Characterizing Output Bottlenecks in a Supercomputer

Bing Xie * Jeffrey Chase* David Dillow† Oleg Drokin‡ Scott Klasky† Sarp Oral † Norbert Podhorszki†
*Duke University
Durham, NC, 27708
Email: {bingxie, chase}@cs.duke.edu
†Oak Ridge National Laboratory
Oak Ridge, TN, 37831
Email: {dillowda, oralhs, klasky, pnorbert}@ornl.gov
‡Intel Corporation
Knoxville, TN, 37919
Email: oleg.drokin@intel.com

***Abstract*—Supercomputer I/O loads are often dominated by writes. HPC (High Performance Computing) file systems are designed to absorb these bursty outputs at high bandwidth through massive parallelism. However, the delivered write bandwidth often falls well below the peak. This paper characterizes the data absorption behavior of a center-wide shared Lustre parallel file system on the Jaguar supercomputer. We use a statistical methodology to address the challenges of accurately measuring a shared machine under production load and to obtain the distribution of bandwidth across samples of compute nodes, storage targets, and time intervals. We observe and quantify limitations from competing traffic, contention on storage servers and I/O routers, concurrency limitations in the client compute node operating systems, and the impact of variance (stragglers) on coupled output such as striping. We then examine the implications of our results for application performance and the design of I/O middleware systems on shared supercomputers.**

I. Introduction

Output performance is crucial to harnessing the computational power of supercomputers. Some HPC applications [1], [2], [3], [4] run on the scale of hundreds of thousands of compute cores and produce terabyte-scale output bursts for intermediate results and checkpointing 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, extending application runtime and compromising system throughput. We find that 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. But these hardware resources are expensive, and supercomputers are designed with a careful balance of I/O and computational capabilities. By the classical Amdahl's 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 [5], 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 the 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 tradeoffs at the application level. This problem has motivated development of adaptive I/O middleware systems, such as ADIOS [6], [7], [8], to present a uniform API to applications and adapt their I/O patterns to the underlying storage system.

This paper characterizes output burst absorption on Jaguar, a 2.33 petaflop Cray XK6 housed at the Oak Ridge Leadership Computing Center (OLCF) at Oak Ridge National Laboratory (ORNL). Storage for Jaguar is provided by Spider [9], the 10 petabyte, 240 GB/s Lustre [10] file system at OLCF. The key contribution of our study is to enhance understanding of performance behaviors for state-of-the art software as currently deployed in a leadership-class facility. One purpose of our study is to inform ongoing development of integrated software stacks for parallel storage including parallel file systems and I/O middleware systems such as ADIOS. In particular, our study is an important step toward quantitative models of storage system performance behaviors for use by I/O middleware systems. Models can guide choices made at the middleware layer, including dynamic adaptation to "cross-traffic" from competing workloads on shared supercomputers.

We use a sampling methodology to address challenges in benchmarking a shared supercomputer. At the time of our study Jaguar was the third-fastest disclosed supercomputer in the world, serving multiple user communities. We are unable to reserve it for exclusive use or replace any part of its system software. We use various configurations of the IOR benchmark [11] to focus traffic on specific stages of the multi-stage write path. We analyze distributions of saturation bandwidths across multiple sample trials in different parts of the machine and at different times. These techniques

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

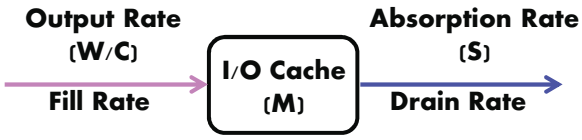

Fig. 1. **The I/O burst model.** An idealized iterative application computes for time C and produces an output burst of size W on each iteration. Each burst enters a client-side I/O cache (write buffer) of size M. The storage system absorbs output at rate S, which varies according to various factors.

enable us to obtain a statistically valid characterization of idealized system performance. They also enable us to quantify the frequency and severity of adverse events that degrade performance and/or cause differential performance behaviors. In some cases the distributions suggest specific causes of the behaviors we observe. Here is a summary of the primary conclusions:

- Storage targets in the Jaguar/Spider deployment deliver close to their full hardware bandwidth under ideal conditions, but delivered performance is often highly variable. Slow storage targets (*stragglers*) limit the aggregate bandwidth of parallel coupled writes. The problem increases with the degree of parallelism. More uniform contention effects in the Jaguar interconnect are clearly visible at larger scales, but the impact of stragglers dominates. These results motivate adaptive selection of compute nodes and storage targets to absorb writes. Avoiding the worst client-target pairs has potential to more than double aggregate bandwidth in highly parallel scenarios.
- Stragglers limit striping bandwidth and reduce the benefits of parallelism. Our results suggest that convoy effects in the Lustre clients exacerbate the problem by delaying issue of writes to striped targets in the presence of stragglers. On Jaguar, the peak per-client bandwidths for writes to a single file are obtained with 4-way striping, but they are less than half of the peak median output bandwidths of clients for the experiments that do not use striping. Striping performance is also sensitive to burst size and requires very large bursts (16GB). These results motivate a looser coupling of parallel I/O (e.g., no striping) for mixed workloads on shared machines.
- As configured on Jaguar, the Lustre write pipelines do not allow a single client to obtain the full bandwidth of a storage target. The results suggest that in the ideal case each client writes to multiple files spread across multiple targets.
- Write-shared files show significantly lower bandwidth. Locking injects bubbles into the Lustre write pipeline and may cause contention in Lustre's metadata service. These results suggest that in the ideal case each client writes to a different set of files.

II. Background

It is estimated that up to 75% of I/O operations in HPC are writes; write-heavy workoads result from state snapshots and defensive writes (e.g., restart files), and most of this data is never retrieved [12], [13]. 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 the rate 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 (rounds) in a loosely synchronous fashion, in which all cores alternately compute and output data. To simplify the model, suppose further 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.

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 machine.

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 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 machine's peak; equivalently, x is an upper bound on mean CPU utilization. With core efficiency x the computation stalls at rate $1 - x$, and 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

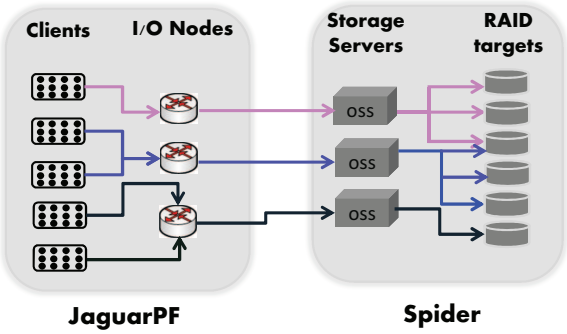

Fig. 2. **The multi-stage write path** in Jaguar/Spider. Writes originate in 16-core client nodes; each client issues RPCs to storage servers through the internal Gemini interconnect and external SION storage network. The server buffers each write and directs it to an attached RAID target. Any client may write to any target; all writes to a given target pass through a single server.

compute time. $W - SC$ is the residual that must be absorbed before the next round 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 round, and $SC < M$, else we are in Case 1.

It is easy to see that C is immaterial and the maximum efficiency of the machine is given by the ratio of the drain rate to the fill rate. Suppose we normalize the fill rate to one unit of output data per second. In Case 2 the drain rate is x with $0 < x < 1$. Then the program produces an excess of $1 - x$ units of output data per second of useful computation, requiring an output stall of $(1 - x)/x$ per second of useful computation to absorb the excess output. The CPU utilization is then the compute time (1) over the total time $(1+(1-x)/x)$. Thus the core efficiency is bounded by x.

Anecdotal evidence from the ADIOS group indicates that output performance observed by real applications is often lower than expected when running on Jaguar at scale. The result distributions reported in this study confirm that delivered performance of output absorption varies within a wide range on Jaguar. The model is useful for predicting the impact of low output bandwidths on application runtime. Applications may compensate by reducing their output frequency, but this presents a tradeoff of recovery times given the failure rates on supercomputers [14].

Although some factors may be unique to Jaguar, we expect that our observations are representative of leadership-scale computing facilities, and I/O performance problems are common. ADIOS implements a variety of techniques to improve I/O performance, and many applications that have encountered these problems in production runs now use ADIOS, e.g., S3D [1], XGC [15] and M8 earthquake simulation[16]. For example, ADIOS enables applications to configure their output buffer size M. 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.

A. Lustre on Jaguar and Spider

The Lustre software deployed on Jaguar is a widely used open-source parallel file system. Lustre runs on about half of the top 30 disclosed supercomputers. This section introduces terms and concepts used in the performance study, summarizing from [17] and other sources on Lustre and Jaguar.

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 direct-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.

Figure 2 depicts the Lustre write path as it is configured on Jaguar and Spider, a center-wide file store hosting a group of Lustre file systems shared across Jaguar and other computing facilities in the center. A Cray Gemini 3D torus interconnect supports messaging among the nodes. The compute nodes access external storage through 192 I/O nodes, which are also attached to the Gemini interconnect. The I/O nodes bridge the internal interconnect to an external storage network called SION (Service I/O Network), a multi-stage InfiniBand network. The SION network provides access to Spider, which comprises 192 storage servers, each mediating access to 7 RAID storage targets (block device LUNs).

Each Lustre write originates with a system call from a user process on a compute node. Each Jaguar 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 Object Storage Client or **OSC** to handle file operations and I/O on Lustre file systems. The OSC performs I/O by issuing Lustre RPC calls to storage servers; each storage server runs a Linux system with a Lustre service called Object Storage Service or **OSS**. Each Lustre I/O operation to an OSS is a read or a 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.

Our experiments run on Widow1, a Lustre file system residing on 672 of the Spider storage targets. We measure write bandwidths to files after they are open, so the metadata service affects the results only for the write-shared file tests (Figure 11).

B. 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

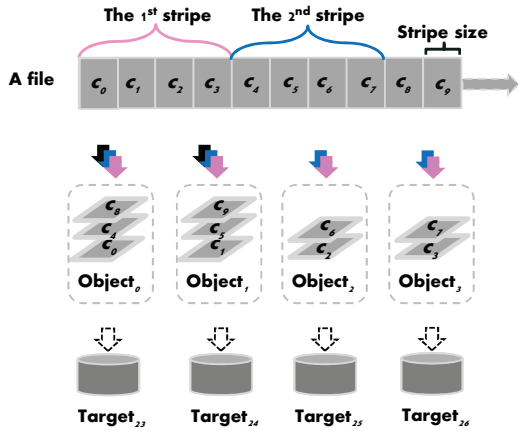

Fig. 3. **Data Striping in Lustre.** Each file is a sequence of chunks of *stripe_size* bytes each, distributed round-robin across a fixed set of *stripe_width* objects created for the file on sequentially ranked targets (OSTs).

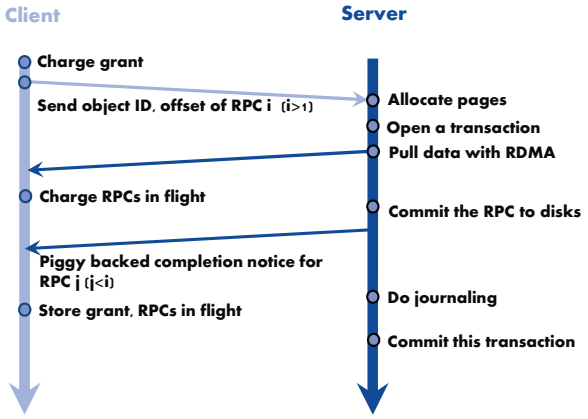

Fig. 4. **RPCs and flow control for asynchronous writes in Lustre.** The client sends an RPC for each write to the object server for the target. The server replies to accept the write and sends a completion notice later when it completes. The client bounds the number of outstanding RPCs and the total size of outstanding writes to each target (a configurable "grant").

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 3 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" or *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 [18], which grow a file by appending new fixed-size objects to it.

C. 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 4KB 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 is 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 design challenge is to keep all pipelines flowing whenever there is data to transfer, throttling them just enough to prevent any buffer overflow.

Lustre clients and servers coordinate to control data flow through the pipelines for normal asynchronous writes. For example, on Jaguar 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 4 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 [17].

III. Methodology

This section introduces a statistical benchmarking methodology, including two parts: targeted focus on specific stages of the write pipeline and statistical analysis across multiple trials. We use IOR [11], a flexible synthetic benchmarking tool for parallel file systems with various interfaces and access patterns. We configure IOR to coordinate simultaneous write bursts from multiple processes and report delivered bandwidth after all data reaches the disks. Each run specifies the number of compute nodes, the number of cores per node, the number of output files, striping parameters, and burst sizes and counts.

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, so 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, our runs are subject to interference and noise from competing traffic and other transient system conditions that we cannot foresee or detect [8].

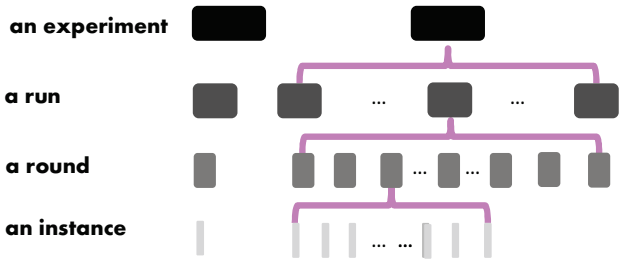

Fig. 5. **Benchmarking Hierarchy**. The graphs plot distributions of results across multiple trials on sampled clients and targets at different times. A round is discarded if conditions change significantly during the round.

To overcome noise and randomness, we obtain a distribution of measures across samples of compute nodes, shared resources and time intervals. 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 5 depicts the structure of an experiment.

- Each IOR job execution is an *instance* of the experiment across a sample of N compute nodes and T storage targets (OSTs). The job scheduler selects the compute nodes. The instance selects the start target randomly. 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 1GB 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.
- A sequence of instances of an experiment is called a *round*. The instances within each round vary the value of a single parameter across a sequence of values; all other parameters are fixed across all instances of each experiment. We submit one instance at a time to the job scheduler, and wait a minimum of 5 seconds after the instance completes before submitting the next instance.
- Each *run* of an experiment is a sequence of identical rounds. We observe that traffic on Jaguar tends to vary little within a run, but may vary significantly across runs.

Each experiment produces a set of points, each giving the output bandwidth measured for one instance. The graphs use box plots to display the quantile distribution and "whiskers" of outliers of sample points. By focusing on the distributions we can quantify the noise and distinguish fundamental performance behaviors from the noise.

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 the node. We use an estimated peak of 300 MB/s for a Spider target (see Section IV).
- *Effective Aggregate Bandwidth (EAB)* is aggregate bandwidth normalized to the peak bandwidth achievable from the number of targets written by the instance.

IV. Output Absorption on Jaguar

This section presents the measurements of burst absorption behavior on Jaguar and Spider (widow1). Our analysis is based on measurements taken in early 2012, after Jaguar was changed from 12 to 16 cores per node, and the interconnect was upgraded to Cray's Gemini NIC from SeaStar 2+. We used half (120 GB/s) of the available storage from Spider (Widow1). The achievable aggregate I/O bandwidth is further limited due to congestion on the Cray 3D torus and the InfiniBand fabric, resulting from the Lustre routing algorithms in use during our measurement period [9].

We present the data in a sequence of graphs with multiple boxplots arranged along an x-axis. Each point in each boxplot corresponds to an instance, as described in the previous section. Each round produces one point for each boxplot in the graph. Most experiments have 50 runs with 5 rounds each. For one long-running experiment we reduce the total number of rounds to 200 (for Figure 13).

A. Pipeline Efficiency

The first experiment evaluates the efficiency of the write pipeline from a single client to a single target, as a function of burst size. Figure 6 gives the results.

Understanding the boxplot graphs. Figure 6 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 III. 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 percential 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 $Q1-1.5*IQR$ respectively. All y-values outside of this whisker range are outliers and are plotted as individual points.

Figure 6 shows that single-pipeline bandwidth is sensitive to burst size, and that 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.

The results suggest that the conservative flow control configuration for output pipelines on Jaguar (e.g., max 8 outstanding RPCs) prevents a single client from obtaining the full bandwidth of the target. This behavior may result from delayed

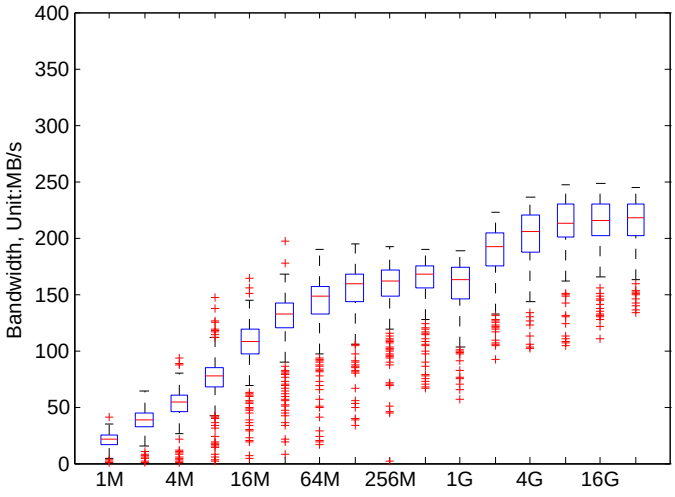

Fig. 6. **Single-pipeline bandwidth** as a function of burst size. The peak median bandwidths are around 75% of the peak median saturation bandwidth of the targets (Figure 7). This graph shows results for a single process on a single core writing a single file on the target. Other results (not shown) indicate that more client processes do not help: the configured pipeline is not deep enough for one client to obtain full bandwidth from a target.

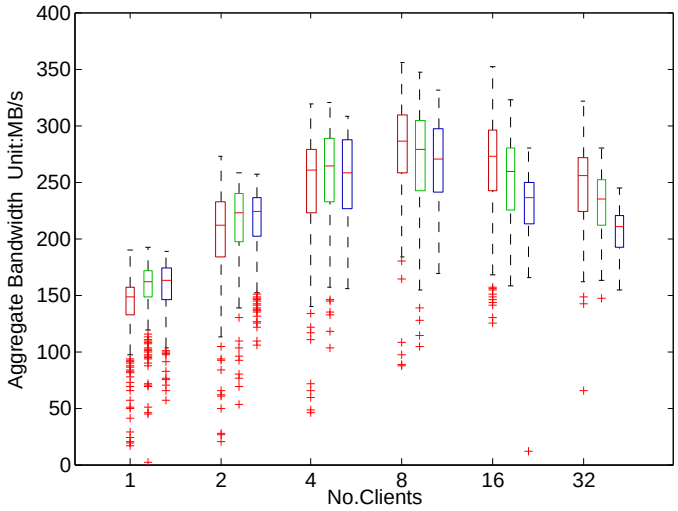

Fig. 7. **Write bandwidth of a target at saturation** as a function of the number of clients. For each fixed number of clients, we plot distributions for the time to absorb a simultaneous burst of 64MB, 256MB and 1024MB from each client (red, green and blue boxes respectively). The small number of outlier points relative to Figure 6 suggests that observed I/O contention occurs mostly within Jaguar itself rather than on SION or Spider.

RPC replies from the targets related to recent enhancements for asynchronous journaling (see [10]).

We ran additional experiments using multiple cores on each client to the same target; the multi-core experiments run multiple single-threaded IOR processes on the same client, each issuing a single output burst to a separate file, synchronized with MPI barriers. However, using multiple cores per client improves bandwidth by at most 5% with asynchronous I/O.

Figure 6 also shows that many trials deliver low bandwidths, presumably due to noise from competing workloads or other system conditions. Results from individual clients show substantial outliers on the low side (3% to 5% of all samples). The next section shows that the incidence of these low outliers decreases as we add more clients (e.g., see Figure 7), suggesting that they are caused by transient severe contention between a client and its I/O router, or within the I/O router, rather than in the SION network, server, or target.

B. Write Bandwidth of OSTs

The next experiments use multiple clients to focus writes on a single target (OST) to measure their peak bandwidths at saturation. The OST saturation experiments follow the template in Figure 8, in which multiple clients write coordinated bursts to the same target. Figure 7 shows the results for large bursts from modest numbers of clients, which generally yield the highest bandwidths. We take 300 MB/s (96*th* percentile) as the peak OST bandwidth in practice, although a few trials deliver close to the hardware bandwidth of the targets under ideal conditions.

In Figure 7 the peak target bandwidths are reasonably stable for all burst sizes. Low-side outliers decrease with both of the number of clients and the size of bursts, suggesting that the results are dominated by contention within Jaguar rather than contention at the target or its disk system.

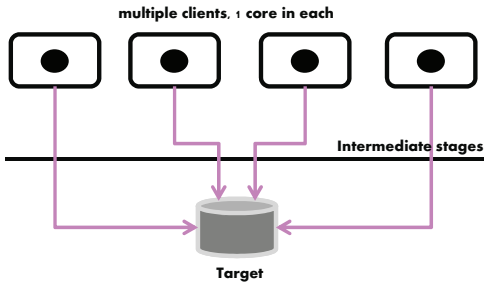

Fig. 8. **Template for target saturation experiments.** Multiple client processes focus simultaneous output bursts (with 64M, 256M, 1024M burst sizes) on a single OST to measure the peak bandwidth of the target at saturation. Each process writes a different file on the OST.

The storage servers (OSS) in the Jaguar/Spider deployment deliver less than the nominal aggregate bandwidth of their attached targets, due at least in part to design choices in provisioning the network. Individual applications are unlikely to observe a bottleneck because the interleaved OST numbering places sequentially numbered targets on successive servers. Thus the default policies stripe each file across the maximal number of servers. We do not quantify it in this paper.

C. Output Bandwidth of Compute Nodes

The next experiments probe the output bandwidth observed by a client writing simultaneously to multiple targets. These experiments test *fan out* parallelism in which the client manages concurrent pipelines to multiple targets. We compare two forms of fan-out parallelism: (1) writes to a single striped file and (2) simultaneous 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 completion notice by pushing more output into the pipeline to

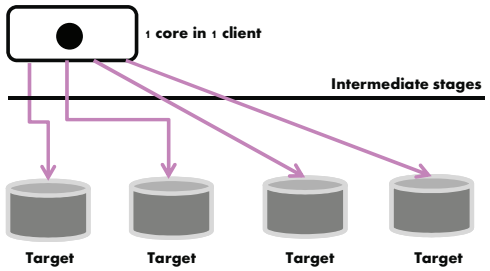

Fig. 9. **Template for striping bandwidth experiment.** A single process issues a write burst to a single file that is striped across multiple OSTs (*stripe_count* targets) at a granularity (*stripe_size*) of 1MB, which is ideal. the burst sizes are separately 1GB, 4GB and 16GB.

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 9: a single process issues asynchronous write bursts to a file striped across a varying number of storage targets (*stripe_count*), with burst sizes of 1GB, 4GB and 16GB. The chunk size is 1 MB. We determined that the 1MB size is a good choice based on other experiments not reported here: delivered bandwidth with striping is insensitive to stripe size up to 32MB, at which point it begins to decline.

Figure 10 shows the measured bandwidths from this experiment, yielding a peak bandwidth of 518.46 MB/s. This peak striping bandwidth is substantially lower than the peak of 800-950 MB/s that we might have expected by extrapolating from Figure 6. The result also indicates that 16GB bursts always obtain the highest observed bandwidth, suggesting that large bursts can use the coordinated pipelines more efficiently. Additionally, increasing stripe width beyond four targets does not increase bandwidth for any burst size. We presume that this effect results from the higher likelihood of encountering straggler OSTs with wider stripes: the bandwidth of striping is gated by the slowest target, offsetting the benefits of higher parallelism. Stragglers are discussed below.

To determine whether the limitation is related to striping, the next experiment writes independent files on multiple targets, following the template in Figure 12. Multiple IOR processes on the same client node write synchronized bursts to multiple unstriped files. Each process writes to a different file on a different OST from the others. We expect that each process executes on a separate core.

Figure 13 shows the results. The client node obtains substantially higher bandwidth than it does using striping: 1GB, 4GB, 16GB overall write size (16 cores with 64MB, 256MB and 1GB each) can all obtain more than 1200MB/s, more than twice the peak bandwidth obtained by the identical write size with a fan-out of four targets. Hence, the results suggest that striping delivers substantially lower output bandwidths than the compute node is capable of. We discuss this further below.

Figure 13 also indicates that a single client performs well when issuing independent pipelines and scales up to 14 cores.

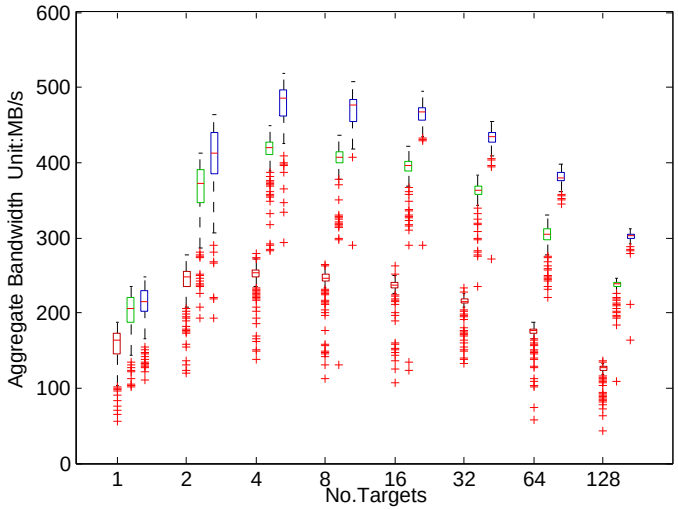

Fig. 10. **Single-client striping bandwidth** as a function of the size of bursts or *stripe_count*. For a fixed number of targets, we issue 1GB, 4GB and 16GB bursts and represent their results with the red, green and blue boxplots respectively. The maximum bandwidth obtained in this experiment is 518.46MB/s, which is obtained by 16GB burst with four targets.

D. Many-Pairs Bandwidth and Stragglers

To measure aggregate I/O bandwidths achievable, we ran at-scale tests using multiple processes on different clients, simultaneously writing to varying numbers of targets, Figure 14 illustrates the template for this experiment. Each client writes to a different target, and we vary the number of client-target pairs. At the largest scale we use 672 compute nodes to write to all 672 targets in widow1. This experiment uses a burst size of 64 MB.

Figure 15 and Figure 16 summarize the results. Although the raw peak bandwidths 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 time for the pairs. The bursts for all pairs are synchronized, and the time interval for the aggregate bandwidth computation is the completion 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. The impact of stragglers grows rapidly as we increase the number of pairs. Stragglers may be caused by bottlenecks in the interconnect, and not necessarily in the targets themselves.

The straggler phenomenon also partly explains the reduced bandwidth of striping, relative to bandwidth obtained by issuing bursts to multiple independent files. The striping pipeline is gated by the slowest of the *stripe_count* targets employed for a striped file.

To quantify the impact of stragglers, Figure 17 plots the cumulative distribution of delivered bandwidth across all client-target pairs for each instance of the experiment. To make the data more descriptive we convert the bandwidth of each pair to the completion time.

In all cases, more than 95% of the synchronized bursts

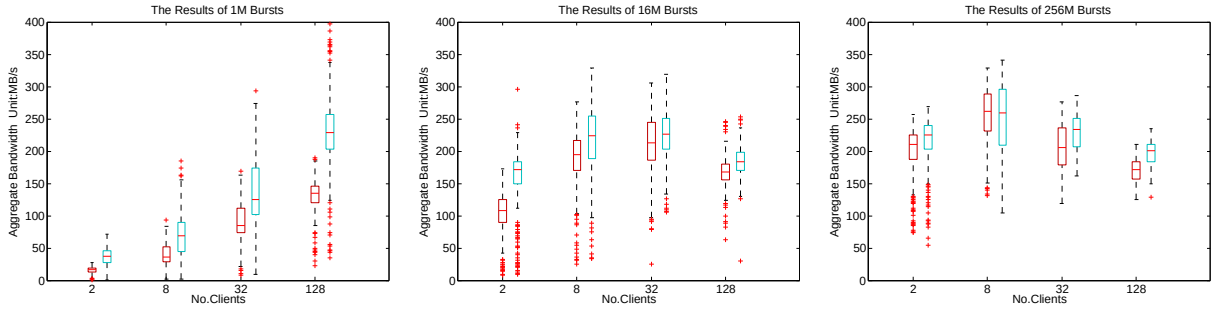

Fig. 11. **The bandwidth of a single target with a shared file** as a function of the number of clients. Three subfigures separately represent the results of 1M, 16M and 256M bursts. In each figure, the red boxes are the results of the synchronized bursts to a single file on the target; the green boxes 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. Write sharing reduces delivered bandwidth by up to a factor of two, due to output pipeline bubbles while a client waits to acquire the necessary locks.

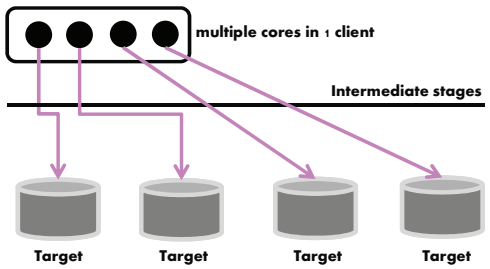

Fig. 12. **Template for client saturation experiment.** Multiple cores (processes) write a coordinated simultaneous burst to multiple targets. The sizes of bursts are respectively 64MB, 256MB and 1024MB. This experiment probes the output limitations of the client for normal asynchronous I/O, but without the complexity of striping.

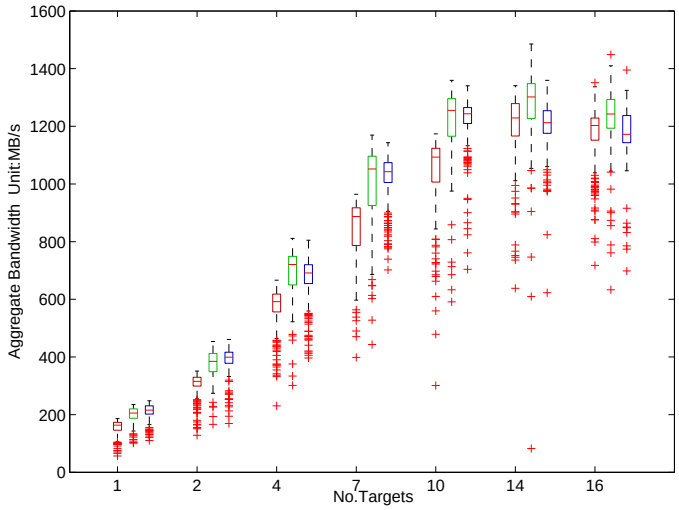

Fig. 13. **Output bandwidth of a compute node** as a function of the number of cores (process-OST pairs) employed, using the template of Figure 12. The client obtains much higher bandwidths by writing multiple independent files than it does using Lustre striping (Figure 10). This result suggests that striping increases the impact of stragglers, e.g., stragglers block the write pipeline and slow the issue of writes to the faster targets (a form of convoy effect).

complete in 2 seconds, but almost every trial has a tail of stragglers. Other pairs are idle while waiting for the stragglers to finish. Aggregate bandwidth is computed over the entire run, which is the completion time of the slowest straggler. As the number of pairs increases, the completion time of the stragglers also increases substantially. Using all 672 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.

These results reflect substantial problems with load balancing in large shared production machines, motivating a looser coupling of parallel I/O pipelines and dynamic selection of compute nodes and targets based on system conditions.

E. Impact of Write Sharing

The next experiment investigates the bandwidth of a single shared target as a function of the number of clients writing to the file, for both shared files and multiple independent files. The experiments use the template in Figure 8, except that we use burst sizes of 1MB, 16MB and 256MB, and we run the experiment for shared files as well. For the sharing tests each client writes an independent region of the same file.

Figure 11 shows that the observed bandwidths with a write-shared file are always lower than those with independent files with the same burst size. Moreover, with smaller burst sizes the cost increases quickly with larger numbers of clients.

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 4KB 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

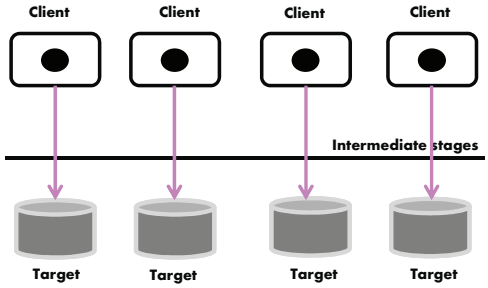

Fig. 14. **Template for the many-pairs experiments.** Each client node runs a process that issues a 64 MB burst to a separate unstriped file on a selected target. Each client uses a different target. The bursts are synchronized. We vary the number of client-target pairs and measure aggregate bandwidth and the bandwidth (or completion time) for each client-target pair.

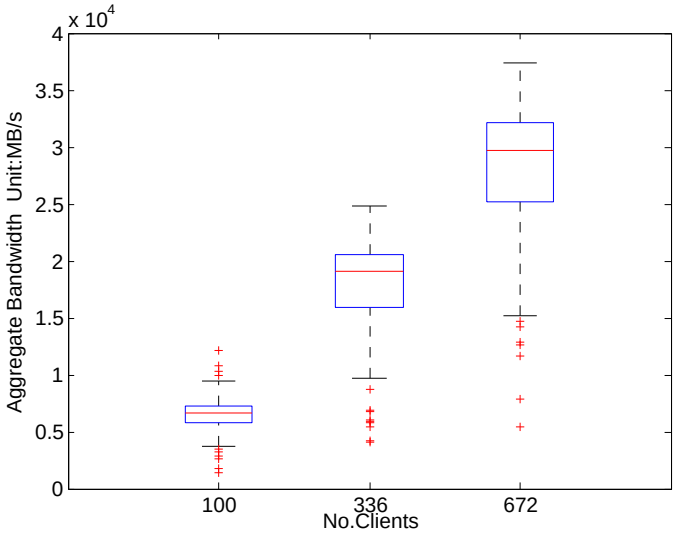

Fig. 15. **Many-pairs bandwidth on Widow1.** Using the experiment template in Figure 14, we obtained about 30 GB/s (36.5 GB/s peak) of aggregate output bandwidth using all 672 targets of Widow1. Aggregate bandwidth is highly variable across multiple runs of this experiment, and varies across more than a factor of three using all pairs.

lock. This locking scheme borrows from the approach used in VAXClusters [19].

V. Other Related Work

Many studies have investigated the performance of HPC file systems. Benchmarking studies commonly take two approaches.

One approach is to measure the performance of file systems under real application workloads. Several influential studies were published in the 1990s [20], [21],[22], [23], [24]. 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 [25] [26].

Uselton et al. [27] also propose a statistical method 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

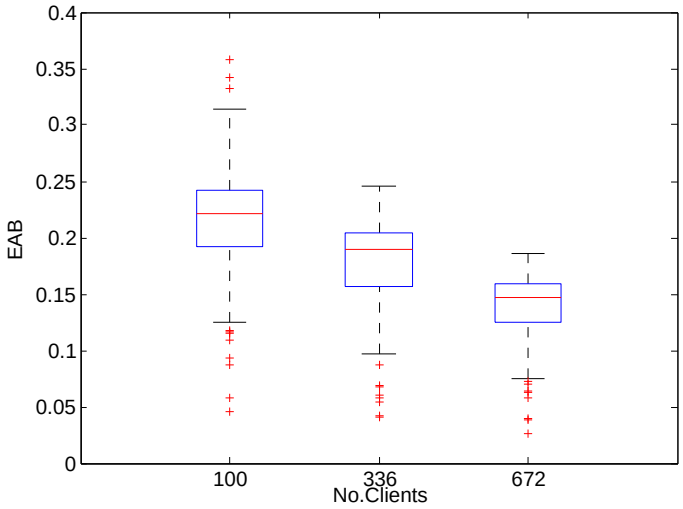

Fig. 16. **EAB of the many-pairs experiment.** This EAB plot shows the same data as Figure 15, but normalized to the nominal potential bandwidth of the targets. With 100 client-target pairs the delivered bandwidth falls below 40% of what could be achieved if all targets yield their expected bandwidths. The results degrade with larger numbers of pairs. Figure 17 shows that interconnect bandwidth is a factor, and stragglers gate the delivered bandwidth.

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, and locate write absorption bottlenecks that influence design choices and configuration choices for adaptive middleware and HPC applications.

A second approach is to stress the file systems 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 FLASH I/O, IOR, BTIO benchmark, etc. 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 [28] 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 are 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 the production runs and with the consideration of competing workloads.

Earlier studies also use configurations of the IOR benchmark to analyze the behavior of HPC systems [29] [11]. The recent paper by Kim et al. [30] also collects I/O performance from Jaguar. That study is complementary to ours: they report monitoring data from the storage servers showing the combined workload on the machine. We focus on the behavior observed by individual jobs, and the impact of write patterns

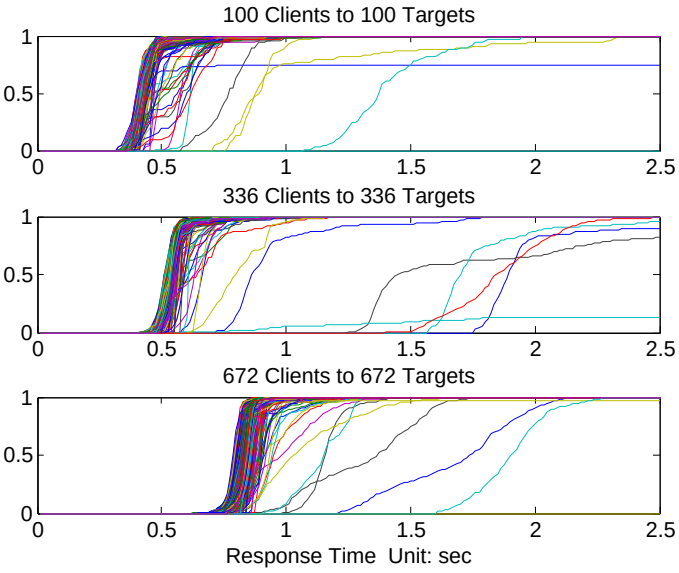

Fig. 17. **CDF of completion times for the instances of the many-pairs experiment.** Each CDF has 250 lines, one for each trial of the instance. Each line shows the distribution of completion times for the pairs of each trial. Each line for the three instances has 100, 336, and 672 points respectively. 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.

and I/O configuration choices.

VI. 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 how the application and I/O middleware uses the storage system APIs. Our study offers a methodology to predict the impacts that result from hardware limitations and file system configuration in a particular facility, by configuring IOR to stress each stage of the write pipeline in turn across a range of parameters.

We apply this approach to map Lustre filesystem output performance in the Jaguar/Spider facility. The measured distributions also quantify the frequency and severity of contention and other transient system conditions. The impacts of these factors are less predictable. Their prevalence motivates adaptive responses in the I/O middleware layer, and structuring choices to loosen the coupling of parallel I/O to maximize the benefits of adaptation.

For example, on Jaguar/Spider 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. This structure is robust to modest burst sizes. More importantly, it offers opportunities for adaptive selection of I/O targets to reduce the impact of stragglers, which can reduce delivered output bandwidth by more than a factor of two.

Our study also suggests that delivered bandwidth is increasingly sensitive to concurrency bottlenecks in the client file system software, which faces the challenge of driving concurrent output pipelines to multiple storage targets. This challenge grows with the number of cores and storage targets.

Acknowledgements: This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

References

[1] L. Chacón, "A non-staggered, conservative, finite-volume scheme for 3D implicit extended magnetohydrodynamics in curvilinear geometries," *Computer Physics Communications*, vol. 163, no. 3, pp. 143 – 171, Nov. 2004.

[2] C. S. Chang and S. Ku, "Spontaneous rotation sources in a quiescent tokamak edge plasma," *Physics of Plasmas*, vol. 15, no. 6, June 2008.

[3] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo, "Terascale direct numerical simulations of turbulent combustion using S3D," *Computational Science and Discovery*, vol. 2, no. 1, Jan. 2009.

[4] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney, "Grid-based parallel data streaming implemented for the Gyrokinetic Toroidal Code," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC '03)*, Phoenix, AZ, Nov. 2003.

[5] D. Nowak and M. Seagar, "ASCI terascale simulation: requirements and deployments," in *ASCI*, Nov. 1999.

[6] Y. Tian, S. Klasky, H. Abbasi, J. Lofstead, R. Grout, N. Podhorski, Q. Liu, W. Yandong, and Y. Weikuan, "EDO: improving read performance for scientific applications through elastic data organization," in *Proceedings of IEEE Cluster 2011*, Austin, TX, Sep. 2011, pp. 93–102.

[7] J. Lofstead, Z. Fang, S. Klasky, and K. Schwan, "Adaptable, metadata-rich I/O methods for portable high performance I/O," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*, Rome, Italy, May 2009.

[8] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing variability in the I/O performance of petascale storage systems," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Washington, DC, Nov. 2010.

[9] D. A. Dillow, G. M. Shipman, H. S. Oral, Z. Zhange, D. Z. Zhang, and Y. Kim, "Enhancing I/O throughput via efficient routing and placement for large-scale parallel file systems," in *Performance Computing and Communications Conference (IPCCC)*, Nov. 2011.

[10] S. Oral, F. Wang, D. Dillow, G. Shipman, R. Miller, and O. Drokin, "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, Feb. 2010, pp. 143–154.

[11] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*, Austin, TX, Nov. 2008.

[12] Y. Kim, G. R., S. G.M., D. D. Z. Zhang, and B. Settlemyer, "Workload characterization of a leadership class storage cluster," in *Petascale Data Workshop (PDSW)*, New Orleans, LA, Nov. 2010.

[13] A. Uselton, K. Antypas, D. M. Ushizima, and J. Sukharev, "File system monitoring as a window into user I/O requirements," in *Proceedings of the 2010 Cray User Group Meeting*, Edinburgh, Scotland, May 2010.

[14] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN '06)*, Philadelphia, PA, June 2006, pp. 249–258.

[15] 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, "Gyrokinetic particle simulation of neoclassical transport in the pedestal/scrape-off region of a tokamak plasma," *Journal of Physics*, vol. 46, no. 1, 2006.

[16] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, J. Levesque, S. M. Day, and P. Maechling, "Scalable earthquake simulation on petascale supercomputers," in *Proceedings of the 2010 ACM/IEEE International Conference*

for High Performance Computing, Networking, Storage and Analysis, ser. SC'10, Washington, DC, Nov. 2010.

[17] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang, "Understanding Lustre filesystem internals," *Technical Report ORNL/TM-2009/117*, Apr. 2009.

[18] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, Nov. 2006, pp. 307–320.

[19] N. P. Kronenberg, H. M. Levy, and W. D. Strecker, "VAXcluster: a closely-coupled distributed system," *ACM Trans. Comput. Syst.*, vol. 4, no. 2, pp. 130–146, May 1986.

[20] A. L. N. Reddy and P. Banerjee, "A study of I/O behavior of perfect benchmarks on a multiprocessor," *SIGARCH Comput. Archit. News*, vol. 18, no. 3a, pp. 312–321, May 1990.

[21] G. R. Ganger, "Generating representative synthetic workloads: an unsolved problem," in *in Proceedings of the Computer Measurement Group (CMG) Conference*, Nashville, TN, Dec. 1995, pp. 1263–1269.

[22] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. L. Best, "File-access characteristics of parallel scientific workloads," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 7, no. 10, pp. 1075–1089, 1996.

[23] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina, "Architectural requirements of parallel scientific applications with explicit communication," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, May 1993.

[24] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed, "Input/Output characteristics of scalable parallel applications," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (SC '95)*, San Diego, CA, Dec. 1995, pp. 59–89.

[25] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *IEEE International Conference on Cluster Computing (Cluster '09)*, New Orleans, LA, Sep. 2009.

[26] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *Trans. Storage*, vol. 7, no. 3, Oct. 2011.

[27] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker, "Parallel I/O performance: from events to ensembles," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)"*, Atlanta, GA, Apr. 2010.

[28] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*, Portland, OR, Nov. 2009.

[29] H. Shan and J. Shalf, "Using IOR to analyze the I/O performance for HPC platforms," in *in Cray Users Group Meeting (CUG)*, Washington, DC, May 2007.

[30] Y. Kim, Gunasekaran, D. R.and Shipman, G.M.and Dillow, Z. Zhang, and B. Settlemyer, "Workload Characterization of a Leadership Class Storage Cluster," in *Petascale Data Storage Workshop (PDSW)*, Nov 2010.