Exploring Data Compression in Distributed File Systems

Dongfang Zhao, Ioan Raicu
Department of Computer Science
Illinois Institute of Technology
dzhao8@hawk.iit.edu, iraicu@cs.iit.edu

Abstract*—The ever-growing imbalance between computation and I/O is one of the fundamental challenges for current petascale and future exascale systems. As these large systems become more compute heavy by design, it is becoming harder to support data-intensive applications with adequate performance. To narrow the gap between computation and I/O, parallel and distributed file systems have been introduced. A popular approach to further ameliorate the I/O pressure is data compression, which aims to reduce the data size and stress on the network. However, it unsurprisingly brings new challenges such as computational overhead, programmability, and co-design with other components in the orchestra of large systems. In order to reduce the computation overhead of data compression in these large scale systems, this paper proposes a new compression mechanism, namely multi-reference compression (MRC), by introducing multiple references to the underlying physical chunks. These multiple references across the physical chunks enable an efficient decompression of requested chunk subsets, thus bypassing the overhead on decompressing the data outside of the requested range. To make MRC easy to use by scientific computing applications, we have adopted the FUSE kernel module to support POSIX. The MRC system can be used in conjunction with parallel file systems (e.g. GPFS) with no modifications needed to either the application or the high level I/O libraries (e.g. netCDF, HDF5). Towards achieving even higher I/O performance, we have proposed and executed a tight integration of MRC with the FusionFS distributed file system, and explored MRC's impact on basic file system components such as metadata management and data movement. We show how MRC is able to improve parallel and distributed file system performance by up to 2X over these file systems without compression.

***Keywords*-data compression; parallel file systems; distributed file systems**

I. Introduction

Today's science generates data at an unprecedented rate. For instance, fusion science data are output at 2 gigabytes/second per core, and 2 petabytes/second of checkpoint data need to be stored every 10 minutes [17]. This amounts to about 3.5 terabytes/second, which is beyond the capability of today's fastest file system (1.4 terabytes/second, reported in [12]). The conventional wisdom to address the I/O bottleneck is to parallelize the data I/O by splitting data into smaller chunks and process them concurrently on massive nodes. Data compression is another way to addressing the I/O bottleneck by shrinking the chunk size and effectively reducing the load on the network and storage sub-system. While data compression indeed reduces the I/O size, it brings several new challenges: computational overhead, programmability, and co-design of file systems.

Computational overhead. The reduced file size does not come for free: the computation of (de)compressing data costs scarce system resources such as CPU cycles and memory usage, and more importantly, takes time that hurts the end-to-end I/O throughput. While the cost on computation is worthwhile in the hope of being outweighed by the gain on reducing the data size, the decompression in many cases induces a significant overhead: the system still needs to decompress the entire chunk even though only a small piece of data needs to be retrieved. Although reducing the chunk size seems to be a plausible solution, it would reduce the compression ratio by compressing a relatively small chunk as discussed in our previous work [50].

Programmability. Most state-of-the-art compression techniques are stand-alone libraries, implying a programming burden to the application developers to manually modify the application source code, or to integrate the compression method to either the high-level I/O library (e.g. netCDF [26], HDF5 [5]) or the MPI-IO middleware (e.g. ROMIO [9]). Our previous work [13] fit in this category by integrating a light-weight and efficient compression method into the netCDF library.

Co-design of file systems. Many existing efforts of embracing data compression in large scale systems reply on introducing an ad-hoc component into the popular parallel file systems (e.g. GPFS [39], PVFS [20], Lusture [40]), and inevitably needs to compromise with the design of these systems. In particular, there is an increasing concern on the scalability of their system designs at extreme scale (e.g. exa-scale). Our previous work [28], [27] showed that the distributed key-value store could be a strong candidate for excellent scalability compared to conventional parallel file systems. However it is still an open question on how to seamlessly support efficient data compression in POSIX-compliant distributed file systems and retain the scalability at large scale.

To solve the dilemma on the chunk size that affects both the compression ratio and the computation overhead, we introduce Multi-Reference Compression (MRC), which significantly reduces the computation overhead while only adding negligible extra information to the compressed data. MRC is not a new compression algorithm or a variant of any, but a general approach that is applicable to specific algorithms by trading a little space with significantly less computation overhead. We provide a quantitative analysis on the optimal configuration of the MRC mechanism, and discuss alternatives to our approach within the context of parallel and distributed file systems.

We make MRC easily adoptable into existing parallel file systems by implementing it with the FUSE framework to support POSIX interface and deploy it as a mount point on each compute node. Thus, to take advantages of MRC, no modification is needed to the application, the high-level I/O library, or the I/O middleware. We evaluate the loosely-coupled MRC system by mounting it to GPFS [39] on an IBM BlueGene/P [1] supercomputer at up to 1024 cores.

To investigate how MRC can be seamlessly integrated into the next generation of scalable storage systems, we design and implement a new distributed file system —FusionFS— with built-in MRC support. We discuss our experience in designing and implementing the metadata management, the data movement, and how data compression interacts with them. We evaluate FusionFS on a commodity 64-node Linux cluster, a 512-node enterprise-class cluster, and an IBM BlueGene/P [1] supercomputer at 2048-core scale.

In summary, this paper makes the following contributions:

1) ***Propose and analyze the multi-reference compression (MRC) mechanism to improve the end-to-end I/O throughput of parallel and distributed file systems***
2) ***Design and implement the integration of the MRC system with the FusionFS distributed file system***
3) ***Evaluate MRC as a loosely-coupled middleware for parallel file systems, as well as a system component for distributed file systems at large scales up to 2048 cores, delivering up to 2X improved throughput***

The remainder of this paper is organized as follows. Section II presents the MRC mechanism. We describe the design and implementation of FusionFS in Section III. Section IV evaluates MRC and FusionFS. We review additional related work in Section V. Section VI concludes this paper.

II. Multi-Reference Compression

A. Overview

When compressing the original data, MRC logically segments the chunk with equidistant starting points (i.e. references), and appends these references to the end of the compressed chunk. These references would allow future reads to only decompress the subset from the lowest upper reference point, as opposed to the very beginning of the chunk. This "incremental" idea was also proposed in author's previous work for large-volume data mining [49], [29], [48]. Two concerns arise though: (1) adding these references would decrease the compression ratio; (2) how many references should we choose to optimize the end-to-end I/O throughput (duplicating every data entry as a reference would only result in compressed data that is twice larger than the original data). For the first question, we will show that adding a very small number of references would have a significant impact on reducing the I/O cost of data-intensive applications (in Figure 9). For the second question, we will provide an analysis on choosing the optimal number of references in §II-C.

Reference points are evenly distributed across all the data entries in the current design of MRC. This is based on the assumption that the data access pattern has a uniform distribution. If there are some "hot data" that are accessed more frequently than others, it makes sense to allocate more reference points in this area. That said, it is desired that the positions of the reference points could be dynamically updated according to user's access pattern. This dynamic approach would require a more delicate approach to dealing with a series of new challenges, e.g. what metric(s) to be used to logically allocate references in accordance to the access pattern, how to efficiently implement the mapping between the lowest upper reference and the requested starting point, etc. These open questions will be studied quantitatively in our future work.

B. Method

We illustrate how MRC works by introducing an over-simplified example on XOR-based delta compression, as shown in Figure 1. The idea of XOR-based delta compression is very straightforward: storing the XOR difference between every pairs rather than the raw data. Suppose the original file has 8 data entries, and we decide to have 2 reference points on Data 0 and Data 4. So in the compressed file, we store 7 deltas and 2 reference points copied from Data 0 and Data 4. Then when users need to read Data 7, we first copy the nearest upper reference point to the beginning of the restored file (i.e. Ref 1 in this case), and incrementally XOR the restored data and the deltas.

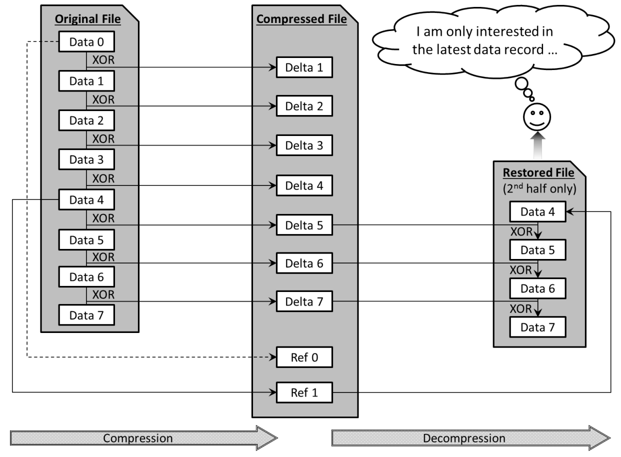


Fig. 1. Two-Reference MRC on XOR compression

Formally, the procedure to compress a file is described in Algorithm 1. For the sake of clear representation, we assume the file content could be split into a logical array, so that each element could be referenced by an index. The first phase of the compression algorithm is to encode the original data entries to the increments of every pairs of neighbor entries in the original file, as shown in Lines 2 - 4. The second phase is to append P reference points (i.e. partitions) to the end of the compressed file, as shown in Lines 6 - 8.

Algorithm 1 Compress a file

```
Require: $F^d$ is the original file to be compressed; $F^e$ is the name of the compressed file; $P$ is the number of partitions to be applied to the original file
Ensure: $F^e$ could be used to recover the content of $F^d$
1: SIZE $\leftarrow$ $F^d$.size()
2: for (int i = 1; i < SIZE; i++) do
3:     $F^e$[i] $\leftarrow$ encode($F^d$[i], $F^d$[i - 1])
4: end for
5: BS $\leftarrow$ SIZE / P
6: for (int j = SIZE; j < SIZE + P; j++) do
7:     $F^e$[j] $\leftarrow$ $F^d$[BS $\times$ (j - SIZE)]
8: end for
```

The decompression procedure, as shown in Algorithm 2, decides the nearest upper reference point, and restores the original data by applying the inverse function of `encode()` (i.e. `decode()`). The nearest upper reference point is located as IDX in Line 2, and GAP indicates the distance between the user-requested starting address and the lowest upper reference point in Line 3. Lines 4 - 7 restore the original data entries by incrementally applying `decode()` from the lowest upper reference point to the end of the requested data. This procedure is applicable to both the entire chunk and its subsets.

Algorithm 2 Decompress a (portion of) file

```
Require: $F^e$ is the encoded file to be decompressed; $F^{d-}$ is the decompressed data for the chunks to be decompressed; $SIZE$ is the original file size; $P$ is the number of partitions to be applied to the original file; $BASE$ is the starting position of the requested (chunks of) data; $LEN$ denotes the number of requested data entries
Ensure: $F^{d-}$ is identical to the (portion of) original file
1: BS $\leftarrow$ SIZE / P
2: IDX $\leftarrow$ BASE / BS
3: GAP $\leftarrow$ BASE % BS
4: $F^{d-}$[0] $\leftarrow$ $F^e$[$F^e$.size() - P + IDX + 1]
5: for (int i = 1; i < GAP + LEN; i++) do
6:     $F^{d-}$[i] $\leftarrow$ decode($F^{d-}$[i-1],$F^e$[BASE-GAP+i])
7: end for
```

In both algorithms, encode() and decode() functions indicate the compression and decompression methods, respectively. They are not necessarily XOR operations as in the illustrating example (Figure 1). In a more general sense, They do not even have to calculate the metric on a pair of neighbors: as long as the compressing method takes a "reference-increment" strategy, encode() and decode() could be implemented accordingly.

C. Analysis

This section answers this question: how many references we should pick to achieve the maximal throughput. By "maximal throughput", we mean the maximal throughput in the worst case where all data are compressed followed by the request (i.e. decompression) of the very last data entry. In general, more reference points consume more storage space, but yield a better chance of a closer lowest upper reference point, which in turn improve the decompression throughput. On the other hand, more references imply longer time to write the compressed data to storage, which reduces the overall throughput. Moreover, weights should be carefully assigned to read and write, respectively. For example, an application with 10:1 read/write ratio should have a higher weight for read than write. Also, we should differentiate write and read bandwidths for data compression and decompression, respectively. We define the parameters for the analysis on the optimal number of reference points in Table I.

TABLE I
MRC PARAMETERS

Variable	Description
B_r	Read Bandwidth
B_w	Write Bandwidth
W_i	Weight of input
W_o	Weight of output
S	File Size
N	Number of Data Entries
R	Number of Reference Points

The overhead to write the extra R reference points for data compression is

$$T_c = \frac{R \cdot S \cdot W_o}{N \cdot B_w},$$

and we would save the following time in decompression:

$$T_d = \frac{(S - \frac{S}{R}) \cdot W_i}{B_r}.$$

Now, we want to maximize

$$F(R) = T_d - T_c.$$

By taking the derivative on R (suppose $\hat{R}$ is continuous) and solving the following equation

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

we have

$$\hat{R} = \sqrt{N \cdot \frac{B_w}{B_r} \cdot \frac{W_i}{W_o}}.$$

Note that

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

since all parameters are positive. And because R is an integer, the optimal R is:

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

Thus we just show that, in order to guarantee the highest end-to-end throughput, the number of reference points should be set roughly to the squared root of the product of 3 factors: the total number of data entries, the ratio of the write bandwidth over the read bandwidth, and the ratio of input weight over output weight. A simplified version of this rule can be stated as: the number of references should be set to the squared root of the total number of data entries, on condition that the read and write throughput/weights are comparable.

D. Discussions

When we designed MRC, we had several alternative designs. This section discusses these ideas, and pinpoints their potential limitations and tradeoffs.

Why does MRC not store references in-place? Instead of storing all reference points at the end of the file, another (and maybe more intuitive) option is to keep them in place. This in-place design offers two benefits over the current MRC design: (1) it saves space of $(P-1)$ encoded chunks (e.g. Delta 4 is not needed in Figure 1); (2) it avoids the computation on locating the lowest upper reference at the end of the chunk, as shown in Line 4 of Algorithm 2. We argue that the space saving of the first benefit is insignificant because encoded chunks are typically much smaller than the original chunks, not to mention this gain is factored by a relatively small number of reference points (comparing to the total number of data entries). The second benefit on saving computation time is also limited. Even though reducing computation overhead is one of MRC goals, the CPU time on locating the reference at the end of the chunk is almost negligible compared to compressing all the data entries. The most critical drawback of the in-place method is, however, on the huge overhead on decompressing large data. For example in Figure 1, if the user requests the entire chunk, then the user needs to read 2 raw data points: Data 0 (i.e. Ref 0) and Data 4 (i.e. Ref 1). Note that Data 0 and Data 4 are original data entries, and are typically much larger than the deltas. Thus, it would induce significantly more overhead by reading these in-place reference points. This issue does not exist in MRC, since all reference points are stored at the end, and the user only needs to retrieve one reference (i.e. Ref 0) and decode with those small deltas.

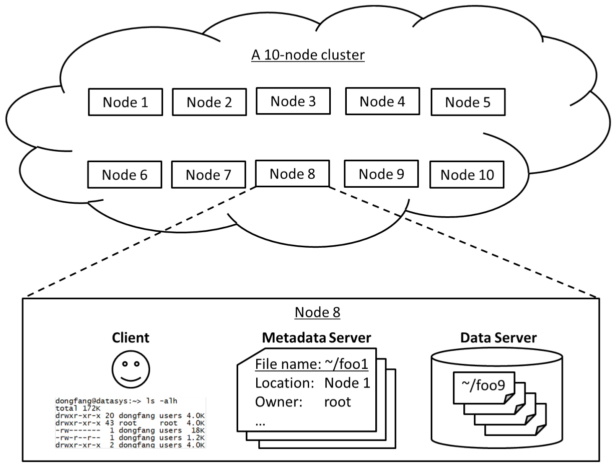
What if the data entries are not sorted by the natural ordering of increments? This would indicate the compression ratio would be low, and reduce the benefit of MRC in terms of I/O throughput. In this case, we propose to add a preprocessing component for the original data, which tries to explore the inner similarity within the chunk and reorder the data if possible. MRC does not provide such a built-in mechanism at the moment, but would employ some existing systems, for example DERD [15], which is a framework that efficiently determines if a sufficient resemblance exists between two objects in a relatively large collection.

What if the data is not compressible at all? MRC itself is not a compression algorithm, but a mechanism to reduce the computation overhead of the underlying compression method. It assumes the compression method to be applied to could yield a decent compression ratio, and selecting an appropriate compression method is beyond what this paper is concerned. Nevertheless, if a file (or chunk) is hardly compressible due to a bad choice of the compression method and/or the workload characteristic, compressing it would only degenerate the performance [24]. In this case, it would be pointless to apply MRC to such a bad choice of compression method. Therefore it is desirable to have a mechanism to check if the underlying compression method is effective. MRC could leverage existing systems (e.g. [21]) to determine if it is worth applying the compression at all.

III. The FusionFS distributed file system

A. Overview

The design principle of FusionFS [45] is to fully exploit the available resources and avoid any centralized component. Thus, we make each participating node play three roles at the same time: client, metadata server, and data server, as show in Figure 2. Users can log in any node to interact with the entire system. Each node is able to pull the global view of all the available data by the single metadata name space, even though metadata is physically distributed on all the nodes. Each node stores parts of the entire metadata and data at its local storage.



```
dongfang@datasys:~> ls -alh
total 172K
drwxr-xr-x 20 dongfang users 4.0K
drwxr-xr-x 43 root     root  4.0K
-rw-------  1 dongfang users  18K
-rw-r--r--  1 dongfang users 1.2K
drwxr-xr-x  2 dongfang users 4.0K
```

Fig. 2. Roles of participating nodes in FusionFS

Even though both are fully distributed on all nodes, 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 more flexible strategies on both, respectively. A similar idea has also been proposed in our previous work [43] for reducing the number of execution nodes for Byzantine replication. In this work, we utilize distributed hash tables to reach load balance for metadata, but for file data we apply a different strategy (driven by the application I/O access pattern with periodic load balance). Distributed hash tables are not selected as the underlying data store because data locality takes a higher priority for high aggregate throughput over good load balance. To further exploit data locality, a remote read would be, if possible, replaced by rescheduling the job to the destination node of the desired data, followed by a local read. On top of the data movement strategies, a local MRC component works as a transparent filter that automatically compresses the assigned data before writing to the persistent storage, and decompresses the compressed data to return the clients the raw data.

These ideas are illustrated in Figure 3, which is an oversimplified 2-node system. The metadata on Node 1 has nothing to do with the files stored on Node 1. Moreover, the metadata is distributed on both nodes, and each node harnesses the data locality by only writing to its local storage and trying to reschedule the job to realize local read if possible, e.g. $Client@Node1 \longmapsto Client@Node2$. Both nodes have their own MRC component to compress and decompress the requested data between the client and the persistent storage via a POSIX API. We will discuss more details on distributed metadata in §III-B, data movement in §III-C, and MRC in §III-D.

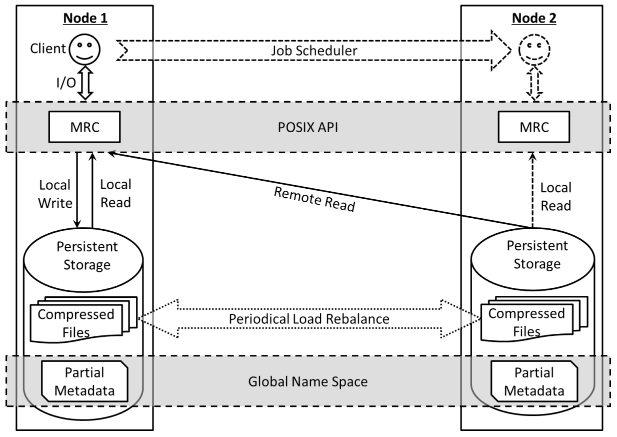


Fig. 3. An example FusionFS deployed on two nodes.

B. Metadata management

We illustrate how to achieve the global single name space by maintaining partial metadata views on local nodes in Figure 4, where an oversimplified example of two nodes is considered. Node 1 and Node 2 only physically keep sub-graphs of the entire metadata (top left and top right portion of the figure), and store the information into the local component of the global Distributed Hash Table (DHT). Nevertheless, client could interact with the DHT for any key-value pair no matter if it is on the local storage or on some remote nodes. Thus, what any client could see is actually the global name space of the entire system, as shown in the bottom portion of the figure. In some sense, DHT is the translator between local partial metadata and the global name space.

The global name space does not need to be aggregated or flushed when local changes occur. In other words, any changes in the local metadata storage are immediately visible to the global name space without extra processing. It is an analogy that modifying a subgraph will automatically update the topology of the entire graph.

In essence, the directory hierarchy of traditional file systems could be abstracted as a tree. In general, there are two options to store a tree: adjacency matrix and adjacency lists. An

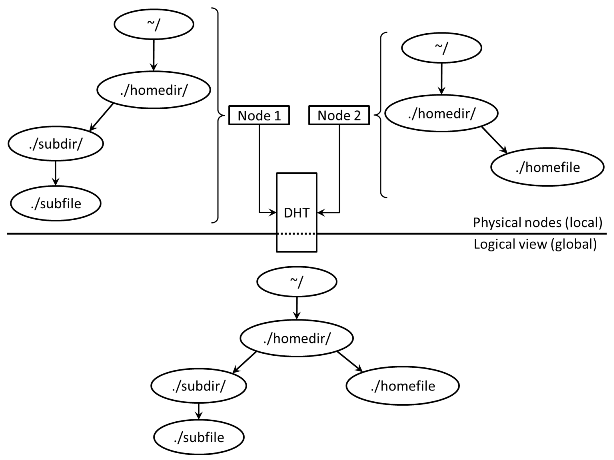
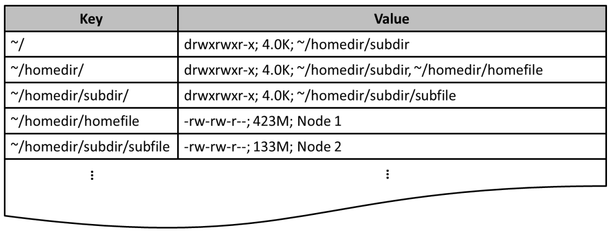


Fig. 4. Directory hierarchy in local physical nodes and global name space

adjacency matrix allows a constant-time lookup operation on the relationship between two vertices, and requires $O(n^2)$ storage, where n is the number of nodes in the graph. The adjacency lists, on the other hand, require much less space in practice particularly when the adjacency matrix is sparse, which is the case for file systems. Another advantage of adjacency lists is that it allows fast content retrieval of a specific directory (i.e. constant time to retrieve the parent node followed by sequential reads of the children), which is a frequent operation in file systems. The potential issue with adjacency lists, however, is that the relationship lookup operation could take up to $O(n)$ in the worst case. However, this lookup operation is relatively rare in file systems. Therefore, we choose adjacency lists as the abstraction of the directory hierarchy, and store them as key-value pairs in the DHT.

To make matters more concrete, Figure 5 shows the distributed hash table in according to the example graph shown in Figure 4. It should be noted this DHT is only a logical view of the aggregation of multiple partial metadata on local nodes (in this case, Node 1 and Node 2). Five entries (3 directories, 2 regular files) are stored in the DHT, with their file names as keys. The value is comprised of a list of properties delimited by semicolons. The first and second portions of the values are for permission flags and file size, respectively. The third portion for a directory is a list of its children delimited by commas, while for regular files is just physical location of the file.



Key	Value
~/	drwxrwxr-x; 4.0K; ~/homedir/subdir
~/homedir/	drwxrwxr-x; 4.0K; ~/homedir/subdir, ~/homedir/homefile
~/homedir/subdir/	drwxrwxr-x; 4.0K; ~/homedir/subdir/subfile
~/homedir/homefile	-rw-rw-r--; 423M; Node 1
~/homedir/subdir/subfile	-rw-rw-r--; 133M; Node 2
⋮	⋮

Fig. 5. A global name space represented by adjacency lists in a DHT

The example in Figure 5 is only to illustrate how the DHT looks like for Figure 4, and we should mention the following clarifications. (1) In real systems there is more metadata information stored in the values, such as modification time, owner ID, etc, that are commonly seen in i-nodes. We do not list all of them here for the sake of limited space. (2) What is shown in the value is a simple string delimited by semicolons. This is only for clear presentation to explain what types of information is stored. In implementation, the value is in fact a serialization of the data structure for metadata. The structured metadata is serialized by Google Protocol Buffer [4] before sending over the network to the metadata servers. Similarly, when the metadata is retrieved, we deserialize the blob back into the structure. All the regular i-node information is tracked, plus the list of children or the node location.

It should be noted that, the location information for a regular file has nothing to do with which node this metadata itself resides on. From Figure 4, we know the $homefile$ metadata is stored in Node 2, and the $subfile$ metadata is in Node 1. However, it is perfectly fine for $homefile$ to reside on Node 1, and $subfile$ reside on Node 2, as shown in Figure 5.

Besides the regular metadata information for files, there is a special flag bit in the value indicating if this file is being written. Specifically, any client who requests to write a file needs to sets this flag before opening the file, and will not reset it until the file is closed. This atomic operation guarantees the consistency of the file data.

One issue with our early implementation on metadata was on big directories, where many files exist in the same directory. In particular, when each of thousands of clients writes thousands of files on the same directory concurrently, the value of this directory in the key-value pair in the hash table gets incredibly large. And the client would need to update the old value with the new one, even though the majority of the old value is unchanged. We designed a new append operation that would allow clients to simply update large key-value pairs efficiently. This is similar to the approach used in the Google File System [18]. In the current implementation, when a new file is created (or removed), we simply append the file and its operation to the value rather than updating it immediately. This append operation is atomic, so no inconsistency will be encountered.

To deal with the high concurrency on metadata servers, epoll is used instead of multithreading. The side effect of epoll is that the received message packets are not kept in the same order as on the sender. To address this, a header [message_id, packet_id] is added to the message at the sender, and the message is restored by sorting the packet_id for each message at the recipient. This is efficiently done by a sorted map with message_id as the key, mapping to a sorted set of the message's packets. The server also periodically triggers a garbage collection for the orphan packets to deal with crashed clients, unreliable network, etc.

C. Data movement

Our strategy to achieve high and scalable write throughput is very straightforward: a client only writes data to its local storage. In other words, it is independent write across data nodes, in the sense that no interference exists on the layer of physical data servers. This local-only write is not really independent, in the sense that the metadata on the global name space is up to date. When we discussed the distributed metadata in §III-B, we showed that the metadata, even though scattered on the same set of nodes as data, is completely decoupled from data. However, the metadata is maintained as a single and global view, so that these data-independent writes are all reflected by the distributed metadata.

The aggregate write throughput is obviously optimal: all writes are associated with local I/O throughput and avoids the following two potential overheads commonly seen in other systems: (1) the procedure to determine to which node the data will be written, normally accomplished by pinging the metadata nodes and/or some monitoring services, and (2) transferring the data to a remote node.

The potential issue with local writes is equally obvious: no guarantee for load balance is provided at all. The assumption that all workloads are uniformly deployed on all nodes is too strong, as we have often seen hot spots in large scale systems. As a consequence, a periodical data migration is used to re-balance the workload.

In some cases, the client is writing to a file that is originally stored in another node. Per our policy, the newly written file will not be sent back to the original node. Rather, the metadata of this file will be updated in the metadata. This saves the cost on transferring the file data over the network by a much faster operation on updating the metadata. Question arises though: what if two clients try to write to the same data? The answer is that a distributed lock service (coordination) is available built on top of the atomic cswap (compare and swap) operation on the underlying DHT. This atomic cswap guarantees that in any given period of time, at most one client could modify the content.

Unlike data write, it is impossible to arbitrarily control where the requested data reside. The location of the requested data is highly dependent on the I/O pattern, and the probability of the requested data residing on the local storage is still quite small (i.e. $P = \frac{1}{n}$, where n is the number of nodes) even assuming the I/O has a uniform distribution.

Nevertheless, we could control which node the job is executed on by some distributed work flow systems. When a job on node A needs to read some data on node B, we reschedule the job on node B. The overhead of rescheduling the job is much smaller than transferring the data over the network. In our previous work [35], [37], [36], we detailed this approach, and justified it with a theoretical analysis and experiments on benchmarks and real applications.

Indeed, some I/O pattern cannot avoid remote reading, e.g. merge sort. The data need to be joined together, and shifting the job does not help. In such cases, we could always transfer the requested data from the remote node to the requesting node; prefetching mechanism could be used to ameliorate the impact of this for predictable I/O access patterns.

D. Compression and decompression with multiple references

The MRC compression is implemented in the *fusionfs_write()* interface, which is the handler for catching the write system calls. *fusionfs_write()* compressed the raw data, cached it in the memory if possible, and wrote the compressed data into the file system. The decompression algorithm was implemented in the *fusionfs_read()* interface, similarly. When a read request came in, this function loaded the compressed data (either from the cache or the disk) into memory, applied the decompression algorithm to the compressed data, and passed the result to the end users.

One obvious advantage of MRC implementation is the high possibility of reusing the decompressed data, since the decompressed data will be cached in the local node. Moreover, because the original compressed chunk is logically split into many partitions each of which can be decompressed by itself, it allows a more flexible memory caching mechanism. We have implemented a simple LRU caching mechanism for caching the intermediate data. A more complicated strategy for the globally optimal caching hit rate is beyond the scope of this paper, and will be the subject of future work.

As another plausible option, the (de)compression could be implemented at the file level, i.e. in file open (*fusionfs_open()*) or file close (*fusionfs_close()*). The issue with *fusionfs_open()* is that when the requested data is opened, the system is yet to know which subsets need to be read into memory. Thus, the entire chunk would be decompressed, which makes MRC pointless by wasting the multiple references. Implementing the (de)compression at file close does not have the issue as in file open. However, it implies the decompressed data need to be buffered until all the requested data are completed, i.e. a blocking read. This is clearly not a desirable feature, since many applications assume the blocks that have been read into memory should be available immediately. Thus, file open and file closes are not good places to implement data (de)compression.

Even though the compression is implemented in the *fusionfs_write()* interface, the compressed file will not be 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 I/O throughput. In some scenarios, users are more concerned on high availability rather than the compressing time. In that case, a *fsync()* could be invoked to the (partially) compressed data to ensure these data is available at the persistent storage so that other processes/nodes could start reading them, even though the end-to-end compressing time would get degraded for the original writing process.

IV. Evaluation

Experiments are conducted on three test beds.

1) *IIT-HEC* is a 64-node Linux cluster at Illinois Institute of Technology. Each node has 2 Quad-Core AMD Opteron 2.3GHz processors with 8GB RAM and 1TB Seagate Barracuda hard drive. All nodes are interconnected with 1Gbps Ethernet.
2) *Kodiak* [8] is a 1024-node cluster at Los Alamos National Laboratory. Each node has an AMD Opteron 252 CPU (2.6GHz), 4GB RAM, and 2 Western Digital 7200rpm 1TB hard drives.
3) *Intrepid* [6] is an IBM BlueGene/P supercomputer [1] of 160K cores at Argonne National Laboratory. We carried out experiments on up to 2048 cores on Intrepid. Each node has a 4-core PowerPC 450 processor (850MHz) and 2GB of RAM. A 7.6PB GPFS [39] is deployed on 128 storage nodes.

We evaluate MRC and FusionFS on two representative data sets in scientific computing.

1) *Global Cloud Resolving Model (GCRM)* [3] models the cloud behavior of the entire globe at a fine scale (2-4 km). It is used to analyze to understand cloud's influence on the atmosphere and global climate, and more importantly to predict cloud's behavior and the consequent climate change. The GCRM data is originally stored as the netCDF [26] format. We extract a subset of temperature information at different latitude, longitude, height, and time stamp.
2) *Sloan Digital Sky Survey (SDSS)* [11] contains a huge collection of astronomical information, which records one-quarter of the entire sky in detail including positions and brightnesses of hundreds of millions of celestial objects. The data are available in multiple formats for different platforms. We obtain a subset of the SDSS data set via the SQL query in the web portal [10], which includes all galaxies whose blue surface brightness is within a particular range.

All experiments are 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. Caching effect is carefully precluded by reading a file larger than the on-board memory before the measurement.

A. Compression ratio of MRC

We show how MRC affects the compression ratio with 244.25GB GCRM data. The compression ratio is shown in Figure 6 when different numbers of reference points (1 to 2000) are stored. As expected, the compression ratio decreases when more reference points are added. However, the loss on compression ratio is almost negligible (i.e. within 0.002 between 1 reference and 2000 references).

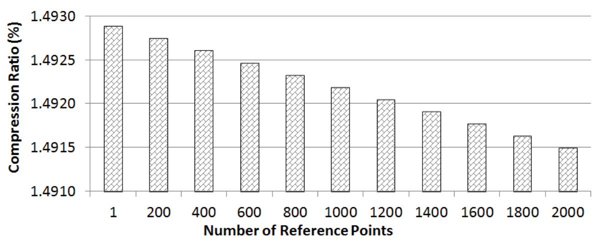


Fig. 6. Compression ratios with MRC

B. Boosting GPFS with MRC

We deploy the MRC implementation on each of the 256 nodes (1024-core) with a FUSE mount point to a 128-node GPFS file system on Intrepid. The dataset is 244.25GB of GCRM climate data. Block size is default to 64KB. Throughput is measured for sequential writes, if not otherwise specified.

Since MRC is implemented with FUSE [2], which adds extra context switches when making I/O system calls, we need to know how much overhead is induced from FUSE. To measure the impact of this overhead, the GCRM dataset was written to the original GPFS and the GPFS+FUSE file system (without MRC), respectively. The difference is only 2.2%, which could be best explained by the fact that in parallel file systems the bottleneck is on networking rather than the latency on local operating systems. Since the FUSE overhead on GPFS is negligible, we will not list two setups (vanilla GPFS and FUSE+GPFS) in the following experiments.

Since more reference points reduce the compression ratio, the I/O time is expected to increase accordingly. However, the extra cost is about negligible (<0.2 second) for 1 - 2000 reference points as shown in Figure 7. The speedup comparing to the original GPFS is also reported.

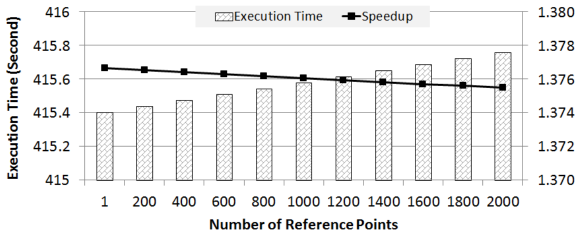


Fig. 7. I/O throughput with MRC

The time breakdown of end-to-end I/O time is shown in Figure 8. The overhead introduced by the compression layer is about 8.4%, and has a slightly smaller portion (hardly visible in the figure though) when increasing the number of reference points. This is because the resultant file size is larger when more reference points are stored, which implies a longer I/O time while the computing cost is relatively constant. The relative low computational overhead infers that for those incompressible data MRC would not cause much performance degradation (compression ratio is 1.492, from Figure 6):

$$\frac{\texttt{Compute\%}}{(\texttt{1-Compute\%})\times\texttt{Ratio}} = \frac{8.4\%}{(1-8.4\%)\times 1.492} = 6.1\%,$$

even though skipping the compression is preferred as discussed in §II-D.

To verify the effectiveness of MRC on reduced decompression overhead, we ran a straightforward application of the routine workload: archival of all the available data followed by the retrial of the latest temperature. The I/O time was reported in Figure 9. We observed that with multiple reference points (200 – 2000), the I/O time reached below 384 seconds

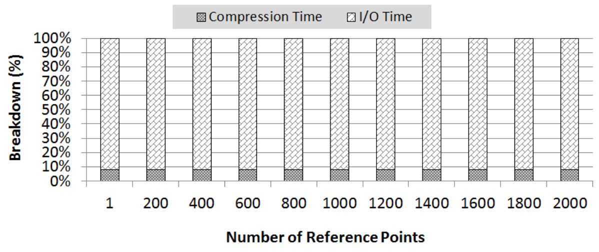


Fig. 8. Breakdown of compute overhead and I/O transfer with MRC

comparing to 501 seconds with a single reference, resulting in 1.31X speedup.

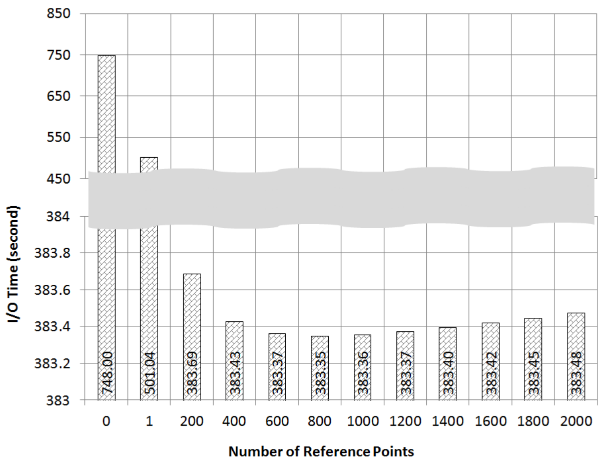


Fig. 9. I/O time to record the GCRM data and retrieve the latest temperature

We then ran the MMAT application [13] that calculated the minimal, maximal and average temperatures on the GCRM dataset. The breakdown of different portions was shown in Figure 10, when compressing the data with $R = 800$ and directly calculating on the raw data (i.e. $R = 0$). After applying the compression layer, the I/O portion was significantly reduced. And this reduction outweighed the overhead introduced by the compression layer, resulting in 1.24X speedup on the overall execution time.

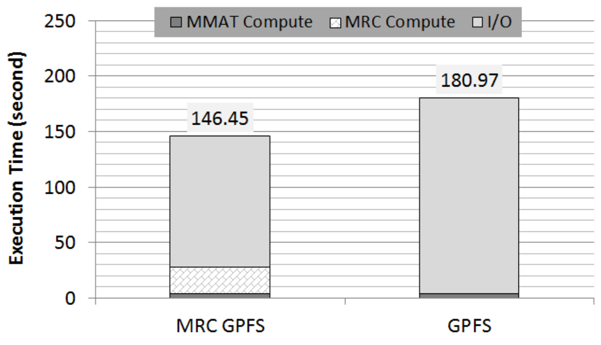


Fig. 10. Execution time of the MMAT application

C. FusionFS performance

We compare the metadata performance between FusionFS and HDFS [42] on Kodiak. Both storage systems have FUSE/POSIX disabled. We have each node create (i.e. "touch") a large number of empty files (with unique names), and we measure the number of files created per second. In essence, each touched file indicates a metadata operation. The aggregate metadata throughput of different scales is reported in Figure 11. The gap between FusionFS and HDFS is about more than 3 orders of magnitude. Note that, HDFS starts to taper off from 128 nodes, while FusionFS keeps doubling the throughput all the way to 512 nodes, ending up with almost 4 orders of magnitude speedup (509022 vs. 57). The reason why HDFS metadata is so slow is twofold: (1) HDFS has a single metadata server that is easily saturated by high concurrency; (2) HDFS' Java overhead is much more significant than the C/C++ implementation of FusionFS.

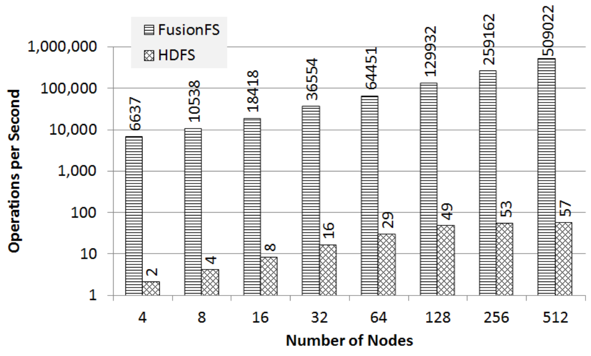


Fig. 11. FusionFS metadata performance on Kodiak

We then compare the metadata performance between FusionFS and PVFS on Intrepid. Both systems turn on the POSIX interfaces. PVFS is deployed on Intrepid with the 1-1-1 mapping between clients, metadata servers and data servers, just like FusionFS. Each client creates 10K empty files on the same directory. The result is reported in Figure 12. FusionFS outperforms PVFS on a single node, which justifies that our metadata optimization for big directory (i.e. append vs. update) is quite efficient. While PVFS is saturated at 32 nodes, FusionFS shows a linear scalability due to its unique design of the DHT-based metadata management.

We illustrate how MRC helps FusionFS to improve the I/O throughput on different data sets in IIT-HEC, as shown in Figure 13. R is set to 2 when MRC 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 a 2X speedup when MRC is enabled (SDSS write: 8206 vs. 4101).

We report the throughput of FusionFS on Intrepid, compared to the default GPFS file system, in Figure 14. Both the read and the write throughput are scalable at up to 2048 cores, which justifies that MRC component does not intervene the scalability of the overall throughput of FusionFS. FusionFS' throughput is orders of magnitude higher than GPFS, which can be best explained by FusionFS' better data locality and the MRC component. In particular, FusionFS reaches over 100GB/s at 2048 cores, which surpasses GPFS' peak throughput of 4.6GB/s at 2048-core scale, as well as the peak throughput of 65GB/s [7] at the full scale of 160K cores.

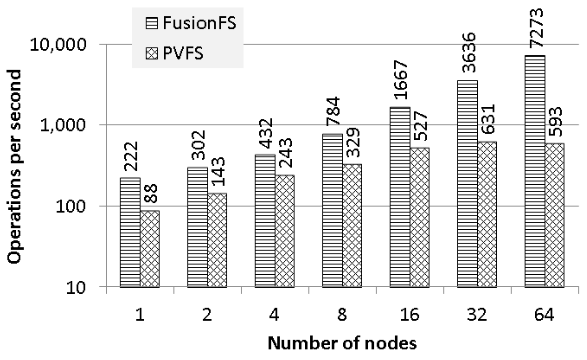


Fig. 12. FusionFS Metadata performance on Intrepid

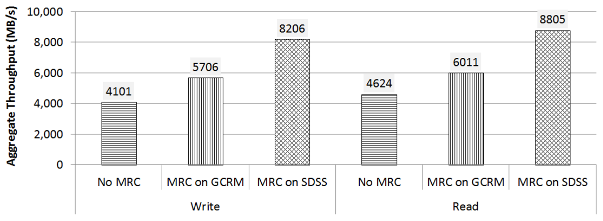


Fig. 13. FusionFS Throughput on IIT-HEC

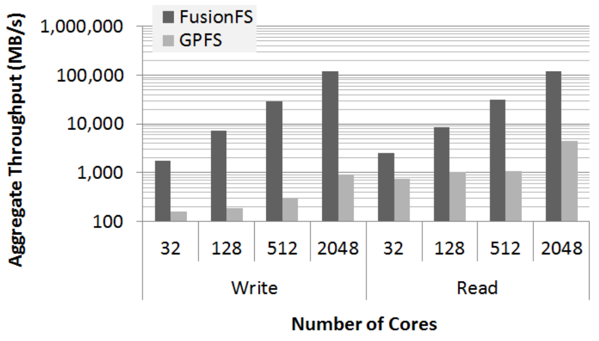


Fig. 14. Throughput on Intrepid

V. Related Work

Some frameworks are proposed as middleware to allow applications call high-level I/O libraries for data compression and decompression, e.g. [13], [38], [23]. 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, FusionFS exploits a finer granularity of chunks and aims to support compression transparently at the file system layer.

Some previous work studied the file system support for data compression. Integrating compression to log-structured file systems was proposed decades ago [14], which suggested a hardware compression chip to accelerate the compressing and decompressing. Later, XDFS [30] described the systematic design and implementation for supporting data compression in file systems with BerkeleyDB [33]. MRAMFS [16] was a prototype file system to support data compression to leverage the limited space of non-volatile RAM. In contrast, FusionFS is a fully-fledged POSIX-compliant distributed file system with built-in efficient MRC support.

Data deduplication is a general inter-chunk compression technique that only stores a single copy of the duplicate chunks (or blocks). For example, LBFS [32] 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 [34] 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 [31]. While deduplication focuses on inter-chunk compressing, MRC 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. [25], [19], [22]. 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. MRC overcomes the large-chunk issue by logically splitting the large chunks with fine-grained partitions while still keeping the physical coherence.

VI. Conclusion

This paper proposes and analyzes the MRC compression mechanism to improve the end-to-end I/O throughput in large scale distributed systems. MRC works as a loosely-coupled middleware on top of parallel file systems, and is integrated into the FusionFS distributed file system with the unique designs of distributed metadata management and location-aware data movement. We deploy MRC-enabled file systems on three test beds from a commodity Linux clusters to a flagship supercomputer, and carry out experiments of both benchmarks and applications at up to thousands of cores. With MRC we observe up to 2X faster I/O throughput for parallel and distributed file systems.

There are two major directions in the future work. First, we will explore how to support inter-node cooperative caching in the data compression layer, to achieve an high cache hit rate from the global perspective by extending our previous work in the single-node setting [46], especially in the contexts of checkpointing [51], reliability [44], and provenance [47], [41]. Second, we will study dynamic reference distributions to better match the data access patterns, which we believe would further improve the I/O performance of parallel and distributed file systems.

Acknowledgments

This work was supported in part by the National Science Foundation under grant OCI-1054974 (CAREER), CNS-1042537, CNS-1042543 (PRObE), and the Office of Biological and Environmental Research, Office of Science, U.S. Department of Energy, under contract DE-ACO2-O6CH11357. 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] Blue Gene. http://en.wikipedia.org/wiki/Blue_Gene.

[2] FUSE Project. http://fuse.sourceforge.net/.

[3] The global cloud resolving model. http://kiwi.atmos.colostate.edu/gcrm/.

[4] Google protocol buffers. http://code.google.com/p/protobuf/.

[5] HDF5. http://www.hdfgroup.org/HDF5.

[6] Intrepid. https://www.alcf.anl.gov/user-guides/intrepid-challenger-surveyor.

[7] Intrepid file system. https://www.alcf.anl.gov/user-guides/intrepid-challenger-eureka-file-systems.

[8] Kodiak. https://www.nmc-probe.org/wiki/Kodiak:Nodes.

[9] ROMIO. http://www.mcs.anl.gov/research/projects/romio/.

[10] SDSS Query. http://cas.sdss.org/astrodr6/en/help/docs/realquery.asp.

[11] Sloan digital sky survey. http://cas.sdss.org/astrodr6/en/.

[12] Supercomputer titan to get world's fastest storage system. http://phys.org/news/2013-04-supercomputer-titan-world-fastest-storage.html.

[13] T. Bicer, J. Yin, D. Chiu, G. Agrawal, and K. Schuchardt. Integrating online compression to accelerate large-scale data analytics applications. IPDPS, 2013.

[14] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. ASPLOS, 1992.

[15] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. USENIX ATC, 2003.

[16] N. K. Edel, D. Tuteja, E. L. Miller, and S. A. Brandt. Mramfs: A compressing file system for non-volatile ram. MASCOTS, 2004.

[17] P. A. Freeman, D. L. Crawford, S. Kim, and J. L. Munoz. Cyberinfrastructure for science and engineering: Promises and challenges. *Proceedings of the IEEE*, 93(3):682–691, 2005.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. SOSP, 2003.

[19] 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. IPDPS, 2012.

[20] I. F. Haddad. PVFS: A Parallel Virtual File System for Linux Clusters. *Linux J.*, 2000(80es), Nov. 2000.

[21] D. Harnik, R. Kat, O. Margalit, D. Sotnikov, and A. Traeger. To zip or not to zip: Effective resource usage for real-time compression. FAST, 2013.

[22] J. Jenkins, I. Arkatkar, S. Lakshminarasimhan, N. Shah, E. Schendel, S. Ethier, C.-S. Chang, J. Chen, H. Kolla, S. Klasky, R. Ross, and N. Samatova. Analytics-Driven Lossless Data Compression for Rapid In-situ Indexing, Storing, and Querying. In *Database and Expert Systems Applications*, Lecture Notes in Computer Science, 2012.

[23] 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. SC, 2012.

[24] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and performance evaluation of lossless file data compression on server systems. SYSTOR, 2009.

[25] 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. HPDC, 2013.

[26] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. SC, 2003.

[27] T. Li, R. Verma, X. Duan, H. Jin, and I. Raicu. Exploring distributed hash tables in highend computing. *SIGMETRICS Perform. Eval. Rev.*, 39(3):128–130, Dec. 2011.

[28] 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. IPDPS, 2013.

[29] R. Lohfert, J. J. Lu, and D. Zhao. Solving sql constraints by incremental translation to sat. In *Proceedings of the 21st International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems: New Frontiers in Applied Artificial Intelligence*, IEA/AIE '08, pages 669–676, 2008.

[30] J. P. MacDonald. File system support for delta compression. Technical report, University of California, Berkley, 2000.

[31] D. Meister, A. Brinkmann, and T. Süß. File recipe compression in data deduplication systems. FAST, 2013.

[32] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. SOSP, 2001.

[33] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. USENIX ATC, 1999.

[34] K. Park, S. Ihm, M. Bowman, and V. S. Pai. Supporting practical content-addressable caching with czip compression. USENIX ATC, 2007.

[35] I. Raicu, I. Foster, M. Wilde, Z. Zhang, K. Iskra, P. Beckman, Y. Zhao, A. Szalay, A. Choudhary, P. Little, C. Moretti, A. Chaudhary, and D. Thain. Middleware support for many-task computing. *Cluster Computing*, 13(3):291–314, Sept. 2010.

[36] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain. Towards data intensive many-task computing. *Data Intensive Distributed Computing: Challenges and Solutions for Large-scale Information Management*, 13(3):28 – 73, Sept. 2012.

[37] I. Raicu, I. T. Foster, Y. Zhao, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain. The quest for scalable support of data-intensive workloads in distributed systems. HPDC, 2009.

[38] 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. HPDC, 2012.

[39] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. FAST, 2002.

[40] P. Schwan. Lustre: Building a file system for 1,000-node clusters. In *Proceedings of the linux symposium*, page 9, 2003.

[41] C. Shou, D. Zhao, T. Malik, and I. Raicu. Towards a provenance-aware distributed filesystem. In *5th USENIX Workshop on TaPP, NSDI 2013*, Lombard, IL, 2013.

[42] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. MSST, 2010.

[43] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. SOSP, 2003.

[44] 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. IEEE CLUSTER '13, 2013.

[45] D. Zhao and I. Raicu. Distributed file systems for exascale computing. SC, Doctoral Showcase, 2012.

[46] 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*, IPDPSW '13, pages 1997–2006, 2013.

[47] D. Zhao, C. Shou, T. Malik, and I. Raicu. Distributed data provenance for large-scale data-intensive computing. IEEE CLUSTER '13, 2013.

[48] D. Zhao and L. Yang. Incremental construction of neighborhood graphs for nonlinear dimensionality reduction. In *Proceedings of the 18th International Conference on Pattern Recognition - Volume 03*, ICPR '06, pages 177–180, Washington, DC, USA, 2006. IEEE Computer Society.

[49] D. Zhao and L. Yang. Incremental isometric embedding of high-dimensional data using connected neighborhood graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(1):86–98, Jan. 2009.

[50] D. Zhao, J. Yin, and I. Raicu. Improving the i/o throughput for data-intensive scientific applications with efficient compression mechanisms. SC, 2013.

[51] D. Zhao, D. Zhang, K. Wang, and I. Raicu. Exploring reliability of exascale systems through simulations. In *Proceedings of the High Performance Computing Symposium*, HPC '13, pages 1:1–1:9, 2013.