Dynamic Virtual Chunks: On Supporting Efficient Accesses to Compressed Scientific Data

Dongfang Zhao, Kan Qiao, Jian Yin, Ioan Raicu

Abstract—Data compression could ameliorate the I/O pressure of data-intensive scientific applications. Unfortunately, the conventional wisdom of naively applying data compression to the file or block brings the dilemma between efficient random accesses and high compression ratios. File-level compression barely supports efficient random accesses to the compressed data: any retrieval request need trigger the decompression from the beginning of the compressed file. Block-level compression provides flexible random accesses to the compressed blocks, but introduces extra overhead when applying the compressor to each and every block that results in a degraded overall compression ratio. This paper extends our prior work that introduces virtual chunks offering efficient random accesses to the compressed scientific data without sacrificing the compression ratio. Virtual chunks are logical blocks pointed at by appended references without breaking the physical continuity of the file content. These references allow the decompression to start from an arbitrary position (efficient random accesses), while no per-block overhead is introduced because the file's physical entirety is retained (high compression ratio). One limitation of virtual chunk is it only supports static references. This paper presents the algorithms, analysis, and evaluations of dynamic virtual chunks to deal with the cases where the references are updated dynamically.

Keywords—File compression, distributed file systems, parallel file systems, big data, data-intensive computing, scientific computing

1 Introduction

As modern scientific applications are becoming data-intensive [1, 2], one effective approach to relieve the I/O bottleneck of the underlying storage system is data compression. For instance, it is optional to apply lossless compressors (e.g. LZO [3], bzip2 [4]) to the input or output files in the Hadoop file system (HDFS) [5], or even lossy compressors [6, 7] at the high-level I/O middleware such as HDF5 [8], NetCDF [9], and GRIB [10]. HDF was originally developed as a general set of file formats at the National Center for Supercomputing Applications (NCSA). NetCDF is a set of self-described libraries for data formats that is hosted at the University Corporation for Atmospheric Research (UCAR). GRIB is the de facto data format for meteorology that is standardized by the World Meteorological Organization (WMO). By investing computational time on compression and decompression, we hope to significantly reduce the file size and consequently the I/O time to offset the computational cost.

State-of-the-art compression mechanisms of parallel and distributed file systems apply the compressor to the data either at the file-level or block-level[1], and leave important factors (e.g. computational overhead, compression ratio, workload's I/O pattern) to the underlying compression algorithms. Such approaches have the following limitations.

First, file-level compression is criticized for its significant overhead for random accesses: the decompression needs to start from the very beginning of the compressed file even though the client might only be requesting a few bytes at an arbitrary position in the file. As a case in point, one of the most commonly used operations in climate research is to retrieve the latest temperature of a particular location. The compressed data set is typically in terms of hundreds of gigabytes; nevertheless scientists would need to decompress the entire compressed file to only access the last temperature reading. This wastes both the scientist's valuable time and scarce computing resources.

Second, the deficiency of block-level compression stems from its additional compression overhead on each block; the overall overhead aggregated from many blocks in a large file easily exceeds the file-level-compression counterpart. To see this, think about a simple scenario that a 64MB (the default chunk size in HDFS [5]) file to be compressed with 4:1 ratio and 4KB overhead (e.g. header, metadata, and so forth). The resultant compressed file (i.e. after file-level compression) is

$$\frac{64\text{MB}}{4} + 4\text{KB} = 16.004\text{MB}$$

If the file is split into 64KB-blocks each of which is

- *D. Zhao and K. Qiao are with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL, 60616. Email: {dzhao8, kqiao}@iit.edu*
- *J. Yin is with the Division of Mathematics & Computer Science, Pacific Northwest National Laboratory, Richland, WA 99354. Email: jian.yin@pnnl.gov*
- *I. Raicu is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, as well as the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. Email: iraicu@cs.iit.edu*

1. The "chunk", e.g. in HDFS, is really a file from the work node's perspective. So "chunk-level" is not listed here.

applied with the same compressor, the compressed file would be (assuming the overhead is also 4KB)

$$\frac{64\text{MB}}{4} + 4\text{KB} \times 1\text{K} = 20\text{MB}$$

Therefore we spend

$$\frac{(20\text{MB} - 16.004\text{MB})}{16.004\text{MB}} \approx 25\%$$

more space in block-level compression.

This paper introduces *virtual chunks* (VC) that aim to better employ existing compression algorithms in parallel and distributed file systems, and to eventually improve the I/O performance of random data accesses in scientific applications and high-performance computing (HPC) systems. The key idea of virtual chunks is: we do not break the original file into physical chunks or blocks, but append a small number of references to the end of file. Each of these references points to a specific block that is considered as a boundary of the virtual chunk. Because the physical entirety (or block continuity) of the original file is retained, the compression overhead and compression ratio keep comparable to those of file-level compression. With these additional references, a random file access need not decompress the entire file from the beginning, but could arbitrarily jump onto a reference close to the requested data and start the decompression from there. Therefore, virtual chunks achieve the best of both file- and block-level compression: high compression ratio and efficient random access.

Virtual chunks might raise concerns about the cost of the additional references: they are not free, and take extra storage space, hurting both the overall compression ratio and the end-to-end I/O time (at least when writing files with VC enabled). We argue, and will justify in Section 2.3 and Section 3.1, that it would not be an issue if the number of additional reference is wisely chosen. It would definitely not be a good choice to store a reference for each and every original data entry: the resultant "compressed file" would become significantly larger than its original size, making data compression meaningless. Few references do not make sense either since the whole point of virtual chunks is to provide a finer granularity for efficient random accesses; as an extreme example, a single reference makes VC-compression degenerate to simple file-level compression. Therefore, the number of additional references must be balanced between compression ratio and compression granularity (i.e. size of virtual chunks, or number of references). We will present theoretical analyses (Section 2.3) and experimental results (Section 3.1) to justify that the space overhead from the additional reference is negligible in terms of end-to-end I/O performance.

This paper is an extension of our prior work [11, 12] that advances the understanding of the system support for data-intensive scientific applications in the following perspectives:

- *Proposal of virtual chunk mechanism to flexibly apply the conventional compression algorithm to parallel and distributed file systems to improve random data accesses while retaining high compression ratios*
- *Design of procedures to manipulate virtual chunks, and provide theoretical analysis on how to set up the parameters to achieve the optimal performance*
- *Implementation of virtual chunks in a production parallel file system GPFS [13] and a distributed file system FusionFS [14, 15]*
- *Evaluations of virtual chunks with real-world scientific data (e.g. GCRM [16], SDSS [17]) at large scale, on up to 1024 cores on a leadership-class supercomputer (Intrepid [18])*

One limitation of equidistant virtual chunks is that it only supports static references, which is not be always the case in practice. To this end, this paper extends static VC to a more general form, namely dynamic VC. In other words, this journal extension makes VC more practical without the constraint of equidistant references (i.e., uniformly distributed).

If the I/O pattern is uniformly distributed, dynamic VC is essentially degraded to static VC. All the study of static VC is also applicable to dynamic VC, as long as the I/O distribution is uniform. However, it is obviously more desirable to allow users to be able to specify an arbitrary distribution. Therefore, in addition to the study on static virtual chunks this paper makes more contributions from the angle of supporting dynamic changes to virtual chunks:

- *We devise the procedures to update virtual chunks (Section 2.6)*
- *We analyze the benefit of dynamic virtual chunks over the original static version (Section 2.7)*
- *We evaluate dynamic virtual chunks in a large parameter space (Section 3.4)*

The remainder of this paper mainly focuses on the presentation and evaluation of employing virtual chunks in compressible file systems. In Section 2, we analyze how to wisely choose the number of references to append, discuss where to store these references, formalize the procedures to leverage virtual chunks in sequential and random data accesses, describe how to update virtual chunks, and analyze the potential I/O improvement of dynamic virtual chunks. Section 3 describes two forms of implementation of virtual chunks: a middleware on a parallel file system (GPFS [13]) and a built-in module in a distributed file system (FusionFS [14, 15]). Large-scale experiments at up to 1,024-cores show that virtual chunks improve I/O performance by up to 2X on real-world scientific applications, such as climate data GCRM [16] and astronomy data SDSS [17]. In Section 4, we discuss the limitations and open questions of virtual chunks. Section 5 reviews previous work on storage systems, I/O performance, and data compression. We finally conclude this paper in Section 6.

2 VIRTUAL CHUNKING

Virtual chunking is applicable as long as the compression algorithm is splittable. To make matters more concrete, we illustrate how virtual chunks work with an splittable XOR-based delta compression [19] that is applied to parallel scientific applications. The idea of XOR-based delta compression is straightforward: calculating the XOR difference between every pair of adjacent data entries in the input file, so that only the very first data entry needs to be stored together with the XOR differences. This XOR compression proves to be highly effective for scientific data like climate temperatures, because the large volume of numerical values change marginally in the neighboring spatial and temporal area. Therefore, storing the large number of small XOR differences instead of the original data entries could significantly shrink the size of the compressed file.

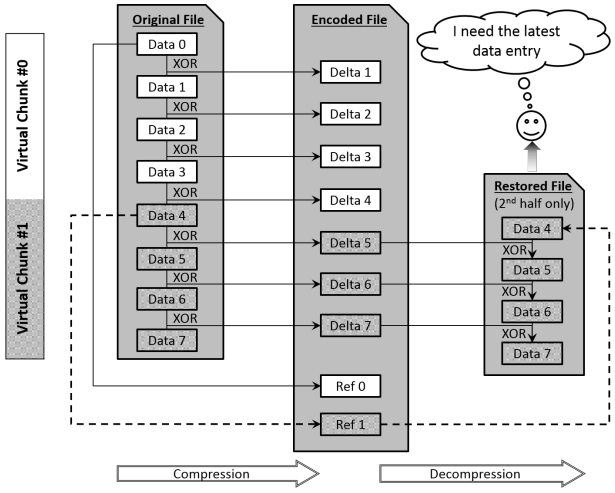


Fig. 1. Compression and decompression with two virtual chunks

Figure 1 shows an original file of eight data entries, and two references to Data 0 and Data 4. That is, we have two virtual chunks of Data 0 – 3 and Data 4 – 7, respectively. In the compressed file, we store seven deltas and two references. When users need to read Data 7, we first copy the nearest upper reference (Ref 1 in this case) to the beginning of the restored file, then incrementally XOR the restored data and the deltas, until we reach the end position of the requested data. In this example, we roughly save half of the I/O time during the random file read by avoiding reading and decompressing the first half of the file. It should be noted that this is an oversimplified example to only illustrate the key idea of our proposed approach; in practice, data could be significantly more complicated.

For clear presentation of the following algorithms to be discussed, we assume the original file data can be represented as a list $D = \langle d_1, d_2, \ldots, d_n \rangle$. Since there are n data entries, we have $n-1$ encoded data elements, denoted by the list $X = \langle x_1, x_2, \ldots, x_{n-1} \rangle$ where

$$x_i = d_i \texttt{ XOR } d_{i+1}\texttt{, for } 1 \leq i \leq n-1$$

We assume there are k references (i.e. original data entries) that the k virtual chunks start with. The k references are represented by a list $D' = \langle d_{c_1}, d_{c_2}, \ldots, d_{c_k} \rangle$, where for any $1 \leq i \leq k-1$ we have $c_i \leq c_{i+1}$. Notice that we need $c_1 = 1$, because it is the basis from where the XOR could be applied to the original data D. We define $L = \frac{n}{k}$, the length of a virtual chunk if the references are equidistant. The number in the pair of square brackets `[]` after a list variable indicates the index of the scalar element. For example $D'[i]$ denotes the i^{th} reference in the reference list D'. This should not be confused with d_{c_i}, which represents the c_i-th element in the original data list D. The sublist starting at s and ending at t of a list D is represented as $D_{s,t}$.

In the following sections, we will discuss the design tradeoff and some analytics of virtual chunking. If not otherwise stated, we assume virtual chunks are distributed in an equidistant manner, i.e. static virtual chunks. Section 2.6 and Section 2.7 will particularly study dynamic virtual chunks where the reference could be positioned arbitrarily.

2.1 Storing References

We have considered two strategies on where to store the references: (1) put all references together (either in the beginning or in the end); (2) keep the reference in-place to indicate the boundary, i.e. spread out the references in the compressed file. Current design takes the first strategy that stores the references together at the end of the compressed file, as explained in the following.

The in-place references offer two limited benefits. Firstly, it saves space of $(k-1)$ encoded data entries (recall that k is the total number of references). For example, Delta 4 would not be needed in Figure 1. Secondly, it avoids the computation on locating the lowest upper reference at the end of the compressed file. For the first benefit, the space saving is insignificant because encoded data are typically much smaller than the original ones, not to mention this gain is factored by a relatively small number of references $(k-1)$ compared with the total number of data entries (n). The second benefit on saving the computation time is also limited because the CPU time on locating the reference is marginal compared to compressing the data entries.

The in-place method has a critical drawback that may not be so obvious: it introduces significant overhead when decompressing a large portion of data spanning over multiple logical chunks. To see this, let us imagine in Figure 1 that Ref 0 is above Delta 1 and Ref 1 is in the place of Delta 4. If the user requests the entire file, then the file system needs to read two raw data entries: Ref 0 (i.e. Data 0) and Ref 1 (i.e. Data 4). Note that Data 0 and Data 4 are original data entries, and are typically much larger than the deltas. Thus, reading these

in-place references would take significantly more time than reading the deltas, especially when the requested data include a large number of virtual chunks. This issue does not exist in our current design where all references are stored together at the end of file: the user only needs to retrieve one reference (i.e. Ref 0 in this case).

2.2 Compression with VC

We assume the underlying compression algorithm used by VC (for example, XOR compression) is comprised of a pass of processing two consecutive data points. This is not always the case, yet greatly simplifies the mathematical notation in the following discussion whose validity is not affected by this simplification. We will discuss the applicability of virtual chunks in more details in Section 4.1. In the rest of this section we will use `encode()` and `decode()` to indicate the compression and decompression of two consecutive data points, respectively.

The procedure to compress a file with multiple references is described in Algorithm 1. The first phase of the virtual-chunk compression is to encode the data entries of the original file, as shown in Lines 1 – 3. The second phase appends k references to the end of the compressed file, as shown in Lines 4 – 6.

Algorithm 1 VC Compress

Input: The original data $D = \langle d_1, \cdots, d_n \rangle$
Output: The encoded data X, and the reference list D'
1: **for** (int i = 1; i < n; i++) **do**
2: $X[i] \leftarrow$ `encode(`d_i, d_{i+1}`)`
3: **end for**
4: **for** (int j = 1; j < k; j++) **do**
5: $D'[j] \leftarrow D[1 + (j-1) * L]$
6: **end for**

The time complexity of Algorithm 1 is $O(n)$. Lines 1 – 3 obviously take $O(n)$ to compress the file. Lines 4 – 6 are also bounded by $O(n)$ since there cannot be more than n references in the procedure.

2.3 Optimal Number of References

The current section answers this question: how many references should we append to the compressed file, in order to maximize the end-to-end I/O performance?

In general, more references consume more storage space, implying longer time to write the compressed data to storage. As an extreme example, making a reference to each data entry of the original file is not a good idea: the resulted compressed file is actually larger than the original file. On the other hand, however, more references yield a better chance of a closer lowest upper reference from the requested data, which in turn speeds up the decompression for random accesses. Thus, we want to find the number of references that has a good balance between compression and decompression, and ultimately achieves the minimal overall time.

Despite many possible access patterns and scenarios, in this paper we are particularly interested in finding the number of references that results in the minimal I/O time in the worst case: for data write, the entire file is compressed and written to the disk; for data read, the last data entry is requested. That is, the decompression starts from the beginning of the file and processes until the last data entry. The following analysis is focused on this scenario, and assumes the references are equidistant.

A few more parameters for the analysis are listed in Table 1. We denote the read and the write bandwidth for the underlying file system by B_r and B_w, respectively. Different weights are assigned to input W_i and output W_o to reflect the access patterns. For example if a file is written once and then read for 10 times in an application, then it makes sense to assign more weight to the file read (W_i) than the file write (W_o). S indicates the size of the original file to be compressed. R is the compression ratio, so the compressed file size is $\frac{S}{R}$. D denotes the computational time spent on decompressing the requested data, which should be distinguished from the overall decompression time (D plus the I/O time).

TABLE 1
Virtual chunk parameters

Variable	Description
B_r	Read Bandwidth
B_w	Write Bandwidth
W_i	Weight of File Read
W_o	Weight of File Write
S	Original File Size
R	Compression Ratio
D	Computational Time of Decompression

The overall time difference between conventional single-reference compression and our proposed multi-reference compression comes from only the I/O but not the computation. This is because the computational time spent in compression is independent on the number of references. The assumption is that when comparing both cases, we need to apply the same compression algorithm that takes the same computation time.

Let t_c indicate the time difference between multiple references and a single reference, we have

$$t_c = \frac{(k-1) \cdot S \cdot W_o}{n \cdot B_w} \tag{1}$$

where k denotes the number of references and n denotes the total number of data points, respectively.

Similarly, to calculate the potential gain during decompression with multiple references, t_d indicating the time difference in decompression between multiple references

and a single reference, is calculated as follows:

$$t_d = \frac{(k-1)\cdot S \cdot W_i}{k \cdot R \cdot B_r} + \frac{(k-1)\cdot D \cdot W_i}{k} \tag{2}$$

The first term of the above equation represents the time difference on the I/O part, and the second term represents the computational part.

To minimize the overall end-to-end I/O time, we want to maximize the following function (i.e. gain minus cost):

$$F(k) = t_d - t_c \tag{3}$$

Note that the I/O time is from the client's (or, user's) perspective. Technically, it includes both the computational and I/O time of the (de)compression. By taking the derivative on k (suppose $\hat{k}$ is continuous) and solving the following equation

$$\frac{d}{d\hat{k}}(F(\hat{k})) = \frac{S \cdot W_i}{R \cdot B_r \cdot \hat{k}^2} + \frac{D \cdot W_i}{\hat{k}^2} - \frac{S \cdot W_o}{B_w \cdot n} = 0, \tag{4}$$

we have

$$\hat{k} = \sqrt{n \cdot \frac{B_w}{B_r} \cdot \frac{W_i}{W_o} \cdot (\frac{1}{R} + \frac{D \cdot B_r}{S})} \tag{5}$$

To make sure $\hat{k}$ reaches the global maximum, we can take the second-order derivative on $\hat{k}$:

$$\frac{d^2}{d\hat{k}^2}(F(\hat{k})) = -\frac{S \cdot W_i}{R \cdot B_r \cdot \hat{k}^3} - \frac{D \cdot W_i}{\hat{k}^3} < 0 \tag{6}$$

since all parameters are positive real numbers. Because the second-order derivative is always negative, we are guaranteed that the local optimal $\hat{k}$ is really a global maximum.

Since k is an integer, the optimal k is given as:

$$\arg\max_k F(k) = \begin{cases} \lfloor \hat{k} \rfloor & \text{if } F(\lfloor \hat{k} \rfloor) > F(\lceil \hat{k} \rceil) \\ \lceil \hat{k} \rceil & \text{otherwise} \end{cases} \tag{7}$$

where $\lfloor x \rfloor$ denotes the largest previous integer of x and $\lceil x \rceil$ denotes the smallest following integer of x, respectively.

Therefore the optimal number of references k_{opt} is:

$$k_{opt} = \begin{cases} \lfloor \hat{k} \rfloor & \text{if } F(\lfloor \hat{k} \rfloor) > F(\lceil \hat{k} \rceil) \\ \lceil \hat{k} \rceil & \text{otherwise} \end{cases} \tag{8}$$

where

$$\hat{k} = \sqrt{n \cdot \frac{B_w}{B_r} \cdot \frac{W_i}{W_o} \cdot (\frac{1}{R} + \frac{D \cdot B_r}{S})} \tag{9}$$

and

$$F(x) = \frac{(x-1)\cdot S \cdot W_i}{x \cdot R \cdot B_r} + \frac{(x-1)\cdot D \cdot W_i}{x} - \frac{(x-1)\cdot S \cdot W_o}{n \cdot B_w} \tag{10}$$

Note that the last term $\frac{D \cdot B_r}{S}$ in Eq. 9 really says the ratio of D over $\frac{S}{B_r}$. That is, the ratio of the computational time over the I/O time. If we assume the computational portion during decompression is significantly smaller than the I/O time (i.e. $\frac{D \cdot B_r}{S} \approx 0$), the compression ratio is not extremely high (i.e. $\frac{1}{R} \approx 1$), the read and write throughput are comparable (i.e. $\frac{B_w}{B_r} \approx 1$), and the input and output weight are comparable (i.e. $\frac{W_i}{W_o} \approx 1$), then a simplified version of Eq. 9 can be stated as:

$$\hat{k} = \sqrt{n} \tag{11}$$

suggesting that the optimal number of references be roughly the square root of the total number of data entries.

2.4 Random Read

This section presents the decompression procedure when a request of random read comes in. Before that, we describe a subroutine that is useful for the decompression procedure and more procedures to be discussed in later sections. The subroutine is presented in Algorithm 2, called `DecompList`. It is not surprising for this algorithm to have inputs such as encoded data X, and the starting and ending positions (s and t) of the requested range, while the latest reference no later than s (i.e. $d_{s'}$) might be less intuitive. In fact, $d_{s'}$ is not supposed to be specified from a direct input, but calculated in an ad-hoc manner for different scenarios. We will see this in the complete procedure for random read later in this section.

Algorithm 2 DecompList

Input: The start position s, the end position t, the latest reference no later than s as $d_{s'}$, the encoded data list $X = \langle x_1, x_2, \ldots, x_{n-1} \rangle$
Output: The original data between s and t as $D_{s,t}$

1: $prev \leftarrow d_{s'}$
2: **for** $i = s'$ to t **do**
3: **if** $i \geq s$ **then**
4: $D_{s,t}[i-s] \leftarrow prev$
5: **end if**
6: $prev \leftarrow$ `encode`$(prev, x_i)$
7: **end for**

In Algorithm 2, Line 1 stores the reference in a temporary variable as a base value. Then Lines 2 – 7 decompress the data by increasingly applying the decode function between the previous original value and the current encoded value. If the decompressed value lands in the requested range, it is also stored in the return list.

Now we are ready to describe the random read procedure to read an arbitrary data entry from the compressed file. Recall that in static virtual chunks, all references are equidistant. Therefore, given the start position s we could calculate its closest and latest reference index $s' =$ `LastRef`(s) where :

$$\texttt{LastRef}(x) = \begin{cases} \frac{x}{L} + 1 & \text{if } 0 \neq x \texttt{ MOD } L \\ \frac{x}{L} & \text{otherwise} \end{cases} \tag{12}$$

So we only need to plug Eq. 12 to Algorithm 2. Also

note that we only use Algorithm 2 to retrieve a single data point, therefore we can set $t = s$ in the procedure.

The time complexity of random read is $O(L)$, since it needs to decompress as much as a virtual chunk to retrieve the requested data entry. If a batch of read requests comes in, a preprocessing step (e.g. sorting the positions to be read) can be applied so that decompressing a virtual chunk would serve multiple requests.

It should be clear that the above discussion assumes the references are equidistant, i.e. static virtual chunks. And that is why we could easily calculate s' by Eq. 12. It needs a more complex procedure for dynamic references that will be discussed in Section 2.6.

2.5 Random Write

The procedure of random write (i.e. modify a random data entry) is more complicated than the case of random read. In fact, the first step of random write is to locate the affected virtual chunk, which shares a similar procedure of random read. Then the original value of the to-be-modified data entry is restored from the starting reference of the virtual chunk. In general, two encoded values need to be updated: the requested data entry and the one after it. There are two trivial cases when the updated data entry is the first or the last. If the requested data entry is the first one of the file, we only need to update the first reference and the encoded data after it. This is because the first data entry always serves as the first reference as well. If the requested data entry is the last one of the file, then we just load the last reference and decode the virtual chunk till the end of file. In the following discussion, we consider the general case excluding the above two scenarios. Note that, if the requested data entry happens to be a reference, it needs to be updated as well with the new value.

For XOR-based delta compression, modifying one original data point changes two deltas in the compressed file: (1) the delta between the modified data point and the one before it, and (2) the delta between the modified data point and the one after it. The procedure of updating an arbitrary data point is described in Algorithm 3. The latest reference no later than the updated position q is calculated in Line 1, per Eq. 12. Then Line 2 reuses Algorithm 2 to restore three original data entries in the original file. They include the data entry to be modified, and the two adjacent ones to it. Line 3 and Line 4 re-compress this range with the new value v. Lines 5 – 7 check if the modified value happens to be one of the references. If so, the reference is updated as well.

The time complexity is $O(L)$, since all lines take constant time, except that Line 2 takes $O(L)$. If there are multiple update requests to the file, i.e. batch of requests, we can sort the requests so that one single pass of restoring a virtual chunk could potentially update multiple data entries being requested.

Algorithm 3 VC Write

Input: The index of the data entry to be modified q, the new value v, encoded data $X = \langle x_1, x_2, \cdots, x_{n-1} \rangle$, and the reference list $D' = \langle d_1, d_2, \cdots, d_k \rangle$
Output: Modified X

1: $s' \leftarrow \texttt{LastRef}(q)$
2: $\langle d_{q-1}, d_q, d_{q+1} \rangle \leftarrow \texttt{DecompList}(q-1, q+1, d_{s'}, X)$
3: $x_{q-1} \leftarrow \texttt{encode}(d_{q-1}, v)$
4: $x_q \leftarrow \texttt{encode}(v, d_{q+1})$
5: **if** $0 = (q-1)$ $\texttt{MOD}$ L **then**
6: $\quad D'[\frac{q}{L} + 1] \leftarrow v$
7: **end if**

2.6 Updating VC

If the access pattern does not follow the uniform distribution, and this information is exposed to users, then it makes sense to specify more references (i.e. finer granularity of virtual chunks) for the subset that is more frequently accessed. This is because more references make random accesses more efficient with a shorter distance (and less computation) from the closest reference, in general. The assumption of equidistant reference, thus, does not hold any more in the following discussion.

While a self-adjustable mechanism to update the reference positions is ongoing at this point, this paper expects that the users would specify the distribution of the reference density in a configuration file, or more likely a rule such as a decay function [20]. For those users who are really familiar with their data, a function that adjusts any particular range of data with an arbitrary number of references is also desirable. That is, the second type of users would need to access a lower level of reference manipulations. Note that, the specifications and distributions required by the first type of users could be implemented by the functions for the second type of users. Therefore, we decide to expose the interface to allow users (i.e. the second type of users) to control the finer granularity of reference adjustment. It should be fairly straightforward for the first type of users to meet their needs by extending the provided interfaces.

Before discussing the procedure to update the references, we will first describe some auxiliary functions. The `FindRef` function finds the latest reference no later than the given data index. It takes two inputs: the list of references and a data index, then applies a binary search to return the closest reference that is not later than the input data index. Since this only trivially extends the standard binary search, we do not give the formal algorithm in this paper. This procedure takes $O(\log k)$ time, where k is the list length of all the references. Then we define the `FindSublist` function that extends `FindRef` with two input data indexes s and t such that $1 \leq s \leq t \leq n$ and the return value as a sublist $D'_{s',t'}$ such that $s' = \texttt{FindRef}(D', s)$ and $t' = \texttt{FindRef}(D', t)$.

To make the virtual chunks adjustable we will introduce the `RefUpdate` procedure that allows users to

specify a linear transform of the existing virtual chunks within a particular range. Not surprisingly, this procedure requires more computation and possibly more parsing time if the updating rules are specified in a user-defined configuration file. This tradeoff between performance and flexibility is highly application-dependent. Thanks to the `RefUpdate` procedure, it is relatively straightforward to extend the file operations described in the static virtual chunks to their dynamic parities.

We assume there are m disjoint subsets of D that will be updated with a new number of references. Users are expected to specify the following parameters: the starting and ending position of a subset (s_i, t_i), as well as the coefficients in the linear transform α_i and β_i, where $1 \leq i \leq m$. Note that both s_i and t_i are the distances from the beginning of D where $1 \leq s_i < t_i \leq n$.

In the updating procedure, a sublist of D', namely $D'' = \langle d_{c_b^i}, d_{c_{b+1}^i}, \ldots, d_{c_e^i} \rangle$ is affected when the granularity within this range is updated. Note that c_b^i should be the immediate precedent of s_i, and c_e^i should be the immediate precedent of e_i. That is, there does not exist such a b' that $b' > b$ and $c_{b'}^i \leq s_i$; and there does not exist such an e' that $e' > e$ and $c_{e'}^i \leq t_i$. α is a float number meaning that we want α times as many as the original number of virtual chunks between s_i and t_i. β is the constant adjustment in the linear transform. We also compute a sublist D^* where $|D_i^*| = \alpha_i |D_i''| + \beta_i$, which will replace D_i'' and then be inserted into D'. The list $\overline{D_i} = d_{c_b^i}, d_{c_b^i+1}, \ldots, d_{c_e^i}$ (i.e. another sublist of D) is also needed for the computation.

The procedure is presented in Algorithm 4. Lines 1 – 5 compute affected sublists of references D_i'' and original data entries $\overline{D_i}$ for all the m requested updates. Lines 6 – 13 calculate the values of the affected or newly added references in the compressed data. Line 14 updates the reference values.

Algorithm 4 RefUpdate

Input: For $1 \leq i \leq m$, (s_i, t_i), α_i, β_i, D', X.
Output: Modified D'.

1: **for** $i = 1$ to m **do**
2: $D_i'' \leftarrow$ `FindSublist`(s_i, t_i, D')
3: $s_i' \leftarrow$ `FindRef`(D', s_i)
4: $\overline{D_i} \leftarrow$ `DecompList`$(s_i, t_i, d_{s_i'}, X)$
5: **end for**
6: **for** $i = 1$ to m **do**
7: **for** $j = 0$ to $|\overline{D_i}| - 1$ **do**
8: $l \leftarrow \left\lfloor \frac{|\overline{D_i}|}{\alpha_i |D_i''| + \beta_i} \right\rfloor$
9: **if** $0 =$ j `MOD` l **then**
10: Add $d_{c_b^i + j}$ to D_i^*.
11: **end if**
12: **end for**
13: **end for**
14: Replace D_i'' by D_i^* in D' for $1 \leq i \leq m$.

The time complexity of Algorithm 4 is as follows. In m iterations, Lines 2 – 3 take $O(m \log k)$ in total. Line 4 takes at most $O(n)$ in m iterations because each interval (s_i, t_i) is disjoint to others. Similarly, Lines 6 – 13 take at most $O(n)$ in m iterations. Line 14 also takes at most $O(n)$. So the overall time complexity is $O(m \log k + n)$. In practice, Algorithm 4 would not be frequently called, because users normally do not need to adjust the virtual chunk granularity for every change to the data.

Once the references are updated, we cannot simply locate the reference by dividing the total number of data entries by the size of the virtual chunk as we did in the static case. However, with the help of the `FindRef` function, we can still retrieve the closest reference before the given data index by a binary search. For example, Line 1 of Algorithm 3 needs to be replaced by

$$s' \leftarrow \texttt{FindRef}(D', s)$$

if the references are not equidistant. Similarly, the `LastRef` function call in the random read procedure (Section 2.4) needs to be replaced by `FindRef`. The time complexity of the dynamic-reference algorithms (i.e. random read in Section 2.4 and random write in Section 2.5) is $O(L' + \log k)$, where L' indicates the size of the affected virtual chunk (not equidistant anymore) and $O(\log k)$ represents the time of `FindRef`.

If a subset is frequently accessed, the rule of thumb is to increase the reference density of this area. In this case, L' becomes small to indicate such fine granularity. It then implies that the overall complexity would not become significantly high even for frequent reference updates. So the random read and random write are still flexibly and efficiently maintained without much overhead compared to the static case. Section 2.7 will provide a detailed analysis on the potential I/O improvement by paying such overhead.

2.7 I/O Improvement from Dynamic VC

This section analyzes the I/O benefit from dynamic virtual chunks. As discussed in Section 2.6, updating static virtual chunks into dynamic ones introduces the overhead of adjusting the references (Algorithm 4). The goal of paying this overhead is to place more references to a frequently accessed subset of data.

To make a clear presentation, we make the following assumptions. Suppose n is dividable by k (i.e. n `MOD` $k = 0$), so that in the static setting all k virtual chunks are of the same size $L = \frac{n}{k}$. We assume there are two updates to the static references on $(1, \frac{n}{c})$ and $(\frac{n}{c} + 1, n)$, respectively, where c is a integer to control the boundary between the two portions. This c variable is supposed to be significantly larger than one (i.e. $c \gg 1$), and $\frac{1}{c} \ll \frac{c-1}{c}$. The first update has parameters α_1, β_1, $s_1 = 1, t_1 = \frac{n}{c}$, and the second one has parameters α_2, β_2, $s_2 = \frac{n}{c} + 1, t_2 = n$. To make the dynamic case comparable to the static virtual chunk, the total number of virtual

chunks after both updates is kept the same:

$$\alpha_1 \cdot \frac{k}{c} + \beta_1 + \alpha_2 \cdot \frac{(c-1)\cdot k}{c} + \beta_2 = k$$

On the other hand, we assume the smaller portion on $(1, \frac{n}{c})$ has a finer granularity of virtual chunks:

$$\alpha_1 \cdot \frac{k}{c} + \beta_1 \gg \alpha_2 \cdot \frac{(c-1)\cdot k}{c} + \beta_2$$

Finally, we assume there are f consecutive random I/Os to be applied to the portion on $(1, \frac{n}{c})$.

Without the two reference updates, the cost of f I/Os is simply $O(f \cdot L)$, since each I/O can take up to $O(L)$. Now we consider the dynamic case. By our previous analysis (Algorithm 4, Section 2.6), it takes $O(m \cdot \log k + n)$ to complete m updates if the references are already updated for dynamic virtual chunks. In this scenario, we only have two updates ($m = 2$) and the virtual chunks before the updates are static ($\log k \to 1$) since Line 3 of Algorithm 4 can be directly calculated by `LastRef`, and the total cost of the updates is just $O(n)$. Thus the total cost of f I/Os in dynamic virtual chunks is

$$O(n + f \cdot \log k + f \cdot L \cdot \frac{\frac{k}{c}}{\alpha_1 \cdot \frac{k}{c} + \beta_1})$$

Note that the cost of static virtual chunks is $O(f \cdot L)$. Therefore, to make the dynamic reference beneficial in terms of the overall I/O performance, we need

$$f \cdot L > n + f \cdot \log k + f \cdot L \frac{\frac{k}{c}}{\alpha_1 \cdot \frac{k}{c} + \beta_1}$$

or

$$L \cdot (1 - \frac{\frac{k}{c}}{\alpha_1 \cdot \frac{k}{c} + \beta_1}) - \log k > \frac{n}{f} \tag{13}$$

In practice, the condition in Eq. 13 is easy to satisfy. On the left hand side of Eq. 13, since the number of references is significantly increased on $(1, \frac{n}{c})$, $\frac{\frac{k}{c}}{\alpha_1 \cdot \frac{k}{c} + \beta_1}$ is significantly smaller than 1 so that $L \cdot (1 - \frac{\frac{k}{c}}{\alpha_1 \cdot \frac{k}{c} + \beta_1})$ is close to L. Also note that $\log k$ is a lot smaller than L since it is the logarithmic of the reference number. On the right hand side, because we assume the portion on $(1, \frac{n}{c})$ is frequently accessed, easily making $\frac{n}{f}$ smaller than the left hand side.

3 Evaluation

We have implemented a user-level compression middleware for GPFS [13] with the FUSE framework [21]. The compression logic is implemented in the *vc_write()* interface, which is the handler for catching the write system calls. *vc_write()* compresses the raw data, caches it in the memory if possible, and writes the compressed data into GPFS. The decompression logic is implemented in the *vc_read()* interface, similarly. When a read request comes in, this function loads the compressed data (either from the cache or the disk) into memory, applies the decompression algorithm to the compressed data, and passes the result to the end users.

The virtual chunk middleware is deployed on each compute node as a mount point that refers to the remote GPFS file system. This architecture enables a high possibility of reusing the decompressed data, since the decompressed data are cached in the local node. In fact, prior work [22, 23] shows that caching plays a significant impact to the overall performance of distributed and parallel file systems. Because the original compressed file is split into many logical chunks each of which can be decompressed independently, it allows a more flexible memory caching mechanism and parallel processing of these logical chunks. We have implemented a LRU (Least Recently Used) replacement policy for caching the intermediate data.

We have also integrated virtual chunks into the FusionFS [15, 24] distributed file system. FusionFS is designed to ultimately address the I/O bottleneck of conventional high-performance computing systems, as the state-of-the-art storage architecture would unlikely scale to the next generation extreme-scale systems [25]. The key feature of FusionFS is to fully exploit the available resources and avoid any centralized component. That is, each participating node plays three roles at the same time: client, metadata server, and data server. Each node is able to pull the global view of all the available files by the single namespace implemented with a distributed hash table [26], even though the metadata is physically distributed on all the nodes. Each node stores parts of the entire metadata and data at its local storage. Although both metadata and data are fully distributed on all nodes, the local metadata and data on the same node are completely decoupled: the local data may or may not be described by the local metadata. By decoupling metadata and data, we are able to apply flexible strategies on metadata management and data I/Os.

On each compute node, a virtual chunk component is deployed on top of the data I/O implementation in FusionFS. FusionFS itself has employed FUSE to support POSIX, so there is no need for VC to implement FUSE interfaces again. Instead, VC is implemented in the *fusionfs_write()* and the *fusionfs_read()* interfaces. Although the compression is implemented in the *fusionfs_write()* interface, the compressed file is not persisted into the hard disk until the file is closed. This approach can aggregate the small blocks into larger ones, and reduce the number of I/Os to improve the end-to-end time. In some scenarios, users are more concerned for the high availability rather than the compressing time. In that case, a *fsync()* could be called to the (partially) compressed data to ensure these data are available at the persistent storage in a timely manner, so that other processes or nodes could start processing them.

The remainder of this section answers the following questions:

1) How does the number of VC affect the compression ratio and sequential I/O time (Section 3.1)?
2) How does VC, as a middleware, improve the GPFS [13] I/O throughput (Section 3.2)?
3) How does VC, as a built-in component, help to improve the I/O throughput of FusionFS [14, 15] (Section 3.3)?
4) How do the parameters affect the cost of updating VC (Section 3.4)?

All experiments were repeated at least five times until results became stable (i.e. within 5% margin of error); the reported numbers are the average of all runs.

3.1 Compression Ratio

We show how virtual chunks affect the compression ratio on the Global Cloud Resolving Model (GCRM) data [16]. Note that climate data is a good example to leverage the XOR-delta compression; however, selecting the appropriate compression algorithm for a particular data set is beyond the scope of this paper. GCRM consists of single-precision float data of temperatures to analyze cloud's influence on the atmosphere and the global climate. In our experiment there are totally $n = 3.2$ million data entries to be compressed with the aforementioned XOR compressor. Each data entry comprises a row of 80 single-precision floats. Note that based on our previous analysis in Section 2.3, the optimal number of references should be set roughly to $\sqrt{n} \approx 1,789$ (Eq. 11, Section 2.3). Thus we test up to 2,000 references, a bit more than the theoretical optimum.

From 1 to 2,000 references, the compression ratio change is reported in Table 2, together with the overall wall time of the compression. As expected, the compression ratio decreases when more references are appended. However, the degradation of compression ratio is almost negligible: within 0.002 between 1 reference and 2000 references. These small changes to the compression ratios then imply negligible differences of the wall time also: within sub-seconds out of minutes. Thus, this experiment demonstrates that adding a reasonable number of additional references, guided by the analysis in Section 2.3, only introduces negligible overhead to the compression process.

TABLE 2
Overhead of additional references

Number of References	Compression Ratio	Wall Time (second)
1	1.4929	415.40
400	1.4926	415.47
800	1.4923	415.54
1200	1.4921	415.62
1600	1.4918	415.69
2000	1.4915	415.76

The reason of the negligible overhead is in fact due to Eq. 9 (or Eq. 11 as a simplified version) discussed in Section 2.3. The total number of data entries is about quadratic to the optimal number of references, making the cost of processing the additional references only marginal to the overall compression procedure, particularly when the data size is large.

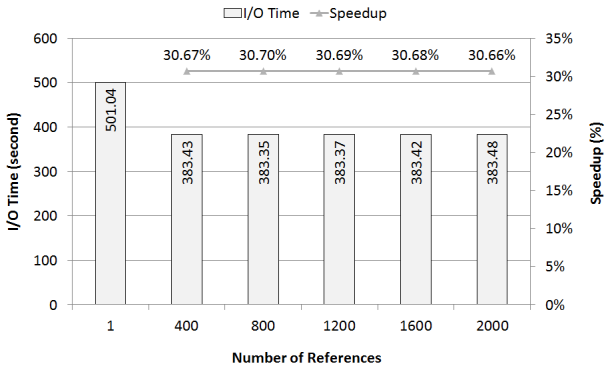
3.2 GPFS Middleware

We deployed the virtual chunk middleware on 1,024 cores (256 physical nodes) pointing to a 128-nodes GPFS [13] file system on Intrepid [18], an IBM Blue Gene/P supercomputer at Argonne National Laboratory. Each Intrepid compute node has a quad-core PowerPC 450 processor (850MHz) and 2GB of RAM. One I/O node exists for every 64 compute nodes, and the I/O time measured in the following experiments is the end-to-end time including the overhead on the I/O nodes. The dataset is 244.25GB of the GCRM [16] climate data. Since GPFS is a shared parallel file systems, when carrying out our experiments we make sure there are no other I/O-intensive jobs; This is achieved by asking each compute node to write 1 GB data to GPFS shortly before the experiments and the measured throughput is within the range specified by the vendor as well as previous measurements. Afterwards, we checked system logs to guarantee there were no new jobs submitted during the experimentation.

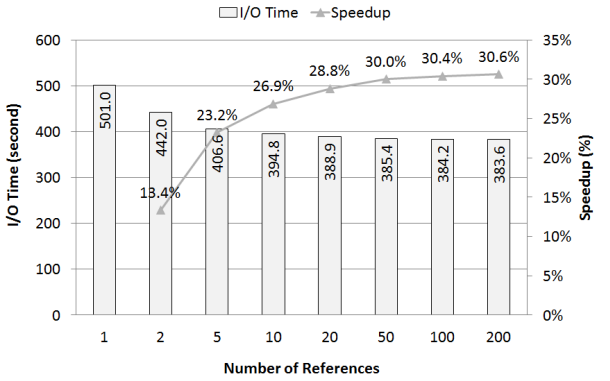
Since virtual chunk is implemented with FUSE [21] that adds extra context switches when making I/O system calls, we need to know how much overhead is induced by FUSE. To measure the impact of this overhead, the GCRM dataset is written to the original GPFS and the GPFS+FUSE file system (without virtual chunks), respectively. The difference is within 2.2%, which could be best explained by the fact that in parallel file systems the bottleneck is on the networking rather than the latency and bandwidth of the local disks. Since the FUSE overhead on GPFS is smaller than 5%, we will not distinguish both setups (original GPFS and FUSE+GPFS) in the following discussion.

We test the virtual chunk middleware on GPFS with two routine workloads: (1) the archival (i.e. write with compression) of all the available data; and (2) the retrieval (i.e. read with decompression) of the latest temperature, regarded as the worst-case scenario discussed in Section 2.3. The I/O time, as well as the speedup over the baseline of single-reference compression, is reported in Figure 2(a). We observe that multiple references (400 – 2000) significantly reduce the original I/O time from 501s to 383s, and reach the peak performance at 800-references with 31% (1.3X) improvement.

An interesting observation from Figure 2(a) is that, the performance sensitivity to the number of references near the optimal k_{opt} is extremely low. And if we consider that all our experiments are within 5% margin of error, the differences across 400 – 2000 are essentially negligible. The optimal number of references seems to be 800



(a) Coarse Granularity 1 – 2000



(b) Fine Granularity 1 – 200

Fig. 2. I/O time with virtual chunks in GPFS

(the shortest time: 383.35 seconds), but the difference across 400 - 2000 references is marginal, only within sub-seconds. This phenomenon is because that beyond a few hundreds of references, the GCRM data set has reached a fine enough granularity of virtual chunks that could be efficiently decompressed. To justify this, we re-run the experiment with finer granularity from 1 to 200 references as reported in Figure 2(b). As expected, the improvement over 1 – 200 references is more significant than between 400 and 2000. This experiment also indicates that, we could achieve a near-optimal (within 1%) performance (30.0% speedup at $k = 50$ vs 30.70% at $k = 800$) with only $\frac{50}{800} = 6.25\%$ cost of additional references. It thus implies that even fewer references than $\sqrt{n}$ could become significantly beneficial to the overall I/O performance.

To study the effect of virtual-chunk compression to real applications, we ran the MMAT application [19] that calculates the minimal, maximal, and average temperatures on the GCRM dataset. The breakdown of different portions is shown in Figure 3. Indeed, MMAT is a data-intensive application, as this is the application type where data compression is useful. So we can see that in vanilla GPFS 97% (176.13 out of 180.97 seconds) of the total runtime is on I/O. After applying the compression layer ($k = 800$), the I/O portion is significantly reduced from 176.13 to 118.02 seconds. Certainly this I/O improvement is not free, as there is 23.59 seconds overhead for the VC computation. The point is, this I/O time saving (i.e. 176.13 - 118.02 = 58.11 seconds) outweighs the VC overhead (23.59 seconds), resulting in 1.24X speedup on the overall execution time.

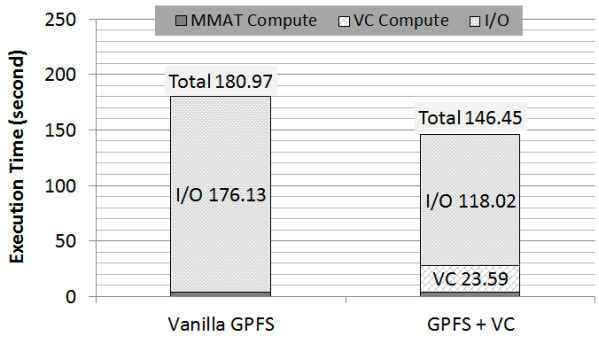


Fig. 3. Execution time of the MMAT application

3.3 FusionFS Integration

FusionFS [14, 15, 27–29] is a new distributed file system driven by our initial simulation study [25] that addresses the I/O bottleneck of high-performance computing. It is designed with unique features such as distributed metadata [26, 30–32], cooperative caching [22, 23, 33], virtual chunking [11, 12], distributed provenance [34, 35], and GPU acceleration [36]. It also serves as a test bed for other system components such as next-generation job scheduler [37, 38].

We have deployed FusionFS integrated with virtual chunks to a 64-nodes Linux cluster at Illinois Institute of Technology. Each node has two Quad-Core AMD Opteron 2.3 GHz processors with 8 GB RAM and 1 TB Seagate Barracuda hard drive. All nodes are interconnected with a 1Gbps Ethernet. Besides the GCRM [16] data, we also evaluated another popular data set Sloan Digital Sky Survey (SDSS [17]) that comprises a collection of astronomical data such as positions and brightness of hundreds of millions of celestial objects. The SDSS dataset is from a SQL query submitted to the SDSS web portal and has the same size of GCRM.

We illustrate how virtual chunks help FusionFS to improve the I/O throughput on both data sets in Figure 4. We do not vary k but set it to $\sqrt{n}$ when virtual chunk is enabled. Results show that both read and write throughput are significantly improved. Note that, the I/O throughput of SDSS is higher than GCRM, because the compression ratio of SDSS is 2.29, which is higher than GCRM's compression ratio 1.49. In particular, we observe up to 2X speedup when VC is enabled (SDSS write: 8206 vs. 4101).

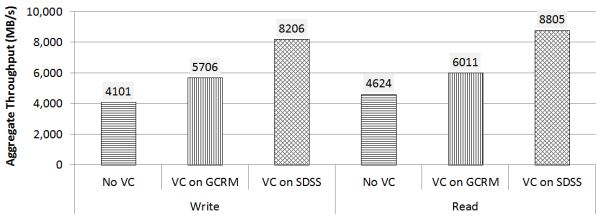


Fig. 4. FusionFS throughput on GCRM and SDSS datasets

3.4 Parameter Sensitivity of Dynamic VC

As discussed in Section 2.6, it is nontrivial to update the granularity (i.e. linear transform by α_i and β_i) of virtual chunks within a particular range (i.e. (s_i, t_i)). We will show quantitatively how costly this update computation is with respect to static virtual chunks. The experiments assume there is one update applied each time, so the subscripts of α_1, β_1, and (s_i, t_i) are not shown in the following discussion.

It should be clear that the cost in the following discussion is only for the reference update, and does not consider the benefit from consequent I/Os as discussed in Section 2.7. Even large overheads can be compensated by the savings in I/O, resulting in better overall performance.

The experimental setup is as follows. The files being evaluated are again the 244.25GB GCRM data [16]. After being compressed with 2,000 equidistant virtual chunks, the `RefUpdate` procedure is triggered to adjust the virtual chunks. The runtime of this procedure is compared to the time of compressing the data with static virtual chunks; the ratio of the update time over the compressing time is then considered as the cost (in %).

There are two dimensions to control the updating behavior: (1) the affected range length $(s - t + 1)$, and (2) the linear transform with α and β. Intuitively, a larger $(s-t+1)$ indicates more computation, since more references need to be updated within that range. This intuition also applies to the linear transform: larger α and β imply more references to be appended to the end of the compressed file. Note that, even though both α and β are considered as coefficients in the linear transform, it is sufficient to adjust α and set $\beta = 0$ to study the performance with respect to the density of references, or in other words the granularity of virtual chunks. Therefore in the following discussion, β is set to zero.

In order to study the combined effect of both the updated range and chunk granularity, we tune $\frac{s-t+1}{n}$ to be 0.2 – 1.0, and α to be 0.5 – 128 times of the original granularity. Figure 5 shows the cost from different parameter combinations. Unsurprisingly, the results confirm our previous intuition: the peak cost (61.15%) comes from the scenario where: (1) all the references are updated, i.e. $\frac{s-t+1}{n} = 1$, and (2) the granularity of virtual chunks is increased by 128 (the most) times. Similarly, the lowest cost (10.07%) occurs for $\frac{s-t+1}{n} = 0.2$ and $\alpha = 0.5$, the smallest values of both dimensions.

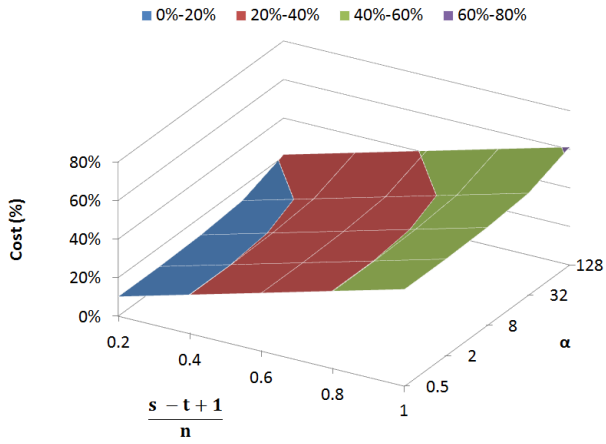


Fig. 5. Parameter sensitivity of dynamic virtual chunks

Now, we turn to discuss a more interesting observation from Figure 5: the updated range seems to have a more significant impact to the overhead, than α does. In particular, for all α's, the increased overhead looks strongly proportional to the increased $(s-t+1)$ range, while the impact from α is less noticeable: increasing α from 0.5 to 128 only adds about 10% cost. To make this more obvious, we slice the 3-D surface on two dimensions when fixing $\frac{s-t+1}{n} = 0.5$ and $\alpha = 2$, as shown in Figure 6 and Figure 7, respectively.

Figure 6 clearly shows that changing α from 0.5 to 128 affects the cost by slightly less than 11%, with a fixed ratio between the updated range and the overall length ($\frac{s-t+1}{n} = 0.5$). This could be best explained by the fact that the number of references is roughly set to the square root of the number of data entries. Thus, even though the number of references (controlled by α and β) is significantly increased (256X from 0.5 to 128), the overall impact to the system performance is diluted by the small factor – the square root of the original scale.

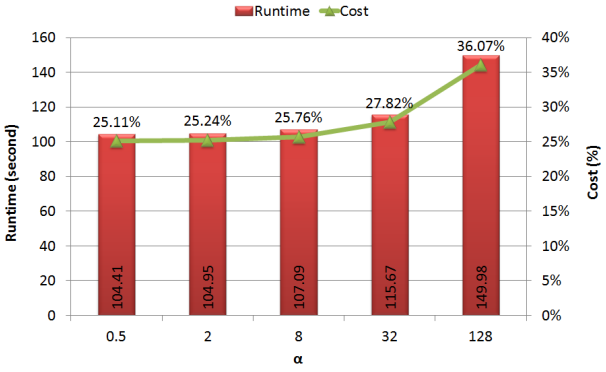


Fig. 6. Parameter sensitivity with fixed range ratio = 0.5

In contrast to α, we observed a strong linearity between the cost and the updated range $(s-t+1)$, as shown in Figure 7. The reason of this phenomenon is that all the encoded data within (s,t) need to be read into memory and then decoded to retrieve the new references. Therefore the cost of this procedure is highly dependent on the number of data within the range; So we see a strong linear relation between the overhead and the updated range.

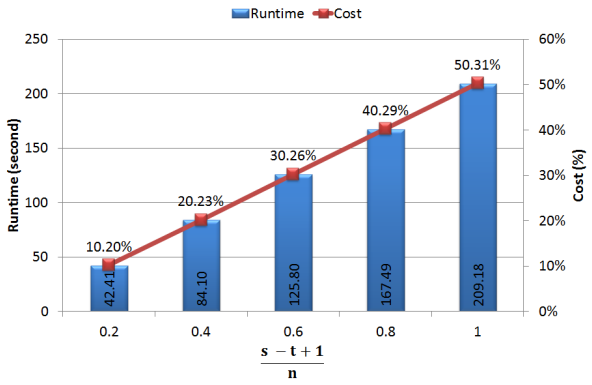


Fig. 7. Parameter sensitivity with fixed α = 2

4 Discussion and Limitation

4.1 Applicability

It should be clear that the proposed virtual chunk mechanism to be used in compressible storage systems is applicable only if the underlying compression format is splittable. A compressed file is splittable if it can be split into subsets and then be processed (e.g. decompressed) in parallel. Obviously, one key advantage of virtual chunks is to manipulate data in the arbitrary and logical subsets of the original file, which depends on this splittable feature. Without a splittable compression algorithm, the virtual chunk is not able to decompress itself. The XOR-based delta compression used through this paper is splittable. Popular compressors, such bzip2 [4] and LZO [3], are also splittable. Some non-splittable examples include Gzip [39] and Snappy [40].

4.2 Autonomic VC Update

Our intuitive solution (Section 2.6) to adjust the virtual chunk granularity is to ask users to specify where and how to update the reference. It implies that the users are expected to have a good understanding of their applications, such as I/O patterns. This is a reasonable assumption in some cases, for example if the application developers are the main users. Besides the manual manipulation approach introduced in Section 2.6, we can also expect that the users would specify the distribution of the reference density in a configuration file, or more likely a rule such as a decay function [20].

Nevertheless we believe it would be more desirable to have an autonomic mechanism to adjust the virtual chunks for those domain users without the technical expertise such as chemists, astronomers, et al. Given the large volume of scientific data sets and their applications, supervised learning techniques have the potentials to predict future I/O patterns on the basis of a training set. This remains an open question to the community and a direction of our future work.

4.3 Data Insertion and Data Removal

We are not aware of much need for data insertion and data removal within a file in the context of HPC or scientific applications. By insertion, we mean a new data entry needs to be inserted into an arbitrary position of an existing compressed file. Similarly, by removal we mean an existing value at an arbitrary position needs to be removed. Nevertheless, this work would be more complete with support of efficient data insertion and removal when enabling virtual chunks in storage compression.

A straightforward means to support this operation might treat a data removal as a special case of data writes with the new value as null. But then it would bring new challenges such as dealing with the "holes" within the file. We would like to have more discussions with HPC researchers and domain scientists regarding this.

5 Related Work

It should be noted that a compression method does not necessarily need to restore the absolutely original data. In general, compression algorithms could be categorized into to two groups: lossy algorithms and lossless algorithms. A lossy algorithm might lose some (normally a small) percentage of accuracy, while a lossless one has to ensure the 100% accuracy. In scientific computing, studies [6, 7] show that lossy compression could be acceptable, or even quite effective, under certain circumstances. In fact, lossy compression is also popular in other fields, e.g. the most widely compatible lossy audio and video format MPEG-1 [41]. This paper presents virtual chunks mostly by going through a delta-compression example based on XOR, which is a lossless compression. It does not imply that virtual chunks cannot be used in a lossy compression. Virtual chunk is not a specific compression algorithm but a system mechanism that is applicable to any splittable compression (either lossy or lossless).

Some frameworks are proposed as middleware to allow applications call high-level I/O libraries for data compression and decompression, e.g. [19, 42, 43]. None of these techniques take consideration of the overhead involved in decompression by assuming the chunk allocated to each node would be requested as an entirety. In contrast, virtual chunks provide a mechanism to apply flexible compression and decompression.

There is previous work to study the file system support for data compression. Integrating compression to log-structured file systems was proposed decades ago [44],

which suggested a hardware compression chip to accelerate the compressing and decompressing. Later, XDFS [45] described the systematic design and implementation for supporting data compression in file systems with BerkeleyDB [46]. MRAMFS [47] was a prototype file system to support data compression to leverage the limited space of non-volatile RAM. In contrast, virtual trunks represent a general technique applicable to existing systems.

Data deduplication is a general inter-chunk compression technique that only stores a single copy of the duplicate chunks (or blocks). For example, LBFS [48] was a networked file system that exploited the similarities between files (or versions of files) so that chunks of files could be retrieved in the client's cache rather than transferring from the server. CZIP [49] was a compression scheme on content-based naming, that eliminated redundant chunks and compressed the remaining (i.e. unique) chunks by applying existing compression algorithms. Recently, the metadata for the deduplication (i.e. file recipe) was also slated for compression to further save the storage space [50]. While deduplication focuses on inter-chunk compressing, virtual chunk focuses on the I/O improvement within the chunk.

Index has been introduced to data compression to improve the compressing and query speed e.g. [51, 52]. The advantage of indexing is highly dependent on the chunk size: large chunks are preferred to achieve high compression ratios in order to amortize the indexing overhead. However large chunks would cause potential decompression overhead as explained earlier in this paper. Virtual chunk overcomes the large-chunk issue by logically splitting the large chunks with fine-grained partitions while still keeping the physical coherence.

6 Conclusion and Future Work

This paper presents a new compression mechanism named virtual chunking (VC). VC avoids the potentially huge space overhead of conventional physical chunking in data compression and supports an efficient random access to the compressed data. In essence, VC achieves the small spacial overhead of file-level compression as well as the small computational overhead of (physical)chunk-level decompression, the best of both worlds.

We discuss VC from two major perspectives on whether or not the I/O pattern follows a uniform distribution over the data: static VC and dynamic VC. Static VC assumes the references are equidistant without a prior knowledge of the application's characteristic and I/O patterns. Although static VC is applicable in many scenarios, it is still highly desirable to allow users to specify the reference position or distribution. To this end, we extend static VC to dynamic VC, a more general form of static VC to make the approach more applicable.

We formulate the procedures to use static and dynamic VC to incorporate splittable compressors. An in-depth analysis is conduced for the abstract model of VC such as the optimal parameter setup. Evaluations show that VC could speed up scientific applications by 1.3X – 2X.

Our future work primarily lies in devising an automatic mechanism to update virtual chunks with machine learning. The learning process in each iteration does not need to start from scratch; it may take the previous result as a feed and incrementally makes the adjustment—some of our previous work [53–55] on incremental algorithms could possibly be leveraged.

Acknowledgment

This work was supported in part by the Office of Biological and Environmental Research, Office of Science, U.S. Department of Energy, under contract DE-ACO2-O6CH11357, and the NSF under awards OCI-1054974.

References

[1] T. Hey, S. Tansley, and K. Tolle, Eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.
[2] P. A. Freeman, D. L. Crawford, S. Kim, and J. L. Munoz, "Cyberinfrastructure for science and engineering: Promises and challenges," *Proceedings of the IEEE*, vol. 93, no. 3, pp. 682–691, 2005.
[3] LZO, "http://www.oberhumer.com/opensource/lzo," Accessed September 5, 2014.
[4] bzip2, "http://www.bzip2.org," Accessed September 5, 2014.
[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies*, 2010.
[6] D. Laney, S. Langer, C. Weber, P. Lindstrom, and A. Wegener, "Assessing the effects of data compression in simulations using physically motivated metrics," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.
[7] S. Lakshminarasimhan, J. Jenkins, I. Arkatkar, Z. Gong, H. Kolla, S.-H. Ku, S. Ethier, J. Chen, C. S. Chang, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "ISABELA-QA: Query-driven analytics with ISABELA-compressed extreme-scale scientific data," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, 2011.
[8] HDF5, "http://www.hdfgroup.org/HDF5/doc/index.html," Accessed September 5, 2014.
[9] NetCDF, "http://www.unidata.ucar.edu/software/netcdf," Accessed September 5, 2014.
[10] GRIB, "http://www.wmo.int/pages/prog/www/DPS/FM92-GRIB2-11-2003.pdf," Accessed April 19, 2015.
[11] D. Zhao, J. Yin, and I. Raicu, "Improving the i/o throughput for data-intensive scientific applications with efficient compression mechanisms," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13), poster session*, 2013.
[12] D. Zhao, J. Yin, K. Qiao, and I. Raicu, "Virtual chunks: On supporting random accesses to scientific data in compressible storage systems," in *Proceedings of IEEE International Conference on Big Data*, 2014.
[13] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.
[14] D. Zhao, N. Liu, D. Kimpe, R. Ross, X.-H. Sun, and I. Raicu, "Towards exploring data-intensive scientific applications at extreme scales through systems and simulations," *Parallel and Distributed Systems, IEEE Transactions on*, 2015 (accepted).
[15] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, "FusionFS: Toward supporting data-intensive scientific applications on extreme-scale distributed systems," in *Proceedings of IEEE International Conference on Big Data*, 2014.
[16] GCRM, "http://kiwi.atmos.colostate.edu/gcrm/," Accessed September 5, 2014.
[17] SDSS Query, "http://cas.sdss.org/astrodr6/en/help/docs/realquery.asp," Accessed September 5, 2014.
[18] Intrepid, "https://www.alcf.anl.gov/user-guides/intrepid-challenger-surveyor," Accessed September 5, 2014.
[19] T. Bicer, J. Yin, D. Chiu, G. Agrawal, and K. Schuchardt, "Integrating online compression to accelerate large-scale data analytics applications," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013.

[20] E. Cohen and M. Strauss, "Maintaining time-decaying stream aggregates," in *Proceedings of the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2003.

[21] FUSE, "http://fuse.sourceforge.net," Accessed September 5, 2014.

[22] D. Zhao, K. Qiao, and I. Raicu, "Hycache+: Towards scalable high-performance caching middleware for parallel file systems," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014.

[23] D. Zhao and I. Raicu, "HyCache: A user-level caching middleware for distributed file systems," in *Proceedings of IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2013.

[24] ——, "Distributed file systems for exascale computing," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12), doctoral showcase*, 2012.

[25] D. Zhao, D. Zhang, K. Wang, and I. Raicu, "Exploring reliability of exascale systems through simulations," in *Proceedings of the 21st ACM/SCS High Performance Computing Symposium (HPC)*, 2013.

[26] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in *Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, 2013.

[27] D. Zhao, X. Yang, I. Sadooghi, G. Garzoglio, S. Timm, and I. Raicu, "High-performance storage support for scientific applications on the cloud," in *Proceedings of the 6th Workshop on Scientific Cloud Computing (ScienceCloud)*, 2015.

[28] D. Zhao and I. Raicu, "Storage support for data-intensive scientific applications on the cloud," in *NSF Workshop on Experimental Support for Cloud Computing*, 2014.

[29] ——, "Storage support for data-intensive applications on extreme-scale hpc systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14), doctoral showcase*, 2014.

[30] T. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, Z. Zhang, and I. Raicu, "A convergence of key-value storage systems from clouds to supercomputer," *Concurr. Comput. : Pract. Exper.*, 2015 (accepted).

[31] T. Li, K. Keahey, K. Wang, D. Zhao, and I. Raicu, "A dynamically scalable cloud data infrastructure for sensor networks," in *Proceedings of the 6th Workshop on Scientific Cloud Computing (ScienceCloud)*, 2015.

[32] T. Li, C. Ma, J. Li, X. Zhou, K. Wang, D. Zhao, and I. Raicu, "Graph/z: A key-value store based scalable graph processing system," in *Cluster Computing, IEEE International Conference on*, 2015.

[33] D. Zhao, K. Qiao, and I. Raicu, "Towards cost-effective and high-performance caching middleware for distributed systems," *International Journal of Big Data Intelligence*, 2015.

[34] D. Zhao, C. Shou, T. Malik, and I. Raicu, "Distributed data provenance for large-scale data-intensive computing," in *Cluster Computing, IEEE International Conference on*, 2013.

[35] C. Shou, D. Zhao, T. Malik, and I. Raicu, "Towards a provenance-aware distributed filesystem," in *5th Workshop on the Theory and Practice of Provenance (TaPP)*, 2013.

[36] D. Zhao, K. Burlingame, C. Debains, P. Alvarez-Tabio, and I. Raicu, "Towards high-performance and cost-effective distributed storage systems with information dispersal algorithms," in *Cluster Computing, IEEE International Conference on*, 2013.

[37] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan, "I/o-aware batch scheduling for petascale computing systems," in *Cluster Computing, IEEE International Conference on*, 2015.

[38] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," in *Proceedings of IEEE International Conference on Big Data (BigData Conference)*, 2014.

[39] Gzip, "http://www.gnu.org/software/gzip/gzip.html," Accessed September 5, 2014.

[40] Snappy, "https://code.google.com/p/snappy/," Accessed September 5, 2014.

[41] MPEG-1, "http://en.wikipedia.org/wiki/MPEG-1," Accessed September 5, 2014.

[42] E. R. Schendel, S. V. Pendse, J. Jenkins, D. A. Boyuka, II, Z. Gong, S. Lakshminarasimhan, Q. Liu, H. Kolla, J. Chen, S. Klasky, R. Ross, and N. F. Samatova, "Isobar hybrid compression-i/o interleaving for large-scale parallel i/o optimization," in *Proceedings of International Symposium on High-Performance Parallel and Distributed Computing*, 2012.

[43] J. Jenkins, E. R. Schendel, S. Lakshminarasimhan, D. A. Boyuka, II, T. Rogers, S. Ethier, R. Ross, S. Klasky, and N. F. Samatova, "Byte-precision level of detail processing for variable precision analytics," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[44] M. Burrows, C. Jerian, B. Lampson, and T. Mann, "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 (ASPLOS)*, 1992.

[45] J. P. MacDonald, "File system support for delta compression," University of California, Berkley, Tech. Rep., 2000.

[46] M. A. Olson, K. Bostic, and M. Seltzer, "Berkeley db," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 1999.

[47] N. K. Edel, D. Tuteja, E. L. Miller, and S. A. Brandt, "Mramfs: A compressing file system for non-volatile ram," in *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, 2004.

[48] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[49] K. Park, S. Ihm, M. Bowman, and V. S. Pai, "Supporting practical content-addressable caching with czip compression," in *2007 USENIX Annual Technical Conference*, 2007.

[50] D. Meister, A. Brinkmann, and T. Süß, "File recipe compression in data deduplication systems," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[51] S. Lakshminarasimhan, D. A. Boyuka, S. V. Pendse, X. Zou, J. Jenkins, V. Vishwanath, M. E. Papka, and N. F. Samatova, "Scalable in situ scientific data encoding for analytical query processing," in *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2013.

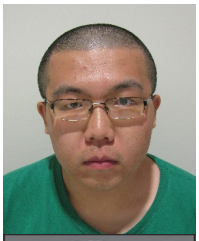
[52] Z. Gong, S. Lakshminarasimhan, J. Jenkins, H. Kolla, S. Ethier, J. Chen, R. Ross, S. Klasky, and N. F. Samatova, "Multi-level layout optimization for efficient spatio-temporal queries on isabela-compressed data," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[53] D. Zhao and L. Yang, "Incremental isometric embedding of high-dimensional data using connected neighborhood graphs," *IEEE Trans. Pattern Anal. Mach. Intell. (PAMI)*, vol. 31, no. 1, Jan. 2009.

[54] R. Lohfert, J. Lu, and D. Zhao, "Solving sql constraints by incremental translation to sat," in *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, 2008.

[55] D. Zhao and L. Yang, "Incremental construction of neighborhood graphs for nonlinear dimensionality reduction," in *Proceedings of International Conference on Pattern Recognition*, 2006.

Dongfang Zhao is working towards his Ph.D. in computer science at Illinois Institute of Technology in Chicago. Previously he obtained his Master's degree in computer science from Emory University in Atlanta. His research interests include high-performance computing, distributed systems, cloud computing, big data, and machine intelligence.



Dr. Kan Qiao received his Ph.D. in computer science from Illinois Institute of Technology in Chicago, 2015. He obtained his bachelor's degree in Electrical Engineering from Beijing University Of Aeronautics and Astronautics (China) in 2010. His research interests include combinatorial optimization and network algorithms. He is now a software engineer at Google.

Dr. Jian Yin is a senior computer scientist at the Pacific Northwest National Laboratory. His current research interests include distributed systems, high-performance computing, data sciences, and large-scale scientific applications. He holds a Ph.D. degree in computer science from the University of Texas at Austin, TX.

Dr. Ioan Raicu is an assistant professor in the Department of Computer Science at Illinois Institute of Technology, as well as a guest research faculty in the Math and Computer Science Division at Argonne National Laboratory. He has received the prestigious NSF CAREER award (2011 - 2015) for his innovative work on distributed file systems for extreme-scales. He obtained his Ph.D. in Computer Science from University of Chicago in 2009.