Characterizing Output Bottlenecks of a Production Supercomputer: Analysis and Implications

BING XIE, SARP ORAL, CHRISTOPHER ZIMMER, and JONG YOUL CHOI,
Oak Ridge National Laboratory
DAVID DILLOW, dave@thedillows.org
SCOTT KLASKY, Oak Ridge National Laboratory
JAY LOFSTEAD, Sandia National Laboratories
NORBERT PODHORSZKI, Oak Ridge National Laboratory
JEFFREY S. CHASE, Duke University

This article studies the I/O write behaviors of the Titan supercomputer and its Lustre parallel file stores under production load. The results can inform the design, deployment, and configuration of file systems along with the design of I/O software in the application, operating system, and adaptive I/O libraries.

We propose a statistical benchmarking methodology to measure write performance across I/O configurations, hardware settings, and system conditions. Moreover, we introduce two relative measures to quantify the write-performance behaviors of hardware components under production load. In addition to designing experiments and benchmarking on Titan, we verify the experimental results on one real application and one real application I/O kernel, XGC and HACC IO, respectively. These two are representative and widely used to address the typical I/O behaviors of applications.

In summary, we find that Titan's I/O system is variable across the machine at fine time scales. This variability has two major implications. First, stragglers lessen the benefit of coupled I/O parallelism (striping). Peak median output bandwidths are obtained with parallel writes to many independent files, with no striping or write sharing of files across clients (compute nodes). I/O parallelism is most effective when the application—or its I/O libraries—distributes the I/O load so that each target stores files for multiple clients and each client writes files on multiple targets in a balanced way with minimal contention. Second, our results suggest that the potential benefit of dynamic adaptation is limited. In particular, it is not fruitful to attempt to identify "good locations" in the machine or in the file system: component performance is driven by transient load conditions and past performance is not a useful predictor of future performance. For example, we do not observe diurnal load patterns that are predictable.

Bing Xie conducted much of this research as a graduate student at Duke University, with support from Duke University and also from the U.S. National Science Foundation under Grant No. CNS-1245997. The work used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract No. DE-AC05-00OR22725. The work also used resources of Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract No. DE-NA0003525.

Authors' addresses: B. Xie, S. Oral, C. Zimmer, J. Y. Choi, S. Klasky, and N. Podhorszki, Oak Ridge National Laboratory, 1 Bethel Valley Rd, Oak Ridge, TN 37830; email: {xieb, oralhs, zimmercj, choij, klasky, pnorbert}@ornl.gov; D. Dillow; email: dave@thedillows.org; J. Lofstead, Sandia National Laboratories, 1515 Eubank SE, Albuquerque, NM 87123; email: gflofst@sandia.gov; J. S. Chase, Duke University, D306 Levine Science Research Center, Durham, NC 27708; email: chase@cs.duke.edu.

2020. 1553-3077/2020/01-ART26 $15.00
https://doi.org/10.1145/3335205

CCS Concepts: • **General and reference → Performance**; • **Computer systems organization → Grid computing**; **Secondary storage organization**;

Additional Key Words and Phrases: High-performance computing, file systems, benchmarking

ACM Reference format:
Bing Xie, Sarp Oral, Christopher Zimmer, Jong Youl Choi, David Dillow, Scott Klasky, Jay Lofstead, Norbert Podhorszki, and Jeffrey S. Chase. 2020. Characterizing Output Bottlenecks of a Production Supercomputer: Analysis and Implications. *ACM Trans. Storage* 15, 4, Article 26 (January 2020), 39 pages.
https://doi.org/10.1145/3335205

1 INTRODUCTION

Output performance is crucial to harnessing the computational power of supercomputers. High-Performance Computing (HPC) applications [4–6, 14] run on the scale of hundreds of thousands of compute cores and produce petascale output bursts for intermediate results, checkpointing, and/or restart files (defensive I/O). If the I/O system does not absorb the output fast enough, then memory to buffer the output is exhausted, forcing the computation to stall before it can output more data. Output stalls leave precious CPU resources underutilized, extend application runtimes, and compromise system throughput. Output stalls are often observed in practice, even with asynchronous writes.

One way to reduce output stalls is to add more memory and disk spindles. However, these hardware resources are expensive and supercomputers are designed with a careful balance of I/O and computational capabilities. By the classical Amdahl rule, a balanced petaflop facility requires 128 TB/s of I/O bandwidth. Technology planning for cost-effective deployments has used a more austere baseline of 2 TB/s per petaflop [25], and some systems are designed with even lower ratios.

As a result, output bandwidth is a precious resource in supercomputers. Trends suggest that this limitation is not likely to change. Therefore, it is crucial for software to make efficient use of bandwidth. In principle, large write bursts can stream effectively and achieve full bandwidth. In practice, delivered bandwidth is highly sensitive to the application's use of storage APIs and its data layout, placing an unwelcome burden on domain scientists to manage I/O performance trade-offs at the application level.

The output bottleneck has motivated the development of adaptive I/O libraries such as ADIOS [19–21, 33] to present a group of easy-to-use APIs to applications and adapt their I/O patterns to the underlying storage system. ADIOS allows users to choose various file formats, to stripe bursts across a chosen set of storage devices, and/or to aggregate bursts to a chosen set of compute nodes with specific burst sizes. The design of effective I/O libraries depends on a deep understanding of file system behaviors.

In this article, we characterize the output behaviors of a production supercomputer called Titan, the 9th fastest supercomputer in the world. Specifically, we take a benchmarking approach by using a synthetic I/O benchmark to produce various output patterns and then to stress the target file systems—Spider in 2013 and Spider 2 in 2015, 2017, and 2018, both of which are based on the Lustre parallel file system [36]. There are three major challenges for benchmarking I/O performance in a production supercomputer:

- Crosstalk interference from other workloads competing for the shared resources on the write path;
- For end users, no monitoring data from intermediate stages of the write path;
- From the view of end users and I/O libraries, no control over the task placements.

To address these challenges, we conduct studies using a *statistical benchmarking* methodology. The results show that statistical benchmarking can extract a performance "signal" from noise and randomness of a shared file system in live use. It enables us to characterize the behaviors of individual stages of the target file system across competing loads, task placements, and times. We present results from experiments that focus output bursts on specific stages of the write path and capture the distributions of performance behaviors that result, and assess the impact of key configuration parameters and choices. By studying the results through sequences of such experiments, we characterize the behaviors of individual components in the write path over time.

The key contribution of our study is to enhance the understanding of performance behaviors for state-of-the art parallel file system software as currently deployed in a production leadership-class facility. The resulting insights can inform design and deployment choices for supercomputing facilities, policies, and configuration choices at the file-system level and technical choices for the ongoing development of integrated software stacks for parallel storage, including parallel file systems and I/O libraries such as ADIOS. It is observed that delivered I/O bandwidth on Titan improved by 30% after a change from round-robin selection of I/O routers [9] to fine-grained load-balancing among nearby I/O routers [11] (see Section 2.6). Insights from statistical performance studies can help to expose such opportunities. For example, Section 5.2 shows how our methodology exposed a persistent load imbalance among InfiniBand ports.

Although some factors may be unique to Titan, to the Spider and Spider 2 file systems, or to Lustre, we expect that our observations are representative of leadership-class computing facilities, and I/O performance problems are common. ADIOS implements a variety of techniques to improve output performance, and many applications now use ADIOS, e.g., S3D [4], XGC [16], and M8 earthquake simulation [8]. For example, ADIOS enables applications to configure their output buffer size. It can issue writes to multiple independent files to avoid performance problems associated with write-shared files and striping, and it reorganizes output data for better read performance. The results in this study provide a foundation to understand and quantify the impacts of these techniques.

Relationship to our previous studies. This article is an extension of our earlier studies [39, 40]. Specifically, [39] characterizes the output performance on the Jaguar supercomputer, a predecessor to Titan. Compared with [39], this work extends the statistical benchmarking methodology developed in that study, repeats many of the original experiments (Section 4), and verifies our benchmarking results on one real application and one real application I/O kernel (Section 6), XGC and HACC IO, respectively. These two are widely used in I/O performance studies, as they represent I/O behaviors of HPC applications. Titan scales Jaguar's computational power by a factor of 10, along with hardware and software upgrades to the I/O system, including the Lustre file systems Spider and Spider 2 (Table 2) used in this study. The results (Section 4) show that the key output behaviors observed on Jaguar continue to hold on Titan, with a few exceptions. Notably, the delivered bandwidth of Lustre striping has almost doubled (Section 4.4) as a result of software upgrades that improve threading in the client. Even so, the Lustre write pipelines on Titan do not allow a compute node to obtain the full bandwidth of a storage target, and locks on write-shared files inject bubbles into the pipeline. These results motivate I/O libraries (e.g., ADIOS) to structure output so that each compute node writes multiple independent files.

The other previous work [40] serves as the excerpt of this article by summarizing some key aspects of the methodology and results. Specifically, it addresses the benchmarking results from 2 of the 6 experiments discussed in Section 4 and includes the profiling results from 2 of the 5 discussions in Section 5.

In addition, this article extends the Jaguar study in a new direction: it evaluates variations in the observed performance of individual components over time (Section 5). The original experiments showed the effect of "straggler" storage targets and their impact on striping performance in Jaguar and now in Titan. Stragglers throttle the tightly coupled write pipelines, limiting striping bandwidth and reducing the benefits of parallelism. The results lead to this question: Can I/O middleware improve output performance by adapting to hot spots in the machine to avoid stragglers? To answer this question, we conducted longitudinal experiments to probe observed performance systematically over time. We find that a small proportion of storage targets (<20%) are straggling at any given interval, but that stragglers are transient: over time, any target may appear as a straggler for some intervals. While the I/O performance delivered on Titan is highly variable, our study suggests that historical performance data and monitoring do not enable adaptive middleware to locate "good spots" in the machine. Local performance behavior is transient and unpredictable.

Finally, we extend the experiments to study observed variability in the I/O performance of compute nodes. In the Jaguar study, we find that transient I/O contention occurs frequently in its I/O system; in this work, we conduct more analysis and conclude that the contention exists within Titan rather than on the SION network or the Spider file system (discussed in Section 4.2). Delivered aggregate output bandwidth is sensitive to location (density) of a job's compute nodes for large bursts under a static node-to-router mapping policy adopted by Titan in its internal network configuration.

2 BACKGROUND AND MOTIVATION

2.1 I/O Cost of Scientific Applications

It is estimated that 50%–60% of I/O operations in HPC are writes [2, 12]; write-heavy workloads result from state snapshots and defensive writes (e.g., restart files). The objective of this study is to characterize the rate at which the storage system can absorb client output. To motivate the study, we introduce a simple bounded buffer model of the impact of output burst absorption on application performance. For the purpose of the model, suppose that the system can absorb output at a constant rate S. In practice, S varies according to system parameters, output sizes and patterns, data layout, hardware capabilities and status, failures, and competing traffic, to name a few key factors.

Consider a program that runs as P processes/threads on P cores and executes a sequence of iterations in a loosely synchronous fashion, in which all cores alternately compute and output data. To simplify the model, suppose that computational load and output are evenly balanced across the processes and iterations. In each iteration, the program computes for time C and then outputs a burst of size W. If the writes are synchronous, then the application stalls for time W/S to wait for a burst to complete after each C time unit of computation.

For asynchronous bursts, we further suppose that the program has at its disposal M bytes of client-side I/O buffer space (I/O cache) to store its output burst until the storage system can absorb it. Each process buffers its output from an iteration by copying it into the local I/O cache; if there is insufficient space in the cache, then the process blocks until space is available (an *output stall*). The next iteration begins after the previous output burst enters the cache. Each client node pushes its buffered output asynchronously to the storage targets to overlap the writes to storage with computation.

Given our idealized assumptions, we may express the parameters M, W, C, and S as per core, per node, or globally across the entire job or the entire system.

Real systems and applications may deviate from this simple idealized model in various ways. For example, the system may delay writes and/or M may vary according to the client strategies

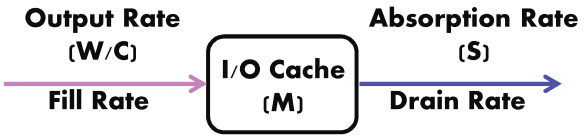


Fig. 1. **The I/O burst model.** An idealized iterative application computes for time C and produces an output burst of size W on each iteration. Each burst enters a client-side I/O cache (write buffer) of size M. The storage system absorbs output at rate S, which varies according to various factors.

for managing the I/O cache. Even so, the model is useful to guide our intuition and estimate the impact of S on output stalls and on application performance. In particular, output stalls limit *core efficiency*, which is the maximum rate x of useful computation normalized to the system's peak; equivalently, x is an upper bound on mean CPU utilization. With core efficiency x, the computation stalls at rate $1 - x$, and the runtime increases by a factor of $1/x$ from the ideal.

Figure 1 illustrates the model. The program produces output into a bounded buffer of size M at an average rate $\frac{W}{C}$, not counting any output stall time. The storage system drains data out of the cache at rate S. There are two cases to consider:

Case 1. $\frac{W}{C} <= S$. The program produces data at an average rate that is slower than the drain rate. If $W <= M$, then the program does not stall. Otherwise, if $W > M$, the program fills the buffer and then stalls until the rest of the burst is absorbed. The stall time per iteration is $\frac{W-M}{S}$.

Case 2. $\frac{W}{C} > S$. The program produces data at an average rate that is faster than storage can absorb it. Once the buffer fills, the program reaches a steady-state behavior in which it stalls $\frac{W-SC}{S}$ per round to throttle the output rate to match the drain rate. SC is the data drained from the cache during the compute time. $W - SC$ is the residual that must be absorbed before the next iteration can begin. The time to reach the steady state is proportional to M. Once the program reaches steady state, the value of M no longer matters because the buffer is full at the end of each iteration, and $SC < M$; else, we are in Case 1.

It is easy to see that C is immaterial and the maximum compute efficiency of the system in steady state is given by the ratio of the drain rate to the fill rate. Suppose that we normalize the fill rate to one unit of output data per second. In Case 2, the normalized drain rate is some s with $0 < s < 1$. Then, the program produces an excess of $1 - s$ units of output data per second of useful computation, requiring an output stall of $(1 - s)/s$ per second of useful computation to absorb the excess output. The efficiency x is then the compute time (1) over the total time $(1 + (1 - s)/s)$. Thus, $x = s$. In the synchronous case, the cores stall for 1 s per s seconds of computation time on average; thus, $x = s/(s + 1)$.

2.2 Application Examples

We describe the XGC [16, 41] code as an example. XGC is a gyrokinetic particle-in-cell code used to simulate tokamak fusion reactor designs, focusing on the multi-scale physics at the edge of the fusion plasma. An XGC run is a simulation for a given 3D tokamak—a magnetically confined torus—that is first decomposed to D poloidal planes with E particles in a plane, and then each plane is partitioned to P subspaces, each containing E/P particles.

A run consists of P identical processes on P cores. Each process solves a fixed set of gyrokinetic equations across particles in one of the P subspaces over a sequence of iterations. In each iteration, particles are "pushed" by a governing gyrokinetic Hamiltonian's equation and gathered onto F grid points of a discrete grid, where the gyrokinetic Poisson or gyrokinetic Maxwell's equations are solved. Periodically each process snapshots particle distributions and features (e.g., electric potential, plasma density, temperature) for its subspace. In addition, D of the processes—one for

Table 1. Write Patterns on Titan

Application Name	Write-share Mode	# Write Types	Burst Size
GTC	0	7	100 B — 1 G
S3D	0	3	100 B — 10 KB
dcaqmcMPI	0	5	100 B — 1 MB
OMEN	1	3	100 B — 10 KB
LMP	1	2	100 B — 1 KB
HACC	1	3	4 MB — 512 MB

Output burst sizes and the number of output burst types for applications in common production use on Titan. The *write-share mode* indicates whether processes share files (write-share is true) or if each process writes to an independent file (see Section 4.6).

each of the D planes—produces periodic bursts of diagnostic output. The frequency of each burst (iterations per burst) is configurable. The data for each burst is stored as a file with a unique file name.

XGC writes its bursts synchronously, and users usually choose to stall the execution until all data reach disks. The computation time for each iteration is fixed and predictable from (D, E, F, P). The burst sizes are also predictable: the total size of each round of subspace snapshots scales with the number of particles, and the aggregate size of the per-plane diagnostic bursts scales with the number of grid points F. The aggregate bursts are approximately balanced across the P (or D) processes generating the bursts. We observed in practice that, for XGC runs, common per-core burst sizes of state snapshots range from 500 MB to 1.2 GB; for diagnosis outputs, the burst sizes range from 1 MB to 400 MB. Scientists use estimates of the absorption rates to select burst frequencies that maintain CPU efficiency around 90%. Our work can help to select burst frequencies precisely.

Table 1 summarizes the write patterns of six other applications in regular use on Titan, as reported in logs recorded by the Darshan [1] tool. For each application, we report the number of burst types and a range of burst sizes. Darshan also records a write-share mode (see Section 4.6): if processes write-share a single file, the mode is "1"; otherwise, each process writes to an independent file and the write-share mode is "0."

2.3 Lustre on Titan

The Lustre software deployed on Titan's I/O system is a widely used open-source parallel file system. Lustre runs on ~75% of the top 100 disclosed supercomputers. This section introduces terms and concepts used in the performance study, summarizing from [36] and other sources on Lustre and Titan.

Lustre is an object-based file system: the data in each Lustre file resides in one or more objects. An *object* is a variable-length sequence of bytes with a unique name. Each object is part of exactly one file and resides on exactly one storage node. A storage node is a RAID array (*target*) that is directly attached to a Lustre storage server. Lustre clients are compute nodes that access the storage servers over a network. A Lustre *metadata service* manages the file name space, file attributes, mappings of files onto objects, and locking for shared access to objects by multiple clients. This article measures write bandwidths to files only after they are open; thus, the metadata service affects the results only for the write-shared file tests (see Figure 20).

Each Lustre write originates with a system call from a user process on a compute node. Each Titan compute node is a multi-core node running a Linux operating system, which maintains the node's file cache as its output buffer. The compute node kernel invokes a local Lustre kernel module called an Object Storage Client or **OSC** to handle file operations and I/O on Lustre file systems.

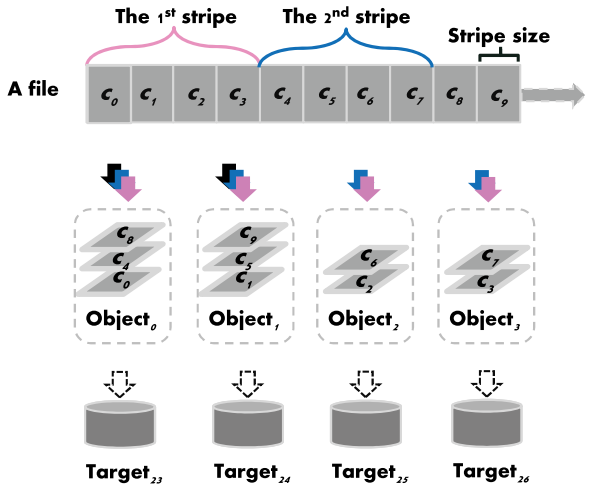


Fig. 2. **Data striping in Lustre.** Each file is a sequence of chunks of *stripe_size* bytes each, distributed in a round-robin fashion across a fixed set of *stripe_width* objects created for the file on sequentially ranked targets (OSTs).

The OSC performs I/O by issuing Lustre remote procedure calls (RPCs) to storage servers; each storage server runs a Linux system with a Lustre service called an Object Storage Service or **OSS**. Each Lustre I/O operation to an OSS is a read or write on exactly one object, which resides on a named storage target attached to that OSS. The Lustre targets are known as Object Storage Targets or **OSTs**. We use the terms *client*, *server*, and *target* to refer to the OSC, OSS, and OST, respectively.

2.4 Lustre Data Striping

A file is a sequence of bytes or fixed-size logical blocks. In Lustre, the blocks of each file are distributed across a fixed set of N storage objects, determined when the file is created. The objects in the set are numbered in a sequence. The current policy assigns the objects in sequence to sequentially numbered targets (OSTs). Lustre allows the user process that creates the file to specify the starting OST; otherwise, the system selects the starting OST at random. The Metadata Server (MDS) creates the object set and stores a list of the file's objects while the file exists. When a client opens a file, it fetches the object list and caches it while the file is open.

The logical file blocks are striped across the file's objects according to a static pattern. Figure 2 depicts an example of this pattern. Sequential runs of blocks are grouped into fixed-size *chunks*, and the chunks are assigned to the objects in a round-robin fashion. The chunk size of a file is called the "stripe size" and the number of objects in the set (N) is called the "stripe count" (*stripe width*). A *stripe* is an aligned sequential run of N chunks.

When a client grows a file by appending bytes to it, it creates and appends new stripes as needed. Each new stripe extends the length of the objects in the object set: the number of objects is fixed for each file, but the objects grow as needed. This policy differs from systems such as Ceph [37], which grows a file by appending new fixed-size objects to it.

We use the *fprof* profiling tool to characterize striping patterns for production files on Titan. We profiled files with ≥ 4 MB file sizes in the Spider 2 file system on Titan (see Table 2) on April 21, 2017 and April 25, 2017, obtaining results for 0.98 billion files and 0.12 billion directories. At that time, roughly 99.6% of the files set $stripe\_count = 4$, the default Lustre configuration of Spider 2. We discuss the write performance with different stripe-count configurations in Section 4.4.

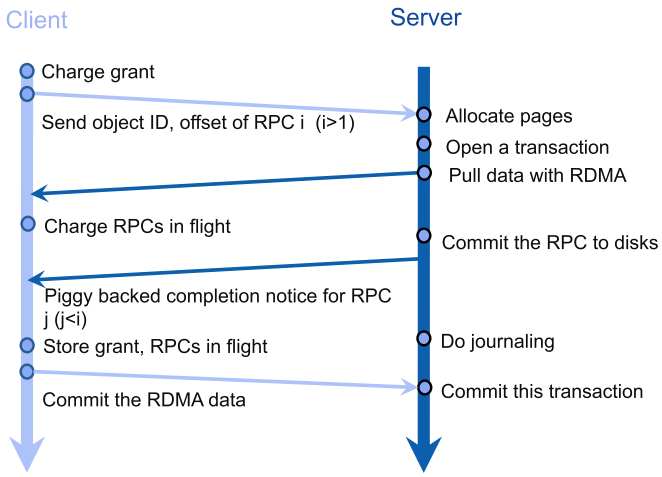


Fig. 3. **RPCs and flow control for asynchronous writes in Lustre.** The client sends an RPC for each write to the object server for the target. The server replies to accept the write and sends a completion notice later when it completes. The client bounds the number of outstanding RPCs and the total size of outstanding writes to each target (a configurable "grant").

2.5 The Write Path

We summarize the write path for ordinary asynchronous I/O. A Linux write system call copies data from the user process into page buffers in the client kernel's I/O cache, blocking (stalling) if the amount of dirty data in the cache exceeds a threshold. The system call path allocates 4 KB pages (blocks) in sequence and aligns the data within the buffers according to their logical offset. We consider only the case in which the process writes (or overwrites) each block in its entirety; we do not consider read/modify/write behavior.

As dirty blocks accumulate in the cache, the client (OSC) groups them into chunk writes. The client targets each dirty block to a specific offset in a specific chunk in a specific object on a specific OST on a specific OSS, according to the striping policy summarized above. The client issues concurrent RPC calls to the servers to write the chunks (or partial chunks) to their targets, releasing the buffers as the writes complete. A write *completes* on a target when the data are safe on disk.

The Lustre OSC module manages concurrent write pipelines to multiple servers and targets. Similarly, each OSS must manage concurrent write pipelines from many clients. A key challenge is to keep all pipelines flowing whenever there are data to transfer, throttling them just enough to prevent any buffer overflow.

Lustre clients and servers coordinate to control dataflow through the pipelines for normal asynchronous writes. For example, on Titan, each client limits the number of RPCs in flight on each target to 8. Each client also limits its pending (incomplete) writes on each target. Figure 3 depicts the steps to generate and process a chunk write request on a given target.

Application processes may request synchronous I/O by opening a file with a specific option flag (O_DIRECT). Direct writes use different mechanisms with less asynchrony [36].

2.6 Titan Interconnects: Gemini and SION

We conducted experiments and real application runs in 2013, 2015, 2017, and 2018 separately on the Spider (2013) and Spider 2 (2015, 2017, and 2018) file stores of Titan (see Table 2). Spider and

Table 2. File Systems on Titan

File Systems	Service Time	Lustre Version	Partitions	Routing Policy	I/O Nodes	OSSes	OSTs	disks per OST
Spider	Jan.2008–Dec.2013	1.8	4	fine-grained	192	192	336 × 4	RAID 8+2
Spider 2	Nov. 2013–present	2.5	2	fine-grained	432	288	1008 × 2	RAID 8+2

A Lustre client (compute node) issues I/O operations to RAID targets (OSTs) attached to Object Storage Servers (OSSs). The I/O path traverses the internal interconnect to a selected I/O node, which acts as a router to forward I/O traffic between the internal interconnect and an external storage network. In Titan the mapping of compute nodes to I/O nodes is static ("fine-grained") when all I/O nodes are functioning normally.

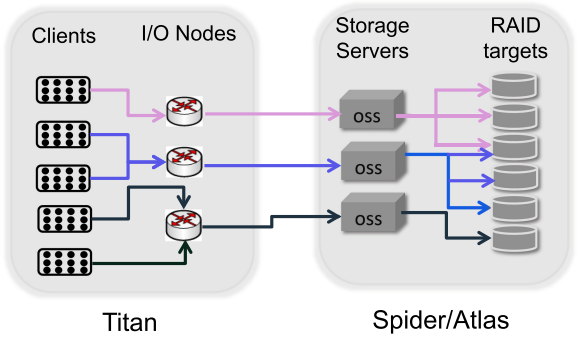


Fig. 4. **Multi-stage write path** in Titan and Spider (or Atlas since 2014). Writes originate in 16-core client nodes; each client issues RPCs to storage servers through the internal Gemini 3D torus interconnect and external SION storage network. The server buffers each write and directs it to an attached RAID target. All writes to a given target pass through a single server.

Spider 2 (Atlas) are center-wide file stores hosting a group of Lustre file systems shared across Titan and other computing facilities in the center. Figure 4 depicts the Lustre write path as it is configured on Titan and its file stores.

Titan comprises 18,688 Cray XK7 nodes linked by a Cray Gemini 3D torus interconnect: a Titan node consists of an AMD 16-core Opteron 6274 processor and an NVIDIA Kepler K20 GPU. Each pair of compute nodes shares a Gemini ASIC with a YARC-2 router and two network interface controllers (NICs), each dedicated to one node in the pair and connecting the node to the Gemini fabric.

The compute nodes (clients) access the storage system via a group of I/O nodes (I/O routers) distributed through the 3D torus interconnect. An external multi-stage InfiniBand (IB) network called SION (Scalable I/O Network) connects the I/O nodes to the storage servers. SION is based on Cisco 7024D IB switches with a fat-tree topology.

Titan configurations bind each compute node to a group of I/O nodes that are "close" to it in the 3D torus; the I/O node for each request is chosen by a fine-grained load-balancing algorithm, which maps compute nodes to I/O nodes in a static way [11]. Section 5.5 discusses the performance impact of this static node-router mapping policy.

In both Spider and Spider 2, each server (OSS) mediates access to 7 RAID storage targets (OSTs). A DataDirect Network (DDN) controller groups the targets and presents them to the server as LUNs; each LUN represents a different target. The targets (LUNs) are sequentially numbered and chosen to map adjacent targets to different servers so that striped accesses spread across multiple servers. Figure 5 presents the OSS to OST (LUN) mapping.

Spider has 4 file system partitions; each partition comprises 48 servers and 336 (48 × 7) targets. Each of the 48 servers connects to 2 IB ports, with each of the 7 managed targets assigned to one

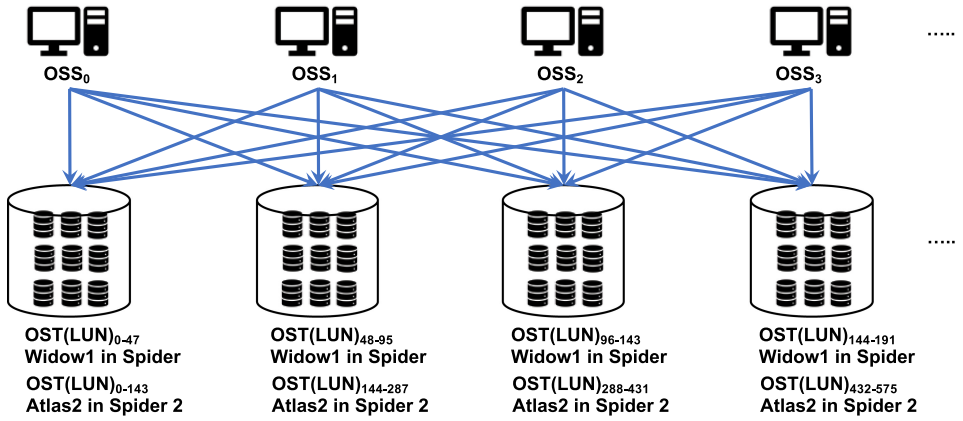


Fig. 5. **OSS to OST (LUN) mapping**. Spider and Spider 2 (see Table 2) follows the same mapping policy: in each partition of Spider or Spider 2, each LUN is a different OST; the 48 or 144 sequentially numbered OSSs connect to the sequence of 336 or 1,008 OSTs in a round-robin way. For example, in Spider/Widow1, OSS_0 connects to 7 OSTs: OST_0, $OST_{48}, \ldots, OST_{240}$, OST_{288}.

of the two IB ports. As a result, the load on these I/O channels may be unbalanced, as one of the IB ports carries traffic for 3×48 of the targets and the other carries traffic for 4×48 targets. Section 5.2 returns to this issue.

The 2013 experiments use the Spider file system Widow1 that spans 48 servers and 336 targets. The 2015 and 2017 experiments and 2018 application runs use the Spider 2 file system Atlas2, which comprises 144 servers and 1,008 targets.

3 A STATISTICAL BENCHMARKING METHODOLOGY

This section introduces a statistical benchmarking methodology, including two parts: focusing writes on specific stages of the write pipeline and statistical analysis across multiple trials. We use IOR [28], a flexible synthetic benchmarking tool for parallel file systems with various interfaces and access patterns. We run multiple IOR instances within a benchmarking harness to coordinate simultaneous write bursts from multiple processes and nodes, and report delivered bandwidth after all data reach the disks.

Our approach is designed to overcome challenges of benchmarking in a production environment on shared hardware. There is no monitoring in the various stages of the write path; thus, we design the runs to focus traffic on specific stages to measure their performance behaviors. Performance is sensitive to location, but we cannot control the compute nodes for our runs: the system's batch job scheduler chooses them "randomly" for each run. In addition, the runs are subject to interference and noise from competing traffic and other transient system conditions that we cannot foresee or detect [21].

To overcome interference noise, we obtain a distribution of measures across samples of compute nodes, storage targets, time intervals, and other instantaneous system conditions. Each experiment is a set of IOR jobs measuring the impact of a single parameter on delivered bandwidth under some set of conditions. The parameters include the number of compute nodes (N); the number of OSTs (T); parameters to the job script, such as the number of cores per client,; or parameters to the IOR benchmark, such as burst size. Figure 6 depicts the structure of an experiment.

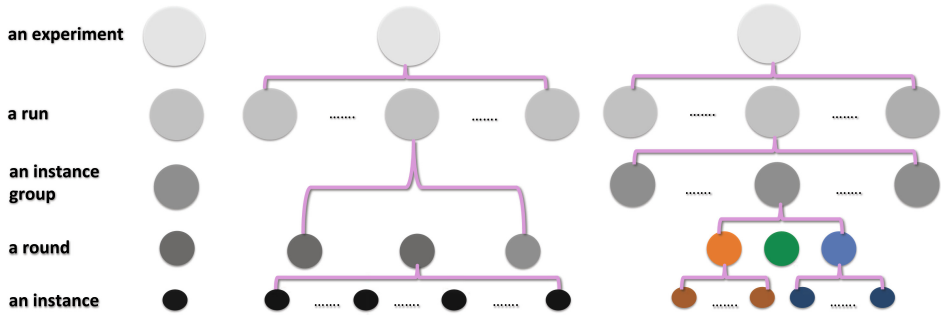


Fig. 6. **Benchmarking hierarchy**. Each experiment produces results of a varying parameter across clients, servers, and targets at different times and associates with a boxplot graph in Section 4.

- Each IOR execution is an *instance* of the experiment across a sample of N compute nodes and T storage targets (OSTs). The N compute nodes are assigned by the system scheduler and the T OSTs are selected as an OST sequence with a randomly chosen starting OST. Thus, in the instance, the IDs of its compute nodes and OSTs are known. Each process in an instance issues a single write burst. A *burst* is a single POSIX *write* system call, or a loop of write system calls if the burst size exceeds the maximum 1 GB for an individual write. The burst from each process is synchronized with the other processes using MPI barriers and is followed by an *fsync*, which blocks until all writes in the burst complete. For each instance, we report the delivered performance for the full burst (Section 4). We also record the transfer start and end times of each node-target pair to evaluate component performance over time (Section 5).
- Instances execute in sequences called *rounds*. Each round varies a single parameter across a range of values, selecting a different value for each instance in the round.
- Some experiments execute a sequence of rounds called an *instance group*, varying the burst size for each round across a sequence of values. This approach enables us to explore the interaction of burst size with other parameters.
- The results for each experiment are gathered from a sequence of runs, varying the selection of client nodes. Each *run* of an experiment is an identical sequence of rounds (or instance groups). Each run is submitted to the scheduler as a job: the job scheduler selects the compute nodes, which are then used throughout the run. Runs are separated in time, e.g., by scheduling delays, which may be hours or days. We observe that traffic on Titan tends to vary little within a run but may vary significantly across runs. A run is discarded if conditions change significantly during the run.

Each experiment produces a set of points, each giving the output bandwidth measured for one instance. The graphs in Section 4 use boxplots to display the quantile distribution and "whiskers" of outliers of sample points. The plots report output bandwidths using four different measures over the same data:

- *Bandwidth* is measured in MB/s per client node.
- *Aggregate Bandwidth*, measured in MB/s, is bandwidth summed across all nodes in an instance.
- *Effective Bandwidth (EB)* is per-node bandwidth normalized to the peak bandwidth achievable from the number of targets written by each node.

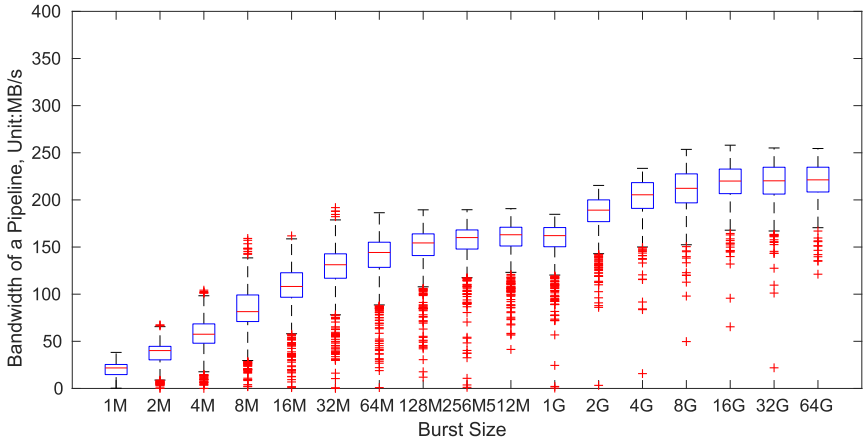


Fig. 7. **Bandwidth of a single pipeline** as a function of burst size. This graph shows results for a single process on a single core writing a single file on a single target. The peak bandwidths obtain around 70% of the peak saturation bandwidth of a target (Figure 9). Other results (not shown) indicate that more client processes from the same node do not help: the configured write pipeline is not deep enough for one client to obtain full bandwidth from a target.

- *Effective Aggregate Bandwidth (EAB)* is aggregate bandwidth normalized to the peak bandwidth achievable from the number of targets written by each instance.

4 OUTPUT ABSORPTION ON TITAN

This section presents the measurements of burst absorption behavior on Titan and Spider (Widow1). The study is useful to understand the performance behaviors of the implementation and to build models that predict output absorption bandwidth as a function of various parameter settings [38].

The data are based on measurements taken from January to December 2013 on Spider/Widow1 (see Table 2). Each experiment has 200 runs with 3 rounds each: it yields a graph with multiple boxplots arranged along an x-axis, with one boxplot for each value of the varied parameter. In a graph, each boxplot contains 600 points associated with 600 instances from 600 rounds (200×3) of 200 runs conducted by using the methodology introduced in Section 3.

4.1 Pipeline Efficiency

The first experiment evaluates the efficiency of the write pipeline from a single client: a single process running on a single core from a single client and writing to a single target (OST), as a function of burst size. Figure 7 gives the results.

Understanding the boxplot graphs. Figure 7 is representative of the box-and-whisker graphs used to report the results of each experiment. The x-axis shows the range of values of the single parameter that varies across the instances of the experiment, as described in Section 3. The boxplot for each x-value reports the distribution of measured output bandwidths, given on the y-axis. The upper and lower borders of each box are the 25th and 75th percentile values (lower quartile Q1 and upper quartile Q3). The band within each box denotes the median value. The value Q3-Q1 is the interquartile range or IQR; thus, 50% of the y-values reside within the box and the IQR is the height of the box. The upper and lower whiskers cover the points outside of the box, except that the upper and lower bounds of the whisker do not extend beyond $Q3 + 1.5 * IQR$ and $Q1 - 1.5 * IQR$,

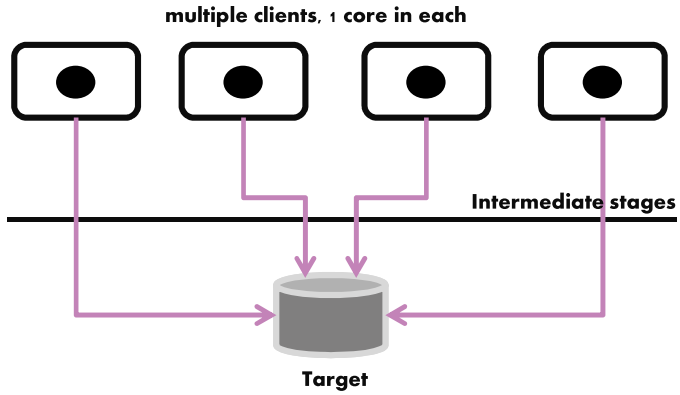


Fig. 8. **Template for target saturation experiment.** Multiple client processes focus simultaneous output bursts (with 64 MB, 256 MB, 1,024 MB burst sizes) on a single OST to measure the peak bandwidth of the OST at saturation. Each process writes a different file on the OST.

respectively. All y-values outside of this whisker range are outliers and are plotted as individual points.

Figure 7 shows that single-pipeline bandwidth is sensitive to burst size. Specifically, for a single pipeline, larger bursts push more data to the pipeline, use the pipeline's I/O resources more efficiently, and deliver higher bandwidths accordingly. The write pipeline obtains its maximum overall bandwidth with a write burst of 2 GB or more. With these burst sizes, the pipeline runs at full bandwidth for long enough to dominate the time to fill and drain the pipeline.

Figure 7 also reports that many of the trials deliver low bandwidths. The results show substantial outliers on the low side (3%–5% of all samples). The next section shows that they are due to intermittent contention on the Titan interconnect. Moreover, we determined from the multi-core experiment (not shown) that using multiple cores on a client does not help. In the multi-core experiment, multiple single-threaded IOR processes run on the same client, each issuing a single output burst to a separate file on the single target, synchronized with MPI barriers. Using multiple cores from a client improves the delivered bandwidth by at most 5%.

Conclusions. The results of single-pipeline experiment suggest that the conservative flow control configuration for output pipelines on Titan (e.g., at most eight outstanding RPCs per client-target pair) prevents a single client from obtaining the full bandwidth of the target. Recent enhancements for asynchronous journaling (see [27]) may delay the RPC replies from the targets, requiring a larger number of outstanding RPCs for effective write streaming.

4.2 Write Bandwidth of an OST

The next experiment uses multiple clients to focus writes on a single target (OST) to measure the peak bandwidths at target saturation: the OST saturation experiment. This experiment follows the template in Figure 8, in which multiple clients (one process per client) write concurrent bursts to the same target, synchronized with MPI barriers. Based on the results, we take 300 MB/s (97.3*th* percentile) as the peak OST bandwidth in practice, although a few trials deliver close to the ideal hardware bandwidth of the targets.

Figure 9 shows the peak target bandwidths for varying numbers of clients and for three different burst sizes. It suggests that the target bandwidth is saturated with 4 to 16 clients and declines when adding more. This kind of bandwidth decline is also observed in other experiments (e.g., the server saturation experiment in Section 4.3 and single-node experiment in Section 4.4). We conclude that

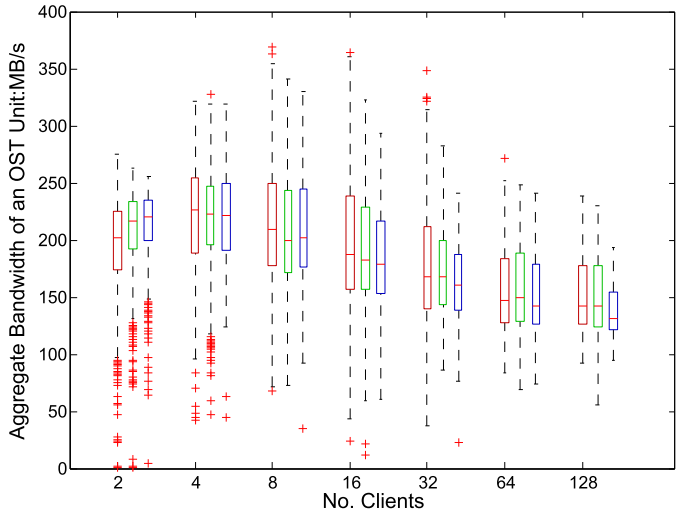


Fig. 9. **Write bandwidth of a target (OST)** as a function of the number of clients. For each fixed number of clients, we plot distributions of the observed target bandwidths for synchronized bursts of 64 MB, 256 MB and 1,024 MB from each client (red, green, and blue boxplots, respectively, from left to right). The results show that bursts from modest numbers of clients yield the highest bandwidths. The low-side outliers decline with larger numbers of clients, suggesting that observed I/O contention occurs within Titan rather than on SION or Spider.

the decline is related to the resource contention at specific I/O stages: after resource saturation, the stages become bottlenecks and, consequently, deliver poorer performance with more I/O load. The decline comes from target-bandwidth saturation in this experiment, server-bandwidth saturation in Section 4.3, and node-bandwidth saturation in Section 4.4.

Figure 9 also shows that the incidence of low-side outliers decreases as we add more clients. We conclude that these outliers result from transient contention on the Gemini interconnect and/or I/O routers rather than on any other stage on the write path (including the SION, server, and target). If the contention appeared in any part of the write path that is shared among all clients of an instance, then it would affect instances with larger numbers of clients as well.

To derive the root cause of the outliers, we compared bandwidths for all clients in this experiment. Figure 10 presents cumulative distribution functions (CDFs)of the bandwidths observed by all individual clients. For small numbers of clients (two and four), the CDFs present long tails: a large number of clients deliver bandwidths well below the fastest client. These low-bandwidth clients correspond to the low-side outlier instances for two and four clients in Figure 9. In fact, for this experiment, all of the clients in these slow instances were slow: in these outlier instances, all clients deliver low bandwidths within a difference of 10%. In general, the target shares bandwidth fairly among the clients. This means that each client's demand on the Gemini interconnect and/or I/O routers is smaller for instances with larger numbers of clients. Thus, congestion in the interconnect has less impact on these larger instances.

We dug deeper to examine the locations of the selected clients in the interconnect. For each outlier instance, the engaged clients are close to each other in the Gemini: 90% of the instances with two clients and 80% of the instances with four clients have the maximum node distances within one and two hops, respectively. These results suggest that the contention may be local to specific regions of the Titan interconnect and/or to a specific set of I/O routers. Although the median observed bandwidth is generally insensitive to burst size in this experiment, the low-side outliers

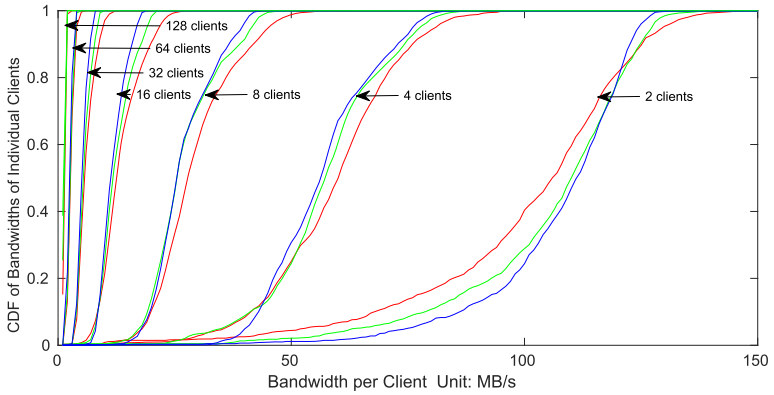


Fig. 10. **Distribution of target bandwidths observed by individual clients** for varying numbers of clients. The data is from the same experiment as Figure 9. The figure shows the CDFs of bandwidths observed by the individual clients with per-client bursts of 64 MB, 256 MB, and 1,024 MB (red, green, and blue, respectively). The variance drops with more clients: a target delivers the write bandwidth fairly to the concurrent clients.

also decrease with larger burst sizes. This effect suggests that the contention of interconnect and I/O routers is transient and of short duration.

In this experiment, the aggregate data size written to the target scales with the number of clients. Figure 9 shows that as the number of clients writing to a target increases, the target delivers a lower share of its potential bandwidth (EAB). This effect is robust across the per-client burst sizes. In this experiment, we chose the burst sizes because Figure 7 motivates large bursts. However, in this experiment, the large bursts do not help: in a few cases, the target delivers even lower bandwidth with multiple clients than it does with a single client using the same burst size. Moreover, the experiment presented later in Figure 20 shows that much smaller bursts are more effective with larger numbers of clients: for example, with 128 clients, the best results are obtained with 1-MB bursts. Clearly, the ideal burst size depends on the number of clients. In particular, large bursts from many clients trigger a thrashing behavior in the targets. Figure 10 shows that the clients observe the slowdown uniformly.

Conclusions. The results suggest that bursts from modest numbers of clients (4–16 clients) yield the highest bandwidths of a single OST. When we increase the numbers of clients to an OST, the low-side outliers decline, suggesting that transient I/O contention occurs within Titan rather than on the SION or Spider file system.

4.3 Write Bandwidth of an OSS

To determine the peak bandwidth of a server (OSS), we conducted an experiment in which groups of clients write evenly across targets of a given server: the OSS saturation experiment. This experiment follows the template in Figure 11: the clients focus synchronized write bursts to the server with aggregate burst sizes of 7 GB, 14 GB, and 28 GB. The files and bursts are distributed round-robin style across the server's targets. Figure 12 plots the observed server bandwidths as a function of the number of clients.

In Figure 12, the measured peak bandwidth of a storage server (OSS) is 1.247 GB/s, around 0.6 EAB of its seven targets. This result is close to the rated hardware bandwidth of the Spider OSS (1.25 GB/s) [31], suggesting that in this experiment, the OSS itself is the bottleneck and is leaving the available disk bandwidth of its targets underutilized.

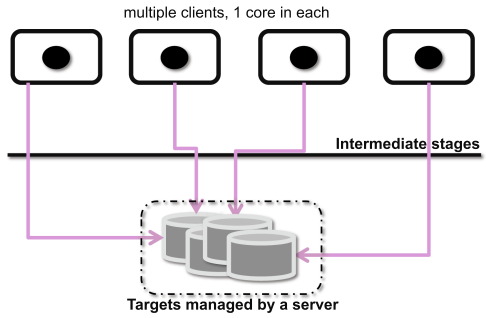


Fig. 11. **Template for server saturation experiment.** Synchronous client processes/cores issue bursts to the sequence of OSTs on a single storage server (OSS) in a round-robin way to measure the peak output bandwidth from the server. Each process writes a different file to a target, sized equally for aggregate bursts of 7 GB, 14 GB, and 28 GB.

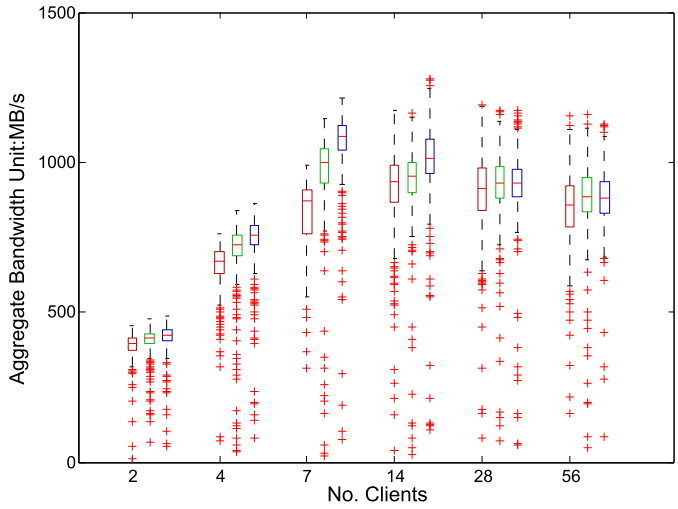


Fig. 12. **Write bandwidth of the storage server** as a function of the number of clients. When x value > 7, all 7 targets on the same server are in use, each target receiving an equal-sized burst from a varying number of clients. For each given number of clients, we plot distributions of the bandwidths of a server with simultaneous output bursts of 7 GB, 14 GB, and 28 GB aggregate burst sizes (red, green, and blue boxplots, respectively, left to right). Large bursts obtain the highest write bandwidths until the server becomes a bottleneck with 28 clients or more, similar to the target behavior at saturation in Figure 9.

Conclusions. The server's bandwidth limitation may be a result of limited buffer capacity or memory bandwidth, or a concurrency limitation within the server.

4.4 Output Bandwidth of a Compute Node

The next experiments probe the output bandwidth observed by a single client writing simultaneously to multiple targets each under a different server. These experiments test *fan out* parallelism in which the client manages concurrent pipelines to multiple targets, stressing the client. We compare two forms of fan-out parallelism: (1) writes to striped files and (2) writes to multiple unstriped files. The purpose is to determine how effectively a client manages the concurrency to keep all of its pipelines full: to stream writes, the client must respond to an incoming RPC reply or

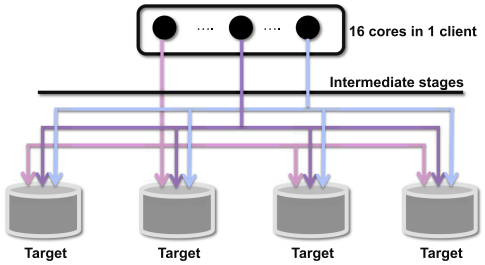


Fig. 13. **Template for striping bandwidth experiment.** We issue 16 concurrent processes (cores) from a single client, with each writing a burst to a separate file. The files are the same size and are striped across the same set of target OSTs (*stripe_count*) at a granularity (*stripe_size*) of 1 MB per OST per stripe, which is ideal. The number of OSTs varies. Figure 14 shows the results.

completion notice by pushing more output into the pipeline to keep it full. Synchronization bottlenecks or internal threading limitations may cause reaction delays, leaving bubbles in the pipeline that reduce delivered bandwidth.

The first experiment follows the template in Figure 13: we issue 16 processes (cores) from a single client, with each writing a burst to one of 16 different files striped across a varying number of storage targets (*stripe_count*; see Section 2.4). Each stripe stores 1 MB of data per target (*stripe_size*). We determined from other experiments (not shown) that the 1 MB per-target (*stripe_size*) and the process count of 16 are ideal. Those experiments show that delivered bandwidth with striping is insensitive to *stripe_size* up to 32 MB per target (the configured grant size in Lustre) and then declines. Moreover, striping delivers the highest bandwidth from the client when all cores are writing [39].

Figure 14 shows the measured bandwidths from this experiment. The maximum bandwidth obtained in this experiment is 1.09 GB/s from a 3.75-GB aggregate burst to 16 targets. All median bandwidths are substantially below 1 GB/s. The result also shows that the delivered bandwidth declines as the client engages more than 16 targets. It suggests that, for a single-client burst, the limitation is on the client side such that the client cannot send enough data to each of the engaged targets. This limitation leaves the concurrent pipelines underutilized and eventually results in a lower aggregate bandwidth. We conclude that good stripe widths on Widow1 are in the range of 8 to 16 targets.

To determine whether the limitation is related to the client's handling of striping, we conducted a similar experiment to measure the peak output bandwidth of the client under similar conditions, but without striping. This experiment follows the template in Figure 15. As in the previous experiment, the client writes synchronized bursts in parallel across a varying number of targets. However, in this experiment, each client process issues its write burst to a file on a single OST, with no striping. Each process writes to a different file on a different OST on a different OSS from the others.

Figure 16 shows the results. The peak bandwidths are obtained with 12 cores and 12 OSTs. The client node obtains higher bandwidth than it does using striping: the client obtains peak bandwidths of more than 1.4 GB/s with all burst sizes.

However, it is worth noting that striping performance has improved substantially since our original 2012 study [39] on the Jaguar machine, prior to the Titan upgrade. The Titan nodes show roughly the same peak output bandwidth as on Jaguar, but the striping bandwidth has almost doubled. This improvement likely results from client software upgrades that introduced a pool of threads to handle RPC load [30].

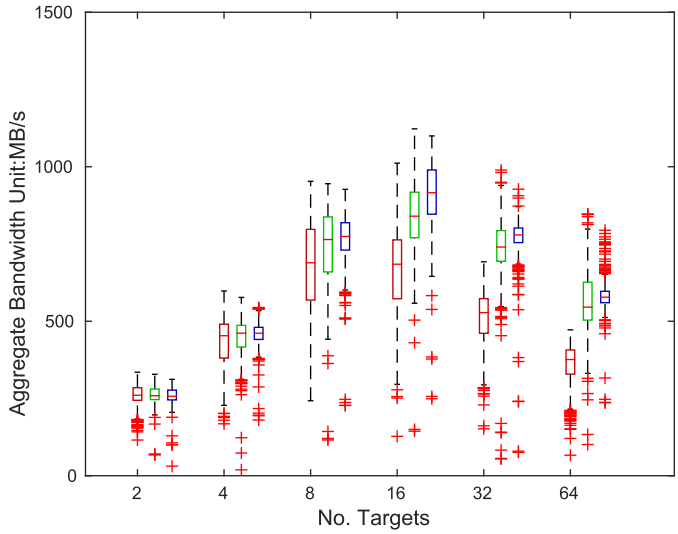


Fig. 14. **Single-client striping bandwidth** as a function of the number of target OSTs—the stripe width or *stripe_count*, following the experiment template in Figure 13. For each *stripe_count* target, 16 processes (cores) of a client issue 16 write bursts of total size 960 MB, 3.75 GB, and 15 GB striped across the targets. The graph represents the measured bandwidths with red, green, and blue boxplots for each respective burst size, from left to right.

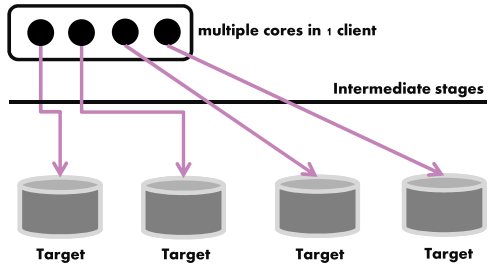


Fig. 15. **Template for client saturation experiment.** Multiple processes (cores) on a client issue synchronized concurrent write bursts to different target OSTs on different servers. This experiment probes the output limitations of the client without the complexity of striping.

4.5 Many-Pairs Bandwidth and Stragglers

The next experiment probes the aggregate I/O bandwidths achievable on Titan and the consistency of performance in different parts of the machine. The runs use equal numbers of clients and targets grouped in client-target pairs: each client runs a single process that writes a single file on a single target following the template of Figure 17. At the largest scale, we use 336 compute nodes to write to all 336 targets in the Widow1 storage system on Titan. This experiment uses a burst size of 64 MB.

Figure 18 and Figure 19 summarize the results. These graphs plot different views of the same data. Although the raw peak bandwidths (24.2 GB/s) are impressive with more pairs, the results are highly variable across runs and the overall output bandwidth utilization is low.

A key factor in this experiment is the variance in completion times for the pairs. The bursts for all pairs are synchronized, and the time interval for the aggregate bandwidth figure is the completion

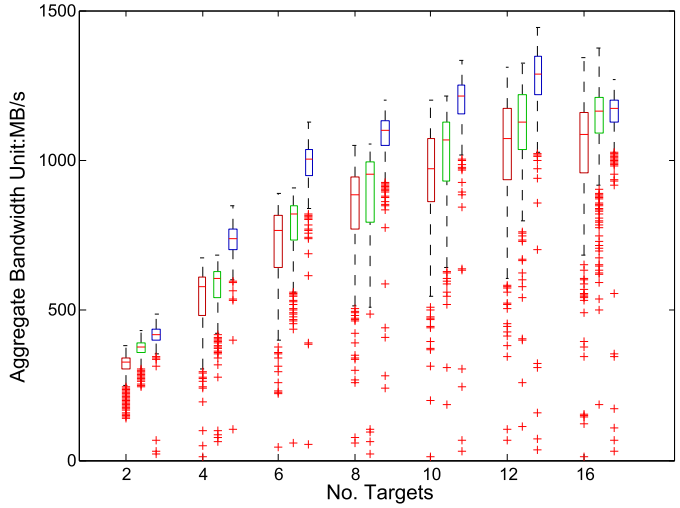


Fig. 16. **Output bandwidth of a compute node** as a function of the number of targets engaged, using the template of Figure 15. The client issues synchronized bursts to multiple files on multiple targets, with no striping: each process (core) writes to a single target. The observed aggregate bandwidths are shown in the boxplots for total burst sizes of 960 MB, 3.75 GB, and 15 GB (respectively, red, green, and blue, from left to right). The client's peak bandwidths and peak median bandwidths are 1.43 GB/s and 1.25GB/s, about 30% and 50% higher than obtained using striping (see Figure 14).

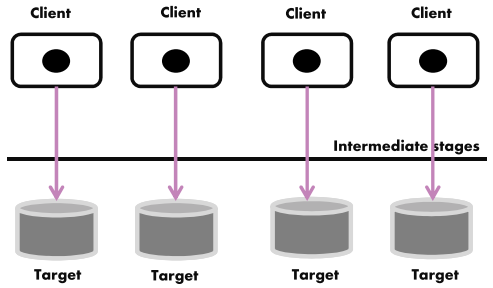


Fig. 17. **Template for the many-pairs experiment.** Each client node runs a process that issues a 64-MB burst to a separate unstriped file on a selected target. Each client uses a different target. The bursts are synchronized. We vary the number of client-target pairs and measure aggregate bandwidth and the bandwidth (or completion time) for each client-target pair.

time of the slowest pair. In each instance of the experiment, some pairs complete quickly while others are "stragglers" that limit the computed aggregate bandwidth.

To quantify the impact of stragglers, Figure 19 plots the cumulative distributions of completion times across all client-target pairs for each instance of the experiment. In all cases, more than 95% of the synchronized bursts complete in 2 s, but almost every trial has a tail of stragglers. In the trial, other pairs are idle while waiting for the stragglers to finish. The impact of stragglers grows as we increase the number of pairs: both the number of stragglers and their completion times increase substantially. Stragglers may be caused by bottlenecks in the interconnect and not necessarily in the targets themselves. Using all 336 targets, even the completion times of the fastest pairs are noticeably greater, indicating that the run has triggered congestion in intermediate stages, uniformly affecting all pairs.

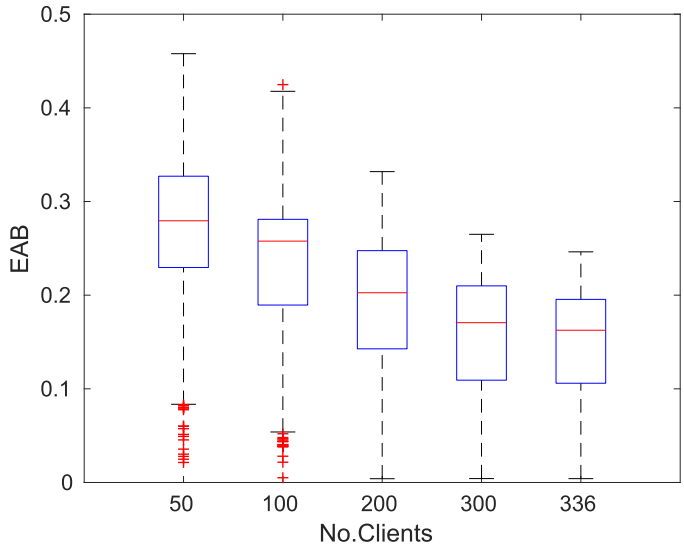


Fig. 18. **EAB of the many-pairs experiment.** This EAB plot shows the aggregate bandwidths obtained following the template in Figure 17 as a function of the number of client-target pairs. The bandwidths are normalized to the nominal aggregate bandwidth of the targets used in each run. With 200 client-target pairs, the delivered bandwidth falls below 40% of what could be achieved if all targets yield their expected bandwidths. The effective bandwidths degrade with larger numbers of pairs.

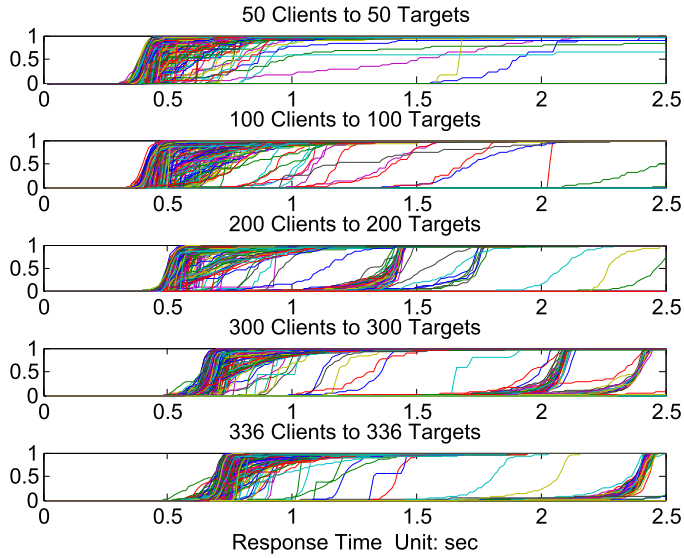
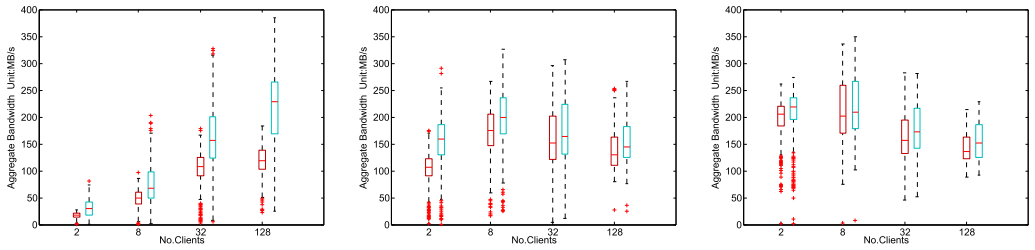


Fig. 19. **CDFs of completion times for the instances of the many-pairs experiment.** Each subgraph has 600 CDF lines, one for a trial of an instance. Each line shows the distribution of completion times for the pairs of one trial. Each line of the five types of instances has 50, 100, 200, 300, and 336 points (pairs), respectively. It is easy to see that almost every trial has good performance in some parts of the machine as well as stragglers that limit the computed bandwidth.



(a) results of 1MB bursts (b) results of 16MB bursts (c) results of 256MB bursts

Fig. 20. **Bandwidth of a single target with a shared file** as a function of the number of clients. Three subfigures separately represent the results of 1-MB, 16-MB and 256-MB bursts. In each figure, the red (left) boxes are the results of the synchronized bursts to a single file on the target; the green (right) boxes are the results of the identical bursts to independent files on the target. The observed bandwidths of the shared file with different bursts are consistently lower than those of independent files.

Figure 19 shows that interconnect bandwidth is a factor, and stragglers gate the delivered bandwidth. These results reflect substantial problems with load balancing in large shared production machines. The straggler problem motivates a looser coupling of parallel I/O pipelines. It may also motivate adaptive selection of compute nodes and targets based on system conditions, *if* contention effects are sufficiently regular and long lived. Section 5 addresses this issue.

Conclusions. For this synchronized burst experiment, the aggregate bandwidths of Titan are lower than half of the achievable performance (Figure 18); the straggler problem is the major factor for this performance degradation (Figure 19).

4.6 Impact of Write Sharing

The experiment is similar to the template in Figure 8, except that we use burst sizes of 1 MB, 16 MB, and 256 MB; we add tests for shared files as well. For the sharing tests, each client writes an independent region of the same file.

Figure 20 shows that the observed bandwidths with write-shared files are always lower than those with independent files with the same burst size. Moreover, with smaller burst sizes, the cost of write sharing increases quickly with larger numbers of clients. We conclude from these results that write sharing of files is undesirable if high output bandwidth is a goal.

These results reflect the impact of locking, which is necessary when multiple clients write share a file. Lustre uses locking to synchronize accesses to each file from multiple clients. In particular, Lustre object servers support range locking on objects at the granularity of 4-KB blocks. Lustre grants object locks greedily to reduce overhead. A client requests an exclusive lock covering the block range of any expected write. The server grants a lock on the maximal enclosing range that is free of conflict with any lock held by another client. If the requested range conflicts with an existing lock, then the server calls back to the lock holder to reclaim the lock on any conflicting part of the range. The lock holder flushes any pending writes on the reclaimed range before releasing the lock. This locking scheme borrows from the approach used in VAXClusters [15].

Conclusions. Compared with independent writes, write sharing reduces delivered bandwidth by up to a factor of two (Figure 20) owing to output pipeline bubbles while a client waits to acquire the necessary locks.

5 UNDERSTANDING PERFORMANCE VARIATIONS IN HARDWARE COMPONENTS

This section investigates the relative performance obtained from individual components in Titan and its storage system during parallel I/O. We analyze and compare measures obtained from

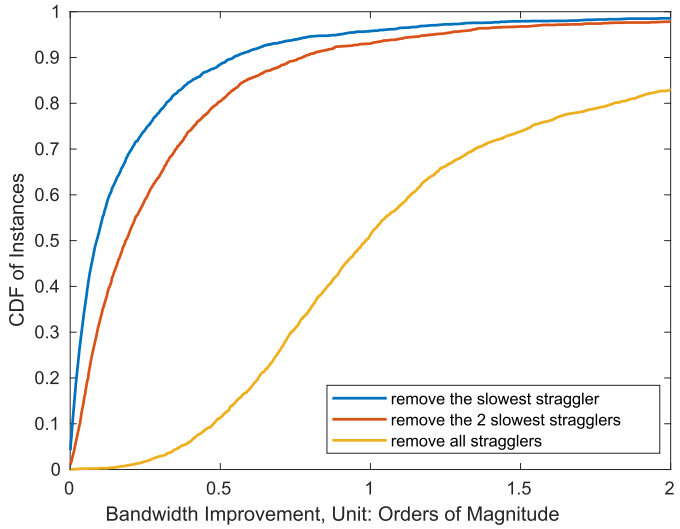


Fig. 21. **Bandwidth improvement from eliminating stragglers**. The figure shows a CDF for bandwidth improvements possible over all instances (samples) of many-pair experiments (6,000 instances). Blue, red, and yellow lines show improvements from eliminating the slowest straggler (blue line), the 2 slowest stragglers (red line), and all stragglers (yellow line) from each of these instances, respectively.

specific components (compute nodes, servers, and targets) at different stages of the Titan I/O system. The goal is to understand performance variation within the machine, the temporal patterns of variation, and the degree to which we can attribute performance issues to specific components. The analysis uses additional data collected from the same experiments as in Section 4 and new data collected in 2015 (Section 5.5) and 2017 (Section 5.4).

We limit our attention to the subset of experiments in which each file is stored on a single target, with no striping and no sharing of files among multiple compute nodes. For these experiments, we also measured the write bandwidths observed for each individual *node-target pair* in each experiment instance. A *pair measure* is the output bandwidth observed from a specific compute node to a specific target during an instance. This section analyzes the dataset of pair measures.

Section 4 shows that parallel write performance depends on balanced load and balanced capability within the machine. The pairs in each instance synchronize the start times for their writes, but their completion times vary according to the bandwidth obtained by each pair. A *straggler* is any pair that obtains lower bandwidth than another pair of the same instance. Since all pairs in an instance transfer the same amount of data, the stragglers take longer to complete than the faster pairs. They delay write completion and limit overall performance of parallel I/O because they continue transferring after other pairs have completed their transfers and idled at the IOR barrier.

Reducing or eliminating stragglers could improve delivered performance substantially. We use the data collected from the many-pairs experiment (Section 4.5) as an example. Figure 21 illustrates the potential effect of eliminating stragglers in the experiment instances of interest. It shows the bandwidth improvements that these instances would have obtained if their straggler pairs had kept up with the group. If we eliminate the slowest straggler from each instance, 60% of the instances improve bandwidth by more than 5.9%, and the worst 1% of the instances yield improvements ranging from a factor of 3.04 to 31.98. Moreover, if we remove all stragglers, so that all pairs of each instance match the fastest pair, then 90% of instances improve by more than 47.75%, and the worst 1% of instances yield improvements ranging from a factor of 10.83 to 120.71.

One goal of the analysis is to identify these stragglers and understand their impact on the results and why they occur. In particular, if some components are more likely to be members of straggler pairs than others, it suggests that they are faulty or affected by some persistent load imbalance. Recognizing these anomalies when they occur can help to enhance system design, deployment, configuration, and maintenance.

5.1 Methodology

To characterize how the choice of components affects write bandwidths, we compare the relative bandwidths observed at different node-target pairs across IOR instances. In each instance, a number of IOR processes from compute nodes/cores issue output bursts concurrently to one or more OSTs (defined in Section 3).

Since we are able to control the selection of targets, we obtained a minimum of 2,114 pair samples for each target, with a median of 3,652 samples per target. The scheduler selects the compute nodes for each run, but we can determine their identities. We obtained samples for 17,470 compute nodes (91% of Titan), with at least 20 samples for 81% of the Titan nodes, with stable configuration and deployment. For these nodes, the median number of samples is 123.

To compare the full set of pair measures in a uniform way, we introduce a method to filter out the effects of varying background load through time and parameter settings that vary across the instances. These parameters include burst size, number of cores per client, number of storage targets, and so on. In this method we use two measures, each obtaining a set of normalized scores for each component.

(1) Assign each node-target pair (N_i, T_j) two scores for each pair measure: LEB (*Lag Effective Bandwidth*) and PEB (*Pair Effective Bandwidth*).
   - The LEB score is the pair's bandwidth normalized to the fastest pair in its instance. The fastest pair ($LEB = 1$) gives a rough measure (a lower bound) of the performance achievable under the parameters and general system conditions for that instance.
   - The PEB score is the pair's bandwidth normalized to the fastest pair measure within similar instances, which share identical parameter settings but run at different times. Such similar instances form an *equivalent instance set*. This analysis includes 91 instance sets with 600 instances per set. The fastest pair in a set ($PEB = 1$) gives a rough measure of the performance achievable for the set under ideal conditions: it is the best observed performance for any pair using those parameter settings.

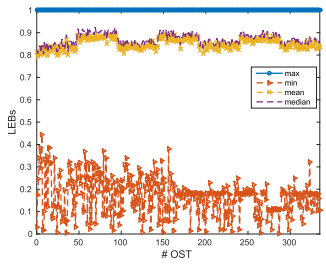
   Both scores are directly comparable across components. For example, if a component is faulty, its LEB/PEB scores are likely to be lower than those of components that are operating normally.

(2) Cluster the LEBs and PEBs by the specific components used for each pair across all instances and experiments, yielding a *component LEB set* and a *component PEB set* for each compute node and target. For each pair (N_i, T_j), $LEB_{(N_i,T_j)} \in LEB_{N_i}$ and $LEB_{(N_i,T_j)} \in LEB_{T_j}$, $PEB_{(N_i,T_j)} \in PEB_{N_i}$ and $PEB_{(N_i,T_j)} \in PEB_{T_j}$.

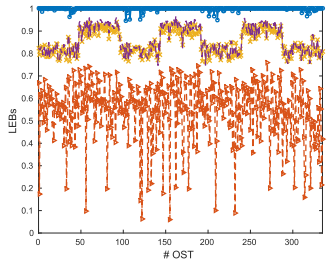
(3) For each node or target, compute the max, median, mean, and min of all LEB measures in the component LEB set and of all PEB measures in the component PEB set. We use these measures to compare component performance over time (Section 5.6).

In summary, a component's *LEB* score captures its relative behavior across instances; the *PEB* score captures the behaviors across experiment settings[1]. We employ these two relative measures

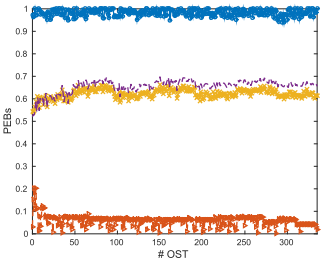
[1]For the instances in the same setting, the same number of IOR processes write from the same number of nodes/cores to the same number of OSTs (defined in Section 3).



(a) LEBs for 336 OSTs from all experiments



(b) LEBs for 336 OSTs from server saturation experiment



(c) PEBs for 336 OSTs from all experiments

Fig. 22. **Relative OST performance.** LEBs (the left two subfigures) and PEBs (right) for each of the 336 storage targets in Titan/Widow1, showing the max, min, median, and mean of all scores for each target. The leftmost and the rightmost subfigures plot the results across all experiments; the middle subfigure plots the LEBs of storage targets from the server saturation experiment, in which clients direct concurrent bursts across all targets of an OSS. The results show clearly that the target scores (median and mean LEB/PEB) vary according to the target's numbering.

to capture the consistent behaviors of a component at the I/O stages of compute nodes, OSSs, and OSTs.

5.2 Relative OST Performance

To demonstrate the methodology, this section analyzes the relative performance of the storage targets (OSTs). Figure 22 presents the median and mean LEBs and PEBs of OSTs; the experiment data was collected from January to December 2013 on Titan and Spider (see Table 2).

Figure 22 presents two patterns: for each block of 48 sequentially numbered OSTs, the LEB scores are equivalent. However, there are persistent differences between the alternating OST blocks: the OSTs in the first block have lower performance than the OSTs in the second block, and this pattern repeats across the 7 alternating blocks of Widow1 (shown in Figure 5). The difference in mean/median LEB scores can be as high as 20% or more. They are most pronounced in the LEB scores from the server saturation experiments (shown in Figure 22(b)).

These results are consistent with the potential for a load imbalance in the disk accesses from each server. Relative to Section 2.6, at the time of these experiments in 2013, each 48 adjacent OSTs form an OST block, each server accesses OSTs in seven distinct blocks, and a server's I/O operations are distributed across two IB ports according to the OST's numbering. One port carries traffic for 4 blocks, and the other carries traffic for 3 blocks. The more heavily loaded ports are slower, on average, as reflected by the normalized LEB and PEB scores.

Moreover, OST 0 to OST 9 have, respectively, 10% to 3% lower mean/median LEB and PEB scores than other OSTs. This may occur because programs that do not explicitly select an initial OST for striping may receive OST 0, resulting in heavier load on these low-numbered OSTs.

We took the same measurements on Titan and Atlas 2 in 2015 after an upgrade (see Table 2) that addressed this imbalance and verified that the effect was no longer present.

Conclusions. The emergence of these patterns identifies the load imbalance on the intermediate device (IB ports). The results also validate the sampling and analysis methodology: they give us confidence that the LEB and PEB scores can capture performance anomalies and offer insights into component performance.

5.3 Relative OSS Performance

As described in Section 2, an OSS is an intermediate stage on the write path: each write request for a target is received and served by the OSS that controls the target. We assess OSS performance indirectly by analyzing the performance of the targets that they mediate. Each pair measure is associated with exactly one target and, therefore, with exactly one OSS. The data are collected from the experiments on Titan and Spider (see Table 2) from January to December 2013.

We analyze the relative performance measures for the 48 OSS storage servers (336 targets) available in Widow1. We find that performance of any given OSS varies widely across instances—from 0.07 LEB to 1 LEB and from 0.03 PEB to 1 PEB—but that they exhibit the same medians for both scores over time.

Conclusions. The results suggest that transient load imbalances are common within the storage system, but that one cannot predict with any accuracy which servers will perform well or badly at any given time. The differences in relative performance show no discernible pattern in our dataset.

5.4 Component Performance Variability

This section analyzes performance variability of storage targets through time. The premise of the analysis is that the sequence lengths are a good measure of performance variability. For example, if OSTs continue to perform poorly for long periods, then it may be worthwhile to adaptively avoid those OSTs at runtime. Long sequences of poor performance are indications of unbalanced I/O load, faulty OSTs, or persistent hot spots. On the other hand, short sequence lengths suggest that OST performance is driven by transient load conditions and past performance is not a useful predictor of future performance.

This analysis is based on a new set of experiments that follow the experiment template of measuring the write bandwidth of an OST (see Figure 8). Similar to that experiment (Section 4.2), in each instance 4 synchronous processes from 4 clients write to an OST. In contrast, for the experiments in this section, continual instances in a run write bursts to a sequence of 4 OSTs in a round-robin fashion. We design and conduct 252 such experiments on Atlas 2 with 1,008 OSTs (see Section 2.6): in each experiment, 4 coordinated clients focus bursts on a different sequence of 4 OSTs with 16-MB bursts. We choose a small burst size to avoid triggering the self-contention in Titan.

We ran these experiments on Titan and Spider 2 (see Table 2) from June to July 2017 and collected 1 to 4 time series for each OST. Figure 23 reports the CDF of EAB scores of individual OSTs across time series and experiments. Figure 24 presents the CDF of time intervals of consecutive measures (left) and the CDF of the time series of continual measures on an OST (right). In this set of experiments, the time duration of a run ranges from 0.5 to 0.98 h; the time interval between two consecutive measures in a run ranges from 17.8 to 30 s.

To quantify the performance variability of individual OSTs, we first label an OST's EAB scores as low or normal performance according to a given score determined by a chosen quantile threshold (t): if an EAB measure is below the given score, it is labeled as a low-performance measure; otherwise, it is considered as a normal-performance measure. Secondly, we measure the lengths of the continual low/normal performance sequences for each OST across its time series. According to Figure 23, we take three quantile thresholds across experiments: $t = 0.05$, $t = 0.1$, and $t = 0.2$.

Thus, for a chosen t, a low-performance sequence represents a time length of a storage target: in the time length, the target delivered a sequence of EAB scores below the score associated with the quantile threshold t for the 16-MB bursts. Relatively, for the same t and burst size, a normal performance sequence represents a time length of a storage target: in the time length, the target delivered a sequence of EAB scores above the score associated with the quantile threshold t for

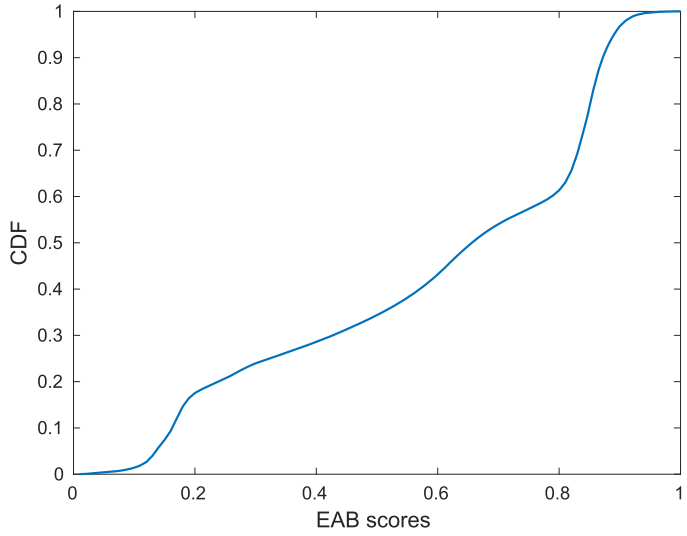
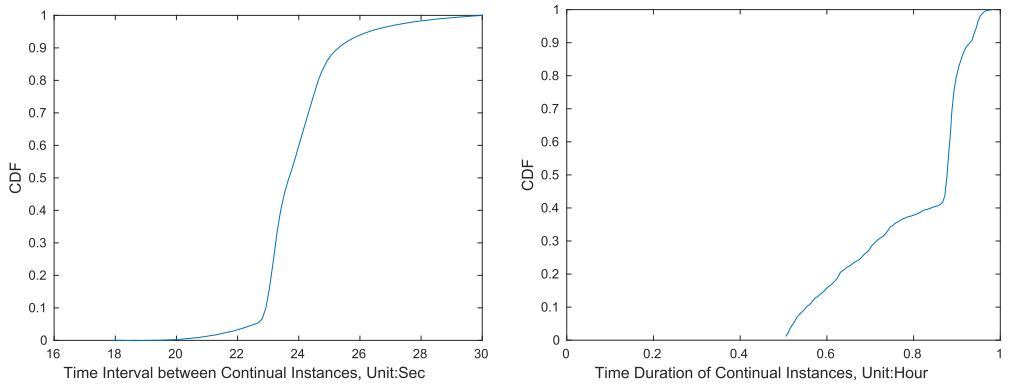


Fig. 23. **CDF of EABs of individual OSTs** with 16-MB bursts.



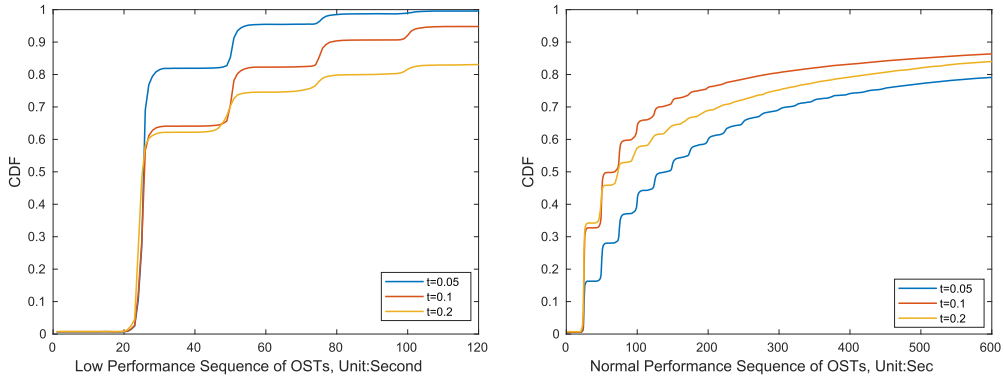
(a) Time intervals of consecutive pair measures (b) Time series lengths of measures

Fig. 24. **CDFs of time intervals and time series.** This figure shows the cumulative distributions of time intervals of consecutive pair measures (left) and the time series lengths of measures on an OST (right).

the 16-MB bursts. We report the CDFs of low/normal-performance sequences of storage targets in Figure 25.

Figure 25 shows that, for all quantile thresholds, over 70% (or 82%) of the low-performance sequences are within 1 min (or 2 min) for storage targets. That is, OSTs that showed low performance generally returned to normal within a short time. Similarly, Figure 25 also shows that normal-behavior OSTs periodically perform poorly: for all ts, above 79% of the normal-behavior sequences terminated within 10 min.

Conclusions. The results suggest that Titan's storage system is highly variable. This complements the observations reported in [40] that the performance behavior of compute nodes in Titan is also highly variable. We conclude that Titan's I/O system is highly variable. Generally speaking, this



(a) CDFs of low behavior sequences of OSTs (b) CDFs of normal behavior sequences of OSTs

Fig. 25. **CDFs of low- and normal-behavior sequences of OSTs**. From left to right, each subfigure presents the CDFs of the time durations of the low-behavior sequences (left) and normal-behavior sequences (right) with 16-MB bursts, respectively. In each subfigure, a line shows the CDF of the low (normal)-behavior sequences associated with an EAB score determined by a quantile threshold (t) of EAB scores (defined in Section 5.4). It suggests that OSTs that showed low performance generally returned to normal within 2 min, and OSTs switched from normal-performance labels to low within 10 min.

high variability suggests that it is not fruitful to identify "good locations" in the machine or in the file system to improve I/O performance.

5.5 Performance Variability and Node Locality

This section probes performance variation across compute node locations. To this end, we examine the many-pairs experiment again (the template in Figure 17) with 16-MB and 256-MB bursts from each of 1,008 compute nodes to a different storage target. We also extend the methodology to group the runs into *sets*, each comprising multiple identical runs with the same group of compute nodes, closely spaced in time. Different sets executed on different groups of compute nodes and at different times.

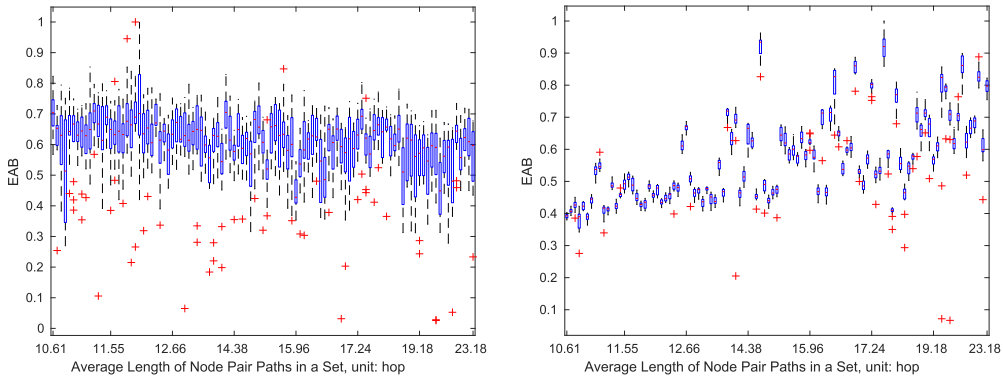
Our analysis is based on measurements taken from May to June 2015 on Titan and Spider 2 (see Table 2). We collected 95 sets with a total of 103 runs; a few sets have multiple runs. Each run comprises 15 rounds of instances with 16-MB and 256-MB bursts, respectively.

To explore the set behaviors for different burst sizes, we estimate the node distribution of each set by measuring the average path length (L) between the nodes in all pairs of nodes drawn from the set. A smaller L indicates a more tightly packed (denser) node set; a larger L indicates a more widely scattered node set. To measure the distance for each node pair in a set, we choose a common metric, L_1 *routing distance*: the length of a path between two points in Titan's 3D torus. For a node pair at positions (x_1, y_1, z_1) and (x_2, y_2, z_2), the distance (d) of the pair is given by

$$d = |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|. \quad (1)$$

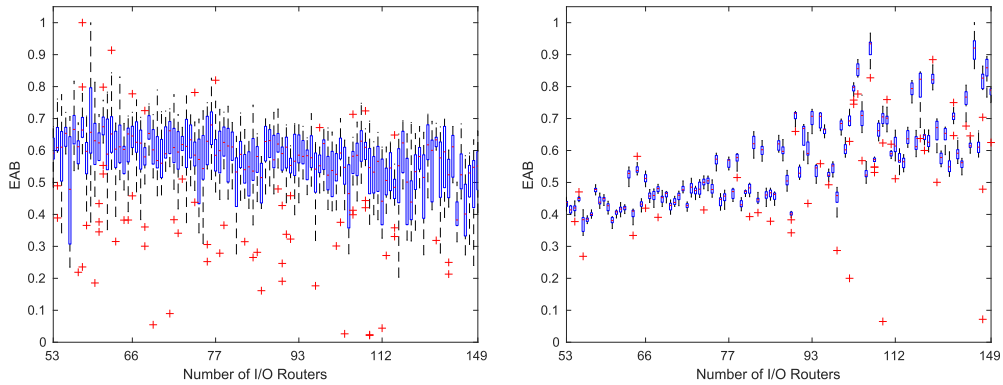
The L values of the 95 sets vary from 10.61 to 23.18 hops. To explore the correlation between output performance and set density, Figure 26 plots their boxplots[2] of EABs ranked by density for 16-MB and 256-MB bursts.

[2]A boxplot shows that the quartiles of the bandwidths collected from the identical instances share the same set of compute nodes. In each boxplot, the bottom, the middle bar, and the top of the box represent 0.25, 0.5, and 0.75 quartiles, respectively. The box contains the middle 50% of samples (called the interquartile range, or IQR); the bottom and the top bars below



(a) Results of 16MB bursts (b) Results of 256MB bursts

Fig. 26. **EABs of the 95 sets** with 16-MB and 256-MB bursts sorted by node density. From left to right, a subfigure reports the boxplots[2] of the EABs of the 103 runs from the 95 sets with 16-MB and 256-MB bursts, respectively. In each subfigure, the x-axis represents the 95 node sets sorted by L (defined in Section 5.5). The corresponding y values summarize the distribution of measured bandwidths (EABs) for that node set.



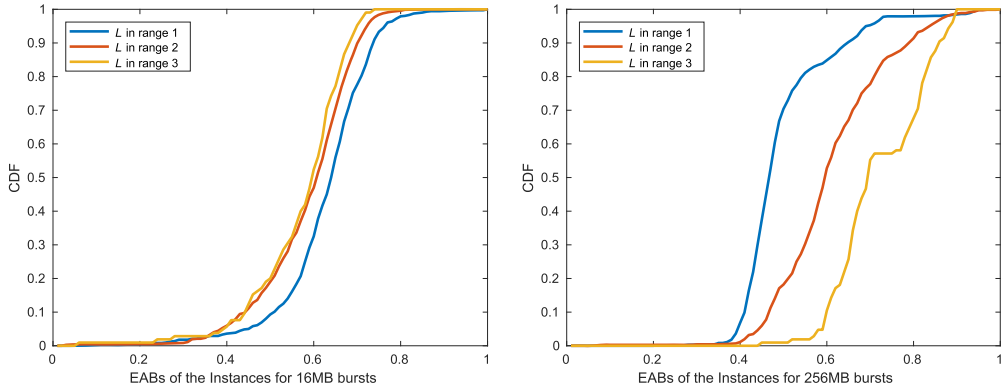
(a) Results of 16MB bursts (b) Results of 256MB bursts

Fig. 27. **EABs of the 95 sets** with 16-MB and 256-MB bursts sorted by the number of I/O routers in use. From left to right, a subfigure reports the boxplots[2] of the EABs of the 103 runs from the 95 sets with 16-MB and 256-MB bursts, respectively. In each subfigure, the x-axis represents the 95 node sets sorted by the number of I/O routers. The corresponding y values summarize the distribution of measured bandwidths (EABs) for that node set.

We also measure the numbers of I/O routers used by the 95 sets according to the Titan's network configuration of I/O routing policy (see Table 2). In summary, the 95 sets use 53 to 149 I/O routers; Figure 27 shows the EABs of the 95 sets ranked by the number of routers in use for 16-MB and 256-MB bursts.

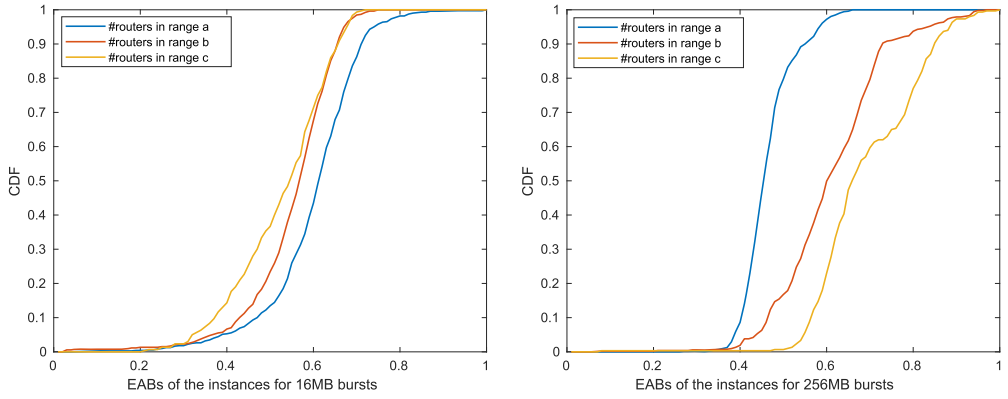
Figures 26 and 27 show that, for 16-MB bursts, the EABs of all sets are distributed in a wide range and are only weakly correlated with density and the number of routers. Small bursts are sensitive to transient contention on the shared intermediate stages or on the storage servers/targets. However,

and above the box report the sample bandwidths at the low and high 1.5×IQR, respectively; the dots on both sides depict outliers.



(a) Results of 16MB bursts (b) Results of 256MB bursts

Fig. 28. **CDFs of the instance EABs** of the 95 sets with 16-MB bursts (left) and 256-MB bursts (right). In each subfigure, the lines (blue, red, and yellow) depict the CDFs of the instances examined on the sets with L in three hop ranges, respectively: 10.61 to 15 (720 instances), 15.01 to 20 (720 instances), and 20.01 to 23.18 (105 instances).



(a) Results of 16MB bursts (b) Results of 256MB bursts

Fig. 29. **CDFs of the instance EABs** of the 95 sets with 16-MB bursts (left) and 256-MB bursts (right). In each subfigure, the lines (blue, red, and yellow) depict the CDFs of the instances examined on the sets with the number of I/O routers in three ranges, respectively: 53 to 84 (720 instances), 85 to 116 (525 instances), and 117 to 149 (300 instances).

for the 256-MB bursts, the sets with larger L and with more routers are more likely to deliver higher aggregate bandwidths.

To quantify the behaviors of the sample-size bursts on various set densities and with different numbers of I/O routers, we further partition the 95 sets based on two rules:

(1) Partition the 95 sets into three ranges of average hop distance: 10.61 to 15 (range 1), 15.01 to 20 (range 2) and 20.01 to 23.18 (range 3), and plot the EABs for the instances in each range in Figure 28.
(2) Partition the 95 sets into three ranges according to the number of I/O routers in use: 53 to 84 (range a), 85 to 116 (range b), and 117 to 149 (range c). Figure 29 shows the output bandwidths (EABs) for the instances of the 3 ranges.

Figure 28 suggests that, for 256-MB bursts, above 80% of the instances in the L range 1, range 2, and range 3 report ~0.4, ~0.52, and ~0.67 EABs, respectively. Similarly, Figure 29 shows that, for 256-MB bursts, above 80% of the instances in the router range a, range b, and range c report ~0.45, ~0.60, and ~0.66 EABs, respectively. For the larger bursts, the sets with larger L and with more routers tend to deliver higher aggregate bandwidths.

It is worth noting that, for 16-MB bursts, the results suggest slightly better performance for denser node sets. Even dense sets are free of self-contention for small bursts and may benefit from the lack of cross-contention in the interconnect and I/O routers from other jobs on the machine. Even so, in a busy and highly contended system such as Titan, the transient system conditions on the storage servers/targets dominate this effect. For longer bursts, the transient hot spots in the storage system tend to cancel out, and the results are dominated by persistent self-contention in the interconnect and I/O routers. This suggests that the I/O routing policy could be improved to spread the load for large bursts. In this scenario, I/O adaptive tools (e.g., ADIOS) might be helpful to move and redistribute the load across the machine.

Conclusions. We conclude that while the bandwidth of small bursts is dominated by transient contention, the performance of large bursts is impaired by denser node sets. Of course, the job scheduler prefers dense node sets because a densely packed job experiences less cross-contention from other jobs on the internal interconnect. However, denser sets may experience self-contention on the internal interconnect and also are locked into using a smaller set of I/O routers, since the binding of nodes to routers is static and determined by node location (see Table 2). More dispersed sets spread their loads across a larger portion of the interconnect and a larger number of I/O routers, and tend to show higher bandwidth accordingly.

5.6 Performance Through Time

We also use the PEB score to measure performance variations through time to identify any predictable or repeating load patterns (e.g., diurnal patterns). We assign a score for each day of the week and for each hour of the day: the score is the mean or median of all scores that occurred during that day or hour across all experiments from January to December 2013. The dataset has good coverage over days and hours. There are at least 103,131 node-target pair samples for each day of the week, with a median of 214,106. There are at least 32,754 instance samples for each hour of the day, with a median of 61.922.

We find that the performance of Titan's I/O system is stable across days in the week and hours in the day: the scores for all days and all hours converge to the overall median score (0.6). Mondays show the worst overall performance (lowest score), but the difference is minor.

Conclusions. This finding confirms that the temporal usage pattern of HPC file systems are notable relative to other common file system settings (e.g., [7]) in that there are no predictable "slack" periods for rebalancing, backup, consistency scrubbing, or other maintenance tasks. Ideally, any such maintenance tasks occur continuously and predictably at an even rate to limit the impact on supercomputer application workloads, which often run continuously over days and weeks and produce periodic outputs.

6 VERIFICATION ON REAL APPLICATIONS

This section shows how the observed properties of Titan's I/O system under synthetic loads (Section 4 and Section 5) reflect in the behaviors of exemplary applications. Specifically, we focus on the analysis of XGC and HACC IO on Titan.

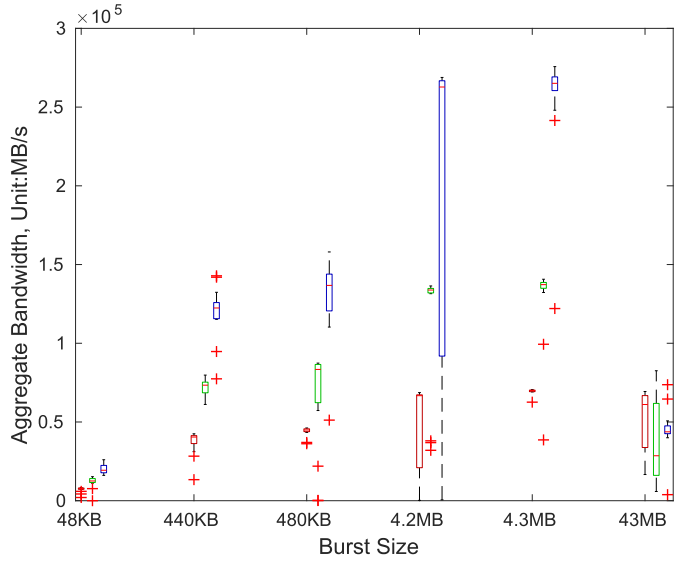
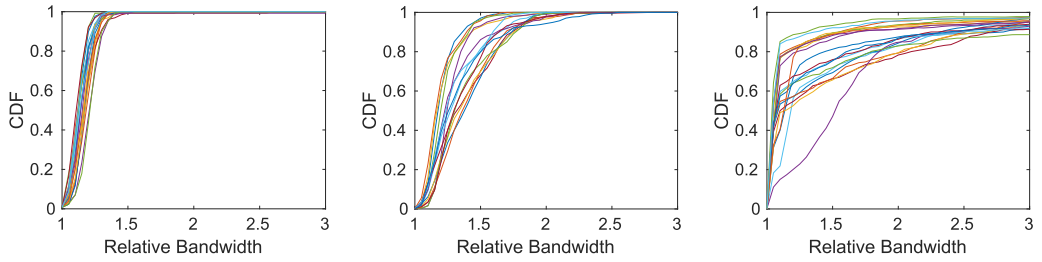


Fig. 30. **Aggregate bandwidths** of XGC runs with 128, 256, and 512 compute nodes. In this figure, each boxplot reports the aggregate bandwidths of 15 sample points; the x-axis represents the burst size; for each x value, we report the results of 128, 256, and 512 nodes with red, green, and blue boxplots, respectively.



(a) Results of 128 nodes with 48KB bursts

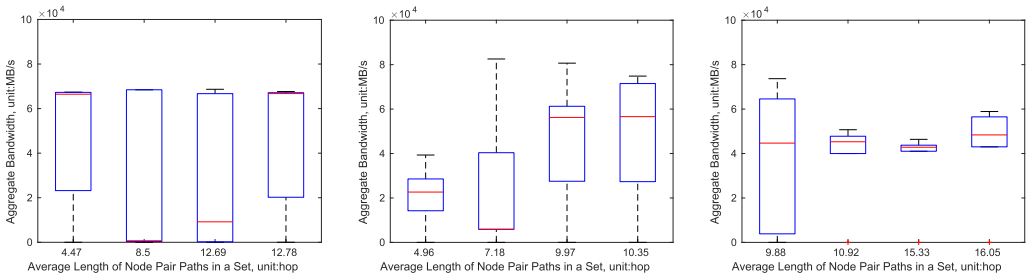
(b) Results of 256 nodes with 4.3MB bursts

(c) Results of 512 nodes with 43MB bursts

Fig. 31. **CDFs of relative bandwidths** of XGC runs with 128, 256, and 512 nodes. From left to right, three subfigures separately represent the results of 128, 256, and 512 nodes. Each figure consists of 20 lines; each line reports the CDF of 128, 256, or 512 relative bandwidths of a sample from 128, 256, or 512 nodes, respectively. Relative bandwidth is defined in Section 6.1.2.

6.1 XGC Code

XGC is an iterative fusion reactor simulator. It generates output bursts between iterations across a group of synchronous processes; each process writes to a different file with a similar amount of data in parallel (as described in Section 2.2). We conducted XGC runs on Titan between November 7 and 20, 2018. The runs are scaled on 128, 256, and 512 compute nodes separately, with each process running on a different node. For the runs, we configure output bursts in two ways: "small" bursts and "large" bursts. In a run with small bursts, each process produces 48-KB, 440-KB, or 4.2-MB bursts; a run with large bursts produces 480-KB, 4.3-MB, or 43-MB bursts. In a run, the 128, 256, or 512 processes write synchronously to 128, 256 or 512 independent OSTs, respectively.



(a) Results of 128 nodes with 4.3MB bursts

(b) Results of 256 nodes with 43MB bursts

(c) Results of 512 nodes with 43MB bursts

Fig. 32. **Aggregate bandwidths** of XGC runs as a function of average length of node-pair paths. Three subfigures separately represent the results for 128, 256, and 512 nodes. In each figure, the x-axis represents the average length of node-pair paths in a sample (defined in Section 5.5); each boxplot reports five samples.

Figures 30, 31, and 32 report the results. In summary, XGC is exposed to three principles of Titan write behaviors: pipeline performance (Section 4.1), straggler issue (Section 4.5), and node locality (Section 5.5).

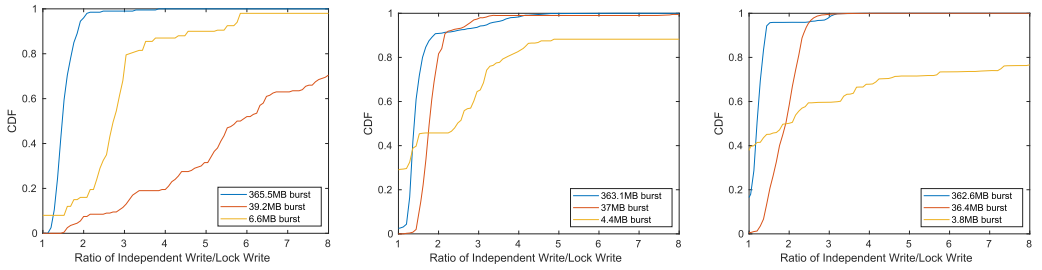
6.1.1 Pipeline Analysis. Figure 30 reports the aggregate bandwidths of XGC runs on Titan. The results show a trend similar to the single-pipeline experiment (see Section 4.1 and Figure 7): for 48-KB to 4.3-MB bursts, aggregate bandwidths grow with increasing burst size. Bandwidth declines with the larger 43-MB bursts, however, and the XGC runs show lower bandwidths than predicted from the single-pipeline results (see Figure 7), even with small bursts. Instead, they are quite similar to the behavior observed in the many-pair experiment (see Section 4.5): it shows a comparable bandwidth profile, including the drop in bandwidth with large burst sizes, as expected. We conclude that XGC's synchronized bursts across many compute nodes trigger congestion in Titan's interconnect; there may also be contention in the I/O network, although each node writes to a different target.

6.1.2 Stragglers. To compare the stragglers across write scales and burst sizes, in each XGC instance, we take relative bandwidths by normalizing the bandwidths of the synchronous processes to the minimum bandwidth among them.

The XGC runs can be grouped into 18 (3×6) settings, each executed on a specific write scale with a specific burst size. Among the 18 settings, we find that the straggler issue falls into 3 patterns. Figure 31 addresses the 3 patterns with the results of 3 settings (128 nodes with 48-KB bursts, 256 nodes with 4.3-MB bursts, and 512 nodes with 43-MB bursts).

In summary, across the 18 settings, the maximum relative bandwidth of a setting is 254.3 to 2781.4, suggesting the highly variable performance of Titan's I/O system (see Section 5). Moreover, as shown in Figure 31, for 48-KB to 480-KB bursts, 80% to 90% of the processes of an instance report $\leq$0.2 bandwidth variance; for 4.2-MB to 4.3-MB bursts, up to 70% of the processes in an instance report $\leq$0.5 bandwidth variance; for 43-MB bursts, up to 60% of the processes in an instance report $\leq$0.7 bandwidth variance. These results are consistent with our observations for single-client performance (see Figure 10): for synchronous writes, longer-lived bursts (larger bursts) show larger performance variance.

6.1.3 Node Locality. Figure 32 reports the node locality of three settings (128 nodes with 4.3-MB bursts, 256 nodes with 43-MB bursts, and 512 nodes with 43-MB bursts), which are representative of the 18 settings of the XGC runs (see Section 6.1.2).



(a) Results of 8 nodes (b) Results of 32 nodes (c) Results of 128 nodes

Fig. 33. **CDFs of the performance of write sharing** for the HACC IO runs with 8, 32, and 128 nodes. Three subfigures separately represent the results of 8, 32, and 128 nodes with *stripe count* = 1. In each figure, a line represents the CDF of the ratio of independent write/lock write (defined in Section 6.2.1).

In summary, the results suggest that the write performance of XGC runs is not affected by the node locality issue. This observation aligns with the conclusions of the node-locality analysis (see Section 5.5): only for large bursts (e.g., ≥256-MB bursts) do dense node sets affect write performance significantly.

6.2 HACC IO

HACC is a cosmological simulation for large-scale sky explorations. HACC IO is an I/O benchmark specifically developed to address HACC's I/O behaviors [26].

Each HACC IO run generates 3 types of output bursts; in each write operation, the synchronous processes write share a single file with the same data size. To evaluate the performance of data locks in write-sharing, we add 50 lines of C++ code in HACC IO to let the synchronous processes write to different files with the same data size as they do for a shared single file.

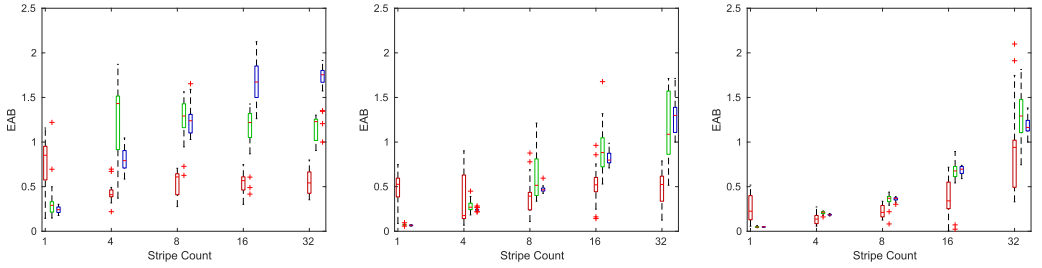
We conducted HACC IO runs on Titan between November 1 and 28, 2018. The runs are scaled for 8, 32, and 128 compute nodes, with each process running on a different node. The 8-node runs generate bursts of 6.6 MB, 39.2 MB, and 365.5 MB; the 32-node runs generate bursts of 4.4 MB, 37 MB, and 363.1 MB; the 128-node runs generate 3.8 MB, 36.4 MB, and 362.6 MB bursts. For a burst type in a run on a write scale, we configure *stripe count* (discussed in Section 2.4) as 1, 4, 8, 16, and 32 and *starting OST* as the same randomly chosen OST for all of the synchronous processes in the run.

This section addresses two of Titan's I/O behaviors on HACC IO runs: write sharing (see Section 4.6) and striping (see Section 4.4).

6.2.1 Performance of Write Sharing. To evaluate the performance of write sharing, we focus on the settings with *stripe count* = 1, in which the synchronous processes write to the same OST.

For each process with a burst size in a HACC IO run, we collect its bandwidths for independent write and write sharing (lock write), then normalize the performance with independent write to write sharing (the ratio of independent write to lock write).

Figure 33 reports the CDFs of the ratios of independent write to write sharing. It is clear that, across burst sizes and the write scales, 64.2% to 100% of independent writes outperform write sharing ≥1.2×; in extreme cases, the outperformance is up to 372.4×. Moreover, for all scales, the performance declines when increasing burst size. These results support the conclusions in the write-sharing analysis (see Section 4.6).



(a) Results of 8 nodes (b) Results of 32 nodes (c) Results of 128 nodes

Fig. 34. **EABs** of the HACC IO runs as a function of stripe count. Three subfigures separately represent the results of write sharing for 8, 32, and 128 compute nodes. In each figure, the x-axis presents stripe count; for each x value, it reports red, green, and blue boxplots for 3.8-MB to 6.6-MB bursts, 36.4-MB to 39.2-MB bursts, and 362.6-MB to 365.5-MB bursts, respectively; each boxplot consists of 25 samples.

6.2.2 Performance of Striping. We focus on the performance of write sharing with striping. Figure 34 reports the results; in each subfigure, for *stripe count* = 1, 4, 8, 16, and 32, the numbers of OSTs in use are 1, 4, 8, 16, and 32, respectively.

Figure 34 shows that, for 3.8-MB to 4.4-MB bursts on the scales of 8 and 32 nodes, when increasing stripe count, the aggregate performance does not change; for 6.6-MB bursts on the scale of 128 nodes, the performance grows when stripe count increases. It is similar to the performance of striping in a single node (see Section 2.4): larger-scale writes with larger data sizes can use each of the concurrent pipelines more efficiently.

For 36.4-MB to 365.5-MB bursts, when increasing stripe count, the aggregate performance increases and the best performance is with *stripe count* = 16 and 32 (same as in Section 2.4).

In summary, all of these results suggest that our conclusions on the striping of a single node (see Section 2.4) hold on large-scale writes.

7 RELATED WORK

A significant recent study installs continuous monitoring software on compute nodes to characterize the I/O requests of real application workloads in real time, modulating the data collected to keep overhead within acceptable limits [2, 3, 22].

Uselton et al. [34] propose a statistical method to collect and analyze I/O events to more fully characterize the I/O behavior of ensembles. They also observe the straggler phenomenon, suggesting that the straggler problem is a general issue in supercomputers. Their work focuses on improving the I/O performance of a given application in a given supercomputer system. Our goal is to characterize the multi-stage write pipeline in a petascale file system, locate write absorption bottlenecks, and capture component performance variability that influences the design and configuration choices for adaptive middleware and HPC applications.

Other previous studies use an approach similar to ours: stress the file system with synthetic benchmarks. A number of HPC I/O benchmarks are designed to be sufficiently flexible to emulate the typical I/O behaviors in supercomputer environments, such as the FLASH I/O, IOR, and BTIO benchmarks. This flexibility enables users to configure the benchmark for a desired pattern approximating an observed application behavior. In our work, we take IOR as a generator and run different patterns and configurations to focus traffic on specific stages and elements of the write pipeline to gain a complete picture of output burst absorption in a production facility.

A recent study [18] uses a similar methodology to measure the performance of the Intrepid file system at the Argonne Leadership Computing Facility. The authors report the capacity of

each I/O stage and measure the behavior of the entire subfile system for large-scale runs of a set of benchmarks. The measurements were taken on dedicated hardware before the supercomputer system was running in production mode. Our work explores the delivered bandwidth of the I/O stages in ongoing production use, reflects the impact of competing workloads under observed usage patterns in production, and shows how to filter noise from competing workloads to obtain insights into the behavior of the underlying hardware and software.

Earlier studies also use configurations of the IOR benchmark to analyze the behavior of HPC systems [28, 29]. Kim et al. [13] collect I/O performance data from Titan's predecessor Jaguar. That study is complementary to ours: it reports monitoring data from the storage servers showing the combined workload on the machine. We focus on the end-to-end behavior observed by jobs running on the compute nodes and the impact of write patterns and I/O configuration choices.

A group of recent studies [10, 17, 24] adopt learning models to predict the I/O behaviors of HPC applications. In particular, Kunkel et al. [17] build decision trees to predict applications' non-contiguous I/O behaviors and further determine the parameter combinations for data sieving in ROMIO [32]. Omnisc'IO [10], inspired by the Sequitur algorithm, introduces a context-free grammar to learn I/O patterns of HPC applications. McKenna et al. [24] model application runtimes and I/O patterns by building a training set from job logs and system monitoring tools. Compared with the works in this group, our study conducts experiments to explore the I/O behaviors of the target systems under various system conditions. Another group of studies [23, 35] use learning models to predict the I/O performance variability of supercomputers. Wan et al. [35] build a hidden Markov model for a Lustre file system to learn the variability of I/O latencies from a single client to a single OST. Madireddy [23] et al. use a Gaussian process to predict the I/O performance variability of a Lustre-based supercomputer and derive system-specific features from Lustre monitoring tools. In contrast to the studies in this group, we address the variability issue with two relative measures (see Section 5.1) and quantify the degree of variability in both compute nodes and storage targets (see Section 5.4).

8 CONCLUSION

I/O bandwidth is a scarce resource on supercomputers. Output burst absorption can have a substantial impact on delivered performance, as demonstrated by a simple performance model. Observed output bandwidth is sensitive to various uses of the storage system APIs and different supercomputer I/O system conditions.

Our study offers a benchmarking methodology to measure the impacts that result from hardware limitations and the choices on designs and configurations of supercomputer file systems in a particular facility by configuring a synthetic I/O benchmark to stress individual stages of the write pipeline in turn across a range of parameters. Moreover, we introduce effective measures to probe and profile the performance of individual components of the stages in the target facility via extracting behaviors of each specific component across experiments, configurations, instances, and times.

We apply the benchmarking approach to map Lustre file system output performance in Titan and its Spider/Atlas file stores. The measured distributions quantify the frequency and severity of contention (stragglers) and other transient system conditions. We show that these imbalances lessen the benefit of coupled I/O parallelism (striping). This effect motivates structuring choices to loosen the coupling of parallel I/O. For example, on Titan's I/O system under typical conditions, the peak median output bandwidths are obtained with parallel writes to many independent files, with no write sharing or striping, and with each target storing files for multiple clients and each client writing files on multiple OSTs.

The prevalence of these imbalances motivates adaptive responses in the I/O middleware layer. To evaluate the potential of adaptation, we studied the behavior of individual components to expose temporal usage patterns, slow components, and system-level performance variability that can lead to imbalances in the write path. Our results show that these performance stutters are difficult to predict and that system changes state quickly and frequently, suggesting that dynamic adaptation to congestion is not a fruitful approach.

Moreover, under a static node-to-router mapping policy adopted by Titan in its network configuration, for large bursts output performance is sensitive to the density of a job's compute nodes, as measured by the mean pairwise routing path distance within the node group.

Here is a summary of the conclusions relating to the design and configuration of HPC file systems and the configuration of I/O middleware systems, such as ADIOS:

- **Stragglers:** We observed that a small proportion of storage targets (<20%) are straggling at any given interval. Stragglers throttle the write pipelines and limit striping bandwidth. Based on this observation, we suggest careful control of target parallelism: a small stripe width limits bandwidth, but wider stripes are increasingly likely to be impacted by stragglers. Therefore, I/O middleware systems should select the number of targets carefully under the guidance of I/O performance models [41].
- **Target placement optimization:** Our results suggest that historical performance data and monitoring do not enable adaptive middleware to locate "good spots" in the supercomputer or in the file system. In particular, stragglers are transient: over time, any target may appear as a straggler for some intervals yet return quickly to normal behavior. We expect minimal potential performance gain from optimizing target placement, since local performance behavior is transient and unpredictable.
- **Client placement optimization:** Delivered aggregate output bandwidth is sensitive to location (density) of a job's compute nodes for large bursts, particularly under a static node-to-router mapping policy adopted by Titan in its internal network configuration. These behaviors are stable, suggesting that I/O performance can benefit from optimizing client placement, such as redistributing the client locations via I/O adaptation and aggregation.
- **Diurnality pattern:** There are no predictable "slack" periods on supercomputers such as Titan, in contrast to many service workloads in industry. We find that observed I/O performance is not correlated with days in the week or hours in the day. This property results from the characteristics of HPC applications that may run continuously over days and weeks and produce regular output bursts. It suggests that the maintenance tasks on supercomputer file systems should be well planned to perform continuously at a low rate to minimize the impact on applications.
- **Single-pipeline performance:** We observed that the performance of a single pipeline is low (≤70% of the peak bandwidth of a target). As configured on Titan and many other supercomputers, the write pipelines of large-scale parallel file systems do not allow a single client to obtain the full bandwidth of a storage target. The results suggest that, to maximize client bandwidth, each client can write to multiple files spread across multiple targets, avoiding the straggler exposure of striping.
- **Write-sharing performance:** Write-shared files show significantly lower bandwidth. Lock contention injects bubbles into the Lustre write pipeline. These results suggest that, to improve I/O performance, it is best not to share files across processes. It is worth noting that this approach will result in a higher number of output files, which, in return, might hinder postprocessing efficiency.

ACKNOWLEDGMENTS

We are thankful to Verónica G. Vergara for her help on the preparation of HACC IO; to Dustin Leverman for his help on understanding the I/O routers and network topology in Titan; and to many other NCCS staff for answering numerous questions on Jaguar and Titan.

REFERENCES

[1] Argonne National Laboratory. 2018. Retrieved November 9, 2019 from Darshan: HPC I/O Characterization Tool. http://www.mcs.anl.gov/research/projects/darshan.

[2] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. 2011. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage* 7, 3, 8–26.

[3] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 2009. 24/7 characterization of petascale I/O workloads. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'09)*. New Orleans, LA, 1–10.

[4] Luis Chacón. 2004. A non-staggered, conservative, finite-volume scheme for 3D implicit extended magnetohydrodynamics in curvilinear geometries. *Computer Physics Communications* 163, 3, 143–171.

[5] C. S. Chang and Susan Ku. 2008. Spontaneous rotation sources in a quiescent tokamak edge plasma. *Physics of Plasmas* 15, 6, 062510.

[6] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. Liao, K. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. Yoo. 2009. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery* 2, 1, 015001.

[7] Yanpei Chen, Kiran Srinivasan, Garth Goodson, and Randy Katz. 2011. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. Cascais, Portugal, 43–56.

[8] Y. Cui, K. Olsen, T. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, J. Levesque, S. Day, and P. Maechling. 2010. Scalable earthquake simulation on petascale supercomputers. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. Washington, DC. 1–20.

[9] David A. Dillow, Galen M. Shipman, Sarp Oral, Zhe Zhang, and Youngjae Kim. 2011. Enhancing I/O throughput via efficient routing and placement for large-scale parallel file systems. In *Proceedings of the 30th IEEE International Performance Computing and Communications Conference (IPCCC'11)*. Orlando, FL, 21–29.

[10] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. 2014. Omnisc'IO: A grammar-based approach to spatial and temporal I/O patterns prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. New Orleans, LA, 623–634.

[11] Matt Ezell, David Dillow, Sarp Oral, Feiyi Wang, Devesh Tiwari, Don Maxwell, Dustin Leverman, and Jason Hill. 2014. I/O router placement and fine-grained routing on Titan to support Spider II. In *Proceedings of the Cray User Group Conference (CUG'14)*. Lugano, Switzerland, 1–6.

[12] Youngjae Kim and Raghul Gunasekaran. 2014. Understanding I/O workload characteristics of a peta-scale storage system. *The Journal of Supercomputing* 71, 3, 761–780.

[13] Youngjae Kim, Raghul Gunasekaran, Galen M. Shipman, David A. Dillow, Zhe Zhang, and Bradley W. Settlemyer. 2010. Workload characterization of a leadership class storage cluster. In *Proceedings of the 5th Petascale Data Storage Workshop (PDSW'10)*. New Orleans, LA, 1–5.

[14] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney. 2003. Grid-based parallel data streaming implemented for the Gyrokinetic Toroidal Code. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'03)*. Phoenix, AZ, 24–36.

[15] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. 1986. VAXcluster: A closely-coupled distributed system. *ACM Transactions on Computer Systems* 4, 2, 130–146.

[16] S. Ku, C. S. Chang, M. Adams, J. Cummings, F. Hinton, D. Keyes, S. Klasky, W. Lee, Z. Lin, S. Parker, and the CPES team. 2006. Gyrokinetic particle simulation of neoclassical transport in the pedestal/scrape-off region of a tokamak plasma. *Journal of Physics* 46, 1, 87–91.

[17] Julian Kunkel, Michaela Zimmer, and Eugen Betke. 2015. Predicting performance of non-contiguous I/O with machine learning. In *Proceedings of the International Conference on High Performance Computing (ISC'15)*. Frankfurt, Germany, 257–273.

[18] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. 2009. I/O performance challenges at leadership scale. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing Networking, Storage and Analysis (SC'09)*. Portland, OR, 40–52.

[19] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, et al. 2014. Hello ADIOS: The challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience* 26, 7, 1453–1473.

[20] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. 2009. Adaptable, metadata-rich I/O methods for portable high performance I/O. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS'09)*. Rome, Italy, 1–10.

[21] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. 2010. Managing variability in the I/O performance of petascale storage systems. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. Washington, DC, 1–12.

[22] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. 2015. A multiplatform study of I/O behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'15)*. Portland, OR, 33–44.

[23] Sandeep Madireddy, Prasanna Balaprakash, Phil Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan M. Wild. 2018. Machine learning based parallel I/O predictive modeling: A case study on Lustre file systems. In *Proceedings of the International Conference on High Performance Computing*. Hyderabad, India, 184–204.

[24] Ryan McKenna, Stephen Herbein, Adam Moody, Todd Gamblin, and Michela Taufer. 2016. Machine learning predictions of runtime and IO traffic on high-end clusters. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'16)*. Taipei, Taiwan, 255–258.

[25] David A. Nowark and Mark Seager. 1999. ASCI terascale simulation: Requirements and deployments. In *Oak Ridge Interconnect Workshop (ASCI-00-003.1)*. Oak Ridge, TN, 1–15.

[26] Oak Ridge National Laboratory. 2018. HACC. Retrieved November 9, 2019 from https://www.olcf.ornl.gov/caar/hacc/.

[27] Sarp Oral, Feiyi Wang, David Dillow, Galen Shipman, Ross Miller, and Oleg Drokin. 2010. Efficient object storage journaling in a distributed parallel file system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. San Jose, CA, 143–154.

[28] Hongzhang Shan, Katie Antypas, and John Shalf. 2008. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*. Austin, TX, 42–54.

[29] Hongzhang Shan and John Shalf. 2007. Using IOR to analyze the I/O performance for HPC platforms. In *Proceedings of the Cray User Group Meeting (CUG'07)*. Washington, DC, 1–15.

[30] Galen Shipman, David Dillow, Douglas Fuller, Raghul Gunasekaran, Jason Hill, Youngjae Kim, Sarp Oral, Doug Reitz, James Simmons, and Feiyi Wang. 2012. A next-generation parallel file system environment for the OLCF. In *Proceedings of the Cray User Group Conference (CUG'12)*. Stuttgart, Germany, 1–12.

[31] Galen Shipman, David Dillow, Sarp Oral, and Feiyi Wang. 2009. The Spider center wide file system: from concept to reality. In *Proceedings of the Cray User Group Meeting (CUG'09)*. Atlanta GA, 1–10.

[32] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation (Frontiers'99)*. Annapolis, MD, 182–189.

[33] Yuan Tian, Scott Klasky, Hasan Abbasi, Jay Lofstead, Ray Grout, Norbert Podhorszki, Qing Liu, Yandong Wang, and Weikuan Yu. 2011. EDO: Improving read performance for scientific applications through elastic data organization. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'11)*. Austin, TX, 93–102.

[34] Andrew Uselton, Mark Howison, Nicholas J. Wright, David Skinner, Noel Keen, John Shalf, Karen L. Karavanic, and Leonid Oliker. 2010. Parallel I/O performance: From events to ensembles. In *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS'10)*. Atlanta, GA, 1–11.

[35] Lipeng Wan, Matthew Wolf, Feiyi Wang, Jong Youl Choi, George Ostrouchov, and Scott Klasky. 2017. Analysis and modeling of the end-to-end I/O performance on OLCF's titan supercomputer. In *Proceedings of the 19th IEEE International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS'17)*. Salt Lake City, Utah, 1–9.

[36] Feiyi Wang, Sarp Oral, Galen Shipman, Oleg Drokin, Tom Wang, and Isaac Huang. 2009. Understanding Lustre filesystem internals. *Technical Report ORNL* TM-2009, 117, 1–80.

[37] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. Seattle, WA, 307–320.

[38] Bing Xie. 2017. *Output Performance of Petascale File Systems*. Ph.D. Dissertation. Duke University, Durham, NC.

[39] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. 2012. Characterizing output bottlenecks in a supercomputer. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*. Salt Lake City, UT, 1–11.

[40] Bing Xie, Jeffrey S. Chase, David Dillow, Scott Klasky, Jay Lofstead, Sarp Oral, and Norbert Podhorszki. 2017. Output performance study on a production petascale filesystem. In *HPC I/O in the Data Center Workshop (HPC-IODC'17)*. Frankfurt, Germany, 1–14.

[41] Bing Xie, Yezhou Huang, Jeffrey Chase, Jong Youl Choi, Scott Klasky, Jay Lofstead, and Sarp Oral. 2017. Predicting output performance of a petascale supercomputer. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'17)*. ACM, Washington DC, 181–192.

Received April 2018; revised March 2019; accepted May 2019