Diving into Petascale Production File Systems through Large Scale Profiling and Analysis

Feiyi Wang[†], Hyogi Sim[†], Cameron Harr[*], Sarp Oral[†]
Oak Ridge National Laboratory[†] Lawrence Livermore National Laboratory[*]

ABSTRACT

As leadership computing facilities grow their storage capacity into the multi- petabyte range, the number of files and directories leap into the scale of billions. A complete profiling of such a parallel file system in a production environment presents a unique challenge. On one hand, the time, resources, and negative performance impact on production users can make regular profiling difficult. On the other hand, the result of such profiling can yield much needed understanding of the file system's general characteristics, as well as provide insight to how users write and access their data on a grand scale. This paper presents a lightweight and scalable profiling solution that can efficiently walk, analyze, and profile multi-petabyte parallel file systems. This tool has been deployed and is in regular use on very large-scale production parallel file systems at both Oak Ridge National Lab's Oak Ridge Leadership Facility (OLCF) and Lawrence Livermore National Lab's Livermore Computing (LC) facilities. We present the results of our initial analysis on the data collected from these two large-scale production systems, organized into three use cases: (1) file system snapshot and composition, (2) striping pattern analysis for Lustre, and (3) simulated storage capacity utilization in preparation for future file systems. Our analysis shows that on the OLCF file system, over 96% of user files exhibit the default stripe width, potentially limiting performance on large files by underutilizing storage servers and disks. Our simulated block analysis quantitatively shows the space overhead when doing a forklift system migration. It also reveals that due to the difference in system compositions (OLCF vs. LC), we can achieve better performance and space trade-offs by employing different native file system block sizes.

1 INTRODUCTION

Present-day large-scale United States Department of Energy (DOE) High Performance Computing (HPC) facilities, such as Oak Ridge Leadership Computing Facility (OLCF) [10], Livermore Computing Center (LC) [14], Argonne Leadership Computing Facility (ALCF) [1], and National Energy Research Scientific Computing Center (NERSC) [8], are equipped with parallel file system capacities in tens of petabytes. Next generation parallel file systems at these facilities will have capacities in the hundreds of petabytes. Understanding the file system metadata can provide useful insight into how these file systems are used and how to develop and deploy better file systems for the future [16, 17, 20, 24]. However, a tool for effectively analyzing the metadata contents of these large-scale file systems has been lacking due to the technical challenges of efficiently profiling billions of file system entries.

Taking one of the most widely deployed parallel file systems, Lustre, as an example, there is little information provided regarding the characteristics of the files beyond overall health status, inode usage, and other common attributes. Third party tools designed for file system analysis at this scale are few in number. Robinhood [3] is perhaps the best known tool and is considered a *policy engine* rather than a profiler, providing policy-based management. Policy engines provide more functionality than *fprof* but operate by collecting and storing the file system metadata in a database. To collect that metadata, Robinhood can either scan a POSIX file system or connect to Lustre's MDT changelog [7] to log all file system changes to its database. Scanning and populating the Robinhood database can take an excessively long time with a moderately to heavily-used file system. For example, we observed processing rates of 50-250 files/sec when using Robinhood with twelve scan threads on production file systems in LC, a rate that would take over two months to finish just one of LC's larger file systems. Additionally, simultaneous I/O from thousands of users can cause the DB to fall behind the active changelog and fail to catch up. Newer versions of Robinhood reportedly provide greatly improved performance, but have not been tested by LC or OLCF.

In this paper, we present *fprof*, a profiling tool that addresses the aforementioned problems. Specifically, *fprof* is scalable and can run in parallel on multiple nodes to speed up the profiling process. In addition, *fprof* is portable, using only standard *Portable Operating System Interface* (POSIX) file system features with no file system-specific features. *fprof* is also flexible in that HPC administrators can run it on demand. Besides the design and implementation of *fprof*, in this paper we present the profiling results of two large-scale production parallel file systems deployed at OLCF and LC. In particular, our analysis focuses on three use cases:

File system snapshot and characterization The file size, file quantity, and file attribute information generated by *fprof* describe the characteristics of the current file system and provide vital input into how users utilize the file system. That user behavior in turn helps us project the requirements for future file system designs. For instance, the file size distribution results of the two profiled file systems (§ 3.1) revealed distinct application access patterns between the two sites, leading to separate block size settings.

Stripe pattern analysis Lustre allows users to control how files are striped across multiple storage targets, i.e., stripe count. However, our analysis reveals over 96% of the files do not exploit this feature at all. Using the default stripe count may inhibit the potential performance increase from wider stripe count, particularly for large files. Such user behavior justifies the need for intelligent features, such as Progressive File Layout (PFL) in Lustre [15].

PDSW–DISCS'17, November 12–17, 2017, Denver, CO, USA

Space utilization for future file systems Using *fprof*, we can quantitatively address the following what-if question for future file system planning: if we were to migrate the entire Lustre-based file system to a GPFS-based file system [2], what would be the space utilization impact, considering different block allocation strategies of the two file systems? The results demonstrably impacted future system planning and acquisitions.

2 DESIGN AND IMPLEMENTATION

fprof [1] takes a target directory path as an argument and profiles all the entries under the target directory. The target can be the root or a subdirectory of a file system. The basic profiling results from *fprof* include the histogram of files according to their sizes and the statistical summary of the file system, such as the number of files/directories, average file size, maximum number of entries in a directory, top *N* largest files in the system, etc. Figure 1 provides a snapshot of the example output. In addition, *fprof* also provides a block-usage space consumption conversion tool. We believe this feature is particularly useful for scenarios where the contents of a given path is transferred from one parallel file system technology to another, e.g., Lustre to GPFS.

2.1 Design Overview

At the core of the lightweight and scalable design of *fprof* is a parallelization engine (PE). The primary function of the PE is to distribute the workload *evenly* across a cluster of machines to scan the entire file system in an efficient manner. The ability to parallelize the workload is paramount to tackling the system at scale.

Specifically, we adopt a *work stealing* algorithm for the workload distribution and the *Dijkstra-Scholten* algorithm for the distributed termination, as suggested for a parallel tree walk [21, 23]. In the work stealing algorithm, each worker process maintains its independent *work queue* and processes *work units* as needed. When the work queue becomes empty, it randomly picks one or more of its peers and sends out a *work request*. A busy worker process then responds by allocating a portion of work units in its work queue to the requester. The highlight of this load distribution method lies in its self-balancing nature, i.e., eventually the workload will be evenly distributed regardless of the initial distribution. It gracefully addresses the runtime load imbalance caused by the heterogeneity of the cluster, e.g., stragglers due to weaker processing power. Since there is not a central coordinator (master), for this fully distributed paradigm to work, we need a mechanism to detect the termination condition when all work items in all work queues are finished. In our current implementation, we adopt the Dijkstra-Scholten algorithm [18] for detecting the completion of the work.

2.2 Implementation

Here, we summarize implementation related issues that we have encountered while developing *fprof*.

First, the length of the local work queue for each worker process is dynamic and dependent on the number of files within a directory and how directories are laid out. We found that multiple sibling directories, each with millions of files, can potentially exhaust the local work queue: e.g., the largest single directory at LC contains

```
Running parameters:
     Num of hosts:        12
     Num of processes:    84
     Syslog report:       no
     Stripe analysis:     no
     Root path:           ['/p/lscratche']
     ...

Fileset histogram:
       Buckets     Num of Files          Size    %(Files)    %(Size)
    <= 4.00 KiB    405,101,078    478.02 GiB      30.17%      0.01%
    <= 8.00 KiB    125,265,283    711.78 GiB       9.33%      0.02%
       ...
    >  4.00 TiB              5     50.87 TiB       0.00%      1.31%

fprof epilogue:
     Directory count:          45,265,268
     Sym links count:          10,430,723
     Hard linked files:        309,219
     File count:               1,342,586,738
     Sparse files:             824,561,829
     Avg file size:            2.83 MiB
     Max files within dir:     26,546,573
     ...
```

Figure 1: Snippet of *fprof* output for the LC file system.

over 26 million files. To curtail the size of the local work queue, *fprof* inserts newly scanned files to the front of the queue and newly scanned directories to the tail of the queue. This optimization, together with a double-ended queue structure, gives priority to files over directories and provides $O(1)$ efficiency for insert with a negligible cost of increased memory overhead.

In addition, after profiling file systems with *fprof*, we found that sparse files can be sufficiently pervasive to affect *fprof's* usage accounting and therefore needed to be taken into account. Detecting sparse files is not trivial and there exist three major approaches: (1) using the FILEMAP directive of the `ioctl(2)` operation; (2) using the SEEK_HOLE directive of the `lseek(2)` function, and (3) using the st_blocks and st_size directives of the `stat(2)` function. Although the first two approaches can efficiently detect holes, they are not supported uniformly across file systems. For instance, the SEEK_HOLE method is not supported with ext4 and NFS file systems in Linux kernels older than version 3.0; FUSE didn't support the feature until Linux 4.5. In our current implementation, we compare the file size with the st_blocks value from the `stat(2)` system call, which specifies the number of 512B blocks allocated. We have found that the `stat(2)` based approach is the most portable and reliable across various system environments. The `stat(2)` approach has its own shortcoming, however, in that it identified as sparse those files that were merely compressed by the ZFS file system that underlies LC's Lustre implementations.

2.3 Deployment

Even though *fprof* is designed to run with multiple processes on multiple nodes to scale, there exist practical concerns and constraints for deploying and running on a production system. For instance, due to the architecture of centralized metadata management in Lustre [22], excessive metadata scanning operations might adversely impact the foreground file system operations. To this end, OLCF ran *fprof* using a single client node for profiling the Lustre-based Spider II file system, while at LC, *fprof* was run on multiple nodes, resulting in a significant performance improvement. Caution should be taken with either model to avoid exhausting resources on the client or metadata nodes. For instance, running *fprof* on a single host caused the Lustre client to crash with an *Out Of Memory* (OOM)

[1] *fprof* is open source and publicly available at https://github.com/olcf/pcircle.

	OLCF (atlas1 & atlas2)	LC (lscratche)
File system	Lustre	Lustre
Back-end local file system	ldiskfs	ZFS
Capacity	32 PB	5.7 PB
File count	0.92 billion	1.3 billion
Directory count	115 million	45 million
Hard link count	4,390,426	309,219
Symbolic link count	7,951,784	10,430,723
Sparse file count	3,240,848	824,561,829
Max # of files in a directory	6,006,529	26,646,573
Average file size	27.67 MB	2.83 MB
Largest files (top N)	32 TB	12.77 TB

Table 1: A summary of two target parallel file systems. The usage patterns are notably different between the two HPC centers.

failure. We found that in Lustre the number of client-side locks to control the *Least Recently Used* (LRU) cache size is set to "unlimited" by default, leading to memory exhaustion when scanning billions of entries in the file system. Fortunately, this can be easily avoided by manually limiting client LRU caches with the following command: `lctl set_param ldlm.namespaces.*.lru_size=2000`.

3 PROFILING AND ANALYSIS

We ran *fprof* on the OLCF's center-wide Spider II file system[22] and the lscratche file system in LC [6] in May 2017. Note the difference in file system architectures of the two HPC centers outlined in Table 1. The Spider II file system in OLCF consists of two symmetric 16 PB Lustre file systems, atlas1 and atlas2, that are built upon Object Storage Targets (OSTs) formatted with the *ldiskfs* filesystem. In contrast, LC hosts multiple classified and unclassified Lustre file systems using OSTs formatted with the *ZFS* file system [9]. The lscratche file system, used in this study, is one of the unclassified Lustre file systems providing a 5.7 PB name space (Table 1).

3.1 File System Snapshot and Characterization

Table 1 summarizes the basic profiling results with *fprof* from two target file systems. One can clearly see that the two file systems exhibit significantly different workload patterns. Specifically, the LC file system has approximately 41% more files despite having a capacity only 18% of that in OLCF. In contrast, the OLCF file system has two and a half times the number of directories and an average file size that is nearly ten times greater than on lscratche. In addition, a single directory in the LC file system holds over 26 million entries, four times more than the one in the OLCF file system, as single large directories are generally frowned upon at OLCF. Therefore, we can conclude that applications at LC tend to create many small files in a directory compared to applications at OLCF, which create fewer but larger files. As a note, both centers have a misleadingly high number of directories due to purge policies, which only remove files, leaving in place many empty directories.

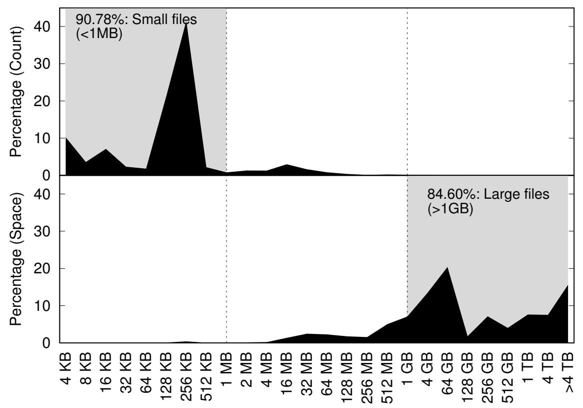
Another notable observation is the stark difference in the number of sparse files on the two targeted file systems. On LC's lscratche, 63% of the regular files are identified as sparse files while only 0.35% of the files on the OLCF file system are sparse. An initial investigation into this disparity suggested that particular libraries or applications created such files. However, this could not account for the large number of other, seemingly random, files that were characterized as sparse. Further investigation led to a very different culprit: compression. LC enables ZFS compression on all the Lustre OSTs, meaning all file objects will be compressed when possible. ZFS aggressively compresses empty spaces in files, and this results in a mismatch in allocated blocks versus the apparent file size.

To verify this effect, we created two 1000 MB files in the ZFS-backed Lustre file system with the *dd* utility. The first file (*randjunk*) was created using random input from */dev/urandom* while the second (*zerojunk*) was filled with zeros from */dev/zero*, which makes the file highly compressible. Using the `stat(2)` system call, we then observed the file size (st_size) and the number of allocated blocks (st_blocks) for each file. Although both files exhibited the same file size (1,048,576,000 bytes), the number of allocated blocks was different. Specifically, ZFS could compress the *zerojunk* file all the way down to a single 512 B block, whereas *randjunk*, which was essentially non-compressible, consumed 2,049,393 blocks. In such a case as the *zerojunk* file that is heavily compressed, the allocated blocks appear much lower than the size suggests, resulting in the file being falsely labeled sparse. Further development of the *fprof* utility will allow the option to search for holes in the file to more accurately determine sparseness.

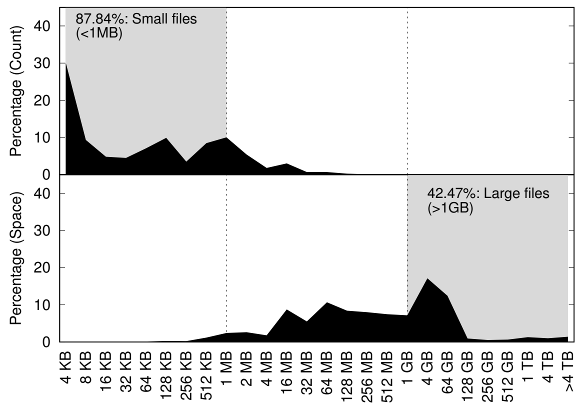
Next, we further analyze the file size distributions in the two target file systems. Figure 2(a) and (b) depict the file count and space occupancy per file size for both the OLCF and LC file systems, respectively. Here, we consider files smaller than 1MB as small and files greater than 1GB as large. We clearly observe that small files dominate the file count in both file systems, i.e., over 90% and 87% in the OLCF and LC file systems, respectively. However, the two file systems exhibit different trends in the space occupancy distribution. Specifically, in the OLCF file system, 84.6% of the total capacity is occupied by the large files that are greater than 1GB, while only 42.5% are occupied by such files in the LC file system. As we see in Figure 2(b), many files in the LC file system are positioned in the middle range, i.e., from 1MB to 1GB. The different profiling results in the two target file systems suggest that each center exhibits different file system workloads and has distinct file system requirements. Such insights from the profiling can be a valuable guidance in designing and planning future file systems.

3.2 Striping Pattern Analysis

Striping is a mechanism for a parallel file system to distribute data across multiple storage targets to improve parallel performance. For Lustre, the stripe count has been a perennial cause of performance issues and an added burden for the end user [5, 13]. Prior to Lustre 2.10, a stripe count must be specified upon the creation of a file (or directly inherited from the parent directory) and can't change afterward unless explicitly restriped, an expensive copy operation of the entire file. At OLCF, the default stripe count is set as four. The performance implication is that the file's data will be served from four OSTs, even for very large files, if the default stripping pattern is used. This limits per-file performance unless a larger stripe count is specified by the user. LC, however, uses a default stripe count of one on its production file systems, writing files to a single OST. This setting is justified by the overwhelmingly small file makeup



(a) OLCF



(b) LC

Figure 2: File size distribution in OLCF and LC.

of LC file systems: nearly 88% of files on lscratche are less than or equal to 1 MB in size, the size of one stripe.

With *fprof*, we have implemented striping analysis which takes a threshold on file size and examines striping information for all files above it. On April 21st and 25th of 2017, we ran a stripe analysis on OLCF's center-wide file system, gathering and combining stripe information for all files at and over 4 GB in size. We set this threshold for two reasons. First, reading the stripe can be an expensive operation, and we want to minimize the overall performance impact on a live production system. Second, we believe that 4 GB and above is a reasonable line to draw - applications *should* pay attention to the data layout once this threshold is crossed. Figure 3 summarizes the results using the scatter plot of file size against stripe width. Each data point is further visually enhanced by the *size* component, where the larger dots indicate a larger file size.

All together we obtained over 513,740 data points allowing us to make the following observations. On both file systems (atlas1 and atlas2), each supporting a maximum of 1,008 OSTs (or stripes), only 21 distinct stripe counts are in use. An overwhelming majority of the files (96.83%) stayed with the default setting. That is, only 3.17% of the files make use of a different stripe pattern. Among those with changed file layout, 2,262 of the files have set the stripe count to 1. Initially we thought these must be test datasets for experimental purposes, but upon close examination, realized they

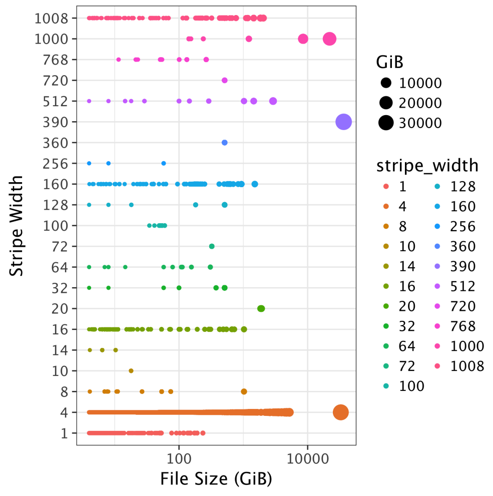


Figure 3: Correlating file size and stripe width on Spider II

are not - the average file size is 14 GB, with a maximum file size of 235 GB. We speculate that applications may perform a 1:1, or file-per-process write. Restricting the stripe width in such cases can avoid contention on OSTs and increase the overall I/O performance.

If we examine the relationship between the file size and its stripe count, as demonstrated in Figure 3, we cannot establish a positive correlation. That is, though there are a few cases where large files are striped larger, we also observe the opposite - where very large files stay with the default. In fact, we see one of the largest files on Spider II, a single 32 TB file, use the default striping of 4. Furthermore, even for users making a conscious choice of changing the file layout, there is not a clear correlation on the file size and choice of stripe width. For example, within each chosen stripe width there are many outliers: any data point 3 interquartile ranges (IQR) below the first quartile or above the third quartile. One extreme case in point: for users or applications using the maximum stripe width of 1,008, the median file size is 4.7 GB while the mean is 26.9 GB. This cannot justify the use of extreme wide stripe count; on the contrary, it most likely hurts the I/O performance.

In summary, our striping analysis shows that the majority of power users at OLCF (applications writing out the largest volumes of data) don't follow the established performance link between stripe count and file size, which suggests that more proactive user outreach and education are needed. In addition, our observation also accentuates the urgent need for deploying recently developed Lustre Progressive File Layout (PFL) [15] into production systems.

3.3 Block Space Analysis of Future File System

For the past decade, OLCF designed and deployed three generations of storage and file systems, all of which are built upon Lustre. However, the upcoming Summit computer [11] will be serviced by a GPFS-based, center-wide Spider III file system. LC's upcoming Sierra system [4] will have a similar GPFS file system. The build proposals for Summit and Sierra suggest employing a large file system block size for better performance. Given the difference between Lustre and GPFS on block management, there have been concerns on how efficient or inefficient disk usage might be. To this end, *fprof* provides a special feature that simulates the *what-if* scenarios for migrating objects in a particular Lustre file system to a fresh GPFS file system, with varying block-sizes. The simulation result helps identify the optimal block size which can maximize the space utilization in the target GPFS file system.

In the following, we first explain unique aspects of GPFS block management compared to Lustre. We then run and simulate three different cases/systems: the first is a dataset from a real-world scientific application (Moment Closure Solver [19]) that is internally used in OLCF to evaluate the storage performance. This represents extreme and ill-fitting cases. In addition, we also conduct the whole system simulation using the snapshots from both OLCF Spider II and LC's lscratche production systems.

3.3.1 Block Management in GPFS. While Lustre delegates the block management to the underlying local disk file system, e.g., ldiskfs or ZFS, GPFS has its own block management with two distinct features relevant to the block usage and its overhead:

Data in inode GPFS supports a *data-in-inode* feature which allows the data portion of small files to be stored in the inode eliminating the I/O needed to read the data segment. It also allows the directory entries of a child directory to be stored in the parent's inode, eliminating the need for a metadata I/O operation for the children directory entries. Each inode has a fixed header of 128 bytes and some optional parts. Assuming the inode does not hold any optional part, then all the space other than the part for the header can be used for data. Thus, for a 512 byte inode, we have 384 bytes of space after the headers for file data, and for a 4,096 byte inode we can fit up to 3,968 bytes of file data.

Sub-blocks For performance reasons, GPFS prefers larger native block sizes. However, larger block sizes can result in wasted space if the block isn't fully used by the file data. To minimize wasted space with large block sizes, GPFS supports the notion of "sub-blocks" - where a native block can be further divided into 32 sub-blocks to improve space efficiency.

The question we sought to answer is, given the scale of storage (e.g., 250 PB usable space in the case of Spider III), what are the trade-offs between increasing the native file system block size (for better performance) and potential wasted capacity? Can we obtain a quantifiable estimate in this early phase to guide the system design?

3.3.2 Comparing Space Utilization. Figure 4 summarizes three runs of the space utilization simulator on (1) an output dataset from a large-scale Moment Closure Solver, (2) the entire OLCF Spider II file system, and (3) LC's entire lscratche file system. All three runs have data-in-inode set to be 4KB, and we vary the file system block size from 256KB to 32MB.

Case 1 presents one of the worst case scenarios: for a dataset has close to 100,000 files, the average file size is only 11.27 KB. It yields only 86.37% efficiency with a 256KB file system block size and plummets to 2.69% with a 32MB block size. Fortunately, our earlier profiling did suggest that even though the majority of the files are small, the majority of the space is consumed by large files. By embedding the estimator logic into the live profiling process, we quantitatively demonstrated that both LC and OLCF production file system fare much better with larger block sizes. In the case of 32MB blocks for a 250PB file system, the wasted space is a little over 5PB or 2%, which is deemed acceptable. Further analysis suggests

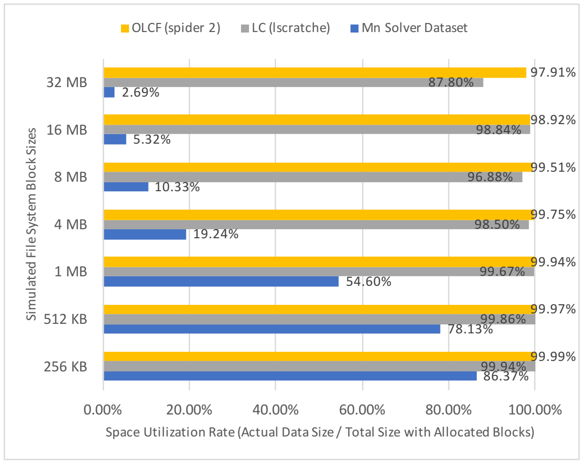


Figure 4: Space consumption comparison: block size vs. efficiency

that there exists a difference between LC and OLCF, particularly at 32MB, where LC's space utilization drops to 87.80%. This number also corroborates the earlier snapshot analysis in § 3.1 where the average file size at LC is smaller than at OLCF. Assuming other things being equal, we claim that a native file system block size of 16 MB would be a better choice and preferred optimization point.

We also note the efficacy of the data-in-inode feature present in GPFS. With this feature turned-on, nearly 60% of the files in the dataset can be stored in their respective inodes without having to use separately allocated data blocks. This drastically improves the overall space utilization.

4 CONCLUSIONS

In this paper, we presented the design and implementation of a lightweight, scalable and portable tool, *fprof*, to efficiently profile large-scale parallel file systems. We also presented three profiling and analysis use cases from two large-scale HPC parallel file systems. Our stripe pattern analysis demonstrated the disconnect between system design and actual user/application behavior and suggested the importance and urgency for features such as the Progress File Layout. Our simulated block analysis not only quantified the space overhead for a complete system migration, but also revealed optimization points for better performance and space trade-offs by employing different native file system block sizes.

Currently, *fprof* has been deployed at OLCF to run on a biweekly basis. The profiling results are further streamed into a Splunk server [12] and displayed on a dashboard. LC has *fprof* scheduled to run quarterly. We are hoping that the periodic profiling will help us to gain a deeper understanding of how the large-scale parallel file systems are used in scientific computing centers. Furthermore, such an understanding will provide practical and valuable insights for future system designs.

Acknowledgment

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. The work at LC was performed under the auspices of the DOE by LLNL under Contract DE-AC52-07NA27344.

REFERENCES

[1] *Argonne Leadership Computing Facility.* https://www.alcf.anl.gov.

[2] *General Parallel File System welcome page - IBM.* https://www.ibm.com/support/knowledgecenter/en/SSFKCN/gpfs_welcome.html.

[3] *Home Âů cea-hpc/robinhood Wiki Âů GitHub.* https://github.com/cea-hpc/robinhood/wiki.

[4] *Livermore Computing - Sierra.* https://hpc.llnl.gov/hardware/platforms/sierra.

[5] *Lustre Basics.* https://www.olcf.ornl.gov/kb_articles/lustre-basics/.

[6] *Lustre Parallel File System | High Performance Computing.* https://hpc.llnl.gov/hardware/lustre-parallel-file-system.

[7] *Lustre* Software Release 2.x.* http://doc.lustre.org/lustre_manual.xhtml.

[8] *National Energy Research Scientific Computing Center.* http://www.nersc.gov.

[9] *Native ZFS for Linux.* http://zfsonlinux.org/zfs-disclaimer.html.

[10] *Oak Ridge Leadership Computing Facility.* https://www.olcf.ornl.gov.

[11] *Oak Ridge Leadership Computing Facility - Summit.* https://www.olcf.ornl.gov/summit/.

[12] *Operational Intelligence, Log Management, Application Management, Enterprise Security and Compliance | Splunk.* https://www.splunk.com.

[13] *Optimizing I/O performance on the Lustre file system - nersc.* http://www.nersc.gov/users/storage-and-file-systems/i-o-resources-for-scientific-applications/optimizing-io-performance-for-lustre/.

[14] *Portal | High Performance Computing - Livermore Computing.* https://hpc.llnl.gov/portal.

[15] *Progressive File Layouts - Lustre Wiki.* http://wiki.lustre.org/Progressive_File_Layouts.

[16] Nitin Agrawal, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2009. Generating realistic impressions for file-system benchmarking. *ACM Transactions on Storage (TOS)* 5, 4 (2009), 16.

[17] Nitin Agrawal, William J Bolosky, John R Douceur, and Jacob R Lorch. 2007. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)* 3, 3 (2007), 9.

[18] Wan Fokkink. 2013. *Distributed Algorithms: An Intuitive Approach.* MIT Press.

[19] C Kristopher Garrett, Cory Hauck, and Judith Hill. 2015. Optimization and Large Scale Computation of an Entropy-Based Moment Closure. *J. Comput. Phys.* 302 (2015), 573–590.

[20] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2012. A file is not a file: understanding the I/O behavior of Apple desktop applications. *ACM Transactions on Computer Systems (TOCS)* 30, 3 (2012), 10.

[21] J. LaFon, S. Misra, and J. Bringhurst. 2012. On distributed file tree walk of parallel file systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC).* 1–11.

[22] Sarp Oral, James Simmons, Jason Hill, Dustin Leverman, Feiyi Wang, Matt Ezell, Ross Miller, Douglas Fuller, Raghul Gunasekaran, Youngjae Kim, Saurabh Gupta, Devesh Tiwari, Sudharshan S. Vazhkudai, James H. Rogers, David Dillow, Galen M. Shipman, and Arthur S. Bland. 2014. Best Practices and Lessons Learned from Deploying and Operating Large-scale Data-centric Parallel File Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14).*

[23] Feiyi Wang, Verónica G Vergara Larrea, Dustin Leverman, and Sarp Oral. 2016. FCP: A Fast and Scalable Data Copy Tool for High Performance Parallel File Systems. In *The 38th Cray User Group (CUG '16).*

[24] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.