PLFS: A Checkpoint Filesystem for Parallel Applications

John Bent[*][†], Garth Gibson[‡], Gary Grider[*], Ben McClelland[*],
Paul Nowoczynski[§], James Nunez[*], Milo Polte[†], Meghan Wingate[*]

ABSTRACT

Parallel applications running across thousands of processors must protect themselves from inevitable system failures. Many applications insulate themselves from failures by checkpointing. For many applications, checkpointing into a shared single file is most convenient. With such an approach, the size of writes are often small and not aligned with file system boundaries. Unfortunately for these applications, this preferred data layout results in pathologically poor performance from the underlying file system which is optimized for large, aligned writes to non-shared files. To address this fundamental mismatch, we have developed a virtual parallel log structured file system, PLFS. PLFS remaps an application's preferred data layout into one which is optimized for the underlying file system. Through testing on PanFS, Lustre, and GPFS, we have seen that this layer of indirection and reorganization can reduce checkpoint time by an order of magnitude for several important benchmarks and real applications without any application modification.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management-File organization

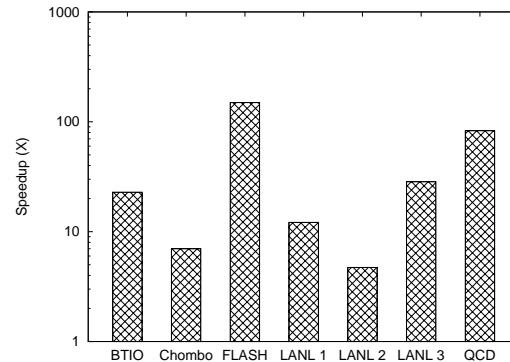## General Terms

Performance, Design

Keywords

High performance computing, parallel computing, checkpointing, parallel file systems and IO

[*]LANL Technical Information Release: 09-02117
[†]Los Alamos National Laboratory
[‡]Carnegie Mellon University
[§]Pittsburgh Supercomputing Center

1: **Summary of our results.** *This graph summarizes our results which will be explained in detail in Section 4. The key observation here is that our technique has improved checkpoint bandwidths for all seven studied benchmarks and applications by up to several orders of magnitude.*

1. INTRODUCTION

In June 2008, Los Alamos National Labs (LANL), in partnership with IBM, broke the petaflop barrier and claimed the top spot on the Top 500 list [3] with the Roadrunner supercomputer [23]. Due to its unique hybrid architecture, Roadrunner has only 3060 nodes, which is a relatively small number for a supercomputer of its class. Unfortunately, even at this size, component failures are frequent. For large parallel simulations at LANL, and other High Performance Computing (HPC) sites, which can run for several months, restarting a calculation from the beginning is not viable, since the mean time to interrupt for today's supercomputers is measured in days or hours and not months [31, 41].

Typically these applications protect themselves against failure by periodically *checkpointing* their progress by saving the state of the application to persistent storage. After a failure the application can then restart from the most recent checkpoint. Due to the difficulty of reconstructing a consistent image from multiple asynchronous checkpoints [18], HPC applications checkpoint synchronously (*i.e.* following a barrier). Synchronous checkpointing, however, does not eliminate the complexity; it merely shifts it to the parallel file system which now must coordinate simultaneous access from thousands of compute nodes. Even using optimal checkpoint frequencies [9], checkpointing has become the driving workload for parallel file systems and the challenge it imposes grows with each successively larger super-

| (a) N-N | (b) N-1 segmented | (c) N-1 strided |
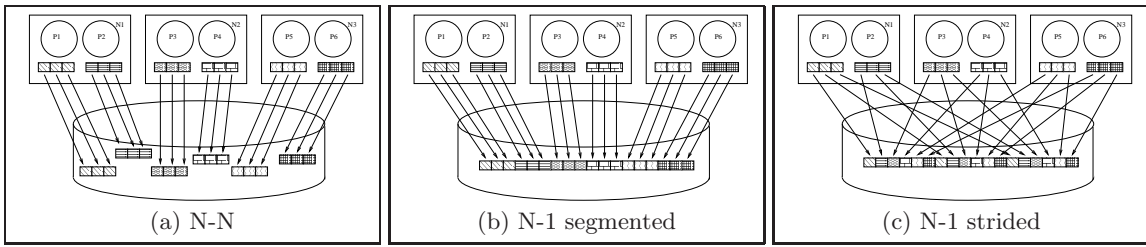
2: **Common Checkpointing Patterns.** *This figure shows the three basic checkpoint patterns: from left to right, N-N, N-1 segmented, and N-1 strided. In each pattern, the parallel application is the same, consisting of six processes spread across three compute nodes each of which has three blocks of state to checkpoint. The difference in the three patterns is how the application state is logically organized on disk. In the N-N pattern, each process saves its state to a unique file. N-1 segmented is simply the concatenation of the multiple N-N files into a single file. Finally, N-1 strided, which also uses a single file as does N-1 segmented, has a region for each block instead of a region for each process. From a parallel file system perspective, N-1 strided is the most challenging pattern as it is the most likely to cause small, interspersed, and unaligned writes. Note that previous work [16] refers to N-N as* file per process *and N-1 as* shared file *but shares our segmented and strided terminology.*

computer [32, 42]. The difficulty of this challenge can vary greatly depending on the particular pattern of checkpointing chosen by the application. In this paper, we describe different checkpointing patterns, and show how some result in very poor storage performance on three of the major HPC parallel file systems: PanFS, GPFS, and Lustre.

We then posit that an interposition layer inserted into the existing storage stack can rearrange this problematic access pattern to achieve much better performance from the underlying parallel file system. To test this hypothesis, we have developed PLFS, a Parallel Log-structured File System, one such interposition layer. We present measurements using PLFS on several synthetic benchmarks and real applications at multiple HPC supercomputing centers. The results confirm our hypothesis: writing to the underlying parallel file system through PLFS improves checkpoint bandwidth for all tested applications and benchmarks and on all three studied parallel file systems; in some cases, bandwidth is raised by several orders of magnitude.

As we shall discuss, and is summarized in Figure 1, PLFS is already showing very large speed ups for widely used HPC benchmarks and important real HPC codes, and at extreme scale. PLFS works on unmodified applications as an optional mount point on the compute nodes and is designed to augment, and not replace, an existing underlying parallel file system. PLFS consists of three thousand lines of code and is publically available at http://sourceforge.net/projects/plfs.

The rest of the paper is organized as follows. We present more detailed background and motivation in Section 2, describe our design in Section 3 and our evaluation in Section 4. We present related work in Section 5, current status and future work in Section 6, and finally we conclude in Section 7.

2. BACKGROUND

From a file system perspective, there are two basic checkpointing patterns: *N-N* and *N-1*. An N-N checkpoint is one in which each of *N* processes writes to a unique file, for a total of *N* files written. An N-1 checkpoint differs in that all of *N* processes write to a single shared file. Applications using N-N checkpoints usually write sequentially to each file, an access pattern ideally suited to parallel file systems. Conversely, applications using N-1 checkpoint files typically organize the collected state of all N processes in

some application specific, canonical order, often resulting in small, unaligned, interspersed writes.

Some N-1 checkpoint files are logically the equivalent of concatenating the files of an N-N checkpoint (*i.e.* each process has its own unique region within the shared file). This is referred to as an *N-1 segmented* checkpoint file and is extremely rare in practice. More common is an *N-1 strided* checkpoint file in which the processes write multiple small regions at many different offsets within the file; these offsets are typically not aligned with file system block boundaries [8]. N-1 strided checkpointing applications often make roughly synchronous progress such that all the processes tend to write to the same region within the file concurrently, and collectively this region sweeps across the file. These three patterns, N-N, N-1 segmented, and N-1 strided, are illustrated in Figure 2. Since N-1 segmented is a rare pattern in practice, hereafter we consider only N-1 strided and we refer to it with the shorthand N-1.

The file system challenge for an N-N workload is the concurrent creation of thousands of files which are typically within a single directory. An N-1 workload can be even more challenging however for several different reasons which may depend on the particular parallel file system or the underlying RAID protection scheme. In the Panasas ActiveScale parallel file system [46] (PanFS), for example, small strided writes within a parity stripe must serialize in order to maintain consistent parity. In the Lustre parallel file system [25], writes to N-1 checkpoint files which are not aligned on file system block boundaries cause serious performance slowdowns as well [28]. Although both N-N and N-1 patterns pose challenges, it has been our observation, as well as that of many others [14, 22, 28, 29, 44], that the challenges of N-1 checkpointing are more difficult. Applications using N-1 patterns consistently achieve significantly less bandwidth than do those using an N-N pattern.

Figure 3 presents experimental data validating this discrepancy: An N-1 checkpoint pattern receives only a small fraction of the bandwidth achieved by an N-N pattern on PanFS, GPFS, and Lustre and does not scale with increased numbers of nodes. The PanFS experiment, run on LANL's Roadrunner supercomputer using its 1000 blade PanFS storage system, shows a maximum observed N-N bandwidth of 31 GB/s compared to a maximum observed N-1 bandwidth of less than 100 MB/s. Although we show PanFS results

(a) PanFS   (b) GPFS   (c) Lustre
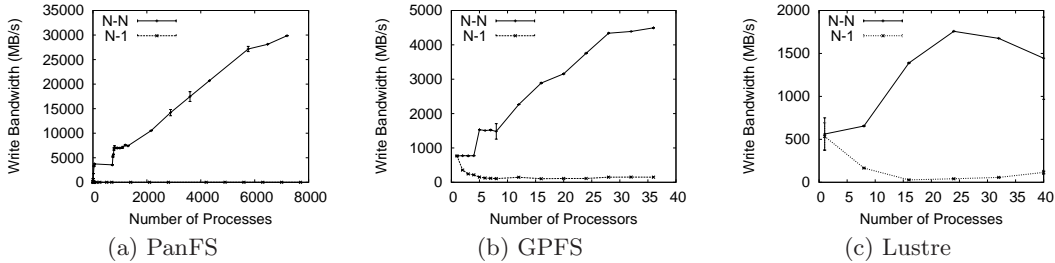
3: **Motivation.** *These three graphs demonstrate the large discrepancy between achievable bandwidth and scalability using N-N and N-1 checkpoint patterns on three of the major HPC parallel file systems.*

using their default RAID-5 configuration, PanFS has also a RAID-10 configuration which reduces the implicit sharing caused by N-1 patterns when two writers both need to update the same parity. While this solution improves scaling and offers much higher N-1 write bandwidth without sacrificing reliability, it does so by writing every byte twice, a scheme that, at best, can achieve only approximately half of the write bandwidth of N-N on RAID-5. PLFS, however, as will be shown in Section 4, can get much closer.

The GPFS and Lustre experiments were run on much smaller systems. The GPFS experiment was run using an archive attached to Roadrunner using its nine, quad-core, file transfer nodes. The Lustre experiment was run using five client machines, each with eight cores, and twelve Lustre servers. All three file systems exhibit similar behavior; N-N bandwidths are consistently higher than N-1 by at least an order of magnitude. Measurements were gathered using the LANL synthetic checkpoint tool, *MPI-IO Test* [12]. For each of these graphs, the size of each *write* was 47001 bytes (a small, unaligned number observed in actual applications to be particularly problematic for file systems). *Writes* were issued until two minutes had elapsed. Although this is atypical since applications tend to write a fixed amount of data instead of writing for a fixed amount of time, we have observed that this allows representative bandwidth measurements with a predictable runtime.

Since N-N checkpointing derives higher bandwidth than N-1, the obvious path to faster checkpointing is for application developers to rewrite existing N-1 checkpointing applications to do N-N checkpointing instead. Additionally, all new applications should be written to take advantage of the higher bandwidth available to N-N checkpointing. Although some developers have gone this route, many continue to prefer an N-1 pattern even though its disadvantages are well understood. There are several advantages to N-1 checkpointing that appeal to parallel application developers. One, a single file is much easier to manage and to archive. Two, N-1 files usually organize data into an application specific canonical order that commonly aggregates related data together in contiguous regions, making visualization of intermediate state simple and efficient. Additionally, following a failure, a restart on a different number of compute nodes is easier to code as the checkpoint format is independent of the number of processes that wrote the checkpoint; conversely, gathering the appropriate regions from multiple files or from multiple regions within a single file is more complicated.

Essentially these developers have once again shifted complexity to the parallel file system for their own convenience. This is not unreasonable; it has long been the province of computer systems to make computing more convenient for its users. Many important applications have made this choice. Of the twenty-three applications listed on the Parallel I/O Benchmarks page [36], at least ten have an N-1 pattern; two major applications at LANL use an N-1 checkpointing pattern as do at least two of the eight applications chosen to run on Roadrunner during its initial stabilization phase. N-1 checkpointing is very important to these applications. For example, at the core of one of LANL's N-1 applications is a twenty-year old Fortran checkpointing library. About a decade ago, in response to a growing clamor about the limitations of N-1 checkpointing bandwidth, developers for this application augmented their checkpointing library with fifteen thousand lines of code. However, instead of changing the application to write an N-N pattern, they added a new IO routine in which interprocess communication is used to aggregate and buffer writes. Although they did not improve the performance to match that of other applications using an N-N checkpoint pattern, this effort was considered a success as they did improve the N-1 performance by a factor of two to three. This checkpointing library, called *bulkio*, has been maintained over the past decade and ported to each new successive supercomputer at LANL [7]. Furthermore, N-1 patterns continue to be developed anew in many new applications. High level data formatting libraries such as Parallel NetCDF [19] and HDF5 [2] offer convenience to application developers who simply describes the logical format of their data and need no longer consider how that data is physically organized in the file system. Once again this convenience merely shifts the complexity to the underlying parallel file system since these libraries use an N-1 pattern.

3. DESIGN OF AN INTERPOSITION LAYER

We start with the hypothesis that an interposition layer can transparently rearrange an N-1 checkpoint pattern into an N-N pattern and thereby decrease checkpoint time by taking advantage of the increased bandwidth achievable via an N-N pattern. To test this hypothesis, we have developed one such interposition layer, PLFS, designed specifically for large parallel N-1 checkpoint files. The basic architecture is illustrated in Figure 4. PLFS was prototyped with FUSE [1], a framework for running stackable file systems [49] in non-privileged mode.
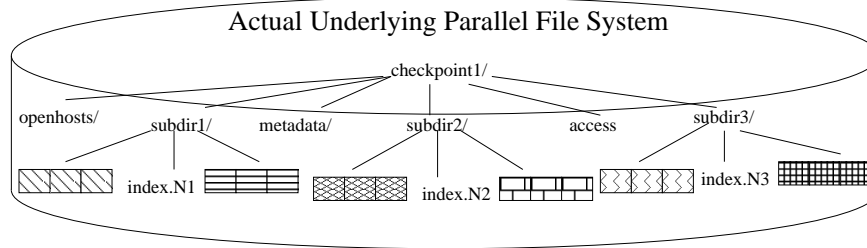
PLFS is a virtual FUSE file system, mounted on the compute nodes, situated between the parallel application and an underlying parallel file system responsible for the actual data storage. As PLFS is a virtual file system, it leverages
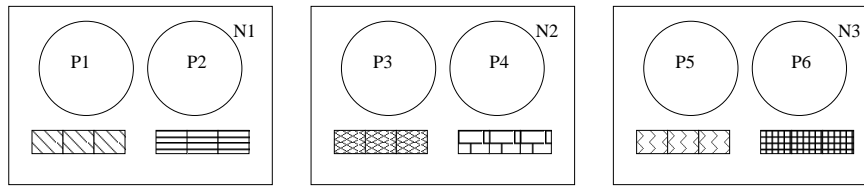
A parallel application consisting of six processes on three compute nodes creates an N–1 strided file, checkpoint1. PLFS preserves the application's logical view of the file.

PLFS Virtual Interposition Layer
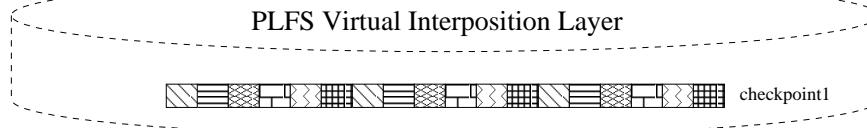
checkpoint1

PLFS then creates a container structure on the underlying parallel file system to hold the data for file checkpoint1.

Actual Underlying Parallel File System

checkpoint1/

openhosts/ subdir1/ metadata/ subdir2/ access subdir3/

index.N1 index.N2 index.N3

4: **Data Reorganization.** *This figure depicts how PLFS reorganizes an N-1 strided checkpoint file onto the underlying parallel file system. A parallel application consisting of six processes on three compute nodes is represented by the top three boxes. Each box represents a compute node, a circle is a process, and the three small boxes below each process represent the state of that process. The processes create a new file on PLFS called checkpoint1, causing PLFS in turn to create a* container *structure on the underlying parallel file system. The container consists of a top-level directory also called checkpoint1 and several sub-directories to store the application's data. For each process opening the file, PLFS creates a* data file *within one of the sub-directories, it also creates one* index file *within that same sub-directory which is shared by all processes on a compute node. For each write, PLFS appends the data to the corresponding data file and appends a record into the appropriate index file. This record contains the length of the write, its logical offset, and a pointer to its physical offset within the data file to which it was appended. To satisfy reads, PLFS aggregates these index files to create a lookup table for the logical file. Also shown in this figure are the access file, which is used to store ownership and privilege information about the logical file, and the openhosts and metadata sub-directories which cache metadata in order to improve query time (e.g. a stat call).*

many of the services provided by the underlying parallel file system such as redundancy, high availability, and a globally distributed data store. This frees PLFS to focus on just one specialized task: rearranging application data so the N-1 write pattern is better suited for the underlying parallel file system. In the remainder of this paper, we refer to PLFS generally to mean this virtual file system which itself is comprised of a set of PLFS servers running across a compute system bound together by an underlying parallel file system; when we refer to a *specific* PLFS we mean just one of these servers.

3.1 Basic operation

The basic operation of PLFS is as follows. For every logical PLFS file created, PLFS creates a *container* structure on the underlying parallel file system. Internally, the basic structure of a container is a hierarchical directory tree consisting of a single top-level directory and multiple sub-directories that appears to users; PLFS builds a logical view of a single file from this container structure in a manner similar to the core idea of Apple bundles [4] in Mac OS X. Multiple processes opening the same logical file for writing share the container although each open gets a unique *data file* within the container into which all of its writes are ap-

pended. By giving each writing process in a parallel application access to a non-shared data file, PLFS converts an N-1 write access pattern into a N-N write access pattern. When the process writes to the file, the write is appended to its data file and a record identifying the write is appended to an index file (described below).

3.1.1 Reading from PLFS

Rearranging data in this manner should improve write bandwidths, but it also introduces additional complexity for reads. In order to read the logical file, PLFS maintains an *index file* for each compute node which records the logical offset and length of each write. PLFS can then construct a *global index* by aggregating the multiple index files into an offset lookup table. This global offset is constructed as needed to satisfy read operations and lookup operations when cached metadata is not available (as will be discussed in Section 3.2).

One difficulty in constructing the global index stems from multiple processes that may write the same offset concurrently. These processes cannot know which will be the ultimate writer, so they need to synchronize to determine the appropriate order. This synchronization needs to be exposed to the file system to be effective [13]. If a logical clock [18]

was associated with these synchronizations, its value could be written to index files so that the merge process could correctly determine write ordering. Since parallel applications synchronize with a barrier called by all processes, a simple count of synchronization calls could be sufficient. In practice checkpoints do not experience overlapping writes, so at this time PLFS has not implemented overlap resolution.

One interesting nuance is that PLFS has a data file for every process but only a single index file per compute node shared by all processes on that node. Sharing an index file is easy; by the time PLFS sees writes, they have been merged by the operating system into a single memory space. The operating systems sends all writes to a single PLFS process which ensures index records are correctly, chronologically appended. Having a single index greatly reduces the number of files in a container since current LANL applications run up to sixteen processes on a node; on the next LANL supercomputer, this could be up to sixty-four. We tried reducing the number of data files similarly. Write bandwidth was not affected, but reads were slowed for *uniform restarts* in which reading processes access the file in the same access pattern as it was written. The pattern of a single reader accessing a single data file sequentially lends itself very well to prefetching. However, due to timing differences between the write and read phases, multiple processes in a uniform restart may not always read from a shared file sequentially.

Having described the basic operation of PLFS, we now present some of its implementation in finer detail. Although there are many interposition techniques available ([43] includes a survey), we have selected FUSE for several reasons. Because FUSE allows PLFS to be accessible via a standard file system interface, applications can use it without modification and files on PLFS can be accessed by the entire suite of existing tools such as *ls*, *diff*, and *cp*. In addition to providing user transparency, using FUSE dramatically simplified our development effort. A file system in userspace is significantly easier to develop than a kernel file system and is more portable as well. However, this convenience is not free as FUSE does add some overhead as shown in Section 4.

3.1.2 Container implementation

Because PLFS leverages the underlying parallel file system as much as possible, we give the container the same logical name as the PLFS file. This allows PLFS to pass a *readdir* system call directly to the underlying parallel file system and return its result without any translation. PLFS also handles *mkdir* system calls without any translation. Leveraging the underlying parallel file system in this way requires that PLFS use some other mechanism to distinguish between regular directories and containers in order to implement the *stat* system call correctly. As the *SUID* bit is rarely used on directories and yet is allowed to be set by the underlying file system, PLFS sets this bit on containers; the implication however is that PLFS must disallow setting this bit on a regular directory.

As we discussed previously, parallel applications do synchronized checkpointing; the implication for PLFS is that multiple processes running on multiple compute nodes writing an N-1 checkpoint file will cause PLFS on each compute node to attempt to create the same container concurrently on the underlying parallel file system. The difficulty arises because each PLFS must first *stat* that path to see whether the path is available, whether that container already exists,

or whether there is a regular directory at that location. Ideally, each PLFS could *stat* the path and, when the location is empty, atomically create a directory with the *SUID* bit set. Unfortunately, the *mkdir* system call ignores the *SUID* bit; each PLFS must therefore first create a directory and then set the *SUID* bit. Doing this naively results in a race condition: if one PLFS *stats* the path after another has made the directory but before it has set the *SUID* bit, then the *stat* will indicate that there is a regular directory in that location. The application issuing the open of the logical file will then receive an error incorrectly indicating that there is a directory already at that location. To avoid this race condition, each PLFS first makes a hidden directory with a unique name, set its *SUID* bit, and then atomically *renames* it to the original container name.

3.2 Metadata operations

Metadata operations against a file include accessing its permissions (including SUID), its capacity, the offset of its last byte and the timestamp of its last update. For a directory on PLFS, these are provided by the underlying file system. But for a PLFS file which is constructed from a container like the example in Figure 4, these metadata operations have to be computed from the many underlying files within the container.

Because the SUID bit on the container itself has been overloaded to indicate that the directory is not a directory at all, but rather a container, it cannot be also used to indicate if the user has set SUID on the PLFS file represented by the container. Instead we use a file inside the container, the *access file*, to represent the appropriate SUID and the rest of the permissions associated with the container. For example, where *chmod* and *chgrp* are directed at the logical file, they are applied to the access file within the container.

Capacity for the logical file is the sum of the capacities of the files inside the container. The last update timestamp is the maximum of the last update timestamps. And the offset of the last byte is the maximum logical offset recorded in any of the index files.

Computing these sums and maximums with every *stat* call on a PLFS file is expensive. Our strategy for quickening this is to cache recently computed values in the *metadata subdirectory*. To make this cache as effective as possible we have each PLFS server cache into this metadata subdirectory any information it has in its memory data structures when the last writer on that node closes the file. On this close, the FUSE process creates a file named $H.L.B.T$, where H is the node's hostname, L is the last offset recorded on that node, B is the sum of the capacity of all files that this node manages, and T is the maximum timestamp among these files.

When no process has this container open for write, a stat call on the container issues a *readdir* on the metadata subdirectory, then reports the maximum of the offsets for last byte offset, the maximum of the timestamps for modification timestamp and the sum of the blocks for capacity.

If one or more processes has the container open for writing, then the corresponding cached metadata values could be stale. PLFS clients therefore create a file in the *openhosts subdirectory* named by its hostname when one or more processes on that node have that file open for writing, and then deleting this file once all *opens* have been closed. *stat* must then do a *readdir* on openhosts as well to discover if any node has the file open for writing, and thereby determine

which metadata cache entries might be stale.

When there are hostname files in the openhosts subdirectory, the node that is executing the *stat* call could read the contents of the index files associated with those hostnames in order to find the largest logical offset, and then combine this with the metadata cache files that are not stale.

In the experience of the HPC community, *stat'ing* an open file is almost always done by a user trying to monitor the progress of their job. What they want is an inexpensive probe showing progress, not an expensive instantaneously correct value [37]. Following this logic, PLFS does not read and proccess the index files associated with hostnames that have the container open for write. Instead it assumes that files are not sparse (*i.e.* every byte has been written) and sums the sizes of all data files within a container to estimate the last offset of the PLFS file. Because writes to each data are always simply appended, this estimation will monotonically increase as additional data is written into the file, allowing users to monitor progress. When the container is closed, the metadata subdirectory contains fully correct cached values, and full accuracy is provided at all times when the container has no processes writing it.

4. EVALUATION

We present the results of our experimental evaluation in Figure 5. Eleven of these twelve graphs present one experiment each. The twelfth, Figure 5k, is a summary. In the majority of these graphs, the write bandwidth is shown on the y-axis in MB/s as a function of the number of processes. We will note it in the text for those few graphs for which we deviate from this general configuration. The write bandwidth that we report is whatever is reported by the particular benchmark; whenever possible, we report the most conservative value (*i.e.* we include open and close times in our write times, and we either barrier after close or we use the time reported by the slowest writer). Benchmarks were run on LANL's Roadrunner supercomputer using the default parameters unless indicated otherwise. Finally we have attempted to run multiple iterations for each experiment; where applicable, the standard deviation is included.

4.1 MPI-IO Test

The top three graphs, Figures 5a, 5b, and 5c, present the results of our study using the LANL synthetic checkpoint tool, *MPI-IO Test* [12], on three different parallel file systems, PanFS, GPFS, and Lustre. There are several things to notice in these graphs. The first is that these are the same three graphs that we presented in Figure 3 except that we have now added a third line to each. The three lines show the bandwidth achieved by writing an N-N pattern directly to the underlying parallel file system, the bandwidth achieved by writing an N-1 pattern directly to the underlying parallel file system, and the third line is the bandwidth achieved by writing an N-1 pattern *indirectly* to the underlying parallel file system through PLFS.

These graphs illustrate how the performance discrepancy between N-N and N-1 checkpoint patterns is common across PanFS, GPFS, and Lustre. Remember, as was discussed in Section 2, switching to N-N is not a viable option for many applications which are inextricably wed to an N-1 pattern and are resigned to the attendant loss of bandwidth. Fortunately, as is evidenced by these graphs, PLFS allows these applications to retain their preferred N-1 pattern while achieving most, if not all, of the bandwidth available to an N-N pattern. Particularly for the PanFS results, which were run on our Roadrunner supercomputer, PLFS achieves the full bandwidth of an N-N pattern (*i.e.* up to about 31 GB/s). In fact, for several of the points, an N-1 pattern on PLFS actually outperforms an N-N pattern written directly to PanFS. Although we have yet to fully investigate the exact reason for this, there are several reasons why this could be the case. The first is that PLFS rearranges writes into a log structured pattern so an N-N pattern which incurs seeks could do worse than a PLFS pattern which appends only. Secondly, the structure of the PLFS container spreads data across multiple sub-directories whereas a typical N-N pattern confines all N files into only a single parent directory.
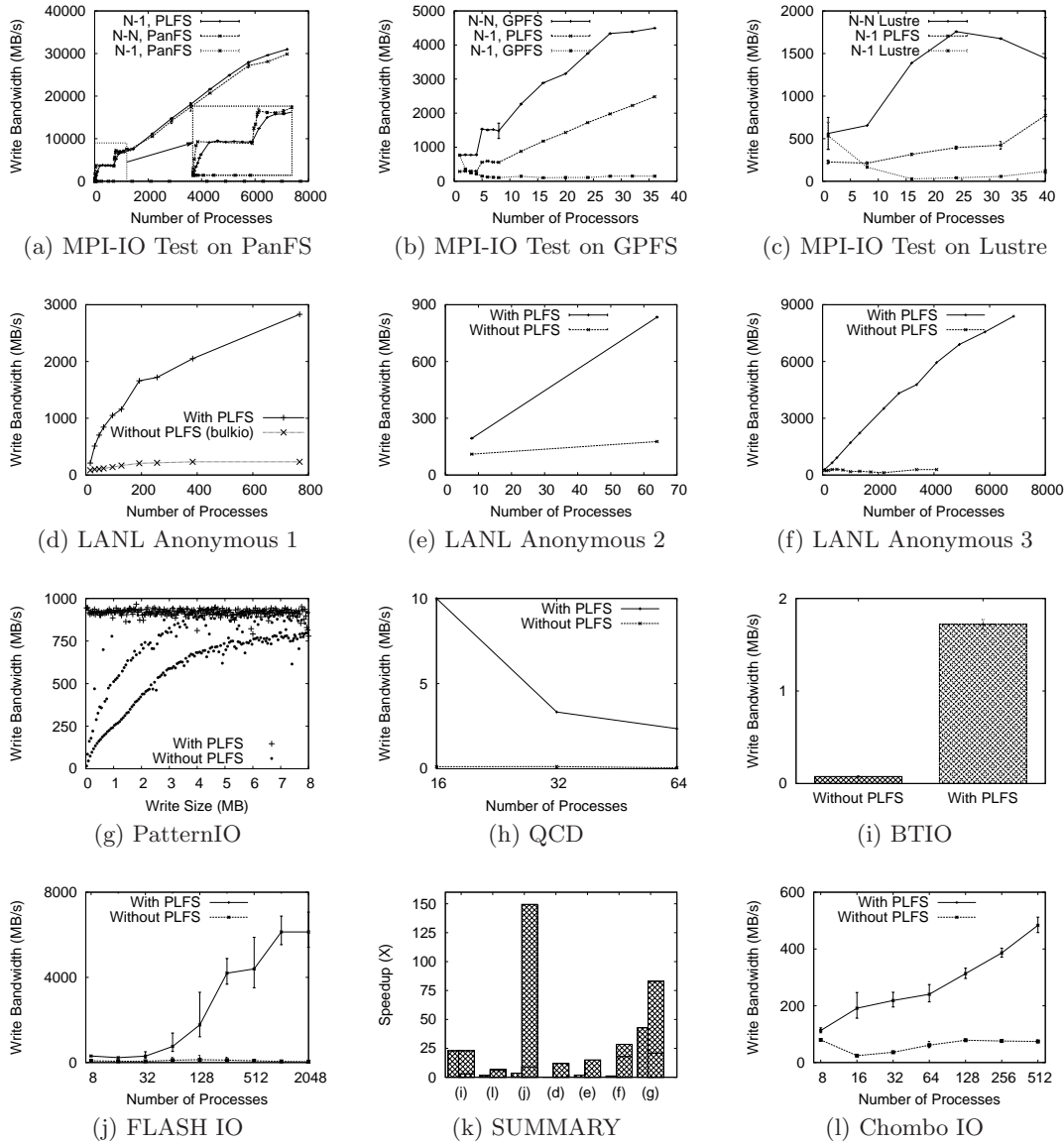
Although PLFS does improve the bandwidth of N-1 patterns on GPFS and Lustre, the improvement is not as large as it is on PanFS. This is because the scale of the experiments on PanFS are 200 times larger than on the other two platforms. In the inset in Figure 5a at the extreme low values for the number of processes, we see that PLFS does not scale N-1 bandwidth as fast as N-N scales for PanFS as well. This is due to overheads incurred by both FUSE and PLFS; these overheads limit the total bandwidth achieved by any single compute node relative to N-N. For HPC systems at extreme scale this limitation does not matter since aggregate bandwidth across multiple nodes is more relevant than the bandwidth from a single node. In this case the real limiting factor is the total bandwidth capacity of the storage fabric, which generally saturates when using only a small fraction of compute nodes. However, with poorly-behaved IO patterns (*i.e.* N-1), even very large jobs may not reach these bandwidth limits because they will be more severely constrained by file system limitations as exemplified by Figure 3. PLFS is designed to remove these file system limitations, so that jobs can achieve higher bandwidth and reach the same limitation of the storage fabric which is constraining N-N bandwidth. We will examine the FUSE and PLFS overheads in greater detail in Section 4.6.

An unusual feature of the inset graph in Figure 5a is that there are localized bottlenecks about 700 processes. This is due to the architecture of Roadrunner and it's connection to its 1000 blade PanFS storage system. Roadrunner is split into seventeen sub-clusters, *CUs*, each of which can run 720 processes. The CUs are fully connected via an Infiniband fat tree, however, in order to minimize network infrastructure costs, the storage system is partitioned into multiple subnets. Each CU currently has six specialized IO nodes, one for each sub-net; these six IO nodes are the storage bandwidth bottleneck. Therefore, as a job grows within a CU, it quickly saturates the IO node bandwidth; when it grows into another CU, its bandwidth increases sharply due to gaining six additional IO nodes: this explains the "stair-step" behavior of the N-N and PLFS lines in this graph.

We do see this same behavior in our GPFS and Lustre graphs, but due to the very small size of our test systems, we do not have enough compute nodes to saturate the available storage bandwidth and thus neither the N-N nor the PLFS lines in these graphs reach a storage bandwidth bottleneck.

4.2 Real LANL Applications

The second row in Figure 5 shows the results of using PLFS to improve the bandwidth of three important LANL applications which use N-1 checkpoint patterns. The appli-

(a) MPI-IO Test on PanFS   (b) MPI-IO Test on GPFS   (c) MPI-IO Test on Lustre

(d) LANL Anonymous 1   (e) LANL Anonymous 2   (f) LANL Anonymous 3

(g) PatternIO   (h) QCD   (i) BTIO

(j) FLASH IO   (k) SUMMARY   (l) Chombo IO

5: **Experimental Results.** *The three graphs in the top row are the same graphs that were presented earlier in Figure 3, except now they have an additional line showing how PLFS allows an N-1 checkpoint to achieve most, if not all, of the bandwidth available to an N-N checkpoint. The bar graph in the center of the bottom row consolidates these results and shows a pair of bars for each, showing both the relative minimum and the maximum speedups achieved across the set of experiments. Due to radically different configurations for these various experiments, the axes for these graphs are not consistent. The relative comparison within each graph should be obvious; absolute values can be ascertained by reading the axes.*

cation shown in Figure 5d is the application whose developers augmented their checkpoint routine with a new checkpoint library called bulkio, which aggregates and buffers writes into larger, contiguous writes more friendly to the underlying parallel file system. Therefore we compare the bulkio method writing directly to PanFS to the MPI-IO method writing indirectly to PanFS via PLFS. This data was collected on Roadrunner. The next two graphs show similar results; using PLFS to rearrange an N-1 pattern yields significantly improved bandwidths. Figure 5e was run on a small Roadrunner test cluster which consists of 128 8-core compute nodes with a 66 blade PanFS storage system, whereas

Figure 5f was run on Roadrunner itself. Note that these are extremely important applications at LANL, consuming more than half of LANL's total cycles. Although they must remain anonymous, traces of their IO are available [8].

4.3 NERSC's PatternIO Benchmark

Figure 5g presents the data derived from our measurements taken using NERSC's PatternIO benchmark [28] which plots write bandwidth as a function of *write* size. Notice that this deviates from the other graphs in Figure 5 which plot write bandwidth as a function of the number of processes. For this experiment, run on the Roadrunner test cluster, the

number of processes was set at a constant 512. This graph also is a scatter plot instead of using lines with standard deviations. The points for writing directly to PanFS show three interesting slopes all converging at about 1 GB/s on the far right side of the graph. The highest of these three regions shows the performance of *writes* when the write size is both block aligned and aligned on file system boundaries. The middle region is when the write size is block aligned only and the lowest region is when the write size is not aligned at all. *This graph demonstrates that PLFS allows an application to achieve a consistently high level of write bandwidth regardless of the size, or alignment, of its writes.*

4.4 Other Benchmarks

The remainder of the graphs show various other parallel IO benchmarks with an N-1 checkpoint pattern. QCD [45] in Figure 5h shows a large improvement using PLFS but this improvement actually *degrades* as the number of processes increases. This is because the amount of data written by QCD is a small constant value and the overhead due to container creation incurred in the *open* becomes proportionally greater as the amount of data written by each process decreases. BTIO [26] in Figure 5i also shows a large improvement but only for one data point. FLASH IO [24] and Chombo IO[27], shown respectively in Figures 5j and 5l, both show improvement due to PLFS which scales nicely with an increasing number of processes. FLASH was run on Roadrunner whereas Chombo was run on the Roadrunner test cluster; both were built with the HDF5 library [2].

4.5 Summary

A summary of the benchmark results, excluding the LANL and NERSC synthetics, is presented in Figure 5k. This summary shows a pair of bars for each benchmark; one for both the worst-case and best-case speedups for PLFS. Only in one case did PLFS actually degrade performance: a slowdown of three percent for LANL 1 when it was run on just a single node. Although we did test on single nodes to be thorough, single node sized jobs are extremely rare in HPC. The relevant results from reasonably sized jobs, showed at least a doubling of performance for all studied benchmarks. QCD and BTIO experienced a single order of magnitude improvement. The best result was observed for FLASH which was improved in the best case by two orders of magnitude.

4.6 Overhead

As was seen in Figures 5a, 5b, and 5c, N-1 patterns written through PLFS only match the bandwidth achieved by N-N patterns written directly to the underlying parallel file system once the storage system bandwidth is saturated. For small number of processes, PLFS cannot match the performance of N-N (nonetheless, it does still improve bandwidth over a direct N-1 pattern). This overhead is measured in Figure 6 which shows results as measured on LANL's GPFS system. In order to measure the both overhead incurred by FUSE as well as any additional overhead incurred by PLFS, we developed a second FUSE file system, *No-opFS*. No-opFS does no extra work, merely redirecting all IO to the underlying parallel file system(*i.e.* GPFS). For those readers familar with FUSE, please note that No-opFS caches the file descriptor created in the *open* into the opaque FUSE file handle pointer, uses it for subsequent *writes* and *reads*, and closes it in the *flush*.
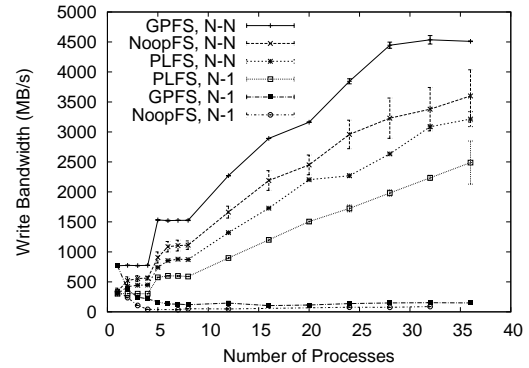
6: **Overhead.** *Ideally, N-1 patterns written through PLFS would be able to achieve the bandwidth available to an N-N pattern written directly to the underlying parallel file system. However, this graph shows that various overheads make this difficult. Even though there is overhead, the important point is that PLFS still allows an N-1 pattern to be written much more quickly than if it was written directly to the underlying parallel file system.*

Figure 6 is almost the same as Figure 5b which compares the bandwidths measured for N-N directly to GPFS, N-1 directly to GPFS, and N-1 indirectly to GPFS written through PLFS. The difference here is that several new measurements have been added. The first is for N-N written indirectly to GPFS through No-opFS. This line is significantly lower than the line for N-N written directly to GPFS; the delta between these lines is the overhead incurred due to FUSE, which is approximately 20%. The next measurement added is running an N-N workload through PLFS; this shows the additional overhead incurred by PLFS, approximately 10%. The delta beween that line and the existing N-1 through PLFS measurements show additional overhead which we believe is due to serializations within FUSE due to multiple processes accessing the same path within FUSE's file table (approximately another 10% loss). For completeness, we also measured the bandwidth achieved by an N-1 workload written through No-opFS; therefore, the bottom two lines on the graph provide another view of the overhead lost to FUSE.

Although this graph shows a total overhead cost of about 40 to 50%, this loss of potential bandwidth is not unduly concerning. As was seen in Figure 5a, the loss of potential bandwidth due to this overhead disappears for reasonable HPC processor counts. Even with a limited per-node bandwidth, a relatively small number of nodes writing through PLFS is able to saturate the storage bandwidth.

4.7 Beyond Writing

Although PLFS is designed primarily for *writing* checkpoints, checkpoint files are still occassionally read. We must therefore weigh improvements in write bandwidth against possible degradations in other operations.

4.7.1 Read Bandwidth

To ensure PLFS does not improve write bandwidth at the expense of read bandwidth, we ran a set of read experiments on Roadrunner test cluster which are shown in Figure 7. We first created two sets of 20 GB files, each written by a different number of writers; all writes were 47001 bytes (in other words, increasing numbers of writers issued decreasing numbers of writes). One set was created directly on PanFS;

(a) Uniform Restart   (b) Non-uniform Restart   (c) Archive Copy
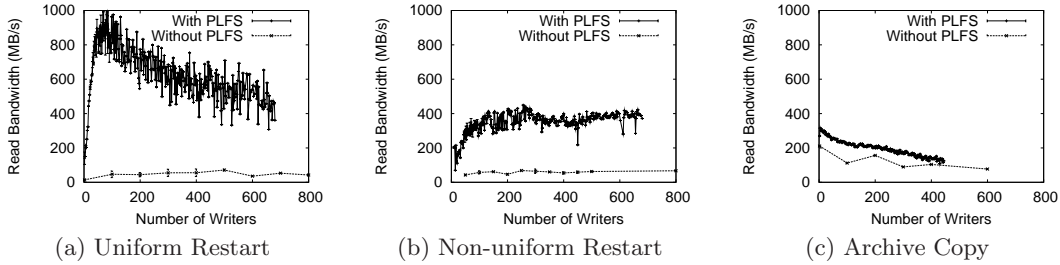
7: **Read Bandwidth.** *These three graphs show the results of our read measurements on the Roadrunner test cluster. We created a set 20 GB N-1 checkpoint files through PLFS and another directly on PanFS. Each file was produced by a different number of writers; all of the writes were 47001 bytes in size. For each graph, the y-axis shows the read bandwidth as a function of the number of writers who created the file. The graph on the left shows the read bandwidth when the number of readers is the same as the number of writers, as is the case in a typical uniform restart; in this case, the size of the reads is the same as the size of the original writes. The graph in the middle emulates a non-uniform restart in which the application resumes on one fewer compute nodes; in this case, the size of the reads is slightly larger than the size of the original writes. Finally, the graph on the right shows the read bandwidth when there only four readers; we used LANL's archive copy utility and modelled the common scenario of copying checkpoint data to an archive system using a relatively small number of readers. To enable comparison across graphs, the axis ranges are consistent.*

the other indirectly through PLFS. In all cases, reads were performed on different nodes than those where the corresponding data was written and caches were always flushed between successive reads.

We measured the time to read these files using three different read scenarios. One emulates a *uniform restart* in which the number of processes resuming the application is the same as the number that wrote the last checkpoint. In this case, each process reads the same data in the same order as one of the previous writers. In contrast, a *non-uniform restart* is one in which the number of readers is different from the number of writers and the read offsets are therefore not aligned with the previous writes. In this case, we emulate the scenario where an application resumes after a failure by simply running on the newly reduced number of nodes. The *number* of reads is not affected by the size of the job for typical N-1 checkpoints since each read is extracting a region for a particular variable within the simulation. However, the *size* of this region depends on the number of processes within the job since each gets $1/Nth$ of the region. Specifically, for a non-uniform restart in which the number of processes, N, has been reduced by M, the size of each read will be $N/(N-M)$ times the size of the original writes. The third scenario emulated is where a relatively, small fixed number of processes reads a checkpoint in order to save it onto an archive system. For these measurements, we used LANL's archive copy utility with fours readers each running on their own node; each reader reads a single contiguous 5 GB region by issuing sequential 1 MB reads.

The results of these experiments can be seen in Figure 7. In order to allow comparison across the three experiments, the ranges of the axes have been made consistent. The y-axis shows read bandwidth as a function of the number of writers who created the checkpoint. We can easily see from these graphs that the highest bandwidth is achieved using PLFS in the uniform restart scenario. This is not surprising: each reader moves sequentially through just one, non-shared file, a pattern easily improved through prefetching performed by the underlying parallel file system. The bandwidth decreases here due to the decreasing amount of data read as the number of readers increases. With less data read by each, the open times begin to dominate, and there is less

potential for prefetching. The very low bandwith observed when the checkpoint is stored directly on PanFS is also not surprising due to its layout of data within RAID groups. Each contiguous GB is spread only across a single RAID group (in this case consisting of just eight storage devices from a pool of around one hundred). The nature of the N-1 pattern and the size of the reads means that all readers will almost always be reading within just a single RAID group. In addition to limiting the bandwidth to only a few storage devices, it also reduces the bandwidth from each of them due to overloading them with non-sequential requests.

The results for the non-uniform restart can be similarly explained. The PanFS results and explanation are essentially the same. The results for PLFS are also very similar; better than PanFS due to spreading the reads across all storage devices and not quite as good as the PLFS results for the uniform restart. The difference between the uniform and non-uniform results for PLFS is only seen for small numbers of readers in the area where PLFS was helped by prefetch in the uniform experiment. Since the readers are reading different offsets than were written in the non-uniform restart, they will read multiple data files instead of just reading a single one. The underlying parallel file system, unsurprising, does not identify this admittedly strange pattern and therefore there is no prefetch benefit. Only when there is no longer any prefetch benefit, do these results converge.

Although we are pleased to see that PLFS also does relatively well for the archive copy experiment, we do not yet fully understand all of these results. We can think of no reason why the bandwidths should be this low and we assumed that PanFS would easily outperform PLFS due to having contiguous data within its RAID groups instead of having data spread across multiple data files within a PLFS container. However, we are not surprised to see the read bandwidths drop as the number of writers increases. Increasing numbers of writers results in a data layout with fewer large contiguous regions of data for both PLFS and PanFS; therefore, the read bandwidth will suffer accordingly.

4.7.2 Metadata Queries

As we discussed in Section 3.2, there are currently two techniques used to discover the metadata for a PLFS file.
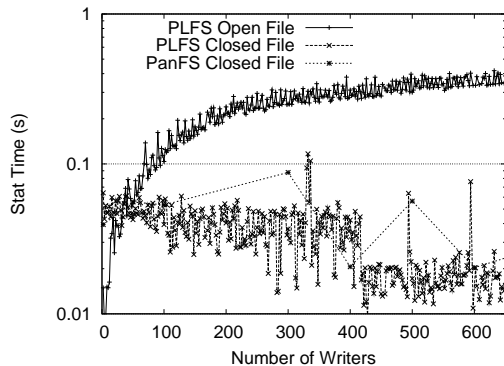
8: **Metadata Queries.** *This graph compares the time to stat a 20 GB file on PLFS with the time to stat a comparable file on PanFS. The y-axis, which is in logscale, shows the stat time as a function of the number of writers who created that file. We compare the PanFS stat rate to the PLFS rate for files opened for writing, and to the PLFS rate for closed files.*

For a file being currently written, metadata is discovered by *stat'ing* individual data files. When a file is closed, metadata information is cached as file names within a specific subdirectory inside the container. Thereafter, metadata information can be discovered merely by issuing a *readdir*. Obviously, the first technique is much slower; if our primary interest was in optimizing metadata query rates, than we could cache this metadata following every write. However, since PLFS is designed for checkpoint writing, we do not consider this technique.

Figure 8 compares these two times against the time it takes to query a closed file written directly to the underlying parallel file system, PanFS. We have not yet measured the time to query an open file on PanFS. For this experiment, conducted on LANL's Roadrunner test cluster, we created two sets of 20 GB files each written by a different number of writers all issuing 47001 byte-sized writes. One set was created directly on PanFS; the other indirectly through PLFS. As the graph shows, the *stat* times for closed files on PLFS and PanFS are approximately the same. However, as expected, the time to query a PLFS file open for writing is greater than to query a closed file.

5. RELATED WORK

Translating random access checkpoint writes into a sequential pattern is an idea which extends naturally from work on log-structured file systems [38] such as NetApp's WAFL [17] and Panasas's Object Storage [47]. While these ideas reorganize disk layout for sequential writing, they do not decouple concurrency caused by multiple processes writing to a single file. Another approach to log-structuring N-1 patterns addressed only physical layout and also did not decouple concurrency [35]. We believe our contribution of rearranging N-1 checkpoints into N-N is a major advance.

Checkpoint-restart is an old and well studied fault tolerance strategy. A broad evaluation of rollback-recovery protocols and checkpointing and a discussion of their future at the petascale can be found in [10, 11].

Berkeley Lab Checkpoint/Restart [15] and Condor Checkpointing [21] both allow unmodified applications to checkpoint the state of a single node. They leverage the operat-

ing system's ability to swap a process to save the totality of the swap image persistently. The great strength of this approach is that it can be used by applications that have no internal checkpoint mechanism. A disadvantage is that these checkpoints are larger then when an application specifies only the exact data that it needs saved. Conversely, incremental checkpointing and memory exclusion [32, 34] reduce the size of checkpoint data by only saving data that has changed since the previous checkpoint. Buffering with copy-on-write [20] can also reduce checkpoint latency.

`stdchk` [40] saves checkpoints into a cloud of free disk space scavenged from a cluster of workstations. A similar approach [39] reserves compute node memory to temporarily buffer checkpoints and then asynchronously saves them to persistent storage. Diskless checkpointing [33] also saves checkpoints into compute node memory, but does not subsequently transfer them to persistent storage. Rather it achieves fault protection by using erasure coding on the distributed checkpoint image. Although these techniques work well for many applications, large HPC parallel applications jealously utilize all memory and demand a high degree of determinism in order to avoid jitter [5] and are therefore seen to be poorly served by techniques reducing available memory or techniques which require background processes running during computation.

The Adaptable IO System (ADIOS) [22] developed by the Georgia Institute of Technology and Oak Ridge National Laboratory provides a high-level IO API that can be used in place of HDF5 to do much more aggressive write-behind and log-like reordering of data location within the checkpoint. While this technique requires application modification, it enables interoperability with other middleware libraries. Similarly, Lustre Split Writing [48], uses a library approach and leverages Lustre's file joining mechanism to decouple concurrent access at runtime as does PLFS. However, Lustre Split Writing is wed to the Lustre file system, requires application modification, and prevents the data from being immediately available following a checkpoint.

ZEST [29], developed at Pittsburgh Supercomputing Center, is a file system infrastructure that is perhaps the most similar in philosophy to PLFS, particularly in its borrowing of concepts from log-structured file systems. Rather than each client process pushing their writes sequentially to storage, in ZEST manager threads running on behalf of each disk pull data from a series of distributed queues, in a technique borrowed from River [6]. The key aspect of ZEST is that no individual write request is ever assigned to a specific disk; disks pull from these queues whenever they are not busy, resulting in high spindle utilization even in a system where some devices are performing more slowly than others. Unlike PLFS, however, data is not immediately available to be read, requiring a subsequent phase to first rewrite the file before it can be accessed. Since this phase happens in the relatively long time between checkpoints and since it happens on the server nodes and not on the compute nodes, this subsequent rewrite phase is not typically problematic for a dedicated checkpoint file system.

6. STATUS AND FUTURE WORK

We initially developed PLFS in order to test our hypothesis that an interposition layer could rearrange checkpoint patterns such that the convenience of N-1 patterns could be preserved while achieving the bandwidth of N-N. However,

	Interposition Technique Used	No Extra Resources Used During	No Extra Resources Used After	Maintains Logical Format	Works with Unmodified Applications	Data Immediately Available
ADIOS	Library	Yes	Yes	Yes	No	Yes
stdchk	FUSE	No (LD, M)	No (LD, N, M)	Yes	Yes	Yes
Diskless	Library	No (M)	No (M)	No	No	Yes
Sp Writ	Library	Yes	Yes	Yes	No	No
ZEST	FUSE	No (RD)	No (RD)	No	No	No
PLFS	FUSE	Yes	Yes	Yes	Yes	Yes

1: **Techniques for improving N-1 Checkpointing** *This table presents a comparison of the various techniques for reducing N-1 checkpoint times. For exposition, we have used various shorthands: Diskless for Diskless Checkpointing, Sp Writ for Lustre Split Writing, LD for local disk on the compute nodes, RD for remote disk on the storage system, M for memory, and N for network.*

after achieving such large improvements with real LANL applications, we were compelled to harden it into a production file system. Consisting of about three thousand lines of code, PLFS is currently an optional mount point on Roadrunner where several applications have begun using it to improve their N-1 checkpoint performance. PLFS is publically available at http://sourceforge.net/projects/plfs.

Although PLFS works transparently with unmodified applications, in practice some applications may need to make minor changes. Many users find it most convenient to stage their input data sets and their executable code in the same directory where they write their checkpoint files. Since PLFS is specifically a checkpoint file system and is not a general purpose file system, this usage pattern may suffer a performance hit. These users should instead stage their input data sets and their executable code in a general purpose file system and merely direct their N-1 checkpoint files to PLFS.

Also, applications that open PLFS files in read-write mode will find that reading from these files can be very slow. As we discussed in Section 3.1.1, reads in PLFS are handled by reading and aggregating the index files. When files are opened in write-read mode, this process must be repeated for every *read* since intervening *writes* may have occurred. Although the vast majority of HPC applications are able to read in read-only mode, we do plan to investigate removing this limitation in the future for those few applications who must occasionally read while writing. One possibility is to introduce metadata servers which can synchronously maintain an aggregated index in memory.

Originally intended merely to address N-1 challenges, PLFS as currently designed, also has the potential to address N-N challenges. One way that PLFS can reduce N-N checkpointing times is by reducing disk seeks through its use of log-structured writing. However, we have yet to measure the frequency of non-sequential IO within N-N checkpoints.

Another challenge of an N-N pattern is the overwhelming metadata pressure resulting from the simultaneous creation of tens of thousands of files within a single directory. Currently HPC parallel file systems do distribute metadata across multiple metadata servers; however they do so at the granularity of a *volume* or a directory (*i.e.* all files within a directory share a single metadata server). PLFS can easily refine this granularity by distributing container subdirectories across multiple metadata servers. At the moment, PLFS only creates container structures for regular files; directories are simply created directly on the underlying parallel file system. By extending PLFS to create a similar con-

tainer structure for directories, we believe that PLFS can effectively address this N-N challenge as well, in a manner similar to [30].

7. CONCLUSION

Large systems experience failures. To protect themselves against these failures, parallel applications running on these systems save their progress by checkpointing. Unfortunately for many of these applications, their preferred checkpointing patterns are extremely challenging for the underlying parallel file system and impose severe bandwidth limitations. In this paper, we have developed PLFS to demonstrate how a simple interposition layer can transparently rearrange these challenging patterns and improve checkpoint bandwidth by several orders of magnitude.

The parallel file system attached to Roadrunner is the largest LANL has ever had; testing it has revealed that the challenges of N-1 patterns are severely exacerbated at this scale. Given current bandwidths, we know of no current N-1 application at LANL that can effectively checkpoint across the full width of Roadrunner. PLFS allows them to do so.

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] FUSE: Filesystem in Userspace. http://fuse.sourceforge.net/.
[2] The HDF Group. http://www.hdfgroup.org/.
[3] Top 500 Supercomputer Sites. http://www.top500.org/.
[4] Apple Inc. Bundle Programming Guide. http://developer.apple.com/documentation/CFBundles, November 2005.
[5] A. C. Arpaci-Dusseau. *Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems.* PhD thesis, University of California, Berkeley, 1998.
[6] R. H. Arpaci-Dusseau. Run-Time Adaptation in River. *ACM Transactions on Computer Systems (TOCS)*, 21(1):36–86, February 2003.
[7] J. Bent. Personal communication with William Rust, Dec 2008.

[8] J. Bent. PLFS Maps. http://www.institutes.lanl.gov/plfs/maps, Mar 2009.

[9] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.

[10] Elnozahy,, E. N. (Mootaz) and Alvisi,, Lorenzo and Wang,, Yi-Min and Johnson,, David B. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[11] Elnozahy,, Elmootazbellah N. and Plank,, James S. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Trans. Dependable Secur. Comput.*, 1(2):97–108, 2004.

[12] Gary Grider and James Nunez and John Bent. LANL MPI-IO Test. http://institutes.lanl.gov/data/software/, jul 2008.

[13] G. A. Gibson, D. Stodolsky, F. W. Chang, W. V. C. Ii, C. G. Demetriou, E. Ginting, M. Holl, Q. Ma, L. Neal, R. H. Patterson, J. Su, R. Youssef, and J. Zelenka. The scotch parallel storage system. In *In Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95*, pages 403–410. IEEE Computer Society Press, Spring, 1995.

[14] G. Grider, J. Nunez, J. Bent, S. Poole, R. Ross, E. Felix, L. Ward, E. Salmon, and M. Bancroft. Coordinating government funding of file system and i/o research through the high end computing university research activity. *SIGOPS Oper. Syst. Rev.*, 43(1):2–7, 2009.

[15] P. H. Hargrove and J. C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *SciDAC 2006*, June 2006.

[16] R. Hedges, B. Loewe, T. T. McLarty, and C. Morrone. Parallel file system testing for the lunatic fringe: The care and feeding of restless i/o power users. In *MSST*, pages 3–17. IEEE Computer Society, 2005.

[17] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, January 1994.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[19] J. Li, W. keng 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 Conference*, 0:39, 2003.

[20] K. Li, J. Naughton, and J. Plank. Low-latency, concurrent checkpointing for parallel programs. *Parallel and Distributed Systems, IEEE Transactions on*, 5(8):874–879, Aug 1994.

[21] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report 1346, University of Wisconsin-Madison Computer Sciences, Apr 1997.

[22] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *CLADE '08: Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24, New York, NY, USA, 2008. ACM.

[23] Los Alamos National Labs. Roadrunner. http://www.lanl.gov/roadrunner/, Nov 2008.

[24] Michael Zingale. FLASH I/O Benchmark Routine – Parallel HDF 5. http://www.ucolick.org/ zingale/flash_benchmark_io/.

[25] S. Microsystems. Lustre file system, October 2008.

[26] NASA Advanced Supercomputing (NAS) Division. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[27] National Energy Research Scientific Computing Center. Chombo IO Benchmark. http://www.nersc.gov/ ndk/ChomboBenchmarks/.

[28] National Energy Research Scientific Computing Center. I/O Patterns from NERSC Applications. https://outreach.scidac.gov/hdf/NERSC_User_IOcases.pdf.

[29] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield. Zest: Checkpoint storage system for large supercomputers. In *3rd Petascale Data Storage Workshop Supercomputing*, Austin, TX, USA, nov 2008.

[30] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte. GIGA+: Scalable Directories for Shared File Systems. In *Petascale Data Storage Workshop at SC07*, Reno, Nevada, November 2007.

[31] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST'07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, page 2, Berkeley, CA, USA, 2007. USENIX Association.

[32] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software – Practice & Experience*, 29(2):125–142, 1999.

[33] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[34] J. S. Plank, J. Xu, and R. H. B. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.

[35] M. Polte, J. Simsa, W. Tantisiriroj, G. Gibson, S. Dayal, M. Chainani, and D. K. Uppugandla. Fast log-based concurrent writing of checkpoints. In *3rd Petascale Data Storage Workshop Supercomputing*, Austin, TX, USA, nov 2008.

[36] Rajeev Thakur. Parallel I/O Benchmarks. http://www.mcs.anl.gov/ thakur/pio-benchmarks.html.

[37] Rob Ross. HEC POSIX I/O API Extensions. www.pdsi-scidac.org/docs/sc06/hec-posix-extensions-sc2006-workshop.pdf.

[38] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10:1–15, 1992.

[39] S.A. Kiswany, M. Ripeanu, S. S. Vazhkudai. Aggregate memory as an intermediate checkpoint storage device. Technical Report ORNL Technical Report 013521.

[40] S.A. Kiswany, M. Ripeanu, S. S. Vazhkudai, A. Gharaibeh. stdchk: A Checkpoint Storage System for Desktop Grid Computing. In *Proceedings of the 28th Int'l Conference on Distributed Computing Systems (ICDCS 2008)*, June 2008.

[41] B. Schroeder and G. Gibson. A large scale study of failures in high-performance-computing systems. *IEEE Transactions on Dependable and Secure Computing*, 99(1), 5555.

[42] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78:012022 (11pp), 2007.

[43] D. Thain and M. Livny. Bypass: A tool for building split execution systems. In *In Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, pages 79–85, 2000.

[44] R. Thakur and E. Lusk. Data sieving and collective i/o in romio. In *In Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1999.

[45] U.S. Department of Energy Scientific Discovery through Advanced Computing (SciDAC). QCD QIO. http://usqcd.jlab.org/usqcd-docs/qio/, Nov 2008.

[46] B. Welch and G. Gibson. Managing scalability in object storage systems for hpc linux clusters. In *In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 433–445, 2004.

[47] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–17, Berkeley, CA, USA, 2008. USENIX Association.

[48] W. Yu, J. Vetter, R. S. Canon, and S. Jiang. Exploiting lustre file joining for effective collective io. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 267–274, Washington, DC, USA, 2007. IEEE Computer Society.

[49] E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *Proc. of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000. USENIX Association.