Discovering Structure in Unstructured I/O

Jun He*, John Bent†, Aaron Torres‡, Gary Grider‡, Garth Gibson§, Carlos Maltzahn¶, Xian-He Sun*
*{jhe24, sun}@iit.edu †john.bent@emc.com ‡{agtorre,ggrider}@lanl.gov §garth@cs.cmu.edu ¶carlosm@soe.ucsc.edu

***Abstract*—Checkpointing is the predominant storage driver in today's petascale supercomputers and is expected to remain as such in tomorrow's exascale supercomputers. Users typically prefer to checkpoint into a shared file yet parallel file systems often perform poorly for shared file writing. A powerful technique to address this problem is to transparently transform shared file writing into many exclusively written as is done in ADIOS and PLFS. Unfortunately, the metadata to reconstruct the fragments into the original file grows with the number of writers. As such, the current approach cannot scale to exaflop supercomputers due to the large overhead of creating and reassembling the metadata.**

In this paper, we develop and evaluate algorithms by which patterns in the PLFS metadata can be discovered and then used to replace the current metadata. Our evaluation shows that these patterns reduce the size of the metadata by several orders of magnitude, increase the performance of writes by up to 40 percent, and the performance of reads by up to 480 percent. This contribution therefore can allow current checkpointing models to survive the transition from peta- to exascale.

I. Introduction

Recent projections by the United States' Department of Energy have predicted extremely challenging storage requirements for exaflop supercomputers. The primary storage driver is checkpointing and the current projections specify that checkpoints of 32 petabytes in size should complete in 300 seconds. The bulk of computational scientists seem to prefer checkpointing into a single checkpoint file over checkpointing into a directory containing tens of thousands of checkpoint fragments [1]. Therefore the performance of shared file writing is critical for effective HPC.

Unfortunately, many otherwise scalable file systems suffer poor performance when a large number of concurrent processes write to the same file [2]. The most powerful way to fix this problem is to transparently transform the representation of a concurrently written file into many exclusively written file fragments, as is done by ADIOS [3] and PLFS [2]. Recent PLFS development however has hit a performance wall as the amount of internal metadata required to reconstruct the file fragments grows with the number of writers. Current petascale size checkpoints are challenging and exascale would be impossible without compressing the internal metadata.

PLFS transparently transforms shared-file writing into log-structured file-per-process writing. As each process writes to the logical file, PLFS appends that data to a unique logfile for that process and creates an index entry in a unique index file for that process that creates a mapping between the bytes within the logical file and their physical location within the logfiles. Therefore, as applications grow in size, the number of index files, and the number of index entries, grows accordingly. The overhead of the index creation is slight, but noticeable, during writes. Read overhead however is much larger; since a reader might read from any portion of the file, every index file and every index entry must be read. There is latency overhead as the on-disk index files are read into an in-memory data structure and there is the disk capacity overhead as well as the in-memory footprint. For example, an anonymous application at Los Alamos National Laboratory (LANL App3) writes a file of 4 GB and creates aggregated index size of 192 MB for a run with 64 processes [4]. In this case, when reading the file with 64 processes, the total memory index footprint is 12 GB since each process has to hold a copy of the whole index. Earlier work [5] addresses the latency of index read; this paper extends that work by further reducing the latency as well as the disk and memory usage. The index compression result of LANL App3 can be found in Figure 1.

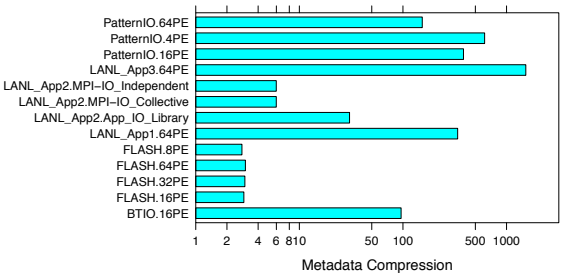

Fig. 1. Compression rates for the index of real applications and benchmarks by discovering structures and representing them in a compact way. The compression rate is represented as (Uncompressed size)/(Compressed size).

We achieve this using a gray-box technique [6] of rediscovering valuable information which was lost as data moved across the POSIX interface. In this case, we rediscover the structure of the checkpoint using pattern detection. Checkpoints are typically the conversion of a distributed data structure into a linear array of bytes. High-level middleware abstractions such as views within MPI and the data types within HDF and NetCDF allow the user to describe the structure of their data structure (e.g. the number and size of the dimensions in a mesh). The middleware then will use the restrictive interface of POSIX to store the data structure with a sequence of writes. Since these writes are storing a structured data set, they will typically follow a regular pattern. By discovering the pattern, PLFS can replace the index entry per write with a single pattern entry describing them all thereby converting the size of the index from O(n) to a small constant value. An alternative approach to reduce the metadata would be to *clean* the logfiles into a single *flat* file. However, this

cleaning is expensive and notoriously difficult; additionally, earlier work [7] has shown that flattening files can lead to slower read performance.

Our main contribution is to propose and evaluate effective algorithms and representations to discover and describe structures in unstructured I/O. Although we note that this technique is useful in a variety of cases such as pattern-aware prefetching and block pre-allocation, as well as for metadata reduction within systems such as SciHadoop [8], we demonstrate its value in this paper exclusively with an evaluation of the compressibility of the PLFS index.

As shown in Figure 1, which presents the compression rates for several benchmarks and real applications, we are able to reduce the size of the index by up to several orders of magnitude. As we will see in Section V, this structure discovery also results in performance improvements of up to 40 percent for writes and up to 480 percent for reads. We also present a visualization of the write patterns of the MILC [9] application to illustrate the inherent structure which our algorithms successfully detect.

I/O access patterns have been studied for decades [1] [10] [11] [12] [13] [14] [15]. However, the majority of them are of coarse granularity or they do not provide effective ways to recognize patterns. A series of other work [16] [17] [18] uses statistics approaches to detect and represent patterns, which are lossy and cannot serve our needs for discovering and storing exact structures.

II. Design of a Pattern Structured PLFS

By default, when an application opens a PLFS file for writing, PLFS opens two *"dropping"* files in the underlying file system for each process involved in the writing of that file. One is for data written by that process, called a *data dropping*, and another is for its associated index. Indices maintain a mapping from offsets within the PLFS file to the physical offsets in each data dropping file. When a read request is performed (e.g. *read(fd, off, len)* is called), PLFS queries the index to find where that actual data resides within the data dropping files. The key variables in a current index entry are:

- *logical offset*: where the data is, from the application's perspective in a single logical file;
- *length*: number of bytes written;
- *physical offset*: the physical offset within a contiguous dropping file;
- *chunk id*: this is the ID of the dropping file where the data is.

Figure 2 is an example of how PLFS works today. When an application makes many small writes, the size of the index will become correspondingly large. This may result in consumption of significant amounts of memory when an application reads a PLFS file due to storing the indices in memory. To use less memory, an alternate option is to not cache entire index data but to access them on disk whenever it is necessary. However, this will be very slow since PLFS has to conduct I/O for each index access.

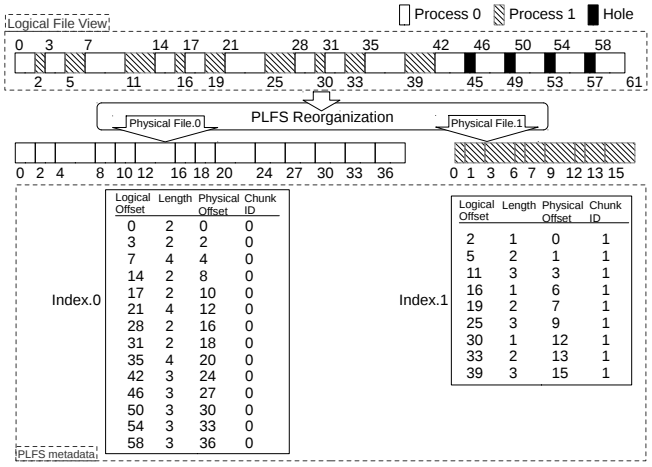

Fig. 2. An example of two processes writing to a traditional PLFS file. If the application writes a lot of data in small extents, the indices shown can become very large.

Our design goal is to discover pattern structures in indices (which can be considered as I/O traces) and represent the mapping in a compact way, so that reading takes less time and uses less space for processing indices. In this paper, we demonstrate the effectiveness of Pattern Structured PLFS. First, we show how we reduce the local (per-process) metadata (indices) size by discovering patterns, and then we further demonstrate how to achieve even better compression by merging local indices into a single global one per PLFS file.

In our design, when writing, Pattern Structured PLFS (Pattern PLFS) buffers traditional indices in raw index buffers for each process. After the buffer is full or at the time of closing, a structure discovering engine starts processing the raw indices and puts the generated pattern structure entries to pattern index buffer and non-pattern ones to non-pattern indices. When an application reads a file, Pattern PLFS reads indices from files, merges pattern entries into global ones whenever possible, and stores the global pattern entries and non-pattern entries in separate buffers. The contents of the buffers are broadcast to other processes that are reading the PLFS file.

We chose the design described above based on efficiency and feasibility. One of the other options is to compress using both local and global patterns at the time of writing in ADIO layer. This approach requires communication and synchronization when writing, which may ruin the biggest advantage of PLFS - fast writing. It becomes worse when the application has more write requests and smaller write extents. Another possibility is to use existing compression libraries, such as zlib [19], to compress indices in memory, write compressed data to files, read them into memory and decompress them. The problem of this is that the eventual memory footprint is still big, although the I/O time of reading indices is reduced due to the compression.

III. Local Pattern Structure

Local pattern structures describe the access behavior of a single process. For example, a process may write to a file

with *(offset, length)* pair sequence such as: *(0, 4), (5, 4), (10, 4), (15, 4)*. This is an example of a typical fixed-stride pattern and can easily be described in a form (e.g., saying start offset is 0; stride is 5; length is 4) of smaller size by checking if the stride is constant. Strided patterns occur when accessing parts of regular data structure (e.g. odd columns of a 2-d matrix). A more complex pattern would occur when accessing discrete parts of an array consisting of complex data types (e.g. MPI file view with complex data types). To compress complex pattern, we need an algorithm to identify the repeating sequences and a structure to represent them in a compact way. The structure should also allow random accesses without decoding. The algorithm proposed in this section can discover complex pattern structures and compress them. Figure 2 shows an example in which two processes write into one file with traditional indices. This section uses this example to show how local pattern structure discovering works.

Figure 3 is the structure of one pattern entry. Chunk *id* is used to find the data dropping file which the pattern is for. Each logical offset pattern may map to many length and physical offset patterns. But if you expand patterns to their original sequences, the number of logical offsets, lengths and physical offsets represented by a pattern entry are exactly the same.

id: chunk id. It is used to locate the corresponding data dropping file.
logical: logical offset pattern unit (See Figure 4).
length[]: an array of pattern units
physical[]: an array of pattern units.

Fig. 3. Structure of a pattern index entry.

$$[i, (d[0], d[1], ...)\hat{}r]$$

Fig. 4. Pattern unit notation. i is the first element of the original sequence. $d[]$ $(delta)$ is the repeating part of an array containing the distances of any two consecutive elements in the original sequence. r is the number of repetitions. For example, $(5, 7, 10, 12, 15)$ can be represented as $[5, (2, 3)\hat{}2]$.

Based on the sliding window algorithm in [20], we propose a new algorithm to discover common patterns in data accesses and store them in a data structure that allows PLFS to conduct lookups without decompressing the index. To demonstrate it, Figure 5 gives an example for discovering patterns in logical offsets of Process 0 in Figure 2. The sequence of logical offsets $(0, 3, 7, 14, 17, 21, 28, ...)$ are preprocessed to deltas $(3, 4, 7, 3, 4, 7, ...)$. Two windows move along the deltas to find repeating subsequences. To represent a pattern of a sequence of numbers in a compact way, we introduced a structure called *pattern unit*, described in Figure 4. The eventual pattern output in Figure 5 is $[0, (3, 4, 7)\hat{}3], [42, (4)\hat{}4]$.

Suppose w is the window size in algorithm demonstrated in Figure 5, the time complexity of finding repeating parts between search window and lookahead window is $O(w)$, since it is essentially a string searching problem and can be solve

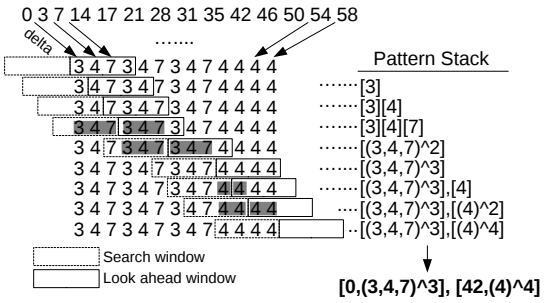

Fig. 5. An example of local pattern structure discovering.

by the KMP algorithm [21] or similar ones. In addition, two windows move forward at least by one position, so the overall time complexity of this pattern recognition algorithm is $O(wn)$. n is the length of the input sequence.

To compress PLFS mappings, given a sequence of tuples (i.e. raw index entries) *(logical offset, length, physical offset)*, they are separated to three arrays, $logical\_offset[]$, $length[]$, $physical\_offset[]$. First, structures in $logical\_offset[]$ are found by a structure discovering engine. Then, elements in $length[]$, $physical\_offset[]$ are grouped within the arrays, according to the structures of $logical\_offset[]$, and their structures are discovered by group. When PLFS receives a read request, it looks up the position of the requested offset in $logical\_offset[]$. After that, it can find the corresponding values in $length[]$ and $physical\_offset[]$. Then, the physical data will be read.

IV. Global Pattern Structure

Global pattern structure is constructed using local pattern structures. To merge local patterns into global patterns, Pattern PLFS first sorts all local patterns by their initial logical offsets. Then it goes through every pattern to check if neighbor patterns abuts one another. Figure 6 is an example of a global pattern. At the beginning of it, a group of three processes (4,7,6) write with a local strided pattern (We call the size of data shared by the same group of processes a *global stride*). After that, (2,8,9) writes the following global stride. Then (4,7,6) repeats the pattern. Global pattern is essentially consecutive repeating local patterns. Since local patterns are repeating, only one local pattern is stored in global pattern structure and the difference between global and local pattern is that global pattern maintains a list of chunk IDs instead of only one chunk id.

Assuming each local pattern repeats twice and physical offset starts at 0, the global pattern structure in Figure 6 can be described by the following:

id[]: [4,7,6,2,8,9,4,7,6,2,8,9]
logical: 1000,(30)$\hat{}4$
length: 10,(0)$\hat{}4$
physical: 0,(10)$\hat{}4$

Of course, there are some more complicated global patterns that the global pattern structure cannot describe. However, in

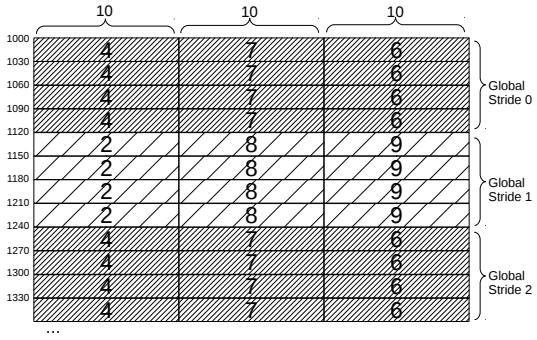

Fig. 6. An example of global pattern. 4,7,6,2 and so on are PIDs. Blocks of same texture represent data area that is shared by a group of processes, e.g. *global strides*.

id[]: an array of chunk id indicating the positions of processes inside the global pattern
logical: a logical offset pattern unit
length: a length pattern unit
physical: a physical offset pattern unit

Fig. 7. Global pattern structure

practice, this simple structure is effective enough and it favors fast lookups.

To look up an offset in a global pattern, we locate which row and column the requested offset is in the imaginary global pattern matrix (e.g. Figure 6). To find the physical offset within a data dropping file, Pattern PLFS needs to figure out how much data has been written to file before the piece of data requested.

V. Evaluation

A. Setup

For a holistic test, several benchmarks and real applications were used to test Pattern PLFS. *FS-TEST* [22] is a synthetic checkpoint tool from LANL. It can be set to write or read with N-N (N processes write N files) or N-1 pattern with many parameters. In addition, we developed a benchmark tool called *MapReplayer*, which can replay traces previously collected by PLFS and show the performance. In order to test structure discovering from unstructured I/O, Pattern PLFS also ran with several real applications. The experiments were conducted on LANL's RRZ testbed, which has eight cores/16GB RAM per node. PanFS was used as the underlying parallel file system.

B. FS-TEST

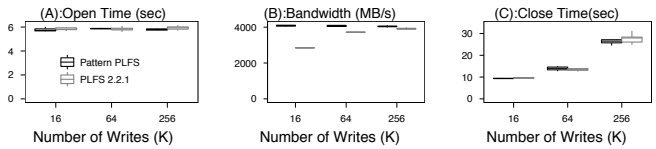

Fig. 8. Write performance of 512 processes with write size of 4K. (Write Open/Close Time: lower is better. Write Bandwidth: higher is better.)

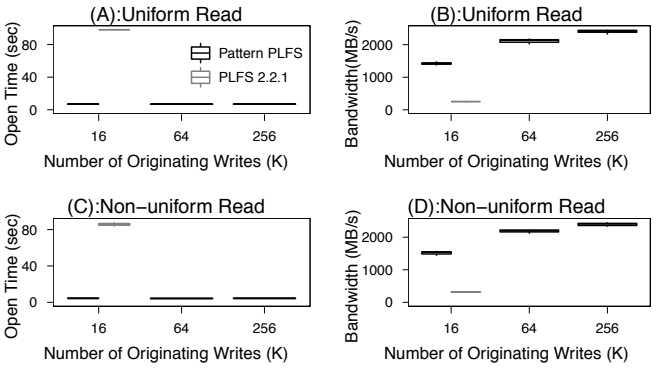

Fig. 9. Performance of uniform read (512 processes) and non-uniform read (256 processes) with originating write size of 4K. Some of the PLFS 2.2.1 data points are missing because large index took too much memory and PLFS crashed when allocating memory. (Read Open Time: lower is better. Read Bandwidth: higher is better.)

FS-TEST has very similar write patterns to many real checkpoint systems. In this experiment, each FS-TEST process writes data stridely, which leads to many index entries in Traditional PLFS (PLFS 2.2.1). The write sizes of all tests are fixed at 4KB. Large amount of indices take lots of space in both disks and memory, resulting in bad I/O performance. Pattern PLFS is expected to reduce index sizes and therefore improve performance.

As shown in Figure 8(A), write open time of Pattern PLFS and PLFS 2.2.1 are very close. In (B), we can observe that write bandwidth of Pattern PLFS is consistently better than that of Traditional PLFS. The reason for this is that Pattern PLFS writes much less metadata (structured index entries), which are much smaller then traditional unstructured index entries, to disks. It is worth noticing that Pattern PLFS is about 1.5 GB/s faster than Traditional PLFS with 512 processes and 16K writes per process. As shown in (C), Pattern PLFS and Traditional PLFS have very similar performance on close, when PLFS flushes data and closes all opened files. Overall, Pattern PLFS has better write performance than Traditional PLFS. In addition, from the experiments we have conducted, we also see a trend towards growing gap between performance of Pattern PLFS and Traditional PLFS as the scale becomes larger.

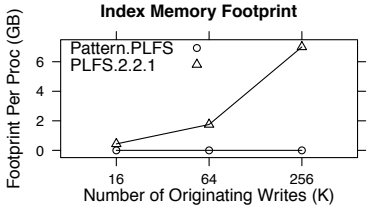

Fig. 10. Index memory footprint of 512 processes. Note that the Y axis shows the per-process memory footprint. For example, an eight-core node needs more than 48 GB memory to hold index for PLFS 2.2.1 if the number of originating writes are 256 K.

As shown in Figure 9, Pattern PLFS has much shorter open time than Traditional PLFS for both uniform and non-uniform read, since indices are read and processed at read open time and Pattern PLFS is able to significantly reduce index size by discovering structures and representing indices as pattern structures. Figure 10 shows the comparison between index memory footprint of Pattern PLFS and PLFS 2.2.1. The reduction of index footprint leads to up to 80 percent and 480 percent higher bandwidth for write (Figure 8) and read (Figure 9), respectively. The improvement is asymmetrical because index write is more parallelized than read in PLFS.

C. Real Applications

We explored writes of several real applications to see if there are any patterns and if Pattern PLFS can discover them. It is really nice that PLFS indices are essentially write traces, by which we can plot and see the patterns if they exist.

1) MILC: MILC is a LQCD application that is widely used to solve real physics problems and to benchmark supercomputers [9] [23]. Figure 11 shows the write patterns of three I/O configurations for saving the same data. All of them are N-1 writes, which are ideal cases for PLFS. In Figure 11 *(A)*, each MILC process writes small fix-size pieces of data with 2-d strided pattern (stride sizes vary). In *(B)*, each process writes to one contiguous portion of the file. The difference between *(C)* and *(B)* is that in *(C)* each process also writes a header at the beginning of the file. The compression rates of *(A) (B)* and *(C)* are 37.0, 3.0 and 3.6, respectively. *(A)* has better compression rate since it has more writes and they have patterns. Pattern PLFS was able to compress by discovering local and global patterns. The other two are both simple and most of the compressions came from using global pattern.

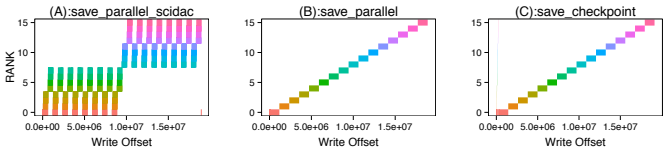

Fig. 11. MILC write patterns. In-memory index compression rates by Pattern PLFS (higher is better): (A):37.0;(B):3.0;(C):3.6

2) Pagoda: Pagoda [24] stands for Parallel Analysis of GeOscience DAta. It is a set of PnetCDF-based tools and APIs that have been developed to mitigate the I/O bottleneck of GCRM (Global Cloud Resolving Model) data analysis, whose scale can be PB's per year. Pagoda conducts N-1 write stridely, which generates a great amount of index entries. By discovering structures out of unstructured writes, Pattern PLFS achieve a compression rate of 2.9 in a typical run.

D. Replay

By using *MapReplayer*, we were able to replay the I/O behaviors of various benchmarks and real applications. The compression rates are already shown in Figure 1. NERSC Pattern I/O [25] is a benchmark in which each process writes with a single fixed-stride pattern. By the local and global pattern structure discovering techniques described in this paper, they can be represented as one global pattern and index size is significantly reduced. Each process of LANL App 3 writes with 2-D strided. Pattern PLFS was able to represent them by one single pattern entry in memory. In LANL App 2 MPI I/O collective and LANL App 2 Independent, each process writes with different strides in different periods of time. The compression was achieved by local pattern compression. Pattern PLFS has better compression rate for LANL App 2 I/O library since the application's own I/O library arranged data to be written with fixed-stride pattern, which made global pattern compression possible. LANL App 1 writes with 2-D strided pattern and global pattern was found. Most of FLASH writes are random, which makes it hard to compress. Each BTIO process writes with a 2-d strided pattern and they are combined to a single global pattern in memory. To sum up, traditional PLFS does not handle these applications very well, while pattern PLFS can discover structures and be able to shrink their index sizes.

VI. Other Possible Use Cases

Discovering structure in unstructured I/O and representing structure in a compact and lossless way are promising techniques and have the potential for being applied in other systems. Two examples are pre-fetching and pre-allocation of blocks in file systems. These eager techniques both use predictions of future access to predictively perform expensive operations asynchronously ahead of time. Our pattern detection of complex access patterns can improve their predictive abilities. Another example is SciHadoop in which the ratio of metadata (keys, which are dimensional information) to data can be high, thereby causing tremendous overhead when transferring this metadata [26]. Our technique can be applied to shrink the size of these keys and eventually reduce overhead by using discovered structures to represent keys.

VII. Conclusion

The era of big data and exascale is nigh and is pushing I/O to its limit. Knowing the structure of I/O can improve performance but discovery is not straightforward. This paper proposes efficient and practical techniques to discover structures from unstructured I/O operations, thereby enabling powerful I/O optimizations. We applied these techniques within PLFS to compress its internal metadata using structured patterns. We demonstrated several orders of magnitude improvement in the size of the metadata and corresponding improvements in write performance up to 40 percent and in read performance of up to 480 percent.

Acknowledgment

The authors are thankful to Michael Lang (LANL) and Adam Manzanares (California State University) for their supports and thoughtful suggestions toward this study.

REFERENCES

[1] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, p. 8, 2011.

[2] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proceedings of the 2009 ACM/IEEE conference on Supercomputing*. ACM, 2009, p. 21.

[3] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. ACM, 2008, pp. 15–24.

[4] J. Bent. (2012) PLFS maps. [Online]. Available: http://www.institutes.lanl.gov/plfs/maps

[5] A. Manzanares, J. Bent, M. Wingate, and G. Gibson, "The power and challenges of transformative i/o," in *IEEE Cluster 2012*.

[6] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, "Information and control in gray-box systems," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001, pp. 43–56.

[7] M. Polte, J. Lofstead, J. Bent, G. Gibson, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, M. Wingate *et al.*, "... and eat it too: high read performance in write-optimized hpc i/o middleware file formats," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. ACM, 2009, pp. 21–25.

[8] J. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt, "Scihadoop: Array-based query processing in hadoop," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 66.

[9] The MIMD Lattice Computation (MILC) Collaboration. [Online]. Available: http://www.physics.utah.edu/ detar/milc/

[10] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.

[11] B. Pasquale and G. Polyzos, "A static analysis of i/o characteristics of scientific applications in a production workload," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. ACM, 1993, pp. 388–397.

[12] E. Smirni and D. Reed, "Lessons from characterizing the input/output behavior of parallel scientific applications," *Performance Evaluation*, vol. 33, no. 1, pp. 27–44, 1998.

[13] B. Pasquale and G. Polyzos, "Dynamic i/o characterization of i/o intensive scientific applications," in *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*. ACM, 1994, pp. 660–669.

[14] S. Byna, Y. Chen, X. Sun, R. Thakur, and W. Gropp, "Parallel I/O prefetching using MPI file caching and I/O signatures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 44.

[15] J. He, H. Song, X. Sun, Y. Yin, and R. Thakur, "Pattern-aware file reorganization in mpi-io," in *Proceedings of the sixth workshop on Parallel Data Storage*. ACM, 2011, pp. 43–48.

[16] T. Madhyastha and D. Reed, "Learning to classify parallel input/output access patterns," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 8, pp. 802–813, 2002.

[17] J. Oly and D. Reed, "Markov model prediction of i/o requests for scientific applications," in *Proceedings of the 16th international conference on Supercomputing*. ACM, 2002, pp. 147–155.

[18] N. Tran and D. Reed, "Automatic arima time series modeling for adaptive i/o prefetching," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, no. 4, pp. 362–377, 2004.

[19] (2012) zlib. [Online]. Available: http://zlib.net/

[20] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337–343, 1977.

[21] D. Knuth, J. Morris, and V. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[22] (2012) LANL FS-TEST. [Online]. Available: http://institutes.lanl.gov/data/software/

[23] J. He, J. Kowalkowski, M. Paterno, D. Holmgren, J. Simone, and X.-H. Sun, "Layout-aware scientific computing: a case study using milc," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems in conjunction with ACM/IEEE SuperComputing 2011*, 2011.

[24] Pagoda website. [Online]. Available: https://svn.pnl.gov/gcrm/wiki/Pagoda

[25] National Energy Research Scientific Computing Center. [Online]. Available: https://outreach.scidac.gov/

[26] J. Buck, N. Watkins, G. Levin, A. Crume, K. Ioannidou, S. Brandt, C. Maltzahn, and N. Polyzotis, "Sidr: Efficient structure-aware intelligent data routing in scihadoop," UCSC, Tech. Rep.