Efficient and Scalable Retrieval Techniques for Global File Properties

Dong H. Ahn[1], Michael J. Brim[2], Bronis R. de Supinski[1], Todd Gamblin[1], Gregory L. Lee[1],
Matthew P. LeGendre[1], Barton P. Miller[2], Adam Moody[1], Martin Schulz[1]

[1]Lawrence Livermore National Laboratory, Computation Directorate,
Livermore, CA 94550, {ahn1, bronis, gamblin2, lee218, legendre1, moody20, schulzm}@llnl.gov

[2]University of Wisconsin, Computer Sciences Department,
Madison, WI 53706, {mjbrim, bart}@cs.wisc.edu

Abstract—**Large-scale systems typically mount many different file systems with distinct performance characteristics and capacity. Applications must efficiently use this storage in order to realize their full performance potential. Users must take into account potential file replication throughout the storage hierarchy as well as contention in lower levels of the I/O system, and must consider communicating the results of file I/O between application processes to reduce file system accesses. Addressing these issues and optimizing file accesses requires detailed runtime knowledge of file system performance characteristics and the location(s) of files on them.**

In this paper, we propose Fast Global File Status (FGFS), a scalable mechanism to retrieve file information, such as its degree of distribution or replication and consistency. We use a novel node-local technique that turns expensive, non-scalable file system calls into simple string comparison operations. FGFS raises the namespace of a locally-defined file path to a global namespace with little or no file system calls to obtain global file properties efficiently. Our evaluation on a large multi-physics application shows that most FGFS file status queries on its executable and 848 shared library files complete in 272 milliseconds or faster at 32,768 MPI processes. Even the most expensive operation, which checks global file consistency, completes in under 7 seconds at this scale, an improvement of several orders of magnitude over the traditional checksum technique.

I. Introduction

Large-scale system sizes continue to grow exponentially [1]. Systems with ten thousand or more compute cores are common and LLNL's recently delivered Sequoia system has over a million cores [2]. This exponential growth in concurrency makes contention within the storage hierarchy common and efficient file access a challenge.

Avoiding contention requires an understanding of the performance and scalability of the entire storage hierarchy. Any software running on large-scale systems, including scientific applications, parallel libraries and tools, must determine dynamically how to adjust their strategies to improve performance. However, determining the properties of the storage hierarchy and its properties for all mounted file systems is nontrivial due to the increasing complexity of file system hierarchies. Further, existing parallel I/O software [3], [4], [5], [6] focuses on the I/O patterns for large data set accesses, and does not suit other I/O access patterns, such as uncoordinated, simultaneous accesses to small files—e.g., launching an executable that depends on many shared libraries triggers vast numbers of simultaneous accesses to the same library files when each process in the application loads the library dependencies. Further, in parallel environments, a file can reside in one or more local or remote file systems. Thus, different physical file systems may serve files with an identical file path to different processes of the same program.

In order to cope with these complexities, high performance computing (HPC) software requires a richer set of abstractions and scalable mechanisms by which to retrieve the performance properties of a file. To close this gap and enable efficient run time access to such information, we propose Fast Global File Status (FGFS), a scalable mechanism to retrieve file information including the degree of replication or distribution and consistency across local or remote file systems. FGFS builds on a simple node-local technique that raises the local namespace of a file to a global namespace using a memory-resident mount points table. FGFS extracts the global properties of a file path by comparing and grouping the global names seen by various processes.

FGFS status queries retrieve global information on both individual files and entire file systems. FGFS supports synchronous and asynchronous file status queries; File systems status queries serve as an inverse classifier that selects those mounted file systems that best match a given set of global properties required by an I/O operation. We design the FGFS Application Programming Interface (API) and its implementation to support the file access and information needs of a wide range of HPC programs, libraries and tools.

This paper makes the following contributions:

- A novel node-local technique to raise locally-defined file names to a global namespace;
- Scalable parallel algorithms based on string comparisons to compute global file properties;
- APIs and their implementations to provide global file information to existing HPC software at run time.

Our performance evaluation on a large multi-physics production application shows that most FGFS file status queries on its executable and its 848 shared libraries completes in 272 milliseconds or less at 32,768 MPI processes. Even the most expensive query that checks the global consistency of these files, takes under 7 seconds at this scale. Compared to the traditional technique in which remote daemons compute and compare checksums, FGFS provides several orders of magnitude improvements.

Additionally, we apply our techniques to three case studies and show how FGFS enables a wide range of HPC software to improve the scalability of its file I/O patterns. The first case study applies FGFS to the Stack Trace Analysis Tool (STAT) [7] and shows that FGFS aids this lightweight debugging tool in choosing between direct file I/O and file broadcasting. This capability results in a 52x speedup at 16,384 MPI processes. Second, we demonstrate that an efficient FGFS file status query is a crucial element for a highly scalable dynamic loading technique called Scalable Parallel Input Network for Dynamic Loading Environment (SPINDLE). The final study shows that FGFS file system status queries help the Scalable Checkpoint/Restart (SCR) library [8] to eliminate the need for arduous manual configuration efforts in discovering the best file system on which to store its multilevel checkpoints.

The remainder of this paper is organized as follows. Section II motivates our work by presenting two file I/O patterns that exposed scalability issues. Section III describes HPC file distribution models. In Section IV, we present the overview of our approach and detail the design and implementation of FGFS. Section V presents our performance results and case studies.

II. Motivating Examples

While systems sizes and the associated concurrency used within applications is growing at an exponential rate, the file I/O subsystems are not able to keep up with this growth. As a direct consequence, file system accesses are turning into one of the most critical bottleneck as we continue to scale HPC systems. In such an environment, uncoordinated file access patterns that do not consider the file system performance characteristics often have an effect similar to site-wide denial-of-service attacks on the shared file systems. Such patterns have been identified as one of major challenges for extreme scale computing [9], yet are becoming increasingly commonplace, impacting the entire workloads of HPC centers.

A. Application Start-up Causes an I/O Storm at Scale

Application start-up of KULL [10], one of LLNL's large multi-physics applications, exposed serious scaling challenges. When it was first run at large scales on DAWN, an IBM Blue Gene/P machine installed at LLNL, application start-up appeared to scale very poorly. For example, at 2,048 MPI processes, loading the executable with its dependent shared libraries took an hour, just to get to the `main` function. At 16,384 processes, it jumped to over ten hours.

Our investigation revealed that the primary cause of the ten hour executable load time was that the 16,384 instances of the dynamic loader (`ld.so`) were making a combined 300 million unnecessary `open` calls to the NFS server (where the application's executable and hundreds of shared libraries are staged) in order to search for the requested shared libraries. Since the POSIX standard prohibits caching of `open` calls to non-existent files, every call must be passed through to the NFS server, and since the compiler placed twenty directories in the default shared library search path, the dynamic loader was searching each of those directories for each of the hundreds of application shared libraries before searching the appropriate user-specified directories. Further, staging the application files into a parallel file system such as Lustre [11] did not improve the performance much, as a metadata operation storm can thrash the parallel file system, too. Simultaneous accesses to the many, typically small library files overwhelmed the file system and significantly disrupted the entire computing facility.

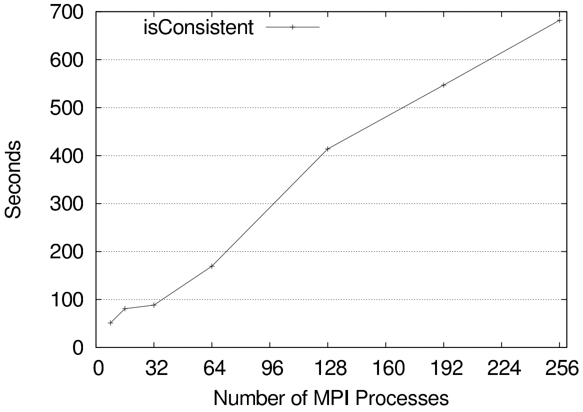


Fig. 1: Checksum-based consistency tests on KULL

To mitigate this problem, the application team staged the shared libraries in a way that minimized the number of unnecessary `open` system calls. As a result of these tuning efforts, the load time was reduced down to 18.6 minutes at 16,384 processes and 37.1 minutes at 32,768 processes. While a major improvement, it was a point solution for this particular setup and, yet, the overhead remained to be too high for a scale representing only a small fraction of today's largest machines. Further, the start-up continues to disrupt the shared file systems. Overall, it became evident that we require a more systematic and generic solution that can effectively manage an uncoordinated file I/O storm exhibited by the dynamic loaders.

B. Checksumming Poses Challenges to Run-time Tools

Accessing application executables and their shared libraries also poses challenges to many run time tools. Debuggers and dynamic instrumentation tools must read the binaries to access their symbols and model the address space of their target application processes. For example, to compute the absolute address of a function found in a shared library, a debugger must fetch the relative address of the function symbol from the library file and add it to the library's load address within the process' address space. Because target processes are distributed and can potentially load different shared library versions through the same file path[1], it is critical for these tools to construct an address-space model that is consistent with each individual target process. To account for this, tools like DDT [12] and TotalView [13] have remote daemons either testing global consistency by taking and comparing the checksums of all involved binaries or by building the model for each target process. Both techniques require daemons to perform extensive file I/O and hence can cause severe performance problems.

Figure 1 illustrates this cost by showing the overhead of the checksum-based scheme on KULL. In this test, daemons take the MD5 checksum of the executable and 848 dependent shared library files of a production KULL installation on a Linux cluster. The file I/O required to compute the checksums is being serialized at the NFS server where the executable along with a majority of its dependent shared libraries are staged. Even at a small scale, like 256 daemons, this scheme has unacceptable overhead for use by an interactive tool.

Similarly to Section II-A, our experiments revealed that staging these application files on a parallel file system did not improve scalability. A typical shared library is small and thus it is divided into only a limited number of smaller chunks to be stored across only few I/O servers of the parallel file system. As a result, highly concurrent file accesses create a contention on the small number of I/O servers, failing to exploit the parallelism and bandwidth that the parallel file system promises. Improving the scalability and disruption therefore requires an efficient global file information retrieval technique with little or no file I/O.

C. Enabling Scalable Solutions

Knowing the global information of a file or a mounted file system allows users to avoid non-scalable read and write I/O patterns. For read accesses, the daemons or loaders in the previous cases can use it to predict contention before accessing a file. If a bottleneck is predicted, the program can then choose to use an alternative file-access technique in which a limited number of representatives are designated to perform the I/O on behalf of the others. For example, a coordinator approach can be used to address the file I/O storm that led to unacceptable start-up overhead for KULL. When a file is found to reside in a non-scalable file system and when a large number of loaders will access it, then it can be read by a few coordinators that then broadcast its contents to the rest of the loaders. However, to enable this scheme, all loaders must be able to obtain the global information of any file that they load. Further, they must get to this information both efficiently and scalably in order not to create a bottleneck by doing so.

Similarly, write patterns can also benefit from global information of the file systems mounted on a machine. For instance, a distributed HPC application often needs to determine the best storage locations across all mounted file systems with respect to their available aggregate space, distribution, performance, and scalability. To make a decision, application processes must be able to construct a query with criteria on the desired global properties of a file system and scalably retrieve the result containing the best-suited locations.

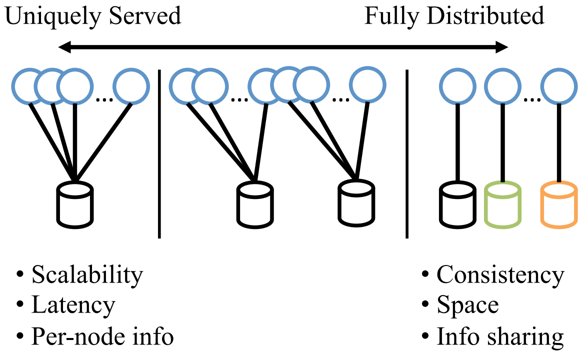


Fig. 2: Issues arising with file distribution models

The FGFS service, introduced in this work, provides the necessary information to enable such optimizations through a reusable, general-purpose query layer. It thereby supports the necessary I/O trade-off decisions for a wide range of HPC software including parallel tools, dynamic loaders, scripting language interpreters, parallel libraries, and further scientific applications themselves.

III. File Distribution Models in HPC

Before going into the details of FGFS, we discuss the overall trade-off space of parallel I/O for HPC systems. A file seen by distributed application processes can be mapped to different physical file distribution models. In one case, a single file system can *uniquely* serve it to all processes requesting it. On the other extreme, a node-local file system, such as a RAM disk, a solid-state drive, or a hard drive, can collectively serve it to all of the processes in a *fully distributed*[2] manner. There also exist hybrid models where multiple file servers collectively serve the file, but each server delivers it to only a subset of the nodes.

Figure 2 depicts the file distribution continuum and contrasts issues that arise at either end of the continuum: with the uniquely-served model, the single file system can become a bottleneck for performance and scalability, as the number of processes accessing its storage increases; while with the fully-distributed model, scalability is less of a problem as all file I/O requests are locally satisfied. However, as a drawback, it becomes impossible for a process to write a file to share with other processes. In addition, even when a file provides identical information to all processes, one must replicate the file across all node-local disks and ensure that copies are *consistent*. Hybrid models suffer these same issues.

Parallel file systems, such as Lustre [11] or GPFS [14], aim to provide the semantics of the uniquely-served model while leveraging storage distribution under the hood, attempting to combine the advantages of both worlds. On a parallel file

[1] This behavior is typically not desirable and systems are typically configured to be consistent. Tools, in particular debuggers, however, must be able to tolerate inconsistencies since they may used to exactly investigate failure scenarios caused by such inconsistencies.

[2] In many cases, a file is simply distributed across all nodes to be replicated. Thus, the set of *distributed* cases is a superset of the set of *replicated* cases.

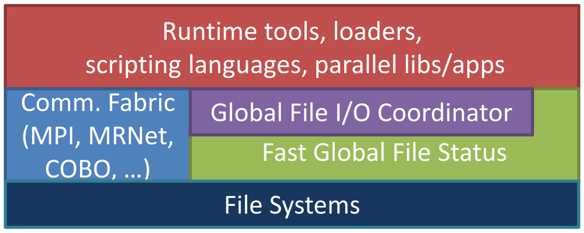


Fig. 3: Global file I/O strategy assistant suite

system, all processes see a file as if it is uniquely served, but it is physically broken down into smaller pieces and stored across multiple I/O servers. This model works well when processes concurrently access different parts of a file—a common access pattern for HPC applications working on a single large-scale data set—but it is ill-suited for concurrent I/O access with no such parallelism: i.e., concurrent reading of an executable, shared libraries, an input deck, or script files. In addition, this model does not provide optimal performance when a file does not need to have global visibility, e.g., the file contains some intermediate state shared by only a subset of processes. On the other hand, hierarchical hybrid approaches pursue building a file system or a library that combines node-local storage, as a high-bandwidth, low-latency cache, with a single remote server, as permanent capacity storage, in an attempt to bring best from both models [15], [8], [16], [17].

Unlike the parallel file system or hybrid approach, FGFS does hide the storage distribution to address the common issues associated with the distribution of a file, but rather exposes the distributed information to a program to assist it in making its own I/O trade-off decisions at scale. Thus, we view FGFS as a complementary approach to existing parallel or hybrid file system approaches.

IV. Approach

The core building block of FGFS is a node-local technique that raises local file identifiers into a global namespace. The global namespace then enables fast comparisons of file properties across distributed components with little or no access requirement on the underlying file systems. Specifically, our name resolution engine turns a local file path into a Uniform Resource Identifier (URI), a globally unique identifier of the file. This resolution process, however, is a purely local memory operation, as our technique builds an URI through a memory-resident table of file system mount points. This technique does not use off-node communication and thus does not impact scalability.

Figure 3 shows the high-level view of our solution, which spans several facets of I/O challenges. The FGFS layer itself is a scalable query layer within our overall I/O architecture responsible for the classification of files or file systems based on our file distribution models in Section III. FGFS provides the global information through easy-to-use, interoperable interfaces and implementations. It is also designed to assist the HPC software either directly or through the global coordinator layer that is responsible for orchestrating independent file I/O accesses. In the following, we describe the core components, techniques, and APIs of the FGFS package.

```
string & resolvePth(const char *pth) {
  string uriStr;
  FileUriInfo uriInfo;

  MountPointInfo mpInfo(true);
  mpInfo.getFileUriInfo(pth, uriInfo);
  uriInfo.getUri(uriStr);

  return uriStr;
}

void manageConfigs() {
  char *pth1="/etc/tool/conf";
  char *pth2="/usr/etc/tool/conf";
  char *pth3="/home/joe/.tool/conf";
  char *pth4="/lscracta/j_cwd/conf";

  string gid1 = resolvePth(pth1);
  string gid2 = resolvePth(pth2);
  string gid3 = resolvePth(pth3);
  string gid4 = resolvePth(pth4);
  ...
}
```

Fig. 4: Code snippet that uses Mount Point Attributes

A. Mount Point Attributes

The main abstractions that enables raising the namespace of local file names are packaged up into the `MountPointAttributes` module. Figure 4 shows an example that uses the high-level API exposed by this module. The `MountPointInfo` class used in the `resolvePth` function is the main data type that parses out a mount point table and aids in resolving the file name with its `getFileUriInfo` method. For example, on a node named `node1`, a node-local path `pth1` would be resolved into `file://node1/etc/tool/conf`, while remote files with paths `pth2`, `pth3` and `pth4` would be resolved into something like `nfs://s1-nfs.llnl.gov:/e/usr/etc/tool/conf`, `nfs://dip-nfs.llnl.gov:/v/joe/.tool/conf`, and `lustre://172.16.60.200:/tmp/j_cwd/conf`, respectively. On another node named `node2`, `pth1` would be turned into `file://node2/etc/tool/conf`, while `pth2` can be turned into `nfs://s2-nfs.llnl.gov:/e/usr/etc/tool/conf`, if served from a different NFS server.

We have ported this scheme to many file systems including local systems such as EXT 2, 3, and 4, various ramdisk-like file systems (i.e., `ramdisk`, `tmpfs`, and `rootfs`), remote file systems like NFS 3 and 4, CIFS, SMBFS, Lustre [11], GPFS [14] and PANFS [18], and hybrid systems such as union file systems. It is generally straightforward to port the

URI-based name resolution scheme to additional file systems; however, some modern HPC file systems impose a slightly higher complexity on this scheme. In the case of parallel file systems such as Lustre and GPFS, which split a file into multiple blocks and store them across multiple I/O servers, care is needed to build a globally unique handle that represents a file as a coherent logical object. Thus, we use the IP address of a mounted Lustre file system's metadata server, and the local device file name for a mounted GPFS. Either is globally unique for a given file, and can also be locally extracted via a mount point table.

B. Global File Status Queries

The global namespace provided by the `MountPointAttributes` module forms a reference space where local parallel name comparisons can compute common global properties. The global information must capture properties like the number of different sources that serve the file to all participating processes as well as the process count and the representative process of each source. FGFS provides low-level primitives on these properties. For example, the `FgfsParDesc` parallel descriptor returns various grouping information such as the number of unique groups and the size and the representative process of the group to which the caller belongs.

FGFS also composes these primitives in a way to capture the main issues that emerge according to our file distribution models described in Section III and exposes this information through a high-level query API. Specifically, the API class, `GlobalFileStatusAPI`, defines five virtual query methods:

- `isFullyDistributed`
- `isWellDistributed`
- `isPoorlyDistributed`
- `isUnique`
- `isConsistent`

Taking the local path of a file as the input, the `isWellDistributed` and `isPoorlyDistributed` queries test whether the file is served by a number of remote file servers that is higher or lower than a configurable threshold, respectively. Further, the `isFullyDistributed` query determines whether the file is served locally, a special case of being *well-distributed*, while the `isUnique` query tests whether it is served by a single remote server, a special case of being *poorly-distributed*. Finally, `isConsistent`, which is implied by `isUnique` evaluates whether the file's content is consistent across all application processes.

C. Support for Synchronous and Asynchronous I/O Patterns

The FGFS queries are designed to support two distinct classes of file I/O exhibited by distributed HPC programs. The first class is synchronous I/O: all processes that make lock-step progress synchronously are also synchronized in their I/O requests. In this case, the processes can call a synchronous FGFS query collectively and act upon the resulting global information. MPI-based bulk-synchronous applications would commonly use this type of queries. The second class is asynchronous, independent I/O: a process independently performs a UNIX file I/O without coordinating it with other processes of the same program. In this case, the process can call an asynchronous FGFS query without having to require other processes to call the same query. Sequential processing elements commonly used in HPC programs, such as dynamic loaders, script languages and run-time libraries can exhibit these I/O patterns. While independent, this I/O class can easily result in a large amount of file system requests in very short bursts at scale. Asynchronous FGFS queries are designed to allow such patterns to avoid thrashing the file system.

`SyncGlobalFileStatus` is the main file status query abstraction for synchronous I/O, which overrides all of the high-level methods that `GlobalFileStatusAPI` defines. Each overridden method requires all processes to participate. Upon a method being entered by all of the processes, `SyncGlobalFileStatus` extracts the global information of a file path on the fly. In contrast, for the asynchronous, independent file I/O, FGFS must build prior knowledge about the global information of all mounted file systems. Specifically, during initialization, `AsyncGlobalFileStatus` *synchronously* classifies all of the mount points and stores the global information into a map. It overrides all but the `isConsistent` method that `GlobalFileStatusAPI` defines. Each method does not require all processes to participate in the query—i.e., a single process can independently make a query. `AsyncGlobalFileStatus` simply uses the map to retrieve the global properties of a mounted file system where the file resides, which is merely a local memory operation.

If a program performs a large number of FGFS queries during its execution, the asynchronous query can perform better as the overhead of building the initial map is amortized over time. Typically, HPC systems contain only few tens of mount points. The major drawback of the asynchronous query is that it is not simple to implement the `isConsistent` semantics. All files that reside under an identical mount point inherit all global properties of the mount point root except consistency: the consistency of files varies at the individual file level. For example, given a mount point `/usr/apps`, a file `/usr/apps/foo` may be perfectly replicated across all file servers that service it, while a different file `/usr/apps/bar` may be inconsistent. A program can currently use the `isUnique` query to test consistency as a special case.

D. Scalable Extraction Algorithm of Global File Properties

When a local file name is raised to the global namespace by N processes, its representation is expanded to a list of N global names with as many as N unique names, and we need to compute this information in a scalable way. FGFS uses a scalable algorithm to identify equivalence classes that group the processes that see the same global name. The output of the algorithm is a reduced list of unique names. At the very least, each name in the reduced list should be associated with the number of equivalent processes, the identifiers of the equivalent processes, and the identifier of the representative process. The ratio of the total process count to the cardinality of the reduced list approximates the degree of distribution or replication of the file. The equivalence class information helps FGFS minimize the use of file I/O for its queries, such as `isConsistent`. If only a single unique name is found, the file is consistent by definition and no file I/O is needed. Further, if the degree of distribution is higher than one but still below a threshold, the representatives can perform I/O to compute and compare the checksums of the file, deducing global consistency.

A tree-based parallel reduce function can be used to implement a simple yet efficient equivalence class algorithm that folds the initial list into a list of unique names. The reduce function takes as input lists of key-value pairs where the URI representation of a file name serves as the unique key. For the implementation of the reduce function, we rely on existing implementations such as collectives in MPI or Tree-Based Overlay Networks (TBŌNs) like MRNet [19]. In the case of MPI, all processes form a binomial tree using point-to-point communication and the reduction operation is recursively applied from the leaf processes to the root, one level at a time. In the case of MRNet, the same reduce function is implemented as a filter that is applied at every level in the MRNet communication tree.

The computational complexity of the tree-based reduction is logarithmic with respect to the process count when the unique item count is small. However, it can pose scaling issues when the item count is very high—the reduced list can grow large enough that the algorithm performs similarly to a concatenation. Since at larger scales both the impact of the I/O issues and the chance of having a higher count of unique items significantly increases, we use a multilevel triaging scheme to put a scalability bound on the reduction operation. As a first-level test, we apply a fixed-sized parallel reduce function, a simple Boolean test across all processes, which determines if a file is served by a local file system. If local, the target file is *fully-distributed* and the equivalence classification is not needed for other queries. If remote, a Bloom filter-based cardinality estimation algorithm is applied as the next refinement test. It represents an efficient triaging that can determine if the target file is *poorly-distributed* or *well-distributed.*

E. Bloom Filter-based Cardinality Estimation for Triaging

A Bloom filter is a bit array of m bits with k defined hash functions. All bits in the array are initially set to the false bit. A hash function hashes each element in a set into a position in this bit array. Thus, the k-way hashing on all elements in the set ends up setting some positions in the array to the true bit. The resulting bit array then represents this set with the required space bound to be m bits. This technique is commonly used to test whether an element is a member of a set when high efficiency and low space usage are required. However, FGFS uses this technique to estimate the cardinality of the set scalably with a space-bounced reduce function.

In our scheme, each process of a distributed program allocates a bit array and applies hash functions to the URI representation of a file name. After computing the hash value and marking its result in the bit array, FGFS applies a parallel reduce function across all Bloom filters with the bitwise OR operator, effectively taking the union of all of the filters and broadcasts the union to all processes. Each process then computes the cardinality estimation on this reduced Bloom filter. The estimation approximates the number of distinct sources of the file, effectively determining if the file is *well-distributed* or *poorly-distributed.*

Although memory efficient, a Bloom filter is a probabilistic data structure. Thus, special care must be taken to ensure sufficient accuracy of the resulting estimate and to impose an error bound. For this we build on some of recent studies in the fields of distributed databases and information systems, in which Bloom filters are extensively used. In particular, we calculate the maximum likelihood value for the number of unique global names, given the state of the reduced Bloom filter, as proposed in [20]: the maximum likelihood value = $\frac{ln(1-\frac{t}{m})}{k\times ln(1-\frac{1}{m})}$, where m is the number of bits, t is the number of the true bits in the union, and k is the number of hash functions.

The accuracy and scalability of this estimation technique is also strongly influenced by the length of the Bloom filter, the number of hash functions, and the number of unique global names. Because a few well-designed hash functions can significantly reduce hashing conflicts and thereby reduce false positives rates, we use only two hash functions. Further, we minimize the error rate by keeping the ratio of the number of true bits to the length of the Bloom filter, also known as the density of a Bloom filter, to be 50% with respect to the worst-case scenario. This density is maintained when $k \approx \frac{m}{n} ln(2)$.

In this scheme, the worst case is not when the maximum number of processes that can run on a machine each contribute unique global names. The first triaging step has already ruled out that possibility. Instead, the worst case at this stage represents the maximum number of remote file systems to that a file path can resolve. This number is necessarily much smaller than the maximum concurrency. Even if we build a billion-core system with LLNL's current Linux cluster approach, in which we designate an NFS server per scalable unit (156 16-way compute nodes), the Bloom filters union can only contain up to 420K unique items (n). Given the required 50% density, the worst case for billion-way concurrency then requires the Bloom filter length to be 1.2 million bits or 150KB. Already, modern high-end systems can efficiently perform a reduce function over a buffer of that size. Additionally, with the expectation of a much higher core counts per compute node on future systems, we expect the actual worst case on real future systems to be much smaller.

F. Global File Systems Status Queries

While file status queries are geared towards the needs of read operations, its inverse function can generally benefit write operations: FGFS' file systems status queries. A file

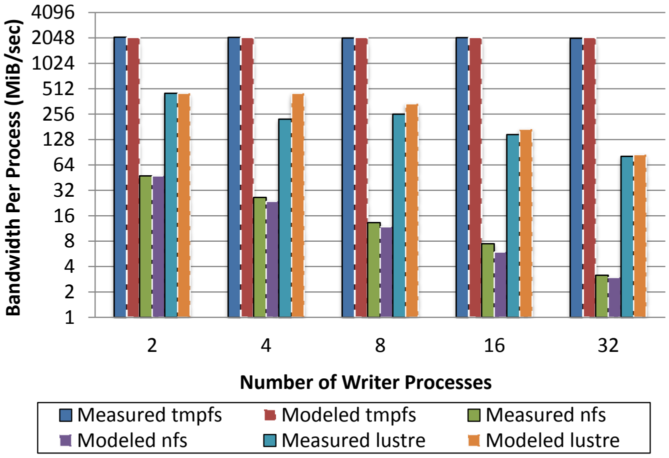


Fig. 5: Measured vs. modeled write performance

systems status query takes as an input a set of required global properties of a file system such as its available aggregate space, distribution, performance and scalability. The query then searches through all of the file systems mounted on the machine and selects the best matching locations.

`GlobalFileSystemsStatus` is the main data type that captures this concept. A distributed program passes to an object of this type the required global properties of a file system in the form of a `FileSystemCriteria` object. The status object then iterates over all mounted file systems and collects the global information of each mount point as a file path in order to test and to score how well the global properties meet the specified criteria.

The only criterion that the program must specify is the space requirement. As the status object iterates through the mounted file systems, it calculates the aggregate number of bytes needed by all equivalent processes and tests whether the target file system has enough free space. In particular, when a mount point resides across multiple remote file servers for the target program, distinct groups of processes write into their own associated file servers. Besides the space requirement, `FileSystemCriteria` provides the following options to allow a program to refine the requirement of a file system further:

- `SpeedRequirement`: the sequential processing performance of a file system with the choices of `LOW` and `HIGH`;
- `DistributionRequirement`: the distribution of a file system with the choices of `UNIQUE`, `LOW`, `HIGH`, and `FULL`;
- `ScalabilityRequirement`: the scalability of a file system with the choices of `SINGLE`—i.e., a plain NFS, and `MULTI`—i.e., a parallel file system.

If multiple file systems match the selected criteria, FGFS orders them using a scoring function: $Score(mp) = \frac{Scalability(mp)}{max(Scalability(mp), Distribution(mp))} \times Speed(mp)$, where mp is a file system represented as a mount point path, *Scalability* is a function that returns the minimum process count that can start to saturate the file system, *Distribution* is the number of requesting processes, and *Speed* is the function that returns the sequential processing performance of the file system. Higher scores indicate better selections. The intuition is that a file system would perform at its peak speed until the number of processes that concurrently access it exceeds its inherent scalability. Once exceeded, performance degrades at a linear rate.

Figure 5 compares the measured performance of write operations with the performance modeled by our score function on three distinct types of file systems: local `tmpfs`, globally mounted NFS and Lustre. It shows the score function closely models the amount of bandwidth that each process gets as increasing numbers of processes simultaneously write to those file systems mounted on an LLNL Linux cluster. The measured performance represents average per-process bandwidth that our Interleaved or Random I/O Benchmark (IOR) [21] tests report, as each IOR process synchronously writes 50-MiB file 10 times on each file system type.

While our performance model is reasonable when the file system is not heavily loaded, we recognize that the current loads of the file system can also significantly affect I/O performance. Thus, we plan to explore mechanisms to increase the accuracy of our score function as part of our future research direction.

G. *Interoperation with HPC Communication Fabrics*

FGFS is designed as a general purpose layer, as shown in Figure 3. Thus, it is critical for FGFS to interoperate well with a wide range of HPC communication fabrics. For this purpose, FGFS builds on a virtual network abstraction that defines a rudimentary collective communication interface containing only the basic operations: a simple broadcast, multicast, and reductions. This layer can be implemented on top of many native communication layers through plug-ins that translates the native communication protocols to this rudimentary collective calls. Thus far, we have developed and tested virtual network plug-ins for MPI, MRNet and LaunchMON [22], but this technique is equally applicable to other overlay networks and bootstrappers such as PMGR Collective [23], COBO [23], [22], and LIBI [24].

The virtual network abstraction allows an HPC program written to a specific communication fabric to instantiate FGFS on that fabric. Further, this approach does not restrict the communication strategy for the global file I/O coordinator layer to a single communication fabric. The coordinator must be able to support various communication scenarios while coordinating independent I/O patterns exhibited by sequential elements. For example, a dynamic loader (e.g., `ld.so`) does not have a communication network on its own and further has to be executed before any fabric bootstrap phase. One way to support this model is through an external service that exploits system-level networks, e.g., provided by scalable resource manager software.

V. Experiments

In this section, we present our performance experiments and three case studies that show our software system's per-

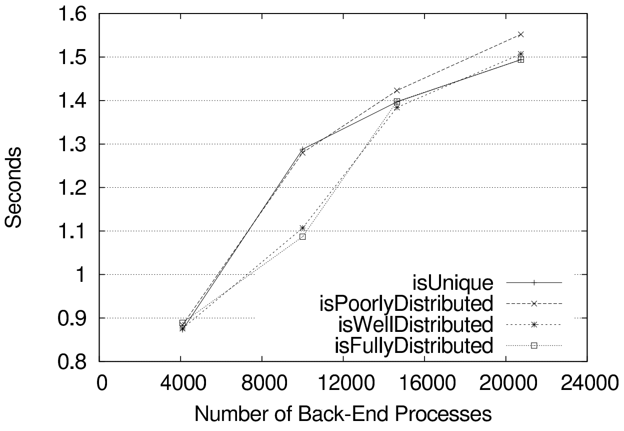


Fig. 7: Performance with MRNet

formance, scalability, and utility.

A. Performance Experiments on Three Applications

We run all of our experiments on two Linux clusters installed at LLNL, named Zin and Merl. Each compute node of these clusters has 2 sockets and 32 GB of RAM. Each socket is populated with an 8-core 2.6 GHz Intel Xeon E5-2670 processor, resulting in 16 cores per node. Zin consists of 2,916 compute nodes totaling 46,656 cores; Merl is a smaller system with the same node type, consisting of 154 compute nodes with a total of 2,464 cores. Nodes are connected by a Qlogic Infiniband QDR interconnect.

We test all high-level FGFS file status queries shown in Section IV-B on the executable and shared libraries of three LLNL multi-physics applications, as they are currently configured and staged on our production systems. The test applications include KULL, with 848 dependent shared libraries. Two other applications are smaller in their aggregate binary sizes and depend on fewer numbers (19 and 31) of generally smaller shared libraries.

The first observation of the experimental results is that `AsyncGlobalFileStatus` queries generally perform better than `SyncGlobalFileStatus` queries in particular when the number of shared libraries to test for an application is substantially high. However, this performance characteristic is reversed with a low count of shared libraries. This is caused by the `SyncGlobalFileStatus` query having to run the classification algorithm on-demand for each file whereas `AsyncGlobalFileStatus` must run the same algorithm on all of the mount point root paths during its initialization. Thus, once the query count becomes higher than the mount point count, the initialization cost of `AsyncGlobalFileStatus` is starting to pay off.

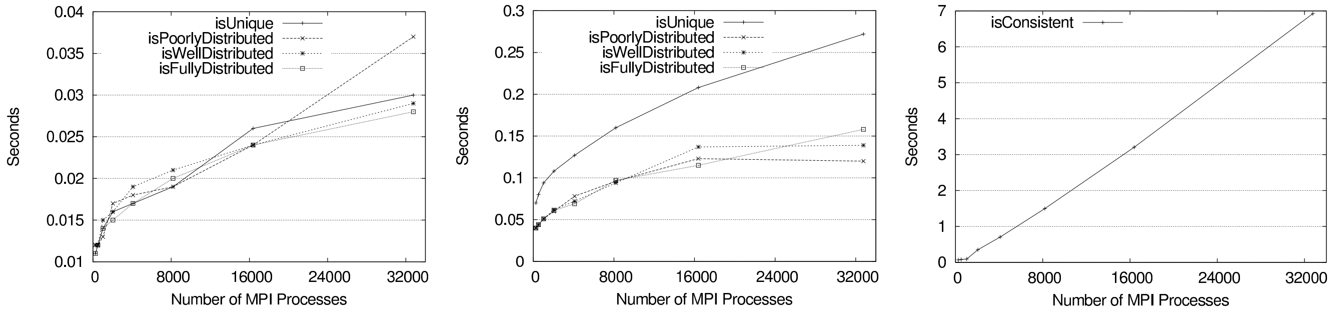
We now detail our results on the KULL application as this application represents the worst case of our three test codes. We use a simple performance benchmark that calls each FGFS file status query on the KULL executable and its many shared library dependencies, and reports the global property statistics at the end. We report timings for the core FGFS section of this benchmark that includes the FGFS initialization API call and a main loop that performs the query on entire target files.

Figure 6 shows the performance of all of the high-level file status queries of FGFS instantiated with the MPI plug-in. We use the default MPI installed on the system, MVAPICH version 1.2. In all but the `isConsistent` case, the FGFS file status queries 849 files (1 executable and 848 shared libraries) are executed in a fraction of a second on up to 32,678 MPI processes. More specifically, at 32K processes it takes `AsyncGlobalFileStatus` 30, 37, 29 and 28 milliseconds to perform `isUnique`, `isPoorly`, `isWell`, and `isFully` file status queries respectively on all files. The synchronous case is about an order of magnitude slower with 272, 120, 139, 158 milliseconds, respectively. More importantly than those absolute numbers, though, Figure 6a and Figure 6b show that most asynchronous and synchronous FGFS queries exhibit strong scaling trends, either logarithmic or linear with a flat slope. The average R^2 value of a logarithmic fit across 8 cases (all but `isConsistent`) is .903.

In the case of the `isConsistent` query shown in Figure 6c, our further analysis shows that the linear scaling is in part due to the poor scaling of `MPI_Comm_split` in the underlying MPI implementation. FGFS splits the MPI communicator when distinct equivalence groups are identified, and a query requires representative processes to compute MD5 checksum by reading the file and to broadcast the checksum to the processes in their group to deduce global consistency. Our measurements show that the cost of `MPI_Comm_split` alone already accounts for over 50% of total overhead at moderate scales with a few thousands of processes and we suspect that the comm-split overhead would dominate the overall performance to a larger extent at higher scales. However, more scalable solutions that exhibit logarithmic scaling behavior exist [25], but have yet to be included into standard MPI distributions. Until this is the case, we plan on integrating them directly into FGFS as a short-term solution.

Figure 7 shows the performance results with FGFS instantiated with a beta MRNet version 3. While its overhead is higher than that of FGFS on MPI, the scaling trends are generally similar—the average R^2 value of a logarithmic fit is .966. FGFS on MRNet incurs higher overhead due to larger communication costs going through MRNet, as it uses TCP/IP. That use allows MRNet to provide a general-purpose, portable overlay network for distributed run-time tools that can run on machines with different architectures and operating systems.

Finally, our measurements show that FGFS scales similarly on the other two multi-physics applications. However, the absolute performance is better as the numbers and sizes of shared libraries to query are substantially smaller. FGFS still proves to be effective on these applications compared to a traditional approach. For example, already at 512 processes, `isConsistent` is measured to be a factor of 9,016 (0.002 vs. 18.032 seconds) better on the first application and a factor of 27,735 (0.002 vs. 55.470 seconds) better on the second application than the traditional checksum scheme.



(a) `AsyncGlobalFileStatus` (b) `SyncGlobalFileStatus` (c) `isConistent`

Fig. 6: Performance with MPI

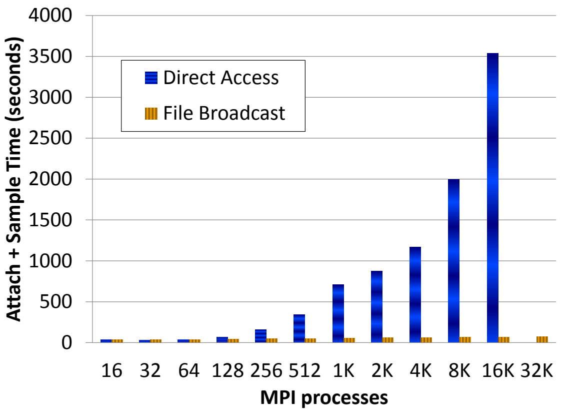


Fig. 8: Attach + sample time for STAT

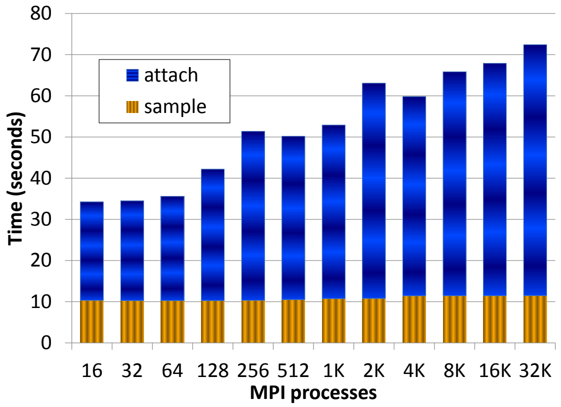


Fig. 9: Attach + sample time for STAT with file broadcasting

B. Case Study on STAT

The Stack Trace Analysis Tool (STAT) [7] is a highly scalable, lightweight debugging tool. STAT gathers stack traces from all processes of a parallel application and merges them into a call-prefix tree, which groups similar processes into equivalence classes. This equivalence means that a single process within a class can serve as a representative of the entire set. Thus, the user can effectively reduce the process search space from the entire application to a single representative process of each equivalence class.

STAT uses MRNet for scalable communication via a tree-based overlay network and the StackWalkerAPI from DynInst [26] for lightweight stack sampling. Our previous work [27] identified and addressed several scalability bottlenecks and demonstrated scaling over 200,000 processes. One of the bottlenecks identified was in fact STAT's file system access and a prototype Scalable Binary Relocation Service (SBRS) feature was developed to address this issue. With FGFS, we were able to overcome the shortcomings of SBRS. For one, SBRS requires broadcasting all of the shared libraries that are linked into the target executable, while FGFS allows us to broadcast only the shared libraries required to resolve the functions in the current execution stack. Furthermore, FGFS adds the ability to handle dynamically loaded libraries, while SBRS could not since it relied on static properties of the executable.

STAT was modified to use `AsyncGlobalFileStatus` and file multicasting. In particular, before accessing a file, each STAT daemon queries FGFS to determine the location of the file. If the file is well distributed, then the daemon directly accesses it through the file system. If it is not well distributed, the daemon sends a request for the file contents to the STAT frontend. The frontend reads the file contents from the file system and sends the contents through the MRNet tree. Additional logic is implemented in the MRNet filter code running in the tree communication processes to ensure that file contents is only sent down paths to daemons that requested the contents. Additionally, the communication processes cache file contents to handle asynchronous requests without performing redundant file reads and sends.

To evaluate the effectiveness of STAT using FGFS and file broadcasting, we ran it on the KULL application on Zin and compared the results to STAT without any alternative I/O scheme. Since a StackWalkerAPI object first accesses the file during attach, but does not read its contents until sampling the stack traces, we measure both the attach time as well as the initial sample time to provide a fair comparison. With the file broadcast version, the file contents is broadcasted during attach, while in the version in which daemons directly access the file system, the file system bottleneck is hit during the actual reads caused by the stack sampling operation.

The build of KULL on which we tested FGFS had all of its packages compiled into the base executable which totaled

1.7GB, rather than a build that links in the packages as shared libraries. StackWalkerAPI uses lazy parsing and thus only needs to access shared libraries that define the functions in the current execution stack. In practice, a hung application will only span a handful of shared libraries, thus our experimental setup with the large base executable represents a worst-case scenario where the STAT daemons need to read in a large quantity of file contents. In all of our tests, we let KULL hang in an interactive prompt to ensure that the application is in a consistent state across test runs. Furthermore, each test is run in a new allocation to ensure that file system caching is not a factor.

Figure 8 shows a comparison of the combined attach and sample time with and without file broadcasting. In the case where we did not use file broadcasting, the time was dominated by the sampling operation. The attach scaled from one quarter of a second up to a second, while the sampling time scales linearly with the size of the system, taking nearly 1 hour at 16K MPI processes. We omitted a test at 32K without file broadcasting since this essentially causes a denial-of-service attack on the file system. By comparison, with our file broadcasting scheme, the attach and sample time was 67.87 seconds at 16K MPI processes and 72.43 seconds at 32K MPI processes. The timing breakdown for the file broadcast case can be seen in Figure 9, which shows that the sample time was consistently around 11 seconds, regardless of scale. An analysis of the attach time, which is dominated by the actual broadcasting of the file contents, shows logarithmic scaling with an R^2 value of .958.

C. Case Study on SPINDLE

The Scalable Parallel Input Network for Dynamic Loading Environment (SPINDLE) takes the role of the global coordinator layer as shown in Figure 3. SPINDLE aims at eliminating the harmful effects of uncoordinated I/O storms, which can lead to unacceptable application start-up overhead and further denial-of-service attacks as described in Section II-A. It consists of a set of load servers that form a forest-based overlay network with support for an arbitrary number of trees in the forest. Only the load servers at the roots of these trees are designated to perform direct file system operations including metadata operations and to broadcast the results back to other servers. The servers then serve the cached results to any sequential processing element including the dynamic loader, which is extended with the SPINDLE client.

We integrated SPINDLE with FGFS on top of LaunchMON and its communication fabric. Upon receiving a file access request from a SPINDLE client, the servers use `AsyncGlobalFileStatus` to choose between a direct file system access and file broadcasting. More specifically, if the file is globally unique, only designated servers at the roots of the trees access the file system and broadcast the result to all others via the trees. If it is well distributed, the request is then passed through to the file system. Instead, if the file is poorly distributed, the `FgfsParDesc` parallel descriptor is used to determine representative load servers that then each access the file system and broadcast the result to other equivalent servers.

Our performance evaluation on LLNL's Sierra Linux cluster indicate that FGFS ensure the alternative scalable I/O scheme of SPINDLE regardless of where the requested file resides. We ran and scaled under SPINDLE the Pynamic [28] benchmark that is designed to stress the underlying dynamic loading system. Our results show that FGFS allowed SPINDLE to improve the benchmark performance by a factor of 3.5 over the traditional approach at 768 MPI processes without make predetermined assumptions about where Pynamic's 495 shared libraries should be staged. This scale represents the limits at which Pynamic could run under the traditional scheme without significantly affecting other jobs. With SPINDLE/FGFS, this benchmark was shown to continue to scale well up to 15,360 MPI processes with no disruption to shared resources.

D. Case Study on SCR

The Scalable Checkpoint/Restart (SCR) library [8] is a multilevel checkpoint system developed at LLNL. In addition to saving checkpoints on the parallel file system for long-term persistence, SCR is designed to save checkpoints to temporary scratch storage located closer to the compute nodes, such as storage installed on each compute node or perhaps burst buffers [29] installed elsewhere in the system. This temporary scratch storage is not reliable, and thus SCR applies redundancy schemes to the data it stores there. The availability, capacity, and mount point location of such storage vary across HPC platforms, and currently a configuration file must be created to specify these details. Someone with an understanding of certain low-level implementation details of SCR must create this configuration file, and in practice, this requirement significantly limits SCR's portability.

Using FGFS, we modified SCR to discover the state of the system dynamically, so that all of this manual configuration is eliminated and replaced by a few API calls. In particular, before an application writes a checkpoint, it specifies an upper bound on the number of bytes that each process will save. Using this information, SCR fills in a `FileSystemCriteria` object and calls upon `GlobalFileSystemsStatus` to discover the fastest available storage location capable of storing the checkpoint. SCR uses the reported speed of the storage location to determine the optimal checkpoint frequency, and by calling `SyncGlobalFileStatus` and `FgfsParDesc` with the corresponding path, we obtain the process groups that share the same physical storage. SCR uses this global information to construct more efficient redundancy schemes. These enhancements ensure that SCR delivers the optimal performance without having to rely on arduous manual configuration efforts, representing significant improvements for effective performance portability across diverse HPC platforms.

VI. RELATED WORK

Our work focuses on supporting efficient file I/O strategies for HPC beyond the common access patterns of applications on large-scale data sets. Much research exists that addresses parallel I/O on large-scale data sets. I/O researchers strive to have the greatest impact to the common access patterns by aiming to improve a layer in the parallel I/O software stack that consists of storage technologies, parallel file systems, I/O forwarding, parallel I/O middleware, and parallel high-level I/O libraries.

FGFS sits above the file systems layer and thus is agnostic to storage technologies. In Section III, we discussed how our work is distinguished from and related to parallel file systems and other hybrid approaches. As previously noted, FGFS complements the file systems layer by characterizing mounted file systems to support HPC software in making intelligent I/O decisions.

The I/O forwarding layer enables scientific applications running on the compute nodes to forward its I/O transparently to dedicated resources like I/O nodes. This approach allows intermediate forwarding agents to use aggressive I/O optimization techniques such as caching and aggregating without introducing noise to the compute nodes. IBM CIOD [30] and Cray Data Virtualization Service (DVS) [17] are the commercial implementations on the IBM Blue Gene family and the Cray XT/XE family, respectively. ZOID [31] is an non-proprietary solution for Blue Gene. Recently, open-source projects such as IOFSL [32], and DIOD [33] have begun to emerge with the goal of providing the benefit of I/O forwarding techniques to a wider range of systems. The aim of these techniques is to transparently reduce the load to any target file system under its control. Thus, it does not expose the characteristics of the file system to a program for use in its I/O strategy. An I/O forwarding topology is typically static and optimized for the common access patterns on large-scale data sets. Researchers have shown that the access pattern of dynamic loaders can be improved by staging all shared libraries in a dedicated server with a dedicated I/O forwarding topology on top of it [34]. In contrast, the goal of FGFS is to support I/O strategies so a program does not experience a bottleneck, regardless of where the file exists.

I/O middleware such as MPI-IO [3] facilitates concurrent access by groups of processes while matching its abstractions with a target programming model. For example, MPI-IO models parallel file I/O as message passing—i.e., a read as a receive and a write as a send—and provides abstractions that describe the partitioning of file data and perform collective I/O among MPI processes. Here, the goal is to help a parallel program written in a particular programming model to realize efficient and portable file I/O beyond UNIX file I/O. A myriad of research exists that design and evaluate efficient and scalable I/O strategies to support middleware concepts such as the description of the file data and collective I/O [35], [36], [37], [38]. Again, FGFS is complementary to this approach by being able to give a hint to the I/O middleware if a file would be better off with collective or serial I/O. High-level I/O libraries such as HDF5 [4], Parallel netCDF [5] and ADIOS [6] map scientific application abstractions onto storage abstractions and provide data portability. They are best suited for large-scale data sets on parallel file systems.

Finally, our work is related to collective OS research [39]. This research shares the common goal of providing the OS with parallel awareness and further collective services. Our global file I/O coordinator concept can be viewed as part of this domain. In identifying OS issues for petascale systems [9], researchers argued for the need on much research and development that prevent I/O nodes and their paired compute nodes from creating what will essentially be denial-of-service attacks on file systems. Specifically, they demanded new caching strategies, collective I/O calls, and automatic I/O reductions and broadcasts through the I/O nodes for petascale machines to achieve scalability. FGFS can provide an OS service with the global information of a file through the mounted file systems so that the service can determine its coordination strategy effectively.

VII. Conclusion

Efficient use of storage is essential for HPC software. To realize the full potential of a system, the software must use efficient mechanisms in accessing every level in the storage hierarchy. In particular, with a growing trend towards very large numbers of compute cores, using an access scheme that avoids contention at lower levels is becoming increasingly important. However, devising efficient alternative file I/O schemes are becoming a greater challenge at the file system level. Today's systems mount many file systems with distinct performance characteristics and a file can reside in any of these file systems. However, this structure is hidden from the user. Thus, highly concurrent access to the file with no detailed knowledge on its global properties often incurs unacceptable performance overhead and further disrupts the entire computing facility. Worse, the existing parallel I/O software stack does not provide rich enough abstractions and mechanisms to support alternative I/O schemes often needed to eliminate serious I/O scaling issues beyond I/O on large-scale data sets.

We have developed Fast Global File Status as a means to retrieve the global information on a file or file systems, filling that gap. The goal of FGFS is to provide a reusable, general-purpose layer that exposes the information relevant to the distribution to a file. It is designed to support alternative I/O schemes beyond direct I/O for a wide range of HPC software. Our performance evaluation shows that FGFS represents orders of magnitude improvements over the traditional approaches. Further, our case studies on STAT, Spindle, and SCR suggest that fast mechanisms to retrieve the global information on files and file systems are widely needed and greatly help a wide range of HPC software improve their file I/O patterns and overall scalability in a portable fashion.

Our results show that our proposed system is already efficient, scalable, and has utility, as a stand-alone software module. But we envision that FGFS will be deeply integrated into various HPC software stacks, further extending its benefits to many essential HPC elements. They include our global file I/O strategy assistant suite, next generation resource management software, and parallel-aware operating systems. Our work on integrating FGFS with SPINDLE and SCR is in fact our first step towards this vision.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-PROC-554055).

References

[1] "Top 500 Supercomputer Sites," http://www.top500.org. [Online]. Available: http://www.top500.org/

[2] Lawrence Livermore National Laboratory, "Advanced simulation and computing sequoia," https://asc.llnl.gov/computing_resources/sequoia.

[3] P. Corbett, D. Fietelson, S. Fineberg, Y. Hsu, B. Nitzberg, J. Prost, M. Snir, B. Traversat, and P. Wong, "Overview of the mpi-io parallel i/o interface," in *Third Workshop on I/O in Parallel and Distributed Systems, International Parallel Processing Symposium*, April 1995.

[4] *Hierarchical Data Format (HDF)*, http://hdf.ncsa.uiuc.edu.

[5] *Network Common Data Form (netCDF)*, http://www.unidata.ucar.edu/packages/netcdf.

[6] *Adaptable IO System (ADIOS)*, http://www.olcf.ornl.gov/center-projects/adios.

[7] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack Trace Analysis for Large Scale Debugging," in *The International Parallel and Distributed Processing Symposium*, Long Beach, CA, 2007.

[8] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC*. IEEE, 2010, pp. 1–11.

[9] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "Operating system issues for petascale systems," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 2, pp. 29–33, Apr. 2006. [Online]. Available: http://doi.acm.org/10.1145/1131322.1131332

[10] J. Rathkopf, D. Miller, J. Owen, L. Stuart, M. Zika, P. Eltgroth, N. Madsen, K. McCandless, P. Nowak, M. Nemanic, N. Gentile, N. Keen, T. P. R. Buck, and J. K. Hollingsworth, "KULL: LLNL's ASCI Inertial Confinement Fusion Simulation Code," *Physor 2000, ANS Topical Meeting on Advances in Reactor Physics and Mathematics and Computation into the Next Millennium*, May 2000.

[11] W. Yu, R. Nononha, S. Liang, and D. K. Panda, "Benefits of High Speed Interconnects to Cluster File Systems: A Case Study with Lustre," in *The International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, 2006.

[12] Allinea Software, "Allinea DDT the Distributed Debugging Tool," http://www.allinea.com/index.php?page=48.

[13] Rogue Wave Software, "TotalView Debugger," http://www.totalviewtech.com/productsTV.htm.

[14] IBM, "General Parallel File System," http://www-03.ibm.com/systems/software/gpfs/.

[15] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: making the best use of solid state drives in high performance storage systems," in *ICS*, D. K. Lowenthal, B. R. de Supinski, and S. A. McKee, Eds. ACM, 2011, pp. 22–32.

[16] J. Bent, G. A. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "Plfs: a checkpoint filesystem for parallel applications," in *SC*. ACM, 2009.

[17] S. Sugiyama and D. Wallace, "Cray dvs: Data virtualization service," in *Cray User Group 2011*, 2008.

[18] Panasas, "PanFS," http://www.panasas.com/products/panfs.

[19] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools," in *SC '03*, Phoenix, AZ, 2003.

[20] O. Papapetrou, W. Siberski, and W. Nejdl, "Cardinality estimation and dynamic length adaptation for bloom filters," *Distributed and Parallel Databases*, vol. 28, no. 2-3, pp. 119–156, 2010.

[21] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 42:1–42:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413413

[22] D. H. Ahn, D. C. Arnold, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Overcoming scalability challenges for tool daemon launching," in *ICPP*. IEEE Computer Society, 2008, pp. 578–585.

[23] *PMGR Collective*, http://sourceforge.net/projects/pmgrcollective.

[24] J. D. Goehner, D. C. Arnold, D. H. Ahn, G. L. Lee, B. R. de Supinski, M. P. LeGendre, M. Schulz, and B. P. Miller, "A framework for bootstrapping extreme scale software systems," in *First International Workshop on High-performance Infrastructure for Scalable Tools (WHIST)*, Tucson, AZ, 2011.

[25] A. Moody, D. H. Ahn, and B. R. de Supinski, "Exascale algorithms for generalized mpi_comm_split," in *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 9–18.

[26] B. R. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[27] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit, "Lessons learned at 208k: towards debugging millions of cores," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–9.

[28] G. L. Lee, D. H. Ahn, B. R. de Supinski, J. Gyllenhaal, and P. Miller, "Pynamic: the Python Dynamic Benchmark," in *Proceedings of the IEEE International Symposium on Workload Characterization*, Boston, MA, September 2007.

[29] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *28th IEEE Storage Conference*, Pacific Grove, CA, April 2012.

[30] J. Moreira, M. Brutman, J. Castaños, T. Engelsiepen, M. Giampapa, T. Gooding, R. Haskin, T. Inglett, D. Lieber, P. McCarthy, M. Mundy, J. Parker, and B. Wallenfelt, "Designing a highly-scalable operating system: the blue gene/l story," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188578

[31] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "Zoid: I/o-forwarding infrastructure for petascale architectures," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 153–162. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345230

[32] N. Ali, P. H. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. B. Ross, L. Ward, and P. Sadayappan, "Scalable i/o forwarding framework for high-performance computing systems," in *CLUSTER*. IEEE, 2009, pp. 1–10.

[33] *Distributed I/O Daemon (DIOD)*, http://code.google.com/p/diod.

[34] S. M. Kelly, R. Klundt, and J. H. L. III, "Shared libraries on a capability class computer," in *Cray User Group 2011*, 2011.

[35] Y. Chen, X.-H. Sun, R. Thakur, P. C. Roth, and W. D. Gropp, "Lacio: A new collective i/o strategy for parallel i/o systems," in *IPDPS*. IEEE, 2011, pp. 794–804.

[36] S. Moyer and V. S. Sunderam, "Pious: A scalable parallel i/o system for distributed computing environments," pp. 71–78, 1994.

[37] D. Levine, N. Galbreath, and W. Gropp, "Applications-driven parallel i/o," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993, pp. 462–471. [Online]. Available: http://doi.acm.org/10.1145/169627.169784

[38] Z. Zhang, A. Espinosa, K. Iskra, I. Raicu, I. T. Foster, and M. Wilde, "Design and evaluation of a collective io model for loosely coupled petascale programming," *CoRR*, vol. abs/0901.0134, 2009.

[39] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, P. Tomlinson, and M. Roberts, "Improving the scalability of parallel jobs by adding parallel awareness to the operating system," in *In SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2003, p. 10.