Virtual Chunks: On Supporting Random Accesses to Scientific Data in Compressible Storage Systems

Dongfang Zhao*†, Jian Yin†, Kan Qiao*‡, and Ioan Raicu*◇

*Illinois Institute of Technology †Pacific Northwest National Lab ‡Google Inc. ◇Argonne National Lab

dzhao8@iit.edu, jian.yin@pnnl.gov, kqiao@iit.edu, iraicu@cs.iit.edu

***Abstract*—Data compression could ameliorate the I/O pressure of scientific applications on high-performance computing systems. 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 can barely support 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 data, but introduces extra overhead when applying the compressor to each every block that results in a degraded overall compression ratio. This paper introduces a concept called *virtual chunks* aiming to support efficient random accesses to the compressed scientific data without sacrificing its compression ratio. In essence, virtual chunks are logical blocks identified by appended references without breaking the physical continuity of the file content. These additional references allow the decompression to start from an arbitrary position (efficient random access), and retain the file's physical entirety to achieve high compression ratio on par with file-level compression. One potential concern of virtual chunks lies on its space overhead (from the additional references) that degrades the compression ratio, but our analytic study and experimental results demonstrate that such overhead is negligible. We have implemented virtual chunks in two forms: a middleware to the GPFS parallel file system, and a module in the FusionFS distributed file system. Large-scale evaluations on up to 1,024 cores showed that virtual chunks could help improve the I/O throughput by 2X speedup.**

I. Introduction

As today's scientific applications are becoming data-intensive (e.g. astronomy [1]), one effective approach to relieve the I/O bottleneck of the underlying storage system is data compression. As a case in point, it is optional to apply lossless compressors (e.g. LZO [2], bzip2 [3]) to the input or output files in the Hadoop file system (HDFS) [4], or even lossy compressors [5, 6] at the high-level I/O middleware such as HDF5 [7] and NetCDF [8]. By investing some computational time on compression, 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, however, simply apply the compressor to the data either at the file-level or block-level[1], and leave the important factors (e.g. computational overhead, compression ratio, I/O pattern) to the underlying compression algorithms. In particular, we observe the following limitations of applying the file-level and block-level compression, respectively:

1) The file-level compression is criticized by the significant overhead for random accesses: the decompression needs to start from the very beginning of the compressed file anyway even though the client might be only requesting some bytes at an arbitrary position of 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.
2) The deficiency of block-level compression stems from its additional compression overhead larger than the file-level counterpart, resulting in a degenerated compression ratio. To see this, think about a simple scenario that a 64MB file to be compressed with 4:1 ratio and 4KB overhead (e.g. header, metadata, etc.). So the resultant compressed file (i.e. file-level compression) is about 16MB + 4KB = 16.004MB. If the file is split into 64KB-blocks each of which is applied with the same compressor, the compressed file would be 16MB + 4KB $\times$ 1K = 20MB. Therefore we would roughly spend (20MB - 16.004MB) / 16.004MB $\approx$ 25% more space in block-level compression than the file-level one.

This paper introduces *virtual chunks* (VC) that aim to better employ existing compression algorithms in parallel and distributed file systems, and eventually to improve the I/O performance of random data accesses in scientific applications and high-performance computing (HPC) systems. The idea of virtual chunks was first presented in the poster session of the Supercomputing 2014 conference [9]. Virtual chunks 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, continuity of blocks) 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

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

close to the requested data and start the decompression from there. Therefore virtual chunks help to achieve the best of both file- and block-level compressions: high compression ratio and efficient random access.

Virtual chunks might raise concerns about the cost of the additional references: they need more storage space and take more I/O time. We argue, and will justify in Section II-C and Section III-A, 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 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 random accesses. For example, a single reference makes the compression essentially applied to the entire file. 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 II-C) and experimental results (Section III-A) to justify that the space overhead from the additional reference is negligible in terms of end-to-end I/O performance.

To summarize, this paper makes the following contributions:

- Propose the 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
- Formalize the procedures to manipulate virtual chunks, and provide theoretical analysis on how to set up the parameters to achieve the optimal performance
- Design and implement virtual chunks in a production parallel file system GPFS [10] and a distributed file system FusionFS [11, 12]
- Evaluate virtual chunks with real scientific data sets (e.g. GCRM [13], SDSS [14]) at large scale, on up to 1024 cores in a leadership supercomputer Intrepid [15]

The remainder of this paper mainly focuses on the presentation and evaluation of employing virtual chunks in compressible file systems. In Section II, we analyze how to wisely choose the number of references to append, discuss where to store these references, and formalize the procedures to leverage virtual chunks in sequential and random data accesses. Section III describes two forms of implementation of virtual chunks: a middleware on a parallel file system (GPFS [10]) and a built-in module in a distributed file system (FusionFS [11, 12]). Large-scale experiments at up to 1,024-cores show that virtual chunks improve I/O performance by up to 2X on real scientific applications, such as climate data GCRM [13] and astronomy data SDSS [14]. In Section IV, we discuss the limitations and open questions of virtual chunks. Section V reviews previous work on storage systems, I/O performance, and data compression. We finally conclude this paper in Section VI.

II. Virtual Chunk

To make matters more concrete, we illustrate how virtual chunks work with an XOR-based delta compression [16] 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.

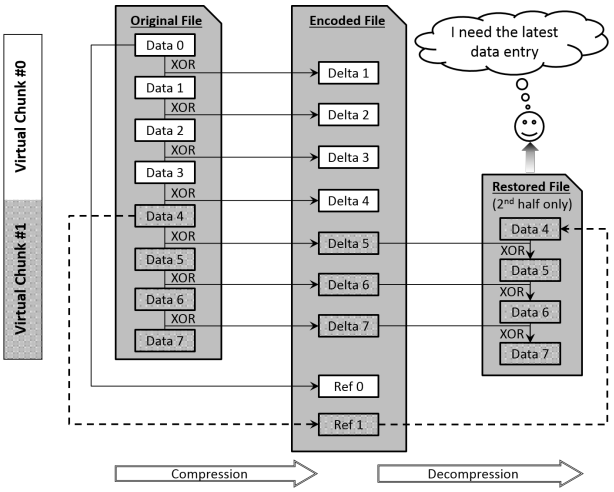


Fig. 1. Compression and decompression with two virtual chunks

Figure 1 shows an original file of eight data entries, and two references to Data 0 and Data 4, respectively. 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.

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, denoted by the list $X = \langle x_1, x_2, \ldots, x_{n-1} \rangle$ where $x_i = d_i$ XOR d_{i+1} 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 square bracket [ ] 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 $D_{s,t}$.

A. 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 reference $(k-1)$ comparing to 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 drawback of the in-place method is, even though not so obvious, critical: 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).

B. Compression with Virtual Chunks

We use `encode()` (or `decode()`) applicable to two neighboring data entries to represent some compression (or decompression) algorithms to the file data. Certainly, it is not always true that the compression algorithm deals with two neighboring data entries; we only take this assumption for clear representation, and it would not affect the validity of the algorithms or the analysis that follows. We will discuss the applicability of virtual chunks in more details in Section IV-A.

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 original 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.

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.

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: $\quad X[i] \leftarrow$ `encode` (d_i, d_{i+1})
3: **end for**
4: **for** (int j = 1; j $<$ k; j++) **do**
5: $\quad D'[j] \leftarrow D[1 + (j-1) * L]$
6: **end for**

C. Optimal Number of References

This section answers this question: how many references should we append to the compressed file, in order to maximize 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.

TABLE I. VIRTUAL CHUNK PARAMETERS

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

A few more parameters for the analysis are listed in Table I. 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 weights 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).

The overhead introduced by additional references during compression is as follows. The baseline is when the file is applied with the conventional compression with a single reference. When comparing both cases, we need to apply the same compression algorithm to be applied on the same set of data. Therefore, the computational time should be unchanged regardless of the number of references appended to the compressed file. So the overall time difference really comes from the I/O time of writing different number of references.

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

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

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:

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

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):

$$F(k) = T_d - T_c$$

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

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

we have

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

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

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

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:

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

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

$$k_{opt} = \begin{cases} \lfloor \hat{k} \rfloor & \text{if } F(\lfloor \hat{k} \rfloor) > F(\lceil \hat{k} \rceil) \\ \lceil \hat{k} \rceil & \text{otherwise} \end{cases} \quad (1)$$

where

$$\hat{k} = \sqrt{n \cdot \frac{B_w}{B_r} \cdot \frac{W_i}{W_o} \cdot (\frac{1}{R} + \frac{D \cdot B_r}{S})} \quad (2)$$

and

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

Note that the last term $\frac{D \cdot B_r}{S}$ in Eq. 2 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. 2 can be stated as:

$$\hat{k} = \sqrt{n} \quad (3)$$

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

D. 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 reference are equidistant. Therefore, given the start position s we could calculate its

closest and latest reference index $s' = LastRef(s)$ where :

$$LastRef(x) \leftarrow \begin{cases} \frac{x}{L} + 1 & \text{if } 0 \neq x \ \texttt{MOD} \ L \\ \frac{x}{L} & \text{otherwise} \end{cases} \quad (4)$$

So we only need to plug Eq. 4 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. 4.

E. 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.

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

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. 4. 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.

III. Evaluation

We have implemented a user-level compression middleware for GPFS [10] with the FUSE framework [17]. 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 [18, 19] 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 replacement policy for caching the intermediate data.

We have also integrated virtual chunks into the FusionFS [11, 12] distributed file system. The FusionFS is proposed 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 [20]. 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 [21, 22], 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. Prior work [23, 24] also shows that a distributed hash table offers a flexible yet efficient means for tracking applications' provenance.

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 III-A)?
2) How does VC, as a middleware, improve the GPFS [10] I/O throughput (Section III-B)?
3) How does VC, as a built-in component, help to improve the I/O throughput of FusionFS [11, 12] (Section III-C)?

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

A. Compression Ratio

We show how virtual chunks affect the compression ratio on the Global Cloud Resolving Model (GCRM) data [13]. 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 II-C, the optimal number of references should be set roughly to $\sqrt{n} \approx 1,789$ (Eq. 3, Section II-C). Thus we tested up to 2,000 references, a bit more than the theoretical optimum.

TABLE II. 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

From 1 to 2,000 references, the compression ratio change is reported in Table II, 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 II-C, only introduces negligible overhead to the compression process.

The reason of the negligible overhead is in fact due to Eq. 2, or Eq. 3 as a simplified version discussed in Section II-C. 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.

B. GPFS Middleware

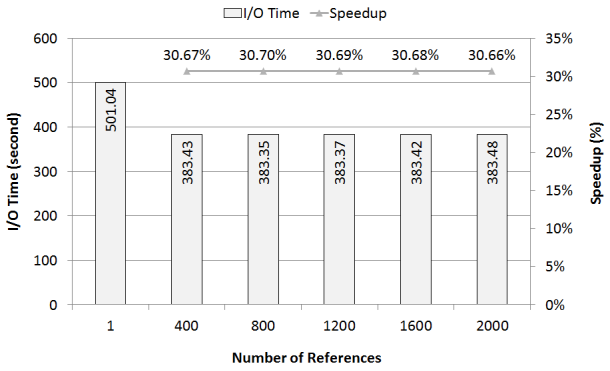
We deployed the virtual chunk middleware on 1,024 cores (256 physical nodes) pointing to a 128-nodes GPFS [10] file system on Intrepid [15], an IBM BlueGene/P supercomputer at Argonne National Laboratory. Each Intrepid compute node has a quad-core PowerPC 450 processor (850MHz) and 2GB of RAM. The dataset is 244.25GB of the GCRM [13] climate data.

Since virtual chunk is implemented with FUSE [17] 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.

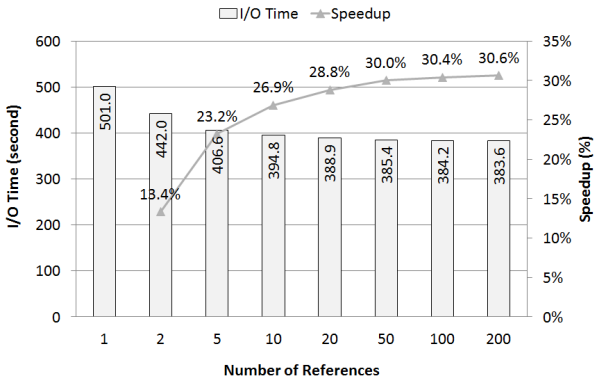
We tested the virtual chunk middleware on GPFS with two routine workloads: (1) the archival (i.e. write with compression) of all the available data; and (2) the retrieval (i.e. read with decompression) of the latest temperature, regarded as the worst-case scenario discussed in Section II-C. The I/O time, 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 relatively low. The optimal number of references seems to be 800 (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 [16] 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



(a) Coarse Granularity 1 – 2000



(b) Fine Granularity 1 – 200

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

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.

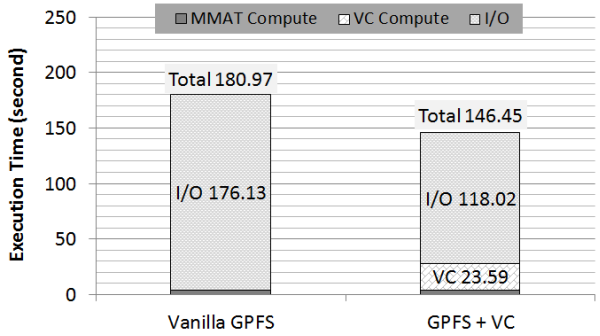


Fig. 3. Execution time of the MMAT application

C. FusionFS Integration

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 2.3GHz processors with 8GB RAM and 1TB Seagate Barracuda hard drive. All nodes are interconnected with a 1Gbps Ethernet. Besides the GCRM [13] data, we also evaluated another popular data set Sloan Digital Sky Survey (SDSS [14]) that comprises a collection of astronomical data such as positions and brightness of hundreds of millions of celestial objects.

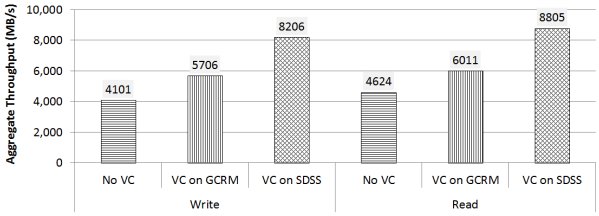


Fig. 4. FusionFS throughput on GCRM and SDSS datasets

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).

IV. Discussion and Limitation

A. 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 clearly a splittable format. Popular compressors, such bzip2 [3] and LZO [2], are also splittable. Some non-splittable examples include Gzip [25] and Snappy [26].

It should also be noted that virtual chunks are not designed for general-purpose compression, but for highly compressible scientific data. This is why this study did not evaluate a virtual chunk version of general compressors (e.g. bzip2, LZO), since they are not designed for numerical data used in scientific applications.

B. Dynamic Virtual Chunks

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 efficiently with a shorter distance (and less computation) from the closest reference, in general. The assumption of equidistant reference, thus, would not hold any more in this case.

One intuitive solution 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. Therefore, we 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 [27].

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, and so on. This remains an open question to the community and a direction of our future work.

C. 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, it would make this work more complete if we supported efficient data insertion and data 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 do not think either is a trivial problem, and would like to have more discussions with HPC researchers and domain scientists before investing in such features.

V. Related Work

While the storage system could be better deigned to handle more data, an orthogonal approach is to address the I/O bottleneck by squeezing the data with compression techniques. One example where data compression gets particularly popular is checkpointing, an extremely expensive I/O operation in HPC systems. In [28], it showed that data compression had the potential to significantly reduce the checkpointing file sizes. If multiple applications run concurrently, a data-aware compression scheme [29] was proposed to improve the overall checkpointing efficiency. Recent study [30] shows that combining failure detection and proactive checkpointing could improve 30% efficiency compared to classical periodical checkpointing. Thus data compression has the potential to be combined with failure detection and proactive checkpointing to further improve the system efficiency. As another example, data compression was also used in reducing the MPI trace size, as shown in [31]. A small MPI trace enables an efficient replay and analysis of the communication patterns in large-scale machines.

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 [5, 6] 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 [32]. 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, not matter if it is 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. [16, 33, 34]. 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 [35], which suggested a hardware compression chip to accelerate the compressing and decompressing. Later, XDFS [36] described the systematic design and implementation for supporting data compression in file systems with BerkeleyDB [37]. MRAMFS [38] 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 algorithms and 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 [39] 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 [40] 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 [41]. 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. [42, 43]. 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.

Data compression attracts a lot of research interests in scientific applications. For instance, AstroPortal [44] shows that working with compressed data for astronomy applications can be beneficial in certain cases. Workflow systems, such as [45, 46], are a good fit for the compressible storage incorporated with virtual chunks.

VI. Conclusion and Future Work

Conventional file- and block-level storage compression have shown their limits for scientific applications: file-level compression provides little support for random access, and block-level compression significantly degenerates the overall compression ratio due to the per-block compression overhead. This paper introduces virtual chunks to support efficient random accesses to compressed scientific data while retaining the high compression ratio. Virtual chunks keep files' physical entirety, because they are referenced by pointers beyond the file end. The physical entirety helps to achieve a high compression ratio by avoiding the per-block compression overhead. The additional references take insignificant storage space and add negligible end-to-end I/O overhead. Virtual chunks enable efficient random accesses to arbitrary positions of the compressed data without decompressing the whole file. Procedures for manipulating virtual chunks are formulated, along with the analysis of optimal parameter setup. Evaluation demonstrates that virtual chunks improve scientific applications' I/O throughput by up to 2X speedup at large scale.

Our future work primarily lies in devising an automatic mechanism to update virtual chunks. Machine learning techniques could possibly be leveraged, for example by analyzing sufficient training data to predict its future access patterns. 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 [47–49] on incremental algorithms could possibly be leveraged. We also plan to explore the feasibility to leverage GPUs to speedup the virtual chunk processing to achieve higher storage and I/O performance, as prior work [50] shows that GPUs have the potential to fundamentally change the conventional wisdom of distributed and parallel file systems.

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. This work was supported in part by the National Science Foundation under awards OCI-1054974 (CAREER). This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain, *Towards Data Intensive Many-Task Computing*, T. Kosar, Ed. IGI Global, 2012.

[2] LZO, "http://www.oberhumer.com/opensource/lzo," Accessed September 5, 2014.

[3] bzip2, "http://www.bzip2.org," Accessed September 5, 2014.

[4] 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.

[5] 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.

[6] 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.

[7] HDF5, "http://www.hdfgroup.org/hdf5/doc/index.html," Accessed September 5, 2014.

[8] NetCDF, "http://www.unidata.ucar.edu/software/netcdf," Accessed September 5, 2014.

[9] 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.

[10] 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 (FAST)*, 2002.

[11] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, "FusionFS: Towards supporting data-intensive scientific applications on extreme-scale distributed systems," in *Proceedings of IEEE International Conference on Big Data*, 2014.

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

[13] GCRM, "http://kiwi.atmos.colostate.edu/gcrm/," Accessed September 5, 2014.

[14] SDSS Query, "http://cas.sdss.org/astrodr6/en/help/docs/realquery.asp," Accessed September 5, 2014.

[15] Intrepid, "https://www.alcf.anl.gov/user-guides/intrepid-challenger-surveyor," Accessed September 5, 2014.

[16] 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.

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

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

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

[20] 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.

[21] 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.

[22] T. Li, R. Verma, X. Duan, H. Jin, and I. Raicu, "Exploring distributed hash tables in highend computing," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 3, Dec. 2011.

[23] 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.

[24] 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.

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

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

[27] 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.

[28] K. B. Ferreira, R. Riesen, D. Arnold, D. Ibtesham, and R. Brightwell, "The viability of using compression to decrease message log sizes," in *Proceedings of International Conference on Parallel Processing Workshops*, 2013.

[29] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann, "McrEngine: A scalable checkpointing system using data-aware aggregation and compression," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[30] M. S. Bouguerra, A. Gainaru, L. B. Gomez, F. Cappello, S. Matsuoka, and N. Maruyam, "Improving the computing efficiency of hpc systems using a combination of proactive and preventive checkpointing," in

Parallel Distributed Processing, IEEE International Symposium on, 2013.

[31] M. Noeth, J. Marathe, F. Mueller, M. Schulz, and B. de Supinski, "Scalable compression and replay of communication traces in massively parallel environments," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC)*, 2006.

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

[33] 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.

[34] 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.

[35] 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.

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

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] I. Raicu, I. Foster, A. Szalay, and G. Turcu, "AstroPortal: A science gateway for large-scale astronomy data analysis," in *TeraGrid Conference*, June 2006.

[45] M. Wilde, I. Raicu, A. Espinosa, Z. Zhang, B. Clifford, M. Hategan, S. Kenny, K. Iskra, P. Beckman, and I. Foster, "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers," *Journal of Physics: Conference Series*, vol. 180, no. 1, 2009.

[46] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *Services, IEEE Congress on*, 2007.

[47] 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.

[48] 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.

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

[50] 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.