Predicting Output Performance of a Petascale Supercomputer

Bing Xie
Duke University
bingxie@cs.duke.edu

Yezhou Huang
Duke University
yhuang@cs.duke.edu

Jeffrey S. Chase
Duke University
chase@cs.duke.edu

Jong Youl Choi
Oak Ridge National Laboratory
choij@ornl.gov

Scott Klasky
Oak Ridge National Laboratory
klasky@ornl.gov

Jay Lofstead
Sandia National Laboratories
gflofst@sandia.gov

Sarp Oral
Oak Ridge National Laboratory
oralhs@ornl.gov

ABSTRACT

In this paper, we develop a predictive model useful for output performance prediction of supercomputer file systems under production load. Our target environment is Titan—the 3rd fastest supercomputer in the world—and its Lustre-based multi-stage write path. We observe from Titan that although output performance is highly variable at small time scales, the mean performance is stable and consistent over typical application run times. Moreover, we find that output performance is non-linearly related to its correlated parameters due to interference and saturation on individual stages on the path. These observations enable us to build a predictive model of expected write times of output patterns and I/O configurations, using feature transformations to capture non-linear relationships. We identify the candidate features based on the structure of the Lustre/Titan write path, and use feature transformation functions to produce a model space with 135,000 candidate models. By searching for the minimal mean square error in this space we identify a good model and show that it is effective.

KEYWORDS

Petascale supercomputer, Output performance, Linear regression

1 INTRODUCTION

Supercomputers and their I/O systems are built to host HPC (High Performance Computing) applications. These applications perform a variety of analyses, experiments and simulations [4–7, 19] from different scientific domains. Typical HPC applications issue periodic bursts of output to the file system for intermediate results and checkpointing; these outputs may total a terabyte or more over a typical run of the application [4, 5, 7, 21]. US-DOE (Department of Energy) leadership computing facilities report that HPC applications generate hundreds of petabytes of science data per year and estimate that this rate will exceed an exabyte per year by 2018 [16]. Trends suggest that HPC applications are likely to generate more and larger output bursts.

Consequently, understanding performance of output burst absorption is crucial for these codes. Many HPC applications are loosely synchronous or bulk synchronous: they execute a sequence of iterations (e.g., for simulation timesteps) interspersed by barriers. These applications often initiate a synchronous output burst between iterations, and then wait for the burst to complete before resuming computation for the next iteration. This pause provides a consistent output image for checkpoints, and in certain other cases when double-buffering requires too much memory to be practical. Therefore, CPUs are left idle during the synchronous burst. As a case study we consider XGC [5], an important code that simulates magnetic confinement of plasma in future fusion reactor designs (§2.1). In practice output times typically comprise 7-20% of XGC's total (wall clock) execution time.

The HPC community recognizes the importance of I/O output performance. There are many efforts to address it. For example, [3, 26] summarize I/O access patterns across scientific domains and supercomputing platforms; [22, 25, 40] investigate behaviors of petascale file systems under various access patterns and system conditions. Others propose techniques, tools and middleware systems to improve I/O performance by reducing metadata operations [23, 33], aggregating/striping/reordering data streams [8, 16, 20, 24], and other techniques [13, 32, 35]. These works on I/O performance range from quantitative analysis on various targets to optimization techniques at various levels.

This paper presents an analysis to derive principles from quantitative models and to specify tradeoffs of techniques across settings and conditions. A key obstacle to meeting this goal is the dynamic nature of our production supercomputer environment. Applications have full ownership of the compute nodes assigned to them, but the interconnect and I/O system are shared and provide no performance isolation. The resulting noise complicates the task of modeling or predicting I/O performance.

We report observations from Titan [13]—a production supercomputer at ORNL—and its Lustre parallel file system (Spider 2/Atlas) to show that although output bandwidth of an HPC file system is highly variable, for a given burst pattern the mean absorption rate across compute nodes and storage targets is surprisingly stable

HPDC'17, June 26-30, 2017, Washington, DC, USA
© 2017 ACM. 978-1-4503-4699-3/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3078597.3078614

over time (§3). We show that this continuity makes it possible to model and predict output performance obtained at a given scale and configuration precisely and accurately (§4). In particular, we conclude:

- For typical well-configured output bursts in Titan the congestion effects are dominated by the primary interconnect, rather than by the I/O system itself.
- At large scale these congestion effects are dominated by self-interference on the compute interconnect rather than by contention from competing workloads. Impacts from competing workloads tend to revert to the mean over relatively short time scales—tens of minutes—so that it does not affect the average absorption rate over an execution (hours or more likely days or weeks).
- The impact of self-interference is determined primarily by the number of *I/O pipes* configured for the burst. An I/O pipe is a logical output path from a task (typically occupying one core of a compute node) to a specific storage target. Well-configured bursts spread their output loads across the storage system so that the target of each I/O pipe is distinct from all other targets in the burst.
- For Titan, output bandwidth at the storage system and the impact of self-interference in the interconnect are predictable from a simple model based on the number of I/O pipes. We gathered data from synthetic probing experiments that emulate output bursts with a variety of configurations and scales on Titan, and use this data to train a regression model to predict the performance of output bursts. We show that the resulting model is precise and accurate in practice (§5).

The output rate model provides useful insight into output behavior on a leadership-class production supercomputer, and can be helpful to auto-configure burst parameters such as striping width for best performance. Moreover, the output model enables *predictions of total application run time*, which can assist with supercomputer scheduling. ORNL requires that submitted jobs include an estimate of wall clock time for use by the scheduler, and kills jobs that exceed their estimates. For XGC and other key applications, good models exist for the computation time for the iterations themselves, but the lack of an I/O model leads researchers to make conservative estimates of job cost ("just double it"). A good model of I/O cost can yield much tighter estimates, which can reduce delays in the job queue and yield more efficient use of resources. These benefits occur in part because the system may schedule mission-critical jobs and maintenance tasks in advance, and it may delay dispatching a job whose estimated run time indicates that the job would conflict with an advance reservation.

Although our current study focuses on a petascale file system, we expect that our approach is also applicable to exascale systems. The current Titan machine approaches an exascale deployment.

2 OVERVIEW

Our benchmarks on Titan and on its predecessor Jaguar [40] show that output performance is highly variable due to contention in the machine over time and across groups of compute nodes, due to transient system conditions resulting from the production workload. The delivered write bandwidth may be distributed across a wide range at small time scales (Figure 5). This makes it a challenge to predict output performance on production supercomputers. However, our benchmarks show that periods of severe congestion are generally of short duration; they are rarely more than tens of minutes—two to three orders of magnitude lower than application run times (e.g., days and weeks). As a result, the output burst times that an application observes across its entire execution are highly likely to regress to the mean. Therefore, in this study, we choose to model the *mean* output burst times as a basis to predict application performance (§3.2).

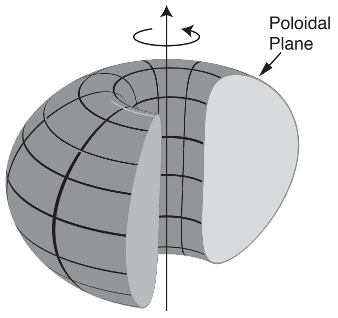


Figure 1: The XGC decomposition. A 3D tokamak is partitioned to D planes and a plane is partitioned to P subspaces. In an XGC run, DP tasks produce DP synchronous bursts for state snapshots on every T_1 iterations; D tasks represent D planes to produce three types of synchronous bursts for diagnostic analysis on every T_2, T_3, and T_4 iterations respectively.

This section presents background to support two key points: (1) the mean write time is effective to address output behavior of a large group of scientific codes that write fixed-size bursts iteratively (§2.1); (2) we can predict mean absorption times for these output bursts based on data from synthetic benchmarks that isolate elements of the multi-stage write path across a range of I/O configurations (§2.2).

2.1 Output Behavior of Scientific Applications

It is widely observed from various supercomputer platforms that 50% — 60% of I/O requests are writes [2, 17]. This section discusses properties of write-intensive applications.

A large group of supercomputer applications are numerical analyses or simulations that compute over iterations or timesteps. We use XGC code as an example to illustrate their behavior.

2.1.1 XGC Code. XGC is a gyrokinetic particle-in-cell code used to simulate tokamak fusion reactor designs, focusing on the multiscale 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. Figure 1 depicts the decomposition of a typical XGC run.

A run consists of DP identical tasks computing across L iterations and synchronizing at the end of each iteration. Each task runs as a process on a different core to solve a fixed set of gyrokinetic equations across iterations on particles in a subspace. Specifically, in each iteration particles are "pushed" by a governing gyrokinetic Hamiltonian equation and gathered onto F grid points of a discrete grid, where the gyrokinetic Poisson or gyrokinetic Maxwell's equations are solved. The computation time for each iteration (t_c) is fixed and predictable from (D, E, F, P).

During the run, each task produces a burst (B_1) as its state snapshot at the end of every T_1 iterations; a fixed set of D of the DP tasks produce summary outputs for the D planes ($D < P$), generating three types of bursts (B_2, B_3, B_4) for diagnostic analysis at the end of every T_2, T_3 and T_4 iterations respectively. For all four types of bursts, the data for each burst is stored as a file with a unique file name. When writing a burst, the entire execution is stalled until all data reaches disks.

The burst size of B_1 is bounded by the number of particles in a subspace (E/P): it preserves fixed-size byte-level data for the grid particle distribution function and numerical values of features (e.g., electric potential, plasma density, temperature) of particles in a subspace. The burst sizes for $B_2 - B_4$ are bounded by the number of grid points reported by the set of D tasks (F/D): it contains numerical values of features of particles on grid points. During an XGC run, particles may die or move, but in most cases any data imbalance across snapshots or grid points is ignorable: the burst sizes of $B_1 - B_4$ are constant and predictable. We observed in practice that for XGC runs common per-core burst sizes of state snapshots range from 500MB to 1.2GB; for diagnosis outputs the burst sizes range from 1MB to 400MB.

In an XGC run, E and F are given parameters as part of a problem setting; D, P, L, $T_1 - T_4$ are configurable parameters; t_c and burst sizes of $B_1 - B_4$ are predictable when the above-mentioned parameters are determined; all parameters are fixed and known before the run starts. Thus, the end-to-end execution time of a run (t_{run}) can be computed by summing up times consumed by computation and four types of bursts across iterations.

Besides t_c, if write times of $B_1 - B_4$ are also predictable, t_{run} can be estimated before a run. The prediction result can help scientists control the cost of their write operations. For example, scientists usually want to keep the time consumption for state snapshots within 10% of the execution time of an application run. But we observed from XGC production runs that the state snapshot cost varies from 7% to 20% of the entire application execution time. Therefore, given the predicted compute time and output times the state snapshot cost can be controlled by choosing T_1 appropriately.

2.1.2 Properties of Iterative Codes. XGC is representative of a large group of *static iterative scientific codes* that take iterative computation structures and produce periodic and predictable bursts [21, 22]. In general, these applications perform iterative scientific simulations on a static multi-dimensional space and produce one or more types of fixed-size bursts periodically.

Another group of scientific codes, e.g., AMR (Adaptive Mesh Refinement) [1], may vary computation space across iterations and produce different size bursts at different iterations accordingly. We call this group of codes *dynamic iterative scientific codes.*

This study focuses on output performance prediction and optimization for static iterative scientific codes. The principles also apply to dynamic codes to the extent that their burst sizes are fixed and predictable.

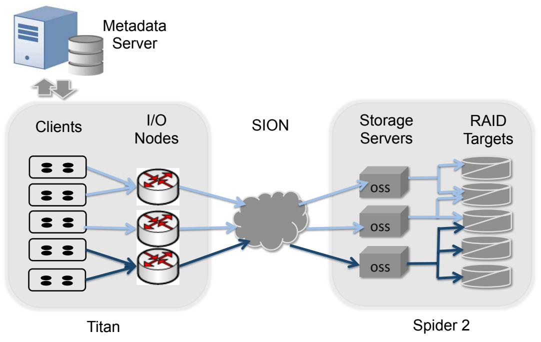


Figure 2: Titan and the Lustre File System (Spider 2)

Similar to our analysis on XGC code in §2.1.1, presume that each run consumes t_c computation time on each of L iterations and produces a B_i type of burst on every T_i iterations ($i = 1, 2, \ldots$). We observed from the production supercomputer Titan and report in §3.2 that although output performance of the file system is highly variable, the mean write time (t_{B_i}) of the burst type B_i is fixed and predictable. As a result, the end-to-end execution time (t_{run}) of an application run can be predicted.

2.2 Titan and its Lustre File System

Our target environment is Titan, the 3rd fastest supercomputer in the world. Titan is a Cray XK7 supercomputer hosted at the Oak Ridge Leadership Computing Facility (OLCF) and serving scientists in disciplines such as climate science, chemistry, molecular science, and materials science. Titan's file system, called Spider 2, is based on Lustre, an object-based parallel file system software that is deployed on ~75% of the top 100 systems [36]. Spider 2 has 32 PB of data storage and above 1TB/s peak I/O bandwidth [31]. This section summarizes Titan/Spider 2 based on materials from [13, 30, 39, 40].

Figure 2 depicts the write path of Titan and Spider 2. Titan is composed of 18,688 compute nodes; these nodes are connected by a 3D torus interconnect; each node has a 16-core CPU and a GPU: it runs the Lustre software and serves as both a Metadata Client (**MDC**) and an Object Storage Client (**OSC**).

A Metadata Server (**MDS**) stores metadata of files and objects (e.g., file namespaces and attributes, object IDs) on RAID devices, called Metadata Targets (**MDTs**). A Lustre file system is a single namespace maintained by one MDS on one or more MDTs. As described in §2.1, for scientific codes a write operation produces a group of synchronous bursts with each burst stored as an independent file. Thus, for each operation compute nodes (clients) only communicate with MDS for $file\_create()$, $file\_open()$ and $file\_close()$ at the start and the end of the operation.

Compute nodes access Spider 2 via I/O nodes (routers) that are evenly distributed through the torus interconnect. Titan is configured to connect a compute node to a fixed group of "closest" I/O nodes in the torus by a fine-grained routing policy [18]. Thus, an output operation with more compute nodes is highly likely to spread across more I/O nodes.

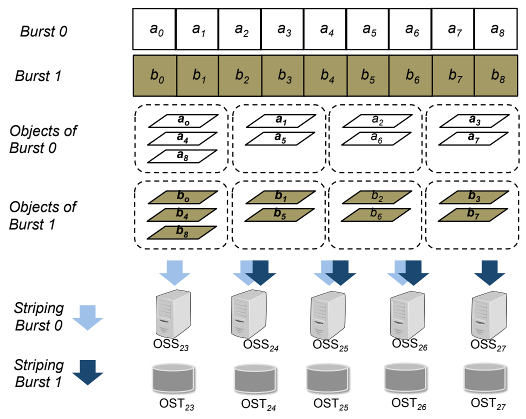

Burst 0
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
Burst 1
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8
Objects of Burst 0
Objects of Burst 1
Striping Burst 0
Striping Burst 1
OSS23 OSS24 OSS25 OSS26 OSS27
OST23 OST24 OST25 OST26 OST27

Figure 3: Striping Bursts/Files for a Write Operation. Users set *stripe_size, stripe_width* and *starting_OST*. Each burst is partitioned into a sequence of chunks with *stripe_size*-byte per chunk; the chunks are distributed round-robin across a fixed set of *stripe_width* objects (Formula 3); the sequentially-numbered bursts are assigned with a sequence of *starting_OSTs* (Formula 2). In this example, the *starting_OST* sequence is $\{OST_{23}, OST_{24}, ...\}$.

I/O nodes link Titan's compute nodes to the Lustre Object Storage Servers (**OSSes**) via a Scalable I/O Network (SION). Each OSS manages 7 Object Storage Targets (**OSTs**), each a RAID array direct-attached to an OSS. To balance the load across OSSes, Spider 2 maps sequentially-numbered OSTs across OSSes in a round-robin fashion. Let M and N represent the total number of OSSes and the total number of OSTs in Spider 2: $N = 7 \times M$. The M OSSes are numbered as $(OSS_0, ..., OSS_i, ..., OSS_{M-1})$; the N OSTs are numbered as $(OST_0, ..., OST_j, ..., OST_{N-1})$. Based on the mapping policy, OST_j is attached to OSS_i:

$$i = j \bmod M \tag{1}$$

For a write operation, each burst is stored as a Lustre file on one or more OSTs: a file is an interleaving of one or more data *objects* with each variable-sized object stored on a different OST. Lustre allows a job to configure three parameters to customize striping: *stripe size*, *stripe width* and *starting OST*: it first partitions a file into a sequence of *stripe_size*-byte data chunks, then spreads them in a sequence of *stripe_width* OSTs from the OST numbered *starting_OST*. In the simplest scheme, all files created by the job start on the same *starting_OST* and are consequently stored on the same sequence of *stripe_width* OSTs. Alternatively, a job may use an MPI-IO primitive to stagger the starting OSTs by offsetting each *starting_OST* by the number of the process that creates the file, as shown in Figure 3. Consider a job with P processes that produces P files $(f_0, ..., f_i, ..., f_{P-1})$, one from each process, and takes OST_* as f_0's *starting_OST*. According to this policy, f_i starts at the jth OST:

$$j = (* + i) \bmod N \tag{2}$$

OSSes and OSTs used in a write operation. According to Formula 2, we can compute the numbers of OSTs and OSSes used in a write operation (Formula 4 and 5). Consider an output pattern that: (1) has P bursts with burst size K; (2) configures *stripe_size, stripe_width* and *starting_OST* as S, W, OST_* respectively; (3) applies the *starting_OST* policy in Formula 2. According to Formulas 1 and 2, the number of OSTs used per burst (N_{per}), the numbers of OSTs (N_{used}) and OSSes (M_{used}) used for this operation are given by:

$$N_{per} = \begin{cases} \lceil K/S \rceil & \text{if } \lceil K/S \rceil < W \\ W & \text{otherwise} \end{cases} \tag{3}$$

$$N_{used} = \begin{cases} P + N_{per} - 1 & \text{if } P + N_{per} - 1 < N \\ N & \text{otherwise} \end{cases} \tag{4}$$

$$M_{used} = \begin{cases} N_{used} & \text{if } N_{used} < M \\ M & \text{otherwise} \end{cases} \tag{5}$$

This study builds models to predict output performance on Titan/Atlas2: one of two equal-size partitions of Spider 2. Each partition consists of an MDS with an attached MDT, 144 OSSes and 1008 OSTs, and is configured in default as: *stripe_size*=1MB, *stripe_width=4*, and a randomly chosen *starting_OST*. We adopt the *starting_OST* policy addressed by Formula 2, and use Formulas 4 and 5 accordingly to build and evaluate models (§4 and §5).

3 OUTPUT BEHAVIOR ON TITAN

This section discusses burst absorption behavior of the Titan I/O system (§2.2), focusing on the metadata service and the stages of the data write path. The summary helps us to build a quantitative understanding of output behavior of a production petascale filesystem, which serves as a basis to model output performance of these filesystems.

3.1 Metadata Behavior

We summarize the logs of Spider 2's two metadata servers (MDS1 and MDS2), collected by MDSTrace [27] and reporting 14-day traces from Sept.10 to Sept.23, 2016. There are 3470 log files with 1735 files per metadata server. Each log file reports a trace sampled over a 60-second interval, giving the total number of requests, the number of requests per operation, the max/min request processing time, and the top ranked application by the number of requests [27]. We expect that the logs have sufficient coverage to capture the metadata behavior for most applications at OLCF.

Table 1 summarizes the logs. We conclude that: (1) The MDS load is balanced: each receives ~28K requests/minute (median) and ~32K requests/minute at the maximum. (2) The metadata cost is low and consistent in general: only one logged request had a processing time over 30 seconds, with 99.5% of requests completing within 6.09 seconds on MDS1 and within 0.9 second on MDS2 respectively. (3) The metadata load is diverse: sometimes it is dominated by a single application; sometimes it is shared by concurrent loads. (4) The *file_open()* requests are bursty and often dominate the metadata load: the median numbers of *file_open()* are 14.9K/minute on MDS1 and 21.6K/minute on MDS2 respectively. In some intervals the *file_open()* requests comprise more than 99% of the overall metadata requests.

In this study, we used a synthetic benchmark that stresses file create/open and file writes on Titan/Atlas2 to train and validate

Metadata Server	# Requests			Max request processing time (unit:sec)			# *file_open()*			#Requests from *the top ranked app*		
	max	*Q*0.995	*median*	*max*	*Q*0.995	*median*	*max*	*Q*0.995	*median*	*max*	*Q*0.995	*median*
MDS1	32K	30.6K	28K	16.62	6.09	0.01	30.2K	25.1K	14.9K	29.1K	27.7K	6.2K
MDS2	32.6K	32.2K	27.8K	567.05	0.90	0.006	32.2K	31.9K	21.6K	31.6K	30.9K	1.1K

Table 1: Metadata Behavior on Spider 2. This table summarizes the logs of the MDS1 and MDS2 metadata servers of Spider 2, focusing on four parameters: the total number of requests, the max request processing time, the number of requests for *file_open()*, the top ranked application by the number of requests. For each parameter, we report the max, quantile 0.995 and the median. In summary, the load on the metadata service is balanced and diverse; and response times are small in most cases.

the models. Under heavy load we observed a maximum *file_open()* time=585.38 seconds. As summarized above, this high response time is rare in practice. The goal of our study is to build a predictive model for expected output behaviors in practice with low benchmarking cost. Therefore, we discard the instances that take beyond 30 seconds for *file_open()*. We return to this topic in §5.3.

3.2 Output Behavior of the Other Stages

We conducted profiling experiments using a statistical benchmarking methodology proposed in [40]. This section sketches the methodology, the experiments and relative conclusions.

3.2.1 *Statistical Benchmarking Methodology.*

The output behavior is derived from a set of experiments, each with a sequence of identical IOR *runs* based on *an experiment setting* consisting of a job script and an IOR configuration file. The job script gives the resource requirements (e.g., the number of nodes) and the job execution instructions specifying how to perform a sequence of IOR executions with varying sets of parameter values (Table 2) within the run. A set of parameter values is a *configuration set*. Each run consists of a sequence of identical *rounds*: a round consists of a sequence of IOR executions; each execution is an *instance* that measures the time for a synchronized output burst. The instances in a round have different configuration sets. Therefore, an experiment produces a set of instances for a configuration set, with one instance from each round. These instances run at different times and perhaps on different sets of compute nodes. Algorithm 1 presents the template of the job execution instructions.

Algorithm 1 The job script for IOR executions in an experiment

```
1: number of Rounds: Rounds = 1, 2, 3, ..., r
2: Parameter 1: $P_1 = \{p1_1, p1_2, ..., p1_i, ..., p1_l\}$
3: Parameter 2: $P_2 = \{p2_1, p2_2, ..., p2_j, ..., p2_m\}$
4: for Round: $1 \rightarrow r$ do
5:     for P1: $p1_1 \rightarrow p1_l$ do
6:         for P2: $p2_1 \rightarrow p2_m$ do
7:             execute $IOR - p1_i - p2_j$
8:             sleep 3
9:         end for
10:    end for
11: end for
```

3.2.2 *Experiments.*

For each experiment, we use IOR as a burst generator: it coordinates P processes from m compute nodes with n cores each ($P = m \times n$) to write P bursts to disks. Each process runs on a different core and produces a burst of size K as a new file with *stripe_width=W*. For each instance, we synchronize bursts before *write_start()* and measure the time from the minimum of *write_start()* to the maximum of *write_end()* among bursts. As discussed in §2.2, Lustre-based file systems support three configurable parameters: *stripe_size*, *stripe_width* and *starting_OST*. We found and reported in [40] that *stripe_size* doesn't affect output performance for its values in 1MB — 32MB, or leads to performance degradation for the values above 32MB. Thus, we choose 1MB as *stripe_size* and assign a randomly chosen OST as *starting_OST* across instances for all configuration sets in all experiments.

We conducted six experiments with varying parameters m, n, K, W (Table 2). The first five experiments ran on Titan and Widow1 [12, 13] from Jan. to Dec. 2013 and produced overall 116 configuration sets with 600 instances per set from 200 runs; the sixth experiment (*client-OST pairs 2*) ran on Titan and Atlas2 (§2.2) from May to June 2015 and produced 3 configuration sets with 1545 instances per set from 103 runs. We take two measures per instance: the aggregate bandwidth and the *effective bandwidth*. We assign an instance an effective bandwidth by normalizing its bandwidth to the maximum bandwidth received by identical instances from the same configuration set. The maximum bandwidth (*effective bandwidth=1*) represents the achievable bandwidth of the set under ideal conditions. We ran six types of experiments:

(1) Saturation of a client measures output performance from a single client to multiple storage targets (OSTs), varying burst sizes and the number of targets.

(2) Saturation of an OST probes the behavior of a single OST, in which processes from coordinated clients focus bursts on the same OST, varying the number of clients and burst sizes.

(3) Saturation of an OSS investigates the behavior of a single OSS by stressing the OSTs attached to it by varying the number of clients and burst sizes.

(4) Performance of Striping explores compute node performance variations on *stripe_width*. In this experiment, we vary burst sizes and *stripe_width*.

(5) Client-OST pairs 1 probes behavior of independent pipes by varying the number of pipes (client-OST pairs).

(6) Client-OST pairs 2 extends (5) by varying burst sizes.

3.2.3 *Results and Conclusions.*

We report the benchmarking results in Figure 4 and Figure 5, and draw four major conclusions. **1. Performance Variability and Stability.** We observed from all of the 119 configuration sets that the effective bandwidth varies significantly from instance to instance. However, Figure 4 suggests that the mean performance of each set converges rapidly to a steady state. It suggests that the mean output performance

Experiments	Performance-correlated Parameters			
Name	#nodes (m)	#cores (n)	burst_size (K)	stripe_width (W)
Saturation of a client	1	2, 4, 6, 7, 8, 10, 12, 14, 16	64MB, 256MB, 1GB, 4GB	1
Saturation of an OST	2, 4, 8, 16, 32, 64, 128	1	64MB, 256MB, 1GB	1
Saturation of an OSS*	2, 4, 7, 8, 14, 16, 28, 32, 56	1	7GB, 14GB, 28GB	1
Performance of striping	1	16	60MB, 240MB, 960MB	2, 4, 6, 8, 10, 12, 16, 32, 64
Client-OST pairs 1	50, 100, 200, 300, 336	1	64MB	1
Client-OST pairs 2	1008	1	16MB, 256MB, 4GB	1

Table 2: Varying parameters for the experiments. In an experiment, a varying parameter has multiple values. Each specific value set of varying parameters {m, n, K, W} is a *configuration set* (defined in §3.2.1) . This table presents overall 119 configuration sets from 6 experiments. *In Experiment *Saturation of an OSS*, each burst size reports the aggregate burst size across all engaged compute nodes.

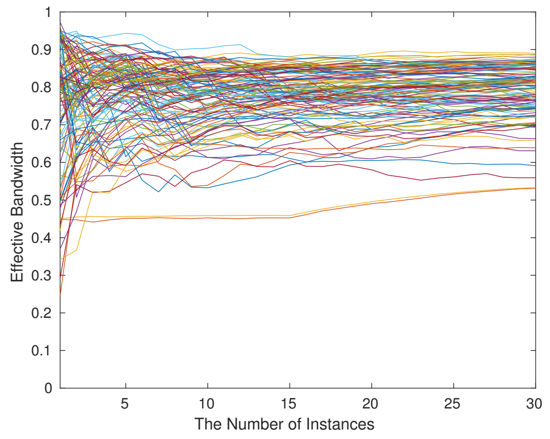


Figure 4: The Mean Effective Bandwidth across 119 configuration sets. Each line presents the data of the first 30 instances from one configuration set. In a line, each y value represents the mean effective bandwidth of the first x instances. It suggests that the mean performance converges to a steady state rapidly for all sets.

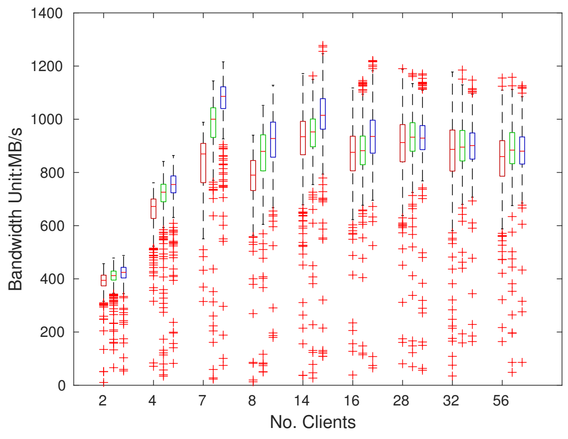


Figure 5: The Boxplots[1] **of Experiment (3).** In this figure, the x-axis represents the varying number of nodes; each x value has 3 boxplots reporting the bandwidths of 7GB, 14GB and 28GB aggregate burst sizes (see Table 2) from left to right respectively. It indicates that output performance is non-linearly related to the number of nodes and the burst size.

of scientific codes (§2.1) is stable and consistent after a few write operations/iterations.

2. Performance-correlated Parameters. Figure 5 presents the boxplots[1] of Experiment (3). It shows that output performance varies on m, K (Table 2). Experiments (1) and (4) (not reported) suggest that output performance also varies on n and W. We also categorized the effective bandwidths of specific *starting_OSTs* across 119 configuration sets, but we did not find a correlation between output performance and *starting_OSTs*. We conclude that output performance of Lustre-based file systems is correlated to m, n, K, W, as well as *stripe_size* (§3.2.2).

3. Behavior of Non-linearity. Figure 5 also suggests that when adding more clients or writing larger bursts, output bandwidth grows rapidly at the start, increases more and more slowly beyond a modest number of clients, and then declines after the saturation point. We also observed similar performance declines from other experiments (not reported), indicating that this behavior corresponds to the peak bandwidth capacity of the hardware on individual stages. It suggests that output performance of multi-stage supercomputer file systems is likely to be non-linearly related to the performance-correlated parameters: feature transformation techniques [15] should be considered for models.

4. Noise. Figure 4 also shows that different configuration sets present different mean effective bandwidths. Sets that run during periods of high contention may receive lower mean effective bandwidths. By summarizing the 119 configuration sets, we find that: (1) the 119 configuration sets receive the mean effective bandwidths ranging from 0.52 to 0.89. (2) the noise is inversely correlated to the aggregate burst size ($\frac{1}{m \times n \times K}$): larger bursts tend to receive higher bandwidth. (3) the noise is positively correlated to m: sets with more compute nodes tend to receive lower effective bandwidth.

4 MODELING OUTPUT PERFORMANCE OF PETASCALE FILESYSTEMS

This section presents how to build performance predictive models for the target file system. In summary, we use machine learning techniques to model the end-to-end burst absorption time for various output patterns and I/O configurations. In a model, a time

[1]Each boxplot depicts the quartiles of the bandwidths of instances of a configuration set in an experiment: the bottom, the middle bar and the top of the box represents 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 and above the box report the sample bandwidths at the low and high 1.5 ×IQR respectively, the dots on both sides depict outliers.

Feature Name	Metadata Cost ($t_{metadata}$)	Titan Cost (t_{titan})		SION Cost (t_{sion})	Spider 2 Cost ($t_{spider2}$)*		Noise Cost (t_{noise})
		Node Cost (t_{node})	Router Cost (t_{router})		OSS Cost (t_{oss})	OST Cost (t_{ost})	
Feature Value	$m \times n$	$\frac{m \times n \times K}{m}$		$m \times n \times K$	$\frac{m \times n \times K}{M_{used}}$	$\frac{m \times n \times K}{N_{used}}$	$\frac{m}{m \times n \times K}$

Table 3: Features for the Output Performance Prediction Model. In this table, all features are addressed by the performance-correlated parameters (§3.2): $\{m, n, K, W\}$, following the definitions in Table 2. Moreover, both *Titan Cost* and *Spider 2 Cost* can be addressed by two types of features alternatively; *Noise Cost* is an optional feature. *: the feature value of $t_{spider2}$ is $\frac{m \times n \times K}{N_{used}}$.

represents the mean of write times (§3.2) of a configuration set (§3.2.1) across times, iterations and various system conditions.

We design features according to performance-correlated parameters (§2.2 and §3.2), transform them to address potential non-linear relationships between features and burst absorption time, introduce a semi-random sampling method to build training set, and propose a systematic modeling methodology to search for the best model from a linear model space.

Although the results are limited to the current Titan deployment and configuration, the selected features, methodology, and sampling method are applicable to other Lustre deployments. Other features may be important for alternative file system structures, but we expect our results to be representative of other parallel file systems in which individual write bursts spread evenly across their selected storage devices.

4.1 Features for a Multi-stage Write Path

In this study, a model describes burst absorption time as a function of *features* (independent variables). We choose features that reflect performance variations of individual stages of the target environment on performance-correlated parameters (§3.2).

Consider an output pattern of a program on a supercomputer that runs P processes/threads on m nodes with n cores per node: $P = m \times n$. The P processes produce P synchronous bursts with burst size K, each process traveling through the write path and eventually residing on disks as an independent file. The pattern is configured as: *stripe_width* $= W$ (§3.2.2 and §3.2). Each specific value set for *{m, n, K, W}* is a *a configuration set* (§3.2.1).

Therefore, a write operation of this configuration set produces $m \times n$ files and overall $m \times n \times K$ size of data. The data travels through stages of the write path: its end-to-end time can be computed by summing up its time consumption on separate stages (features). We list the features below and also present them in Table 3.

Metadata Cost ($t_{metadata}$) is positively correlated to the number of files produced by the configuration set: $t_{metadata} \sim m \times n$.

Titan Cost (t_{titan}). Figure 2 shows that the target write path has two stages in Titan: the compute nodes and the I/O nodes (routers). The number of used I/O nodes is determined by the number of used compute nodes (§2.2). Accordingly, we can design two types of features to address t_{titan}: (1) one feature: t_{titan}; (2) two features: t_{node} and t_{router}. All of these 3 features can be estimated by $\frac{m \times n \times K}{m}$.

SION Cost (t_{sion}) is positively correlated to the aggregate burst size: $t_{sion} \sim m \times n \times k$.

Spider 2 Cost ($t_{spider2}$). Figure 2 depicts two stages in Spider 2: OSSes and OSTs. The number of used OSSes (M_{used}) is determined by the number of used OSTs (N_{used}) (§2.2). Therefore, $t_{spider2}$ can be considered as two types of features: (1) one feature: $t_{spider2}$; (2) two features: t_{oss} and t_{ost}. We can approximate $t_{spider2}$ and t_{ost} by $\frac{m \times n \times K}{N_{used}}$, and t_{oss} by $\frac{m \times n \times K}{M_{used}}$. Here, M_{used} and N_{used} can be computed for m, n, K, W from Formulas 4 and 5.

Noise Cost (t_{noise}). Since our target system is a production supercomputer, we consider the noise cost as an optional feature (§3.2): $t_{noise} \sim \frac{m}{m \times n \times K}$.

We can build a predictive model from any candidate feature set that combines these independent choices for features of the four stages in the target system: two choices for the Titan stage, two for the Spider 2 stage, and two choices for the optional Noise feature (include it or not). Therefore, there are 8 candidate feature sets. For example, we can choose a candidate feature set with 4 features: $\{t_{metadata}, t_{titan}, t_{sion}, t_{spider2}\}$, or a set with 5 features by replacing t_{titan} with t_{node} and t_{router}, or a set with 6 features by adding t_{noise}.

4.2 A Semi-random Sampling Method

A good training set for modeling output performance of a target system should cover the effects of varying key features. This section proposes a semi-random sampling method to build the training set with low cost and that satisfies three goals: (1) covers various configuration sets; (2) captures non-linear behaviors on the target machine (§3.2); (3) separates the stable behaviors of the configuration sets from the noise of samples in a production deployment.

To achieve these goals, we apply the statistical benchmarking methodology (§3.2). We use IOR as the burst generator, design a set of *model experiments*, and follow the experiment script template in Algorithm 1, with a few changes:

1. In an instance, we synchronize bursts before *file_open()* and measure the time from the minimum *file_open()* to the maximum *file_close()* among bursts.

2. We set *the number of Rounds*=1, fix m, and vary the other three performance-correlated parameters (Table 2): n, K and W.

3. For n, K, W in a setting, we produce a sequence of random values per parameter under certain constraints: (1) for n, we produce a fixed number of non-repeating random numbers in 1—16. (2) for K and W, we first choose a value range, then partition the range into several continuous intervals, then produce a random number per interval (details in Table 5). This process yields configuration sets that are both representative and random.

4. For each configuration set, we produce a minimum number of instances to achieve its steady state by making the approximation on the mean observed time with some reasonable confidence interval. After a configuration set reaches its steady state, we take the mean across the observed times of its instances and consider the set with its mean time (t) as *a sample* for model training and validation.

Feature Transformation Function	$()^1$	*log()*	$()^{2/3}$	$()^{3/4}$	$()^{3/2}$
Transformed Feature	$m \times n$	$log(m{\times}n)$	$(m{\times}n)^{2/3}$	$(m{\times}n)^{3/4}$	$(m{\times}n)^{3/2}$

Table 4: Feature Transformation Functions. We use 4 types of common transformation functions (row 1, column 3 — 6). Therefore, for each feature, e.g., Metadata Cost (row2), we produce 5 types of *transformed features*: its original feature and other 4 features transformed by 4 transformation functions respectively.

4.3 A Systematic Modeling Methodology

In this section, we search for the best predictive model from a linear model space built on top of the 8 candidate feature sets (§4.1) with feature transformation.

4.3.1 Feature Transformation. Feature transformation, also called *feature engineering*, represents a group of methods (functions) that transform a feature to new features for addressing the underlying nonlinear relationships between features and the target. In this study, we use this technique to depict the potential nonlinear relationships observed from the target environment (§3.2). For each feature across the 8 candidate feature sets, we take its original form and also apply 4 common transformation functions on it. Thus, for each feature we produce overall 5 types of *transformed features*, shown as Table 4.

A predictive model is built on *a transformed feature set*. Since each feature has 5 transformed features, each of the 8 candidate feature sets produces $5^{\wedge\#features}$ models, in which *#features* represents the number of original features in the feature set. In summary, we search for the best model from a model space with overall 135,000 candidate models.

4.3.2 Training a Model. This section presents how to train a model by using machine learning techniques. Consider a transformed feature set that consists of y transformed features $\{TF_1, TF_2, ..., TF_y\}$. For the target t', we build the linear model as:

$$t' = \alpha_0 + \sum_{j=1}^{y} \alpha_j \times TF_j \tag{6}$$

α_0 is the intercept, α_j is the coefficient of TF_j. In this study, training a model means locating $<\alpha_0, \alpha_1, \alpha_2, ..., \alpha_y>$ for a transformed feature set $\{TF_1, TF_2, ..., TF_y\}$. Presume that a training set has x samples and transformed set is $\{TF_1, TF_2, ..., TF_y\}$. We use *LinearRegression()* method from scikit-learn toolkit [34] to train the model.

The most intuitive approach is to feed the training set directly to *LinearRegression()*, yielding a value set for $<\alpha_0, \alpha_1, \alpha_2, ..., \alpha_y>$. We adopt two techniques to improve result quality: *10-fold cross validation* [15, 34] and mean square error (*MSE*).

10-Fold Cross Validation is an important technique to evaluate model accuracy. It partitions the training set into 10 equal-size subsets with 9 subsets merged for training and the last one used as the *validation set*: it produces the intercept and coefficients for a model by training the merged 9 subsets and receives a model accuracy measurement by testing it on the validation set. In our study, the entire cross validation process repeats *LinearRegression()* 10 times on the same training set by rotating the validation set across its 10 subsets; for each *LinearRegression()*, it produces a value set for $<\alpha_0, \alpha_1, \alpha_2, ..., \alpha_y>$ and also receives an accuracy measurement. Therefore, a trained model has 10 value sets for $<\alpha_0, \alpha_1, \alpha_2, ..., \alpha_y>$ and 10 accuracy measurements. We choose *MSE* as the accuracy metric and take the mean values for both intercept/coefficients and *MSE* for the model, shown as Formulas 7 and 8. In the Formulas, i represents the ith $LinearRegression()$ outputs in a model training process.

$$<\alpha'_0, \alpha'_1, ..., \alpha'_y> = <\frac{\sum_{i=1}^{10}\alpha_{i0}}{10}, \frac{\sum_{i=1}^{10}\alpha_{i1}}{10}, ..., \frac{\sum_{i=1}^{10}\alpha_{iy}}{10}> \tag{7}$$

$$MSE' = \frac{\sum_{i=1}^{10} MSE_i}{10} \tag{8}$$

Mean Square Error is an error estimator that is widely used to quantify accuracy of regression models. Specifically, we use it to measure accuracy of each trained model: in the ith $LinearRegression()$, we feed the validation set to the trained model and get a prediction result set $<t'_1, t'_2, ..., t'_{\frac{x}{10}}>$. If the means of the observed times on the validation set are $<t_1, t_2, ..., t_{\frac{x}{10}}>$, then we compute the ith model accuracy metric: MSE_i, shown in Formula 9.

$$MSE_i = \sum_{j=1}^{\frac{x}{10}} (t'_j - t_j)^2 \tag{9}$$

In summary, a trained model is composed of $\{TF_1, ..., TF_y\}$, $<\alpha'_0, \alpha'_1, ..., \alpha'_y>$ and MSE'.

Algorithm 2 Searching for the best model

```
1:  The Training Set
2:  Candidate Feature Sets: {$t_{metadata}, ..., t_{spider2}$}, ...
3:  Transformed Feature Sets:..., {$TF_1, ..., TF_y$}, ...
4:  for Candidate Feature Set: 1 → 8 do
5:      for Transformed Feature Set: 1 → $5^{\wedge\#features}$ do
6:          10-fold cross validation
7:          The training set, LinearRegression()
8:          → $<\alpha'_0, \alpha'_1, ..., \alpha'_y>, MSE'$
9:          if $MSE' < MSE_{min}$ then
10:             $\{TF_1, ..., TF_l\}^* = \{TF_1, ..., TF_y\}$
11:             $<\alpha'_0, \alpha'_1, ..., \alpha'_l>^* = <\alpha'_0, \alpha'_1, ..., \alpha'_y>$
12:             $MSE_{min} = MSE'$
13:         end if
14:     end for
15: end for
```

4.3.3 Searching for the Best Model. For each pair of trained models, we select one model as better if its MSE' is smaller than the other. Following this rule, we search for the best model with minimum MSE' from a linear model space with 135,000 candidate models (§4.3.2).

Scale (m)	Cores per Node (n)	Burst Size (K)	*stripe_width* ((W))
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 800	8 *from* 16	1MB—5MB, 6MB—25MB, 25MB—100MB, 101MB—250MB, 251MB—500MB, 501MB—1024MB, 1025MB—2560MB	1—4, 5—8, 9—16, 17—32, 33—64
1, 2, 4, 8, 16, 32, 64, 128	4 *from* 16	2561MB—5120MB, 5121MB—7680MB, 7681MB—10240MB	1—4, 5—8, 9—16, 17—32, 33—64

Table 5: Template for the small-scale and medium-scale samples following the sampling method in §4.2. The setting in row1 produces 11 model experiment settings: each examines on m nodes, varies n cores as 8 non-repeating random numbers in 1 — 16, changes K on 7 random numbers with each from one of 7 continuous intervals, and alters W on 5 random numbers with each from one of 5 continuous intervals. Therefore, each setting in row1 produces $8 \times 7 \times 5$ samples; the setting in row2 follows the same rule and has 8 model experiment settings, each producing $4 \times 3 \times 5$ samples.

Consider the best model has l features: $\{TF_1, ..., TF_l\}^*$, $< \alpha'_0, \alpha'_1, ..., \alpha'_l >^*$ and MSE_{min}. At the initial state, $\{TF_1, ..., TF_l\}^* = \{\}$, $< \alpha'_0, \alpha'_1, ..., \alpha'_l >^* = <>$, $MSE_{min} = +\infty$. The searching process is shown as Algorithm 2.

5 EXPERIMENTS

This section evaluates our systematic machine learning analysis on Titan/Atlas2. We collected measures for IOR bursts with varying parameters for use as a dataset for training and validation. These bursts ran on Titan at scales up to 16K cores (1000 nodes) from Aug. 2016 to Jan. 2017.

5.1 Experiment Data

Output bandwidth is a scarce resource on supercomputers; large scale write operations are expensive. To generate models with low cost, we chose to focus on training models with *small-scale* writes ($\leq$ 128 nodes) and testing them on *medium-scale* writes (256—800 nodes) supplemented with a modest set of measures from *large-scale* writes (=1000 nodes) with representative output patterns.

Consequently, we produce experiment data according to two templates: (1) for the small-scale and medium-scale samples, we follow the semi-random sampling method (§4.2) and choose intervals for n, K, W, focusing on typical output patterns and I/O configurations observed from production use; (2) for the large-scale samples, we use output patterns characteristic of production codes, including XGC (§2.1.1), PlasmaPhysics, Turbulence1, Turbulence2 and AstroPhysics reported in [21]. We produced overall 3578 samples. Tables 5 and 6 present details of the two templates respectively.

5.2 Model Evaluation

This section evaluates: (1) effectiveness of the features (§4.1), (2) usefulness of feature transformation (Table 4), and (3) accuracy of the model located by the systematic modeling methodology (§4.3).

To this end, we use a training set with 2720 samples produced by 1 — 128 nodes according to the two settings in Table 5; we

Scale (m)	<*cores per node, burst_size*> ($< n, K >$)	*stripe_width* (W)
1000	<1, 59MB>, <4, 69MB>, <4, 4MB>, <4, 1024MB>, <16, 23MB>, <16, 121MB>, <16, 376MB>, <16, 750MB>, <16, 1280MB>	*4, 5—100*

Table 6: Template for the large-scale samples. This template mimics the output patterns of XGC and other sample production codes in [21], in which $< n, K >$ are given. We choose 2 *stripe_width* for each pattern: *stripe_width=4* (Titan's default configuration) and *stripe_width = a random number* in 5 — 100 (typical configurations). This template produces overall 18 samples.

perform the systematic modeling methodology on 135,000 models across 8 candidate feature sets (§4.1) and pinpoint the model (Model$_{best}$) with MSE_{min}; we also process the methodology on 625 (5^4) models from the candidate feature set with 4 features and locate the best model (Model$_{local\_best}$) for the set; we train the model (Model$_{original}$) from the Model$_{best}$'s candidate feature set and with all original features. In summary, we address three models: Model$_{best}$, Model$_{local\_best}$, Model$_{original}$, shown in Table 7. Besides presenting models, Table 7 also reports the cost consumed to generate a group of models and select the best from the group: for Model$_{best}$ and Model$_{local\_best}$, the cost includes the time to train all models from the space of 135,000 candidate models and the space with 625 models respectively; for Model$_{original}$, the cost is the time to train the single model.

We use *relative true error* (ϵ) to quantify model accuracy. If the observed mean write time of the ith sample is t_i and its prediction result is t'_i then its ϵ_i is:

$$\epsilon_i = \frac{t'_i - t_i}{t_i} \tag{10}$$

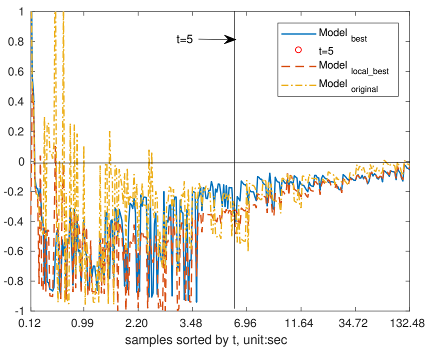
Thus, $\epsilon_i > 0$ suggests that t_i is over estimated; $\epsilon_i < 0$ suggests that t_i is under estimated; $||\epsilon_i||$ quantifies prediction accuracy: smaller $||\epsilon_i||$ indicates higher accuracy. We focus on two thresholds: $||\epsilon||$=0.2 and = 0.3.

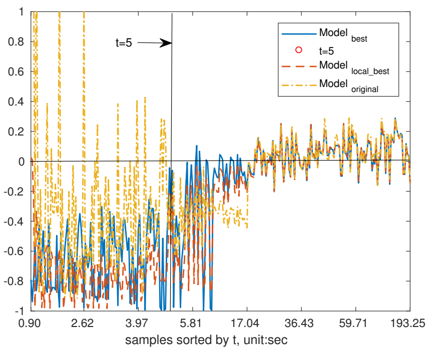
We evaluate Model$_{best}$, Model$_{local\_best}$ and Model$_{original}$ on 4 test sets, plot the results in Figures 6 and 7, and draw 4 major conclusions below.

1. The identified features are effective predictors of output burst performance. Figure 6 shows that, for a model on a test set with $t \geq 5$, 63.28% — 96.51% and 76.84% — 98.84% of samples report $||\epsilon|| \leq 0.2$ and ≤ 0.3 respectively. Similarly, Figure 7 shows that, for a model with $t \geq 5$, 43.75% — 56.25% and 62.5%—75% of samples report $||\epsilon|| \leq 0.2$ and ≤ 0.3 respectively. It suggests that all 3 models are generally accurate across all 4 test sets if a write operation takes $\geq$ 5 seconds: the features we choose are effective to capture output behavior of the target environment.

2. Feature transformation is useful. Figures 6 and 7 also suggest that Model$_{best}$ is generally more accurate than Model$_{original}$ for all 4 test sets: feature transformation is useful to address non-linear relationships between output performance and its features under linear regression.

3. The modeling methodology identifies good models. For $t \geq 5$, Model$_{best}$ is most accurate for all 4 test sets. Moreover, for Model$_{best}$ on a test set with $t \geq 5$ in Figure 6, 83.05% — 96.51%





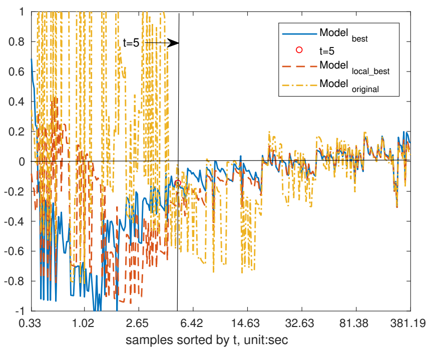


Figure 6: Model Evaluation. From left to right, 3 subfigures present accuracy of the models (Table 7) on 3 test sets produced by 256, 512 and 800 nodes separately by following the row1 template in Table 5. Each test set has 280 samples; in each subfigure, a line plots 280 error measures (ϵ) for a model on a test set; in a line, samples are sorted along the x-axis based on the observed mean write time (t). It suggests that, Model$_{best}$ is generally most accurate for all 3 test sets for $t \geq 5$.

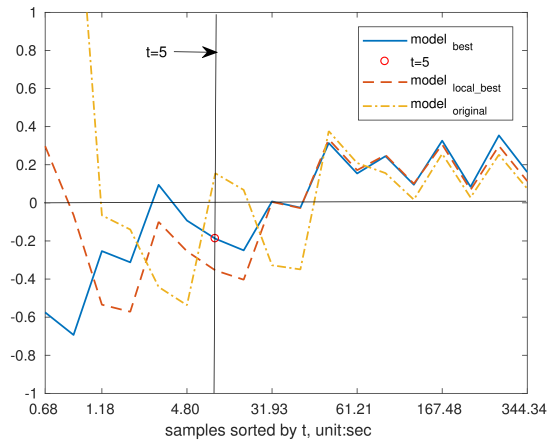


Figure 7: Model Validation on the large-scale samples. The test set has 18 samples; a line plots 18 error measures (ϵ) for a model; in a line, samples are sorted along the x-axis based on the observed mean write time (t). It suggests that Model$_{best}$ is generally most accurate.

and 89.83% — 98.84% of samples report $||\epsilon|| \leq 0.2$ and ≤ 0.3 respectively; for Model$_{best}$ in Figure 7, 56.25% and 75% of such samples report $||\epsilon|| \leq 0.2$ and ≤ 0.3 respectively. It suggests that the methodology locates the best model from the model space; the best model is highly accurate.

4. Limitations. When $t < 5$, all models are less accurate. This is likely results from two issues: (1) Titan is noisy. For short bursts, highly parallel writes are vulnerable to transient system conditions that are less likely to be reflected in the small-scale samples in the training set. (2) Error estimator biases to longer bursts. According to Formula 9, the model searching process searches for the model with minimum $\sum(t' - t)^2$. Therefore, it tends to identify models that perform well for samples with large t. We leave the followup study on this as our future work.

5.3 Metadata Anomaly

To address the expected behavior of Titan/Atlas2, we decided to discard the anomalous instances given how rarely they occur. An instance is considered anomalous if it consumes > 30 seconds on *file_open()* (§3.1). The 35,404 instances in our benchmark data showed 17 anomalous instances. These occurred across a range of configurations and scales (number of nodes, cores, and files) with no detectable pattern. It is possible that a larger dataset would reveal a correlation with scale. We leave a more detailed study for future work.

6 RELATED WORK

Related studies fall into three categories: quantitative I/O performance studies, novel techniques to improve performance, and I/O middleware systems.

1. I/O Performance Studies. Researchers started profiling supercomputer filesystems under production loads in the 1990s [28], [14],[29], [10], [9]. More recently, the Darshan team from Argonne National Lab [3, 26] investigates the I/O behavior of supercomputer platforms by analyzing logs and traces collected by continuous monitoring software installed on compute nodes. [40] and [37] adopt statistical benchmarking methods to analyze I/O behavior of parallel filesystem deployments, and identify factors that can slow down striped I/O operations. [22, 25] also study the I/O performance of Lustre-based file systems: [22] probes the application behaviors by analyzing server side I/O logs, while [25] explores factors inhibiting I/O performance on scientific codes with MPI-IO. Kim et al. [18] studies the combined application behavior on the server side by monitoring the performance of storage servers.

2. Techniques to Improve I/O Performance. Researchers have explored several techniques to improve the performance of Lustre-based file systems: multi-threading in the compute node client OSC [35], asynchronous journaling [32] in the server OSS, and fine-grained load balancing across I/O routers [13]. Another group of studies focuses on reducing the cost of metadata operations [23, 33].

Model Name	Cost	Intercept	$t_{metadata}$	t_{node} (t_{titan}*)	t_{router}	t_{sion}	t_{oss}	t_{ost} ($t_{spider2}$*)	t_{noise}
Model$_{best}$	44.46 minutes	0.65	-0.19	0.005	4.64^{-7}	1.00^{-9}	0.002	0.15	-0.58
			log()	$()^{3/4}$	$()^{3/2}$	$()^{3/2}$	$()^{1}$	log()	$()^{3/2}$
Model$_{local\_best}$	0.18 minute	3.04	-0.51	0.0004		1.65^{-5}		1.32^{-5}	
			log()	$()^{1}$		$()^{1}$		$()^{3/2}$	
Model$_{original}$	0.0003 minute	1.03	-0.001	0.0002	0.0002	-4.13^{-6}	0.003	-0.0004	-2.16
			$()^{1}$	$()^{1}$	$()^{1}$	$()^{1}$	$()^{1}$	$()^{1}$	$()^{1}$

Table 7: Three Trained Models defined in §5.2. In this table, features are defined in §4.1 and given in Table 3; Model$_{best}$ and Model$_{original}$ have the same 7 features with different feature transformation functions; Model$_{local\_best}$ has 4 features. For each model, we report its cost, intercept and features; for each feature (column 4-10), we report its name, coefficient (a numeric value) and feature transformation function (Table 4). *: Model$_{local\_best}$'s features.

3. I/O Middleware Systems. I/O middleware systems provide a high-level I/O API for applications and adapts I/O patterns and configurations automatically to improve I/O performance. ADIOS [8, 20, 24] is a widely used I/O middleware system for HPC applications on supercomputers. Our work is complementary to middleware systems: for example, they can use I/O system performance profiles and prediction results to guide their configuration choices and adapt I/O patterns.

Studies on cloud and data center workloads have investigated machine learning techniques to predict performance. For example, [38] and [11] adopt linear regression and SVD (*Singular Value Decomposition*) separately to predict end-to-end application execution time and drive co-scheduling choices. Compared to their works, we address an I/O system that provides massively parallel I/O for individual HPC applications, and apply a systematic modeling methodology for the multi-stage write path in a petascale I/O deployment under production load. To our knowledge our work is the first to apply machine learning regression models to analyze and predict I/O performance for HPC systems and applications.

7 CONCLUSION

Scientific codes generate periodic parallel output bursts in regular patterns. We propose a statistical approach to benchmark a production petascale I/O system and learn performance models that can predict output performance seen by applications. We show that accurate models can be learned based on a few key features of the deployment and I/O configuration parameters. Our premise is that accurate prediction models can guide configuration choices to reduce I/O cost, and also provide better information to the global scheduler, which can use this information to improve resource utilization.

The major obstacle to building the quantitative I/O analysis on petascale filesystems is the high degree of performance variability in production deployments. We find that although the output performance of petascale filesystems is highly variable, the mean performance for sufficiently large bursts is predictable.

This paper develops a regression approach to predict output performance of petascale filesystems under production load, focusing on the mean write time. We select and transform features to capture the properties of the target multi-stage write path and their impact on I/O performance. We introduce a semi-random sampling method to generate performance datasets to train the models with low benchmarking cost. A systematic modeling methodology obtains the best model from a rich model space of features and transformations. The results suggest that the model is sufficiently accurate to predict performance for sufficiently large write bursts in practice. The key limitation is that small bursts are vulnerable to transient contention, and so are difficult to predict accurately.

8 ACKNOWLEDGMENTS

We are thankful to the anonymous reviewers and our shepherd Gabriel Antoniu for their invaluable feedback; to Choong-Seock Chang and Randy M. Churchill from PPPL for their help on understanding XGC; to Chris Zimmer and Matt Ezell from OLCF for their detailed explanations on Titan and Spider 2; to Norbert Podhorszki from ORNL and Philip Carns from ANL for their advice on I/O behavior of HPC codes; to Sayan Mukherjee at Duke for helpful suggestions on machine learning techniques; to Suzanne T. Parete-Koon at ORNL for her help on arranging Titan time for experiments.

This work was supported by the U.S. Department of Energy, under FWP 16-018666, program manager Lucy Nowell. 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 DE-AC05-00OR22725. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] M. Berger and J. Oliger. 1984. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of computational Physics* 53, 3 (1984), 484–512.
[2] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. 2011. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)* 7, 3 (2011), 8–26.
[3] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 2009. 24/7 characterization of petascale I/O workloads. In *Proceedings of 2009 IEEE International Conference on Cluster Computing (Cluster'09)*. IEEE, New Orleans, LA, 1–10.
[4] L. Chacón. 2004. A non-staggered, conservative, finite-volume scheme for 3D implicit extended magnetohydrodynamics in curvilinear geometries. *Computer Physics Communications* 163, 3 (2004), 143 – 171.
[5] C. S. Chang, S. Klasky, J. Cummings, R. Samtaney, A. Shoshani, L. Sugiyama, D. Keyes, S. Ku, G. Park, S. Parker, and others. 2008. Toward a first-principles integrated simulation of tokamak edge plasmas. *Journal of Physics: Conference Series* 125, 1 (2008), 012042.

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

[7] 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 (2009), 015001.

[8] A. Choudhary, W. Liao, K. Gao, A. Nisar, R. Ross, R. Thakur, and R. Latham. 2009. Scalable I/O and analytics. *Journal of Physics: Conference Series* 180, 1 (2009), 012048.

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

[10] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. 1993. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. ACM, San Diego, CA, 2–13.

[11] C. Delimitrou and C. Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ATM, Houston, TX, 77–88.

[12] D. A. Dillow, G. M. Shipman, S. Oral, Z. Zhang, and Y. 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)*. IEEE, Orlando, FL, 21–29.

[13] M. Ezell, S. Oral, F. Wang, D. Tiwari, D. Maxwell, D. Leverman, and J. 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)*. cug.org, Lugano, Switzerland, 1–6.

[14] G. R. Ganger. 1995. Generating representative synthetic workloads: an unsolved problem. In *Proceedings of the Computer Measurement Group Conference (CMG'95)*. CMG, Nashville, TN, 1263–1269.

[15] J. Han, J. Pei, and M. Kamber. 2011. *Data mining: concepts and techniques* (3 ed.). Morgan Kaufmann, Waltham, MA.

[16] Y. Kim, S. Atchley, G. Vallée, and G. Shipman. 2015. LADS: optimizing data transfers using layout-aware data scheduling. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX, Santa Clara, CA, 67–80.

[17] Y. Kim and R. Gunasekaran. 2014. Understanding I/O workload characteristics of a peta-scale storage system. *The Journal of Supercomputing* 71, 3 (2014), 761–780.

[18] Y. Kim, R. Gunasekaran, G. M. Shipman, D. A. Dillow, Z. Zhang, and B. W. Settlemyer. 2010. Workload characterization of a leadership class storage cluster. In *Proceedings of the 5th Petascale Data Storage Workshop (PDSW'10)*. ACM, New Orleans, LA, 1–5.

[19] 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)*. IEEE, Phoenix, AZ, 24–36.

[20] S. Kumar, J. Edwards, P.-T. Bremer, A. Knoll, C. Christensen, V. Vishwanath, P. Carns, J. A. Schmidt, and V. Pascucci. 2014. Efficient I/O and storage of adaptive-resolution data. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. ACM, New Orleans, LA, 413–423.

[21] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems. In *Proceedings of the 28th IEEE Conference on Massive Data Storage (MSST'12)*. IEEE, Long Beach, CA, 1–11.

[22] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai. 2014. Automatic identification of application I/O signatures from noisy server-side traces. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX, Santa Clara, CA, 213–228.

[23] J. Lofstead, F. Zheng, S. Klasky, and K. 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)*. IEEE, Rome, Italy, 1–10.

[24] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. 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)*. ACM, Washington, DC, 1–12.

[25] J. Logan and P. Dickens. 2008. Towards an understanding of the performance of MPI-IO in Lustre file systems. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'08)*. IEEE, Tsukuba, Japan, 330–335.

[26] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. 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)*. ACM, Portland, OR, 33–44.

[27] R. Miller, J. Hill, D. A. Dillow, R. Gunasekaran, G. Shipman, and D. Maxwell. 2010. Monitoring tools for large scale systems. In *Proceedings of Cray User Group Conference (CUG'10)*. cug.org, Edinburgh, 1–4.

[28] A. L. Narasimha Reddy and P. Banerjee. 1990. A study of I/O behavior of perfect benchmarks on a multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*. ACM, Seattle, WA, 312–321.

[29] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. L. Best. 1996. File-access characteristics of parallel scientific workloads. *IEEE Trans. on Parallel and Distributed Systems* 7, 10 (1996), 1075–1089.

[30] S. Oral, D. A. Dillow, D. Fuller, J. Hill, D. Leverman, S. S. Vazhkudai, F. Wang, Y. Kim, J. Rogers, J. Simmons, and R. Miller. 2013. OLCF's 1 TB/s, next-generation Lustre file system. In *Proceedings of The Cray User Group Conference (CUG'13)*. cug.org, Napa, California, 1–4.

[31] S. Oral, J. Simmons, J. Hill, D. Leverman, F. Wang, M. Ezell, R. Miller, D. Fuller, R. Gunasekaran, Y. Kim, S. Gupta, D. Tiwari, S. S. Vazhkudai, J. H. Rogers, D. Dillow, G. M. Shipman, and A. S. Bland. 2014. Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. ACM, New Orleans, LA, 217–228.

[32] S. Oral, F. Wang, D. Dillow, G. Shipman, R. Miller, and O. 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)*. USENIX, San Jose, CA, 143–154.

[33] K. Ren, Q. Zheng, S. Patil, and G. Gibson. 2014. IndexFS: scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. ACM, New Orleans, LA, 237–248.

[34] scikit learn. 2016. scikit-learn:machine learning in Python. http://scikit-learn.org/. (2016). Accessed: 2016-08-29.

[35] G. Shipman, D. Dillow, D. Fuller, R. Gunasekaran, J. Hill, Y. Kim, S. Oral, D. Reitz, J. Simmons, and F. Wang. 2012. A next-generation parallel file system environment for the OLCF. In *Proceedings of the Cray User Group Conference (CUG'12)*. cug.org, Stuttgart, Germany, 1–12.

[36] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. 2016. Top500 supercomputer sites. http://www.top500.org. (2016). Accessed: 2016-08-11.

[37] A. Uselton, M. Howison, N. J. Wright, D. Skinner, N. Keen, J. Shalf, K. L. Karavanic, and L. Oliker. 2010. Parallel I/O performance: from events to ensembles. In *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS'10)*. IEEE, Atlanta, GA, 1–11.

[38] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. 2016. Ernest: efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*. USENIX, Santa Clara, CA, 363–378.

[39] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang. 2009. Understanding Lustre filesystem internals. *Technical Report ORNL* TM-2009, 117 (2009), 1–80.

[40] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. 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)*. IEEE Computer Society Press, Salt Lake City, UT, 1–11.