment is one with no collisions between randomly written partitions (no step on each other even by 1 byte).

Table 3 show that for Trace1 more than 2/3 of the subsegments (of 100 partitions each) are filled with no collision ("perfectly" - with a small slack) and for the other 2 traces more than 1/3 of subsegments are perfectly filled. In the subsegments that are not perfectly filled the collisions usually come in contiguous chains, indicating that the content changed significantly. Note that in these experiments we report that two compressed partitions collide if they overlap by at least one byte. Also, no additional knots were added, each subsegment has exactly 100 partitions.

In a real implementation we can take advantage of the slack between compressed partitions so we can shift them not to overlap as long as their placement is satisfying the error_window.

The lesson of this experiment is that leaving a small slack in CR smoothes out the noise (small differences) between individual partition compression ratios many subsegments being perfectly filled; however, if subsegments contain a subset of partitions with widely different CRs (like a big JPEG image in the middle of a text subsegment) than leaving a bit of slack (space) in the placement won't smooth out the difference and this case is a candidate for "copying" the group of partitions (representing the JPEG image, for example) into a different extent.

7 Mechanisms to maintain a good (readable) interpolation

We described the initial placement (write) of compressed partitions into extents for various writing scenarios: sequential, quasi-sequential and random. When one or a subset of partitions do not fit in the corresponding area in extent, as required by linear mapping and the error_window, we can do few things:

- Add light weight knots that can be of two types: (1) indicate a change in the interpolation slope and or in the acceptable error threshold, or (2) indicate a change in the target extent.
- Place a redirection record on disk to a location in a different extent than the one used in current subsegment mapping. This won't add any in memory metadata but will require another read access too get to the compressed partitions.
- Migrate the entire subsegment to a new location.
- Leave some slack (free space) when we first place the compressed partitions. One way to achieve this is to slightly decrease the estimated compression ration (or slope) when we first place the partitions. Or we can leave a constant amount of free space. Either way the free space has to be zeroed to be detected when we read a set of compressed partitions. The header of compressed partition contains their IDs (raw address) so we can locate them if they are in the allowed read window.

8 Conclusions and Future Work

We introduced a new way of adding compression in primary storage systems where very little metadata is needed and it is kept in memory. The method relies on approximating the compressed data placement on disk and seems especially simple for object stores /

write once storage applications. We are investigating how efficient it could be in general read / write applications.

References

[1] Cornel Constantinescu and Maohua Lu. Quick estimation of data compression and deduplication for large storage systems. In *CCP-2011*, pages 98–102. IEEE Computer Society, 2011.

[2] Danny Harnik, Ronen Kat, Dmitry Sotnikov, Avishay Traeger, and Oded Margalit. To zip or not to zip: Effective resource usage for real-time compression. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 229–241, San Jose, CA, 2013. USENIX.