Can I/O Variability be Reduced on QoS-less HPC Storage Systems?

Dan Huang, Qing Liu*, Jong Choi, Norbert Podhorszki, Scott Klasky, Jeremy Logan, George Ostrouchov, Xubin He, *Senior Member, IEEE*, and Matthew Wolf

Abstract—For a production high-performance computing (HPC) system, where storage devices are shared between multiple applications and managed in a best effort manner, I/O contention is often a major problem. In this paper, we propose a balanced messaging-based re-routing in conjunction with throttling at the middleware level. This work tackles two key challenges that have not been fully resolved in the past: whether I/O variability can be reduced on a QoS-less HPC storage system, and how to design a runtime scheduling system that can scale up to a large amount of cores. The proposed scheme uses a two-level messaging system to re-route I/O requests to a less congested storage location so that write performance is improved, while limiting the impact on read by throttling re-routing. An analytical model is derived to guide the setup of optimal throttling factor. We thoroughly analyze the virtual messaging layer overhead and explore whether the in-transit buffering is effective in managing I/O variability. Contrary to the intuition, in-transit buffer cannot completely solve the problem. It can reduce the absolute variability but not the relative variability. The proposed scheme is verified against a synthetic benchmark as well as being used by production applications.

Index Terms—High-performance computing, storage, quality of service, variability

1 Introduction

In high-performance computing (HPC) systems, parallel storage systems have become the primary solution for managing big datasets generated from scientific simulations. These systems use a range of 100s to 1000s of storage devices to achieve the throughput and capacity demanded by applications. Despite strong peak performance that were achieved during the maintenance windows when users are disallowed to access the systems and the I/O system is quiet, we have seen subpar performance with significant I/O fluctuations in production environments, observing as much as an order of magnitude variation in throughput across storage devices and runs. A root cause of such performance degradations and variations is the interference incurred by applications running simultaneously and sharing storage resources, as shown by previous work [1], [2], [3], [4]. I/O contention is detrimental to the performance of application whose I/O time is a significant part of run time, as the contention leads to variations in completion times across processes. Most parallel applications are forced to operate in a tightly synchronized manner, which means that a process that finishes its I/O more quickly is still forced to wait for its slower peers. The aggregate computational cycles wasted can be far too costly to ignore, especially for the application with significant I/O time. Furthermore, the variability in I/O time makes the job time less predictable. If a job does not finish on schedule due to the interference from a competing job, the portion of the run starting from the most recent checkpoint has to be re-done.

- *D. Huang and Q. Liu are with the Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ.*
- *J. Choi, N. Podhorszki, S.Klasky, J. Logan, G. Ostrouchov, and Matthew Wolf are with Oak Ridge National Laboratory, Oak Ridge, TN.*
- *X. He is with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA.*
- ** Corresponding author.*

Nevertheless, on next generation HPC systems, increasing I/O throughput is still the overarching theme and I/O variability continued to be less emphasized. As such, an application is typically allowed to access shared or dedicated storage devices to maximize its I/O throughput. For example, on the Cori system[1] at National Energy Research Scientific Computing Center (NERSC), storage devices within the burst buffer and the parallel file system can be accessible to applications, when the applications are allocated the shared buffer and storage resources. A file placed onto the burst buffer is striped onto all storage devices, thus greatly increasing the likelihood for an application to compete with others. With the growing scale of HPC systems and application data, the I/O variations can become worse if left unmanaged, and new approaches are needed to address this issue on HPC systems with massive parallelism.

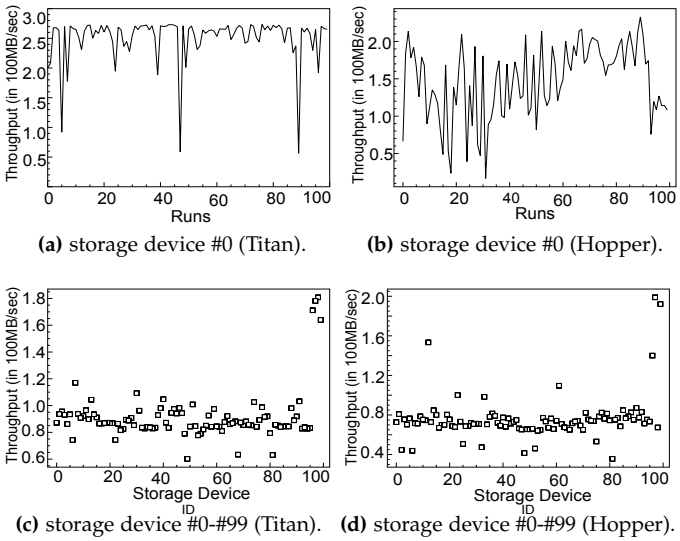
In HPC domain, most existing research on storage and I/O quality of service (QoS) fall into three broad categories: server-side scheduling [5], [6], [7], [8], [9] and log-structured file system (LFS) [10], [11], none of which have been adopted in the current petascale production systems, unfortunately. This is mainly attributed to the following: 1) the I/O scheduling is often made by a single entity, and this greatly limits the number of clients that do simultaneous writes to the file system, 2) the usage of large memory space as cache compromises the resolution and fidelity of numerical calculations. In non-HPC domains, e.g. data centers and cloud storage, the current dominant method is work-conserving resource allocation [12], [13], [14], [15], [16], [17]. A basic example, e.g. WFQ [18], is round-robin scheduling for requests at the same priority level and placing a premium on requests of higher priority levels. However, these methods do not monitor the congestion status on storage devices

1. http://www.nersc.gov/users/computational-systems/cori/

and can not reduce the workload imbalance in a resource-constrained environment. Nevertheless, the throughput of a HPC write workload, as shown in Figure 1a and 1b, exhibits high temporal and spatial speed variance due to the workload imbalance on storage devices. Without knowing the I/O congestion and imbalance status, a work-conserving scheduler's best choice is viewing storage systems as a black-box and sending out re-ordered I/O requests as fast as possible for the purpose of using more shares of the resources. Under such situation, congestion easily occurs at many levels in the system, hurting the overall efficiency [9].

In light of these limitations, this work designs a novel scheme in which clients collectively participate in the I/O scheduling through lightweight messaging to enable I/O re-routing in the presence of hotspots. In contrast to the prior approaches, no additional caching, system-wide or server-side scheduler is used here for mitigating variability, and the scheduling decision is managed in a hierarchical structure to sustain the parallelism up to the 100,000-core level. The major contributions in this paper are as follows.

- We design a two-level *virtual messaging layer* (Section 3.1), which facilitates I/O re-routing and throttling for a large number of clients. The key idea is that via messaging, the I/O activities on each storage device can be coordinated, and the slow storage devices can be discovered and managed. The I/O re-routing is designed to operate in a *lazy* manner so that the constant monitoring can be avoided. In addition, we show that a limited storage probing via a one-time $STATUS\_INQ$ message, which is issued during the initial period of re-routing, can obtain the storage device load, and reduce the failure messages due to race condition.
- The impact of I/O re-routing to the write and read performance is thoroughly investigated, and an analytical model is developed to guide the selection of throttling factor (Section 3.3 and 3.4). As a result of re-routing, each storage device receives a varying amount of data depending on how heavily it is loaded. This has the effect of creating new hotspots for reading. Our approach avoids the excessive re-routing and incorporates an analytical model to guide the selection of throttling factor.
- We further investigate the I/O variability on the emerging burst buffer storage that will be deployed on the next generation HPC systems (Section 4). Contrary to the common intuition, burst buffers, despite being fast, does not cure the variability issue. Performance variability remains a challenge in the next generation HPC systems, which share the emerging burst buffer storage. To the best of our knowledge, our work provides the first deep dive into the performance variability issue on the new HPC storage architecture, that is, burst buffer. Our findings can help HPC developers and platform owners improve I/O performance and motivate further research addressing the problem across all components of the I/O stack on the emerging burst buffer storage.
- This work has been tested using production applications on leadership class systems during the operational hours. This technique effectively controls I/O traffic in a way that write performance can be improved without significantly degrading read performance. In addition, this work is implemented at the middleware level, without imposing major file system or operating system changes. As far as we know, our technique is the only one that can be used in a production setting.



(a) storage device #0 (Titan). **(b)** storage device #0 (Hopper).

(c) storage device #0-#99 (Titan). **(d)** storage device #0-#99 (Hopper).

Fig. 1: Performance of storage devices on Lustre file system (version 2.8.0). This figure shows even with the Lustre internal load balancing via the round-robin data placement, the storage performance is still unbalanced.

The code is publicly available at https://lqnjupt@bitbucket.org/lqnjupt/aio.git.

The remainder of the paper is organized as follows. Section 2 presents the measurement results on two leadership class systems and motivates the I/O variation problem. Section 3 discusses the design and implementation details of I/O re-routing with throttling. Section 4 discusses the in-transit buffering. Section 5 presents performance evaluations, along with related work and conclusions in the end.

2 Observation and Motivation

In this section, we present the measurements collected on two leadership systems, and demonstrate that I/O contention does exist and can be severe on production storage systems. The choice of using shared, best effort storage system is attributed to the fact that shared storage is much more economical and convenient than the dedicated storage. The experiments were done on Titan[2] at Oak Ridge National Laboratory, currently the 5th fastest machine[3], and Hopper at NERSC. Each machine deploys a massively parallel storage system, comprised of 100s to 1,000s storage devices for a given file system partition. For the first experiment, in Figure 1(a) and (b), we write 16 MB blocks to the storage device

2. https://www.olcf.ornl.gov/titan
3. http://www.top500.org

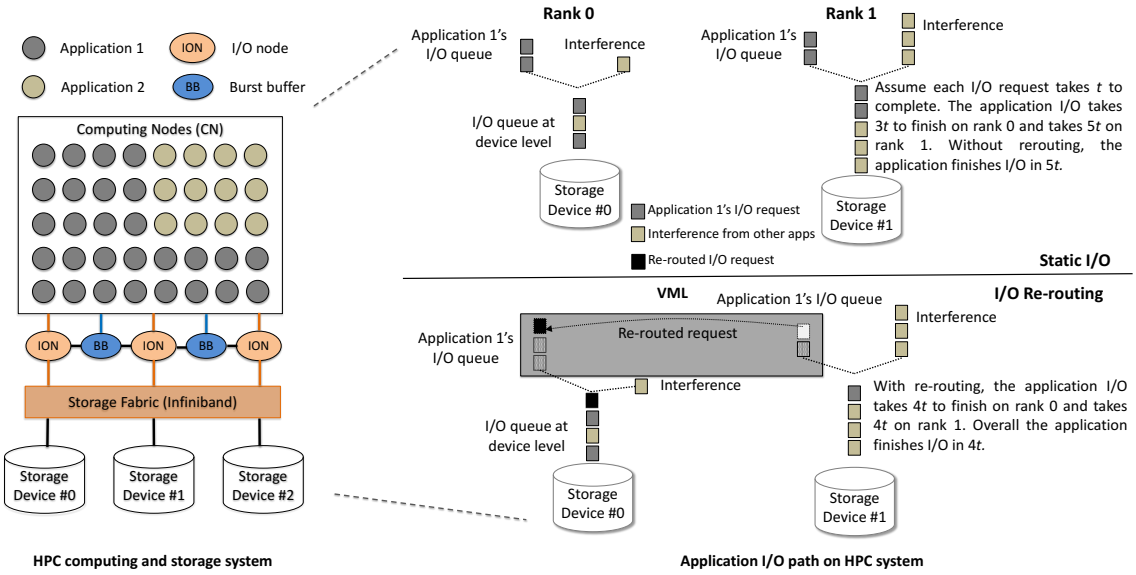


Fig. 2: Re-routing vs. static I/O. The left part illustrates that two applications are running on an HPC system, where the computing nodes are allocated and the storage devices are shared. In the right part, the upper and lower subfigures show how application 1's I/O requests are handled in static I/O and re-routing, respectively. In both cases, I/O requests are issued from two MPI ranks and served by two storage devices. As a result of re-routing (lower figure), one I/O request is re-routed from storage device #1 to #0, thus avoiding the relatively higher interference on storage device #1.

#0 in the system to test the perceived speed of an individual storage device. The results were collected over 100 iterations to show the severity of the throughput fluctuation over time. For example, on Hopper (Figure 1(b)), the peak is more than 10x higher than the lowest performance. The results indicate that the performance of the storage system is indeed very bursty and fairly random. Figure 1(c) and (d) shows the performance snapshot of 100 storage devices. Although most of the storage devices offer a similar throughput, there are a number of outliers that are significantly slower.

The main driver behind this work is that I/O interference has been significantly impacting HPC application performance. The reason is that the majority of scientific codes require tight synchronization, in which processes need to synchronize at some point during the calculation in order to get to a coherent state, and prepare for the next iteration of calculations. Many large scale scientific codes use Message Passing Interface (MPI) collectives to exchange state between processes. For example, the fusion code XGC1[4] decomposes the toroidal simulation space across processes. In each step, electrons are pushed to an adjacent domain by Lorentz force in the electromagnetic field, and hence processes need to communicate the particle data to ensure the correctness. With hotspots in the storage system, each process progresses differently in time, and at some point a fast process has to sync with other processes for the particle data. For leadership systems where there are a large amount of cores, the speed gap between the fastest and slowest device can be huge. For example in the test runs in Figure 1(d), the speed of storage device #98 on Hopper is five times faster than that of storage device #48. The consequence is that CPU cycles are wasted on fast processes as a result of synchronization, which reduces the overall system utilization.

4. http://theorycodes.pppl.wikispaces.net/XGC1

3 I/O Re-Routing

The proposed I/O re-routing mitigates the imbalance in storage systems by re-directing the I/O traffic to less loaded storage locations, thereby reducing the amount of data being written to the congested devices. The core component of I/O re-routing is a virtual messaging layer that serves to disseminate the storage state to each process that participates in I/O. Each process is attached to this messaging layer, and is notified and re-directed to a new target if its current target becomes heavily congested. On the other hand, if a target has accepted too much new traffic, the messaging layer is responsible for identifying and forwarding a request to another location.

TABLE 1: An example of VML queue size over time. Re-routing occurs at $3t$ (shaded cells).

Time	Static I/O		Re-routing	
	Rank 0	Rank 1	Rank 0	Rank 1
0	2	2	2	2
t	1	2	1	2
$2t$	1	2	1	2
$3t$	0	2	1	1
$4t$	0	1	0	0
$5t$	0	0	0	0

3.1 Virtual Messaging Layer (VML)

In this section we address three key research questions in re-routing.

Question 1: *How can an application discover a highly loaded storage device? What is the criteria of marking a storage device as loaded?* A key challenge is that I/O statistics are not easily available in a parallel file system, and fine-grained

system level I/O statistics are typically not maintained due to the associated overhead. For example, on the production Lustre file system on Titan and GPFS on Mira[5], the per storage device statistics, such as bytes read/written and throughput, are not available to applications. On the other hand, the POSIX block interface is designed to write and read a number of bytes from a given offset, and performance information is not available and exposed from it. There has been some recent work on extending this block interface to gain more semantic knowledge [19], [20]. However, it is not straightforward to adapt these to obtaining I/O load on parallel storage systems. A possible yet intrusive approach is to have a monitoring daemon process run in the background at each storage device, which samples the I/O load periodically and reports back to a client process. This approach has its obvious performance disadvantages, as it incurs additional I/O costs to both storage and network.

A key design decision we made is to avoid measuring I/O load directly, and instead re-route I/O requests during the late stage of I/O, i.e., after all I/O requests issued to one of the storage devices are served. Such a storage device is deemed to be *idle*, and I/O re-routing of requests from other loaded devices is initiated. An example of the re-routing process is shown in Figure 2. The rationale is that if a storage device is under more interference than others, its application-level queue will be processed more slowly. Re-routing traffic away from it would reduce its load, while circumventing the expensive I/O monitoring. In Figure 2, we assume each I/O request takes t seconds to complete if no interference. Due to the interference from other applications, static I/O (i.e., no re-routing) takes $5t$ to finish I/O. With re-routing, a request issued to storage device #1 can be re-routed to storage device #0 at time $3t$, thus reducing the total I/O time to $4t$. Table 1 further shows the queue length over time.

Question 2: *How to re-direct a client process to a faster storage device?* In this work, we designed a thin messaging layer called VML to coordinate the I/O activities of clients including registration and re-routing. The purpose of VML is to provide the necessary messaging capabilities to client processes. VML receives the incoming I/O requests from client processes, grants permission to do I/O, and re-directs a request to a faster storage device. It internally uses short messages to disseminate storage state to each participating group so that the occurrence of congestion can be reported quickly and acted upon. On the other hand, the high-speed interconnects today have extremely low latency which also makes this approach favorable. For instance, the Cray Gemini network on Titan has a low message latency of 2.5 μs, which is nearly negligible as compared to disk I/O latencies, and can process tens of millions of MPI messages per second[6].

Question 3: *What are the assumptions about the behavior of the interfering applications? What are the changes to applications?* We treat the interfering applications as a black box and do not make assumptions about their patterns and I/O mechanisms. As compared to CALCioM [3], we do not force all applications on the system to adopt the same I/O APIs, which is in practice impossible. The interfering applications can use any I/O libraries by choice. We implemented the re-routing as a new transport into Adaptable I/O System (ADIOS) [21]. Existing ADIOS-enabled applications can take advantage of this work without changing their code.

TABLE 2: List of variables

Name	Definition
B	the bandwidth of a storage device
D	the application data block size
G	a group of processes
GC	a global coordinator
M	the total number of storage devices
M_{noise}	the total number of storage devices under noise
N	the total number of processes
N_{noise}	the total number of interfering processes
P	a process
SC	sub-coordinator
SD	storage device
TF	throttling factor

Figure 3 illustrates the overall architecture of the I/O re-routing framework. Here a group represents a set of processes that share a common initial storage target location to write (e.g., P_4, P_5, P_6 and P_7 sharing SD_1). This target may change when the ensuing I/O re-routing happens. In this case, the re-routed process will leave its original group and join a new group where the I/O load is expected to be lighter. VML uses a group-based two-level control framework to facilitate message exchange. To enable messages between groups, a *global coordinator* (GC) is selected for all groups and a *sub-coordinator* (SC) is selected to arbitrate the messaging between the SC and the individual *process* (P). The reason for the two-level design is to make the communication layer more scalable and to avoid having the GC handle messages directly from every process, which otherwise would greatly limit scalability. Implementation wise, SC/GC can be a process or a *pthread* launched alongside applications, which allows an application and VML to run concurrently so that the execution of one does not block the other. For the convenience of discussion, a process is denoted as P_i where $0 < i < N$ and N is the total number of processes. The number of storage devices is M and, without loss of generality, we also assume N is multiple of M. Therefore the number of processes that each group has is N/M and hence, for group G_i, the initial process set it contains is $[P_{N/M \cdot i}, P_{N/M \cdot (i+1)-1}]$. The SC_i is the sub-coordinator for group G_i, which manages exclusively the i-th storage device SD_i. The one-to-one mapping between SC_i and SD_i assumed here is for the convenience of discussion and in general an SC can write to any SD. Note SC_i can be attached to one of the processes in the group, e.g., $P_{N/M \cdot i}$, and GC can be attached to one of the SCs.

Question 4: *What are the scalability requirements in HPC systems?* Resource management systems (e.g. YARN [22] and Mesos [23]) in non-HPC clusters are designed to scale with the size of clusters and the number of concurrent jobs. In contrast, the scalability of re-routing framework is constrained by the HPC cluster hardware configuration, which are much less frequently scaled out compared to non-

5. https://www.alcf.anl.gov/mira

6. https://www.cray.com/sites/default/files/resources/CrayXE6 Brochure.pdf

HPC clusters, e.g. ORNL Titan launched in 2012 [24]. HPC jobs are performed with exclusively assigned computing resources and typically one job process is affiliated to a CPU core (Figure 2). Thus, despite the number of co-running job on an HPC cluster, the maximum number of job processes is under the constraint of computing resources (the number of CPU cores). For example, ORNL adopts PBS/Torque as a centralized resource management to allocate compute nodes to perform jobs. PBS/Torque accepts an HPC job, and allocates requested computing resources (e.g. compute nodes and CPU cores) to perform the job [25]. The computing resource capacity of ORNL Titan is the total of 299,008 CPU cores on 18,688 nodes. Thus, the re-routing framework takes group-based two-level control to facilitate message exchange, which is evaluated on the current leading HPC facilities with effective and efficient performance.

Question 5: *Will the I/O re-routing lead to oscillation?* For a storage system that has variable performance, the proposed solution will indeed write to different targets, and that is exactly the goal of this work. For example, if storage device #1 experiences fast, slow, and then fast performance in time, we believe we need to write to device #1 first, then some other device, and write to device #1 again. However, if there are no oscillations, device #1 will not be selected at the later time, and we lose the opportunity to shorten the I/O time. Carefully note that the granularity of oscillation in this work is application-level blocks (typically MB in size for HPC applications), and once a block starts being written, it cannot change its destination. Therefore, the oscillations are not very frequent and do not incur the extra overhead. In addition, our proposed Throttling Factor (TF) can limit the degree of the re-routing oscillation between sub-coordinators (SCs). TF denotes the ratio of the amount of data re-routed to the associated group versus the amount of data that originally belonged to the group and has been written (i.e., local data). If oscillation happens, the amount of re-routed data is increased resulting in the TF value increasing. When TF reaches a threshold, this SC will reject re-route requests and terminate the oscillation. We also explain this in Section 3.4.

3.2 Re-Routing

The re-routing runtime involves both *intra-* and *inter-group* level messages, which are listed in Appendix A. Intra-group messages occur only within a group G_i orchestrated by SC_i. Its main functionality is to provide admission control and re-direction to an individual process and update GC about its status (busy or idle). Before a process can start writing to storage, it must first ask for permission by sending $WRITE\_SUBMISSION$ message to its SC_i. If there are outstanding requests at SD_i being processed, the permission message is held off. Otherwise, a DO_WRITE message is sent off to the client process and the I/O operation proceeds. In the case of re-direction, a remote SC_j in group j will issue DO_WRITE to redirect a process to write to SD_j.

Inter-group messages are exchanged between two groups, e.g.,G_i and G_j, facilitated by SC_i, SC_j and GC. The runtime system utilizes these messages to discover an *idle* group and a *busy* group, and then offload requests from one to another. The I/O re-routing phase is jumpstarted by SC_i issuing $WRITE\_IDLE$ message to GC, indicating group G_i (and hence SD_i) has finished all its pending requests and is in idle state. This only occurs when all pending processes within G_i finish writing. GC is a central controller which keeps track of the busy/idle state of all storage locations, and upon receiving a $WRITE\_IDLE$ message, GC updates the state of G_i to *idle* and searches for a group G_j that is in the *busy* state. If group G_j is found, GC then initiates re-routing process via sending $REROUTE\_REQ$ to SC_j to request offloading portion of its I/O load. When this request is acknowledged by $REROUTE\_ACK$, GC constructs a $WRITE\_MORE$ message with its payload carrying the re-routed process ID - P_k to SC_i, and re-directs process P_k to write to the new storage device via DO_WRITE. If $REROUTE\_REJECT$ is received, for example, due to requests have been offloaded to other groups as a result of race condition, a new $REROUTE\_REQ$ will be constructed and sent out to another group that is busy. At this point, the re-routing phase is ended and no group is in idle state. The process of re-routing will start again when there is a new idle group emerged. When all I/O requests are finished (i.e., when file is closed), VML will be de-attached from each process and then released. An example of the message flow is illustrated in Figure 3.

3.3 Re-Routing with Throttling

The I/O re-routing scheme is aggressive in the sense that it places a larger burden on a fast storage device, and the consequence is that a varying amount of data is written to each storage device, depending on the degree of imbalance experienced. This is problematic for data that will be read back in the future (i.e., post-processing) when hotspots may disappear or transition to a different pattern. The overly aggressive nature of re-routing write operations can create a secondary load imbalance for reading, i.e., some storage devices have significantly more data to read than others. To address this problem, we apply a throttling technique that can limit how much data is allowed to be re-routed during writing, thus avoiding hotspots while lessening the impact of re-routing on read performance. The key idea is that since there are a small number of stragglers and most of clients are relatively fast, there are a large number of candidate storage devices to be offloaded to. Therefore, we should balance the traffic to them, instead of offloading solely to the fastest ones. To achieve this, we introduce the notion of *throttling factor* (TF) for each SC, which is defined as the ratio of the amount of data re-routed to the associated group versus the amount of data that originally belonged to the group *and* has been written (i.e., local data). For each storage device, TF essentially limits the size of new I/O requests that can be accepted inbound by an SC. If the current ratio is no greater than TF, the re-routed request will be granted. Otherwise, the storage device is considered to have accommodated too many re-routed requests and the new request will be rejected. In that case GC will look for the next idle SC.

In Appendix A, we detail the re-routing algorithms at GC, SC and individual processes.

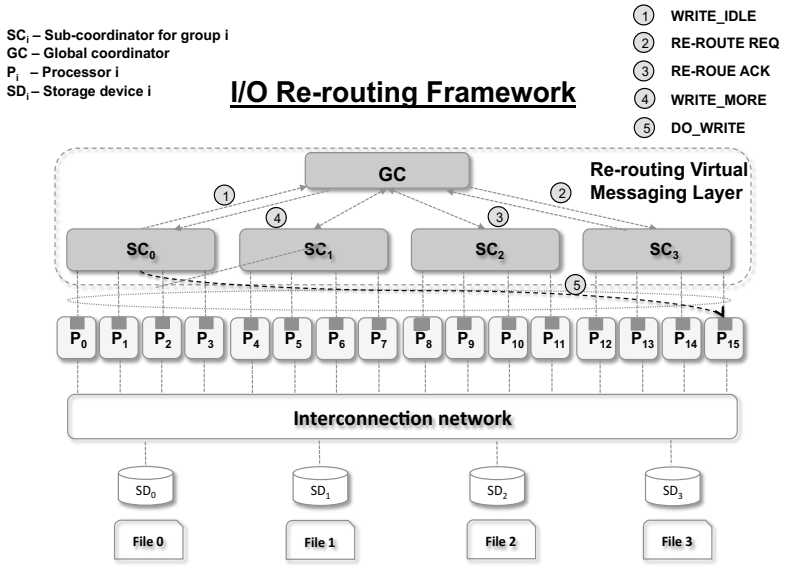


Fig. 3: I/O Re-routing. Note this figure illustrates a sample system that has 16 MPI processes and 4 storage devices. The re-routing starts with SC_0 sending $WRITE\_IDLE$ to GC, notifying that SD_0 is the idle state. GC subsequently identifies SD_3 to be in the busy state, and sends $REROUTE\_REQ$ to request re-routing. SC_3 acknowledges the request via $REROUTE\_ACK$ with payload P_{15}. Next, SC_0 sends DO_WRITE to P_{15} and direct the MPI process to write to SD_0.

3.4 Analytical Modeling of Throttling Factor

This section addresses the question of how to determine the best TF value using a simplistic analytical model. Although in reality, it is nearly impossible to have priori knowledge about the pattern of interference, this section aims to gain a theoretical understanding of the relationships between the intensity of interference and TF. Suppose the number of interfering processes is N_{noise} and the number of storage devices that are under noise is M_{noise}. Without losing generality, we assume SD_0 to $SD_{M_{noise}-1}$ are the ones injected with noise. Therefore there are $\frac{N_{noise}}{M_{noise}}$ competing noise streams on each of the interfered storage devices. To simplify the model, we also assume that noise is continuously streamed into the system and each data block has the same size as the application requests. An application data block is denoted as D_i and the bandwidth of storage device is B. Without noise, each storage should finish within $T_i = \frac{D_i}{B}$ seconds theoretically, neglecting disk seek overhead. After noise is injected, the interfered storage should complete within $T_i = \frac{D_i}{B} \cdot (\frac{N_{noise}}{M_{noise}} + 1)$ seconds, where $0 \leq i < \frac{N_{noise}}{M_{noise}}$. If re-routing is enabled, each storage should have equal or similar timing as a result of offloading, because a slow storage device would continue to offload until it catches up with the fast ones. Suppose a slow storage device, SD_j, offloads $D_{out,j}, 0 \leq j < M_{noise}$, and a fast device receives $D_{in,i}, M_{noise} \leq i < M$. Those SDs under noise should have

$$TF_i = \frac{D_{in,i}}{D} = \frac{N_{noise}}{M_{noise} \cdot ((\frac{N_{noise}}{M_{noise}} + 1) \cdot (\frac{M - M_{noise}}{M_{noise}}) + 1)} \quad (1)$$

The details can be found in prior work [1]. With the threshold, TF_i, I/O re-routing oscillation can also be in control. Here, D is the total amount of written data by a SC. If re-routing is enabled and every process writes equal amount of data, then D is a constant [1]. $\frac{D_{in,i}}{D}$ is determined by $D_{in,i}$, which is the re-routed data. When re-routing oscillation happens, $D_{in,i}$ and $\frac{D_{in,i}}{D}$ are increased. When $\frac{D_{in,i}}{D}$ reaches to the threshold TF_i, the I/O re-routing behavior of SC_i is suspended, resulting in the termination of the oscillation.

Next in Section 5.3, we will examine the analytical model against the experiment results for the synthetic benchmark and Pixie3D. **For the heavy interference** setup in this paper, we have $N_{noise} = 64$, $M_{noise} = 4$, $M = 16$, therefore $TF_i = 0.31$. **For the light interference** setup, we have $N_{noise} = 16$, $M_{noise} = 1$, $M = 16$, therefore $TF_i = 0.06$.

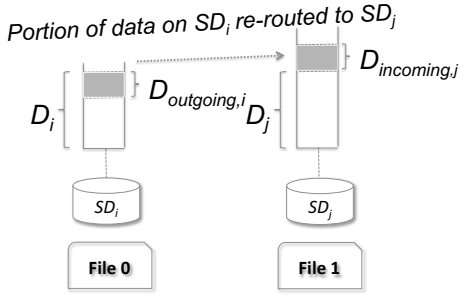


Fig. 4: An analytical model of re-routing.

3.5 Understanding the Message Overhead

I/O re-routing in conjunction with throttling involves short but frequent messages between GC, SCs and clients. Despite that the messaging load at 100K-core (shown in Section 5) is well within the capability of the interconnect, our goal here is to understand the behavior of VML and pinpoint any potential overhead that can be reduced, so that re-routing can be utilized on future larger systems where the messaging load is expected to be higher. Figure 5 illustrates percentage-wise the amount of messages exchanged at GC for XGC1. To understand the messaging overhead,

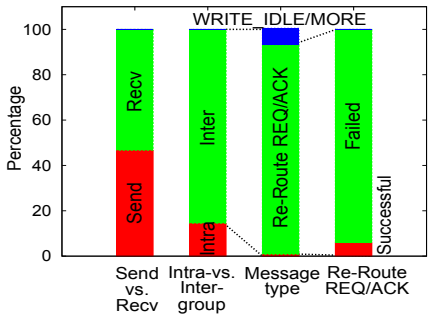


Fig. 5: XGC1 messaging overhead at GC.

we classify the messages into four categories: messages sent vs. received (1st bar), intra-group vs. inter-group messages (2nd bar), the breakdown of inter-group messages (3rd bar), and re-routing success vs. failure rate (4th bar). The inter-group messages that are recorded include (*GROUP_CLOSE*, *GROUP_CLOSE_ACK*), (*REROUTE_REQ*, *REROUTE_ACK* or *REROUTE_REJECT*), ($WRITE\_IDLE$ and $WRITE\_MORE$) pairs. In Figure 5, it is clear that $REROUTE\_REQ$ related messages dominate the inter-group messages and, most importantly, failed re-route messages, i.e., $REROUTE\_REJECT$, constitutes a high percentage of messages exchanged. The occurrence of failed re-routing is due to the race condition where a large number of SCs finish at about the same time and compete for requests from a small number of busy SCs. This is mainly due to that, GC is unaware of the procession of queue size of each SC beyond being busy or idle, and can direct a number of SCs to offload from a busy SC whose queue length is short.

To solve this problem, we aim to provide limited visibility into the queue length via issuing a *one-time* probe message, called *STATUS_INQ*, to each SC as soon as the very first $WRIT\_IDLE$ is received by GC. Once an SC receives *STATUS_INQ*, it immediately sends back $STATUS\_REPLY$ reporting its current queue size. When GC collects all the $STATUS\_REPLY$ messages, it sorts the SCs by the queue size. For the subsequent $WRITE\_IDLE$ messages, *REROUTE_REQ* will be sent to the SC with the highest queue size first. If failed, the SC with the second highest queue length will be tried, and so forth. In doing so, the chance of success can be increased by avoiding those with a short queue.

4 In-transit Buffering

In-memory buffering (e.g., burst buffer) provides an isolation layer between applications and the storage systems, and utilizes fast storage medium to provide higher perceived performance to applications. Data buffered can be drained asynchronously to persistent storage, so that applications are shielded from the potentially higher contention. The in-memory buffering can be local or remote or both. Since HPC simulations typically output large amounts of data, the local buffering may comprise the memory usage of HPC applications, and therefore we choose in-transit buffering where data are streamed to auxiliary compute nodes. The key idea of using in-transit buffering, Figure 6, is that data can be staged from simulations to auxiliary (remote) compute resources through high-speed interconnect, and simulations can proceed asynchronously while data can be drained from the buffer to the storage systems.

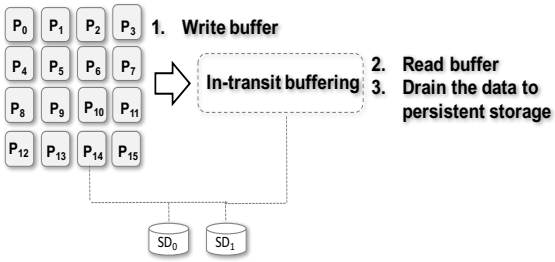
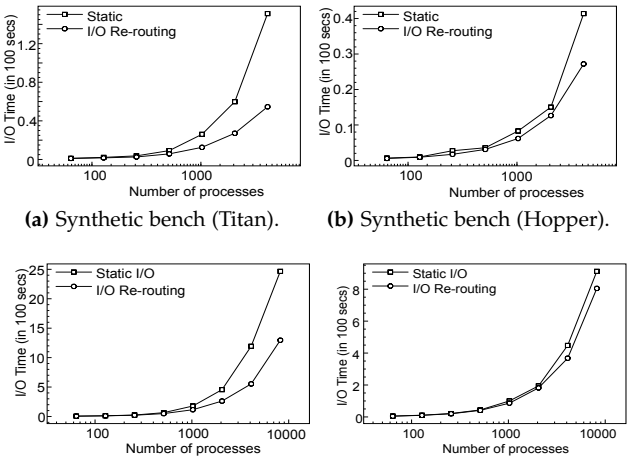


Fig. 6: In-transit buffering.

I/O buffering is not a new idea. In fact, buffering has long been used as a standard method to minimize I/O overhead. However, it is not designed to support large parallel I/O operations and it is unclear that how effective is the in-transit buffering across interconnect and draining in dealing with I/O variability. Moreover, it lacks metadata support, which further limits its broad adoption within HPC I/O. More recently, the creation of hardware based buffering areas, known as burst buffers [26], has started to emerge in a production HPC environment, such as the Cori system at NERSC and Summit[7] at ORNL, with additional featured being developed. This type of system-wide and special-purpose hardware supported buffering may help reduce I/O overhead, but it is unclear its effectiveness in reducing I/O variability. For example, on Cori, the burst buffer is featured as a storage system that absorbs burst. However, the fact that a DataWarp node[8] is shared among all applications, thus still creating opportunities for inter-job interferences. Therefore, this work uses the in-memory buffering enabled by Dataspaces [27], where a small number of dedicated compute nodes are utilized to absorb and write out data to parallel file systems asynchronously. DataSpaces is a scalable data sharing framework designed to support dynamic interaction and coordination amongst large-scale scientific applications. DataSpaces is built on a RDMA-based asynchronous memory-to-memory data transport layer to provide low latency and high throughput data transfer.

We consider the following three quantities to help further characterize I/O variability: mean I/O time (μ), standard deviation of the I/O times from the mean (σ), and the coefficient of variation (c_v). The c_v, also known as relative standard deviation, is defined by σ/μ, as the ratio of the standard deviation σ to the mean μ. As shown later, contrary to the intuition, using the in-transit buffering does not provide a complete solution for the I/O variability woes. The effectiveness of in-transit buffering is contingent upon the volume and velocity of data produced by an application. Furthermore, to fairly evaluate staging solutions, we must weigh the performance gains against the additional cost of compute resources.

7. https://www.olcf.ornl.gov/summit/
8. https://www.cray.com/datawarp



(a) Synthetic bench (Titan). **(b)** Synthetic bench (Hopper).

(c) Pixie3D (Titan). **(d)** Pixie3D (Hopper).

Fig. 7: Runs with manual noise injection.

5 EVALUATION

To gauge the effectiveness of I/O re-routing and throttling, we collected measurements on two leadership systems: Titan and Hopper. Titan is a Cray XK7 machine and has 18,688 compute nodes, in addition to the dedicated login and service nodes. Each compute node contains a 16-core 2.2 GHz AMD Opteron processor and an NVIDIA Kepler GPU. Each node has 2 GB DDR3 memory per core, and a Gemini interconnect. We run a number of production applications on the Lustre file system, which has 1,008 storage devices. Hopper is a Cray XE6 machine and has 6,384 compute nodes, with each consisting of two twelve-core AMD MagnyCours 2.1 GHz processors, with 2 GB DDR3 memory per core. Hopper has a total of 156 storage devices. Our experiments include the runs in which the interference is from other applications, and runs in which we manually inject noise. For the latter, the purpose is to control the interference so that for both re-routing and static I/O, the intensity of interferences can be maintained to be identical to the best of our abilities. For those runs that have manual noise injected, by default 16 MB data blocks are continuously written to the file system, unless otherwise noted. For the results with I/O re-routing, TF is set to 0.1 by default. We tested two simple interference settings: heavy interference and light interference. For *heavy interference*, $N_{noise} = 64$, $M_{noise} = 4$. For *light interference*, $N_{noise} = 16$, $M_{noise} = 1$. Each data point is the average of 20 measurements collected, unless otherwise noted.

We ran four sets of workloads: a synthetic benchmark, Pixie3D (kernel) [28], fusion XGC1(production application) and QMCPack[9] (production application): The **synthetic benchmark** used in our tests is a simple MPI-based parallel code that writes and reads configurable size of data per process, with no computation involved.

Pixie3D [28] is a large extended *Magneto-Hydro-Dynamic* (MHD) code that solves extended MHD equations using fully implicit Newton-Krylov algorithms. The output from Pixie3D contains eight double-precision 3D cubes. Each 3D array is typically sized $256 \times 256 \times 256$ (hero), $128 \times 128 \times 128$ (large), $64 \times 64 \times 64$ (medium) and $32 \times 32 \times 32$ (small), depending on the desired resolution as well as memory constraint. Here we only present the results of $32 \times 32 \times 32$ setup for Pixie3D and other settings bear similar findings.

XGC1 is a gyrokinetic particle-in-cell code that simulates the development of an edge pedestal in the radial density and temperature profiles of tokamak fusion plasmas. XGC1 calculation can scale up very well to a large amount of MPI processes with each process generating tens of megabytes, and therefore fast I/O is crucial to XGC1. In general, XGC I/O outputs both checkpoint and analysis data, whose frequency being adjustable depending the science needs as well as I/O throughput sustained from the storage system. In the runs conducted here, checkpoint period ($sml\_restart\_write\_period$) is set to 10 timesteps, and analysis output period ($diag\_3d\_period$) is set to 5 timesteps (2 MB field data and 2 KB 2D diagnosis output). Although the analysis output is more frequent than the checkpoint, the size is much smaller and is dumped out in serial. Therefore in the XGC1 performance study, we only focus on the more challenging checkpoint output.

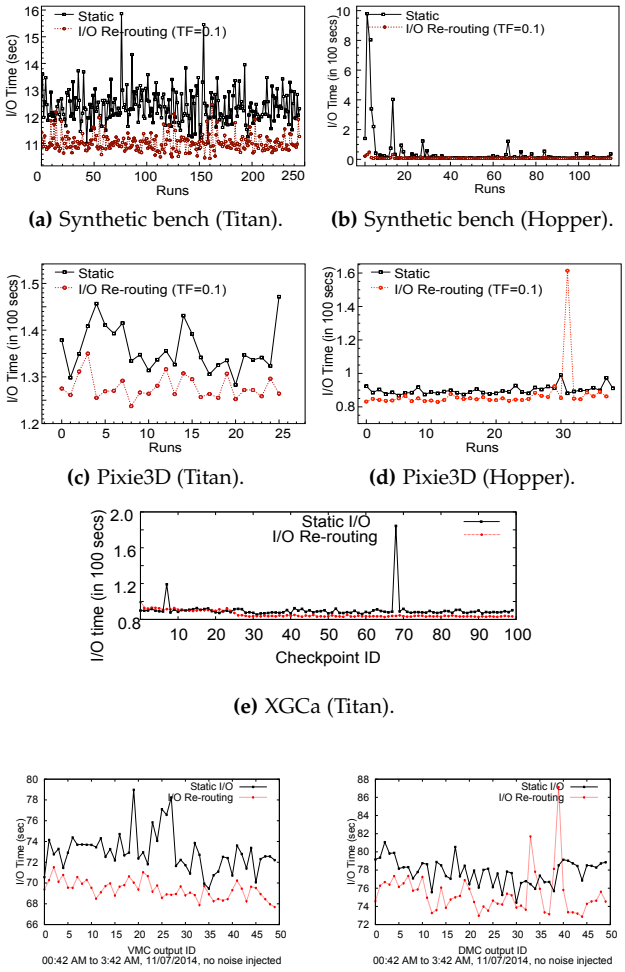
QMCPack is an open-source many-body ab initio Quantum Monte Carlo code for computing the electronic structure of atoms, molecules, and solids. It can be used to perform both *Variational Monte Carlo* (VMC) and *Diffusion Monte Carlo* (DMC) simulations. The QMCPack runs were conducted using weak-scaling, and there are two key parameters: $steps$, which is the total number of simulation steps, and $blocks$, which is the number of VMC/DMC blocks. In VMC runs, we dump out two $10 \times 256 \times 3$ single-precision floating point variables from each process, *momentum* (which is of complex type), and *position*. In DMC runs, the size of the variables is $531 \times 256 \times 3$.

5.1 Write Performance

In this section, we evaluate the write performance of re-routing with and without manual noise injection. Figure 7(a) and (b) show the total write time of the synthetic benchmark on Titan and Hopper, respectively. Herein the benchmark writes 2 MB chunks continuously from 64 to 4,096 processes with light noise being injected. It is clear that the I/O re-routing results in a much improved write performance at all process counts, as compared to the static I/O. The performance gain is particularly notable at a large scale. For example, at 4,096-core, I/O re-routing yields a 67% reduction of write time on Titan and 33% reduction on Hopper, respectively. These runs demonstrate that as the scale becomes larger, the static I/O is more susceptible to the interference and brings larger performance degradations against the re-routing. Similar results of Pixie3D are shown in Figure 7(c) and (d).

Figure 8 shows a set of runs conducted during the production windows of Titan and Hopper, and the interferences are from other concurrent applications. These runs give us an indication of how the I/O re-routing performs in a realistic environment, where interferences are of a stochastic nature. Figure 8(a) shows the benchmark performance observed during a 2-hour production window on Titan and Hopper. The static I/O and re-routing runs

9. http://qmcpack.org



(a) Synthetic bench (Titan). **(b)** Synthetic bench (Hopper).

(c) Pixie3D (Titan). **(d)** Pixie3D (Hopper).

(e) XGCa (Titan).

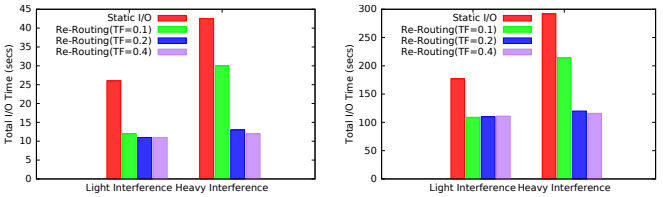
(f) VMC (Titan). **(g)** DMC (Titan).

Fig. 8: Runs during production windows.

were interleaved to achieve maximum fairness. Here the I/O re-routing yields a much lower variance versus the static I/O (28 versus 16,136 on Hopper), as well as a lower average I/O time (8.47 secs vs 41.42 secs on Hopper). Figure 8(c)-(g) similarly show Pixie3D, XGC1, and QMCPack, respectively. They were set up to compute 1,000 timesteps ($sml\_mstep = 1000$) and produce 100 checkpoints from 1,024 compute nodes with one process per core. Figure 8(f) and (g) show VMC and DMC production run performance at 1,024 processes with 50 blocks and 100 steps.

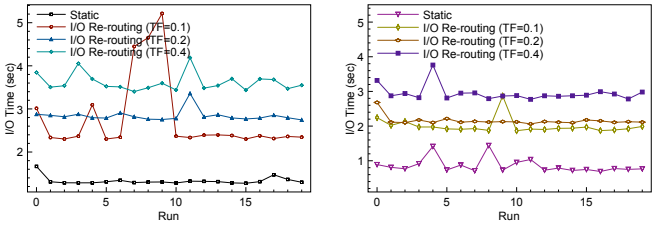
5.2 Impact of TF to Write and Read

In Figure 9, we further evaluate the sensitivity of I/O performance with regard to TF. For Pixie3D runs (Figure 9(b)), the static I/O exhibits a much longer I/O time, particularly under the heavy interference, 290 secs vs. 125 secs of re-routing, resulting in a 57% improvement with TF set to 0.2. We observed that under the light interference, a small TF (i.e., TF=0.1) is sufficient to get the most out of re-routing. Meanwhile, under the heavy interference, a larger TF (i.e., TF=0.2) is needed to allow a higher percentage of traffic being re-routed. This behavior, that a relatively small TF is sufficient to achieve a large performance margin, can be attributed to the observation that a large fraction of storage devices perform well and only a small percentage are stragglers, as observed in Figure 1. A low TF is sufficient in re-routing the stragglers to fast storage devices - if one storage device exceeds TF and cannot take more traffic, GC can find the next available fast storage devices. However, with the intensity of interference getting stronger, increasing TF is needed to allow more traffic to be re-routed.



(a) Synthetic bech. **(b)** Pixie3D.

Fig. 9: Impact of TF to write.



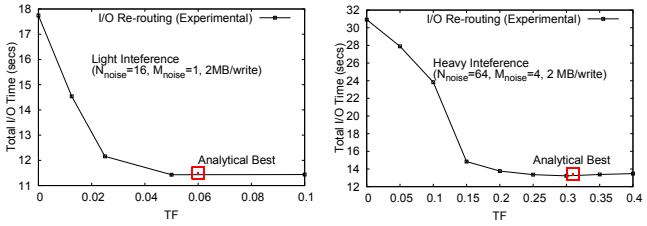
(a) Synthetic bench. **(b)** Pixie3D.

Fig. 10: Impact of TF to read.

Figure 10 shows the time it takes to read the data previously generated under the heavy interference. For the benchmark runs, we read the entire datasets that were written previously. For Pixie3D, we extract the eighth cube from the output data, i.e., post-processing data retrieval. The read times here show a tiered separation among curves with different TF values. Although the static I/O is inefficient for write performance in the presence of interference, it does yield the best performance for reading, as data are evenly distributed across storage devices. As TF increases, the read time also increases. To achieve a balance between write and read, one needs to set a relatively small TF so that re-routing can come into play with minimal impact to read. The details can be found in Section 3.4.

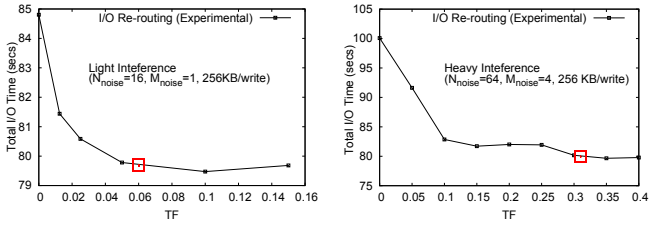
5.3 Analytical vs. Experimental Results of TF

Figure 11 and 12 evaluate the fidelity of the analytical model we developed. The plots here show both the measurements and the optimal value of TF calculated by our analytical model. It is evident that the optimal value of TF calculated from the analytical model approximates the one that yields the lowest I/O time. Enlarging TF beyond this optimal value does not yield substantial write performance improvement. For Pixie3D runs, the light and heavy interference were adjusted to 256 KB/write to match the size of Pixie3D $32 \times 32 \times 32$ cube, to make the model assumption valid. In Figure 12, the model-predicted TF yields an increase of I/O time that is below 0.4% for light interference and 0.6% for heavy interference, respectively.



(a) Light interference. **(b)** Heavy interference.

Fig. 11: Model-predicted TF (Synthetic bench)

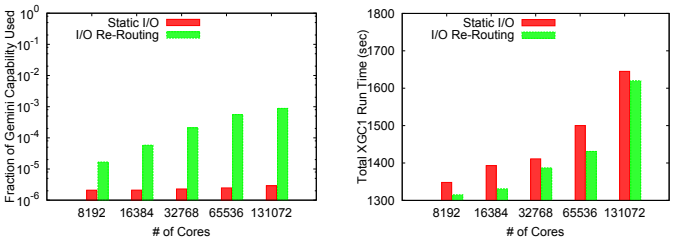


(a) Light interference. **(b)** Heavy interference.

Fig. 12: Model-predicted TF (Pixie3D)

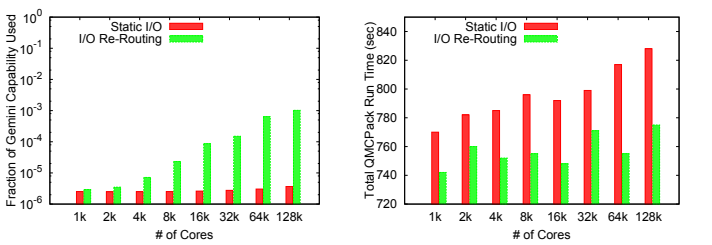
5.4 Messaging Overhead

In Figure 13(a), the messaging load of XGC1 at GC, with respect to the fraction of interconnect capability consumed, is measured till 131,072-core. We scale up the runs using the weak scaling by increasing sml_nphi_total, which is the number of toroidal domains simulated in XGC1. Note that the core-to-storage ratio is kept the same to ensure the storage resources used are proportional to the compute resources. Since GC handles messages from members of its group and each SC to coordinate traffic redirection, GC is anticipated to be the location with the highest messaging load in the system. While we observe a clear increased messaging load of the re-routing over the static I/O, the load is well below what the interconnect can sustain (tens of millions of MPI messages per second). In Figure 13(b), the application run time, including that of I/O and computation, is measured against the number of cores. With the re-routing outperforming the static I/O at all core counts, this result illustrates the overall effectiveness of I/O re-routing, even with the added messaging overhead. Figure 14 shows similar results for QMCPack till 128,000-core.



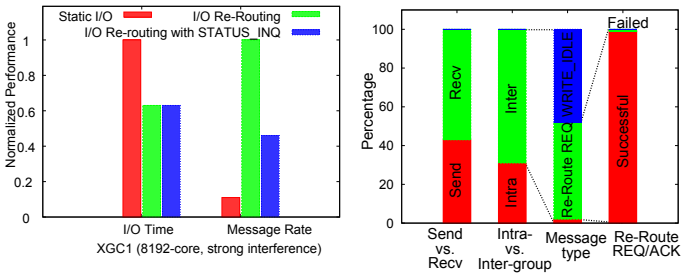
(a) Messaging overhead at GC. **(b)** Total run time.

Fig. 13: XGC1.



(a) Messaging overhead at GC (blocks=10,steps=20). **(b)** Total run time (blocks=10,steps=20).

Fig. 14: QMCPack.



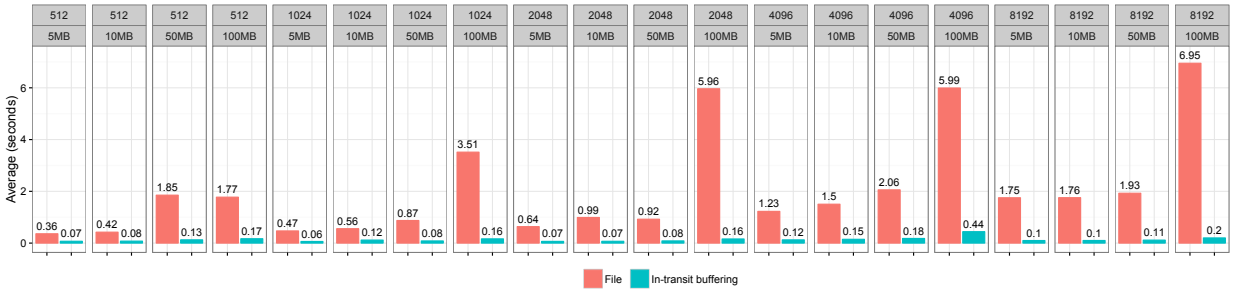
(a) Performance comparison. **(b)** Message classification.

Fig. 15: XGC1 with a one-time $STATUS\_INQ$.

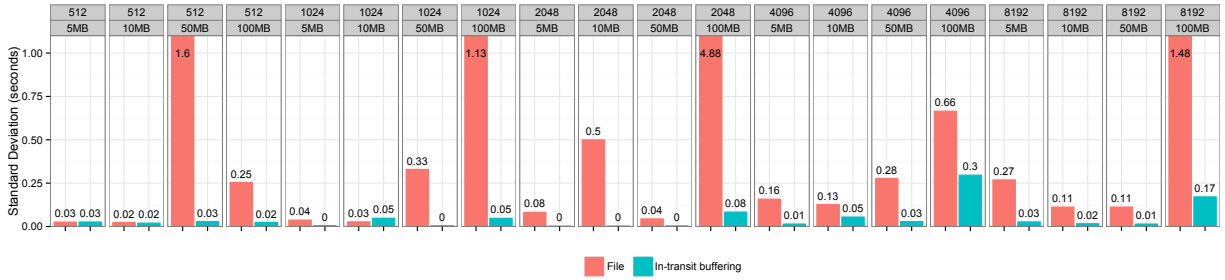
In Figure 15, we measure the effectiveness of the one-time $STATUS\_INQ$ in reducing the messaging load (Section 3.5) for XGC1. In Figure 15(a), the normalized I/O time and message rate are measured, comparing the static I/O and re-routing with/without $STATUS\_INQ$. With $STATUS\_INQ$, the overall messaging load is reduced by about 50%, while achieving the same I/O time as the re-routing without $STATUS\_INQ$. There are two conclusions we can draw from this result. First, even without further message reduction, the incurred messaging load of re-routing can be very well handled by the interconnect on the leadership systems, and therefore, the overall application performance is not negatively impacted by the VML-enabled re-routing. Second, a one-time probe into storage state during the re-routing phase can further help identify the stragglers and reduce the failed messages. Figure 15(b) shows percentage-wise the amount of messages sent and received, intra-group vs. inter-group messages, the breakdown of inter-group messages, and success vs. failure rate of $REROUTE\_REQ$. Compared to Figure 5, the percentage of $REROUTE\_REJECT$ is significantly reduced, resulting in a lower messaging rate.

5.5 The Effectiveness of In-Transit Buffering

To measure file I/O performance in a fair way, we restrict ourselves to use only 512 storage devices for all cases, and maintain the total number of processes to be multiples of 512, so that each storage device can have equal workload. For the in-transit buffering, we test DataSpaces with the direct end-to-end data transport method. DataSpaces requires a staging area consisting of additional compute nodes for running metadata servers and staging nodes. We limit this overhead to be less than 5% of the compute nodes allocated to the main simulation, i.e., we prepare an extra 1/32



(a) Average I/O performance.



(b) Standard deviation.

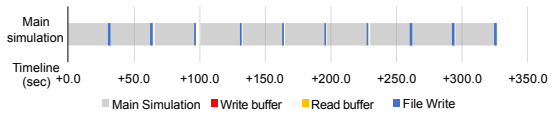
(c) Coefficient of I/O variation.

Fig. 16: Performance comparison between file I/O and in-transit buffering. We conducted experiments over a few weeks of time period with different data volumes (from 5 MB up to 100 MB of data per process) over different scales, ranging from 512 up to 8,192 nodes. All tests were performed on Titan.

of the number of compute nodes for running DataSpaces servers and an additional 1/64 compute nodes for buffering nodes (i.e., 4.7% overhead in total). First, we measure the performance of parallel data writing and compare the performance between file I/O and in-transit buffering in various data volumes and scales. We let each process write four different data sizes (5, 10, 50, and 100 MB data per process, respectively) in five different scales using 512, 1,024, 2,048, 4,096, and 8,192 compute nodes (1 process per node). To capture the variability during production windows, we conducted the experiments several times (ranging from 45 up to 180) for each configuration over a few weeks. Figure 16 shows (a) the average I/O times based on the results we observed, and (b) the absolute variances. In-transit buffering not only outperforms file I/O in all the cases we tested, but also shows smaller absolute variability. However, in Figure 16(c), we observe that the relative variability of in-transit buffering is worse than that of file I/O in a number of cases. Although the I/O time has been reduced, it may still be subject to be affected by the external effects or noises on the interconnect, and thus the relative variance is not reduced.

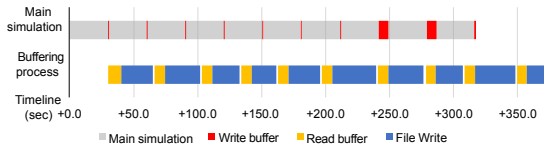
Next, we perform a set of experiments to understand how the interference can affect the performance of in-transit buffering, and in turn can disturb the performance of the main simulation. For example, Figure 17 shows an instance of an application that finishes 10 iterations. At each step, it performs a 30-second computation, followed by 100 GB checkpoint writing. We use three different I/O methods: (a) file I/O without in-transit buffering, (b) in-transit buffering without interference, and (c) in-transit buffering with interference. 1,024 processes are used for the main simulation and 32 processes are assigned for in-transit buffering. To observe the performance under the influence of interference, we manually inject interference using only 4 processes. Each interference process writes 100 MB of data concurrently onto only one storage device that the main simulation works on.

In Figure 17, we can make a few observations. First, compared with (a) and (b), we observe the in-transit buffering can reduce the main simulation time as it hands over its I/O to the interconnect, instead of persistent storage. Second,



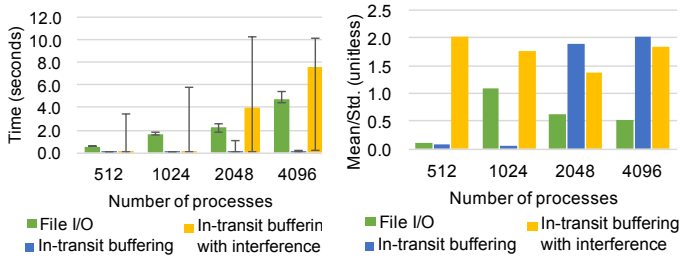
(a) No in-transit buffering

(b) In-transit buffering without interference



(c) In-transit buffering with interference

Fig. 17: An instance of 10 checkpoint writing tasks (a) with only file I/O, (b) with in-transit buffering without interference, and (c) in-transit buffering with interference. 1,024 processes are used for the main simulation and 32 processes are for in-transit buffering. The main computation takes 30 seconds, followed by 100 GB checkpoint writing. To inject interference, we use only 4 processes and each process writes 100 MB of data concurrently



(a) Average checkpoint performance. **(b)** Coefficient of variation.

Fig. 18: The performances of 10 checkpoint writing with 30 second intervals using three different methods: file I/O only, in-transit buffering, and in-transit buffering with interference. (a) shows the average performance at each iteration. The error bars represent the minimum and maximum values, respectively. (b) represent the relative variations.

the file I/O performance in (a) varies at each write but, in the in-transit buffering (b), the variation has been greatly reduced. Third, more importantly, as shown in (c), under the influence of interference, we observe the delays in buffering. Please note the variations of the I/O performance in the in-transit buffering, and they, in turn, cause the disturbance in the main simulation. For the in-transit buffers, data are first held in memory, and then drained to the file system. Due to the limited memory capacity of buffering nodes, any delays in the draining to the file system create corresponding delays in the main simulation running in the compute nodes, to avoid buffer being overwritten.

We repeat the above experiment setting (10 checkpoint writing after every 30 second interval by using three different methods) with different scales. We increase the number processes of the main simulation from 512 up to 4,096. The number of in-transit buffering nodes is proportionally increased from 16 up to 128. We maintain the number of processes causing interference to be 4, and the interference is injected to a single storage device. In Figure 18(a), the in-transit buffering outperforms the file I/O if there is no interference. However, the in-transit buffering with interference show severe performance loss and it is even worse than the file I/O. Also notice that the large variation (depicted in the error bars) in the buffering under the interference. Figure 18(b) shows the coefficient of variation. Both methods (without and with interference) yield a large c_v, which implies the relative variation of in-transit buffering is no better than file I/O.

6 Related Work

In the HPC domain, I/O contentions have demonstrated to be at a whole new level simply due to the level of currency available. With over hundreds of thousands cores available on a HPC system, the likelihood that two clients compete for the same storage resource is fairly high. The earlier work [2], [4] made a number of measurements on a few large HPC machines during their production runs and showed contentions can lead to serious I/O performance degradation in the system. The prior work on the I/O variability [4] tackled the problem by first looking at write optimization only using a simplistic approach. The design of it does not allow for re-routing, and more importantly traffic throttling is not possible using this approach. Moreover, the proposed technique uses a single execution thread for both application write and communication on the coordinator and sub-coordinator processes. This entire design limits how effectively and quickly a coordinator can respond to storage load dynamics, and for light workload, e.g., Pixie3D small (section 4(b) in [4]), I/O requests cannot be processed responsively, thus adaptive I/O performance in this case is on par with non-adaptive I/O. This lowers the adaptability of the system in general when I/O hotspots are present. Very recently, researchers used server-side scheduling to address the problem [7]. While being quite effective, only small scale has been looked at so far and it is unclear whether they will work as efficiently at larger scales, such as leadership class (100,000+ cores) being considered in the paper. Work by Gainaru *et. al.* [7] leverages application past behaviors to tackle the congestion. However, for exploratory science where application patterns change, its effectiveness may be limited. In contrast, our work considers interference as a black box and does not assume that application patterns are known a priori. Dorier *et. al.* [3] proposed CALCioM API which allows an application to nicely inform its I/O intention so that contentions can be managed. Overall the work requires all applications on a system be CALCioM-compliant in order to be effective, which is impractical in general, and the added communication overhead between all applications is unclear.

There are works being done in non-HPC areas using I/O scheduling [29], [30], [31] and analytical approaches [9], [32], [33], [34], [35] to reduce contentions. The most relevant work is BASIL [29], an online storage management system that automatically performs virtual disk placement and load balancing across storage devices. BASIL formulated sampling-based empirical models to facilitate virtual

machine migrations between devices to balance load. Pesto [30], a similar work using enhanced empirical models to schedule workload, resolved a few unsolved challenges in BASIL, for example the model sensitivity to workloads. The time scale considered here is relatively large as frequent VM migrations incur high overhead and service disruptions even though a single migration causes minimal disruption.

In addition, explicit QoS scheduling mechanisms have been investigated in storage management solutions over the years. Wang and Merchant [12] proposed a distributed I/O scheduling algorithm that offers system-wide I/O fairness via distributed SFQ-based [36] scheduling on the coordinators (proxy nodes). In the research on proportionally I/O sharing, several existing schedulers, mclock [14], pclock [5], FSFQ(D) [37], and IOFlow [38], focus on proportionally sharing available I/O resources via allocations of I/O throughput (bytes/s), IOPS (I/O operations/s), or I/O queue depth, and vFair [15] focuses on proportionally sharing cloud storage to VMs with interleaving high/low I/O-concurrency. IBIS [13] is an interposed scheduler in the application-level to assign the I/O resources of HDFS to Hadoop related applications, such as MapReduce and HBase. Gulati [16] proposed a flow control solution, PARDA, to enforce fairness on storage arrays and LUNs. However, these work-conserving scheduling solutions aim to provide fairness among different I/O flows, rather than the workload variability and congestion on the storage nodes of HPC storage system. In addition, as far as we know, the system-wide and server-side I/O schedulers in all production systems in US leadership HPC centers are not allowed to be modified due to user privilege and security. As a result, we provide the unique application-level I/O rerouting that was at least possible to be put into production, without incurring intrusive changes to systems and applications.

In the HPC domain, especially in the next-generation HPC, understanding and solving the performance variability problem is important. Previous works [1], [2], [3], [4] show the root cause of such performance degradations and variations is the interference incurred by applications running simultaneously and sharing storage resources. Yildiz et al. [39] discovered that the performance variability is caused by cross-application I/O interference in HPC exascale storage systems. This work also reveals that in many situations interference is a result of bad flow control in the I/O path. MOANA [40] is a modeling and analysis approach that predicts HPC I/O variability on shared-memory. Some research efforts consider network contention as the major contributor to the I/O interference. Bhatele et al. [41] investigated the performance variability in Cray machines and Blue Gene systems. They found that the interference of multiple jobs that share the same network links is the primary factor for high performance variability. Kuo et al. [42] investigated the influence of file access patterns on the degree of interference observed. They found that chunk size can determine the degree of interference and the interference effect induced by various access patterns in HPC systems can slow down the applications by a factor of 5. Cao [43] reported a study to characterize the amount of variability in benchmarking modern storage stacks, for the purpose of ensuring stable I/O performance. In contrast, our work investigate the I/O variability and observe the severe performance loss due to the large relative I/O variations on the emerging burst buffer storage.

7 Conclusion

This paper attempts to resolve the I/O contention on leadership systems where there are massive parallelism. We propose a balanced client-assisted re-routing + throttling approach to alleviate the contention and a theoretical model to guide the setting of throttling factor (TF). This work tackles two key challenges that have not been fully resolved in the past: how to design a runtime scheduling system that can scale up to a large amount of cores, and how to avoid using on-node memory/cache, which would otherwise compromise fidelity and resolution of HPC applications, to manage contention. In addition, this work discusses the effectiveness of using burst buffers to deal with variability. The performance results indicate that the scheme works well, e.g., achieving 1.8x improvement in write and scaling up to 131,072 cores, for benchmarks as well as production applications on leadership class systems. We also thoroughly analyze the virtual messaging layer (VML) overhead and use a one-time $STATUS\_INQ$ message to probe each storage state and reduce the amount of $REROUTE\_REJECT$ messages as well as the overall messaging load. In contrast to the common belief that burst buffers can absorb I/O burst, it does not necessarily reduce the I/O variability.

8 Acknowledgements

This work is supported in part by the US National Science Foundation Grant CCF-1718297, CCF-1812861 and Department of Energy Advanced Scientific Computing Research. The work performed at Temple is partially sponsored by the US National Science Foundation under grants #1702474, #1717660, and #1813081. The experiments of this work are conducted on the HPC facilities managed by Oak Ridge National Lab and National Energy Research Scientific Computing Center.

References

[1] Q. Liu, N. Podhorszki, J. Choi, J. Logan, M. Wolf, S. Klasky, T. Kurc, and X. He, "Storerush: An application-level approach to harvesting idle storage in a best effort environment," *Procedia Computer Science*, vol. 108, pp. 475 – 484, 2017, the International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[2] Q. Liu, N. Podhorszki, J. Logan, and S. Klasky, "Runtime I/O Re-Routing + Throttling on HPC Storage," in *Proceedings of the 5th USENIX workshop on Hot Topics in Storage and File Systems*, ser. HotStorage'13, 2013, pp. 4–4.

[3] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "Calciom: Mitigating i/o interference in hpc systems through cross-application coordination," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 155–164.

[4] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing variability in the io performance of petascale storage systems," in *SC '10: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*.

[5] A. Gulati, A. Merchant, and P. J. Varman, "pclock: an arrival curve based approach for qos guarantees in shared storage systems," in *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '07, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/1254882.1254885

[6] A. Gulati, G. Shanmuganathan, X. Zhang, and P. Varman, "Demand based hierarchical qos using storage resource pools," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342822

[7] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the i/o of hpc applications under congestion," in *IEEE International Parallel and Distributed Processing Symposium, IPDPS'15*, 2015.

[8] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-side i/o coordination for parallel file systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 17.

[9] Y. Li, X. Lu, E. L. Miller, and D. D. Long, "Ascar: Automating contention management for high-performance storage systems," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. IEEE, 2015, pp. 1–16.

[10] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. [Online]. Available: http://doi.acm.org/10.1145/146941.146943

[11] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "Plfs: A checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 21:1–21:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654081

[12] Y. Wang and A. Merchant, "Proportional-share scheduling for distributed storage systems." in *FAST*, vol. 7, 2007, pp. 4–4.

[13] Y. Xu and M. Zhao, "Ibis: interposed big-data i/o scheduler," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 111–122.

[14] A. Gulati, A. Merchant, and P. J. Varman, "mclock: handling throughput variability for hypervisor io scheduling," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 437–450.

[15] H. Lu, B. Saltaformaggio, R. Kompella, and D. Xu, "vfair: Latency-aware fair storage scheduling via per-io cost-based differentiation," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 125–138.

[16] A. Gulati, I. Ahmad, C. A. Waldspurger *et al.*, "Parda: Proportional allocation of resources for distributed storage access." in *FAST*, vol. 9, 2009, pp. 85–98.

[17] H. Wang and P. J. Varman, "Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation." in *FAST*, vol. 14, 2014, pp. 229–242.

[18] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Transactions on networking*, vol. 4, no. 3, pp. 375–385, 1996.

[19] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "Database-aware semantically-smart storage," in *In Proceedings of the 4th USENIX Conference on File and Storage Technologies.*, 2005, pp. 239–252.

[20] G. R. Ganger, "Blurring the line between oses and storage devices," *Technical Report CMU-CS-01-166*, 2001.

[21] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu, "Hello adios: The challenges and lessons of developing leadership class i/o frameworks," *Concurr. Comput. : Pract. Exper.*, vol. 26, no. 7, pp. 1453–1473, May 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3125

[22] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[24] B. Bland, "Titan-early experience with the titan system at oak ridge national laboratory," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 2189–2211.

[25] "Titan User Guide,"]https://www.olcf.ornl.gov/for-users/system-user-guides/titan/running-jobs/.

[26] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, April 2012, pp. 1–11.

[27] C. Docan, M. Parashar, and S. Klasky, "Dataspaces: An interaction and coordination framework for coupled simulation workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851481

[28] L. Chacn, "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, 2004.

[29] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar, "Basil: automated io load balancing across storage devices," in *Proceedings of the 8th USENIX conference on File and storage technologies*, ser. FAST'10, 2010, pp. 13–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855511.1855524

[30] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal, "Pesto: online storage performance management in virtualized datacenters," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11, 2011, pp. 19:1–19:14. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038935

[31] D. Huang, D. Han, J. Wang, J. Yin, X. Chen, X. Zhang, J. Zhou, and M. Ye, "Achieving load balance for parallel data access on distributed file systems," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 388–402, 2018.

[32] M. Liu, Y. Jin, J. Zhai, Y. Zhai, Q. Shi, X. Ma, and W. Chen, "Acic: automatic cloud i/o configurator for hpc applications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 38.

[33] S. Groot, K. Goda, D. Yokoyama, M. Nakano, and M. Kitsuregawa, "Modeling i/o interference for data intensive distributed applications," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 343–350.

[34] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Server-side log data analytics for i/o workload characterization and coordination on large shared storage systems," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 819–829.

[35] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *Proceedings of the 2013 USENIX Annual Technical Conference*, no. EPFL-CONF-185984, 2013.

[36] P. Goyal, H. M. Vin, and H. Chen, "Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks," in *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 4. ACM, 1996, pp. 157–168.

[37] W. Jin, J. S. Chase, and J. Kaur, "Interposed proportional sharing for a storage service utility," *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, pp. 37–48, 2004.

[38] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "Ioflow: a software-defined storage architecture," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 182–196.

[39] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the root causes of cross-application i/o interference in hpc storage systems," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 750–759.

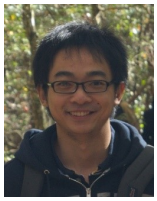
[40] K. W. Cameron, A. Anwar, Y. Cheng, L. Xu, B. Li, U. Ananth, T. Lux, Y. Hong, L. T. Watson, and A. R. Butt, "Moana: Modeling and analyzing i/o variability in parallel system experimental design," 2018.

[41] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: performance degradation due to nearby jobs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 41.

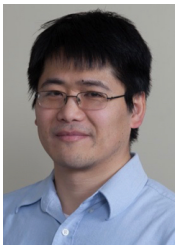
[42] C.-S. Kuo, A. Shah, A. Nomura, S. Matsuoka, and F. Wolf, "How file access patterns influence interference among cluster applica-

tions," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2014, pp. 185–193.

[43] Z. Cao, V. Tarasov, H. P. Raman, D. Hildebrand, and E. Zadok, "On the performance variation in modern storage stacks." in *FAST*, 2017, pp. 329–344.



Dan Huang is currently a postdoctoral researcher in the Department of Electrical and Computer Engineering at New Jersey Institute of Technology, Newark, NJ. Before this, he received his Ph.D. in computer engineering program at University of Central Florida. He received master and bachelor degrees in Southeast University and Jilin University respectively. His research interests are distributed storage systems, virtualization technology and the I/O of distributed system.



Qing Liu is an Assistant Professor in the Department of Electrical and Computer Engineering at New Jersey Institute of Technology, Newark, NJ, and holds a joint faculty appointment at Oak Ridge National Laboratory. Prior to that, he was a Research Scientist at Scientific Data Group, Oak Ridge National Laboratory till 2016. He received his Ph.D. in Computer Engineering from the University of New Mexico in 2008, M.S. and B.S., from Nanjing University of Posts and Telecom, China, in 2004 and 2001, respectively. His research interests include high-performance computing, large-scale data management, and high-speed networking.

Jong Choi is a Research Scientist working in Scientific Data Group, Computer Science and Mathematics Division, Oak Ridge National Laboratory (ORNL), Oak Ridge, Tennessee, USA. He earned his Ph.D. degree in Computer Science at Indiana University Bloomington in 2012 and his MS degree in Computer Science from New York University in 2004. His areas of research interest span data mining and machine learning algorithms, high-performance data-intensive computing, parallel and distributed systems for Cloud and Grid computing. More specifically, he is focusing on developing high-performance data mining algorithms and researching efficient run-time environments in Cloud and Grid systems.

Norbert Podhorszki is a Senior Scientist working in Scientific Data Group, Computer Science and Mathematics Division, Oak Ridge National Laboratory (ORNL), Oak Ridge, Tennessee, USA. He earned his Ph.D. degree and Master in Computer Science at Eotvos Lorand University in 2005 and 1995, respectively. His research interests include scientific data management, storage systems, and scientific workflows.

Scott Klasky is a Distinguished Scientist and the group leader for Scientific Data in the Computer Science and Mathematics Division at the Oak Ridge National Laboratory. He holds an appointment at the University of Tennessee, and Georgia Tech University. He obtained his Ph.D. in Physics from the University of Texas at Austin (1994). Dr. Klasky is a world expert in scientific computing and scientific data management, coauthoring over 200 papers, and leading several key projects in the department of energy.

Jeremy Logan is a Research Scientist at the University of Tennessee working closely with the ORNL Scientific Data Group. Jeremy earned a Ph.D. from the University of Maine in 2010 while studying parallel I/O performance. His current research interests include scientific computing applications, large-scale I/O systems, deep learning, and generative software techniques.

George Ostrouchov (Fellow ASA, Fellow AAAS) is a Senior Research Scientist at the Oak Ridge National Laboratory and Joint Faculty Professor at the University of Tennessee. He obtained his Ph.D. in Statistics from Iowa State University after a B.Math from the University of Waterloo. His interest is in scalable statistical computing for parallel and distributed systems.

Xubin He received the BS and MS degrees in computer science from Huazhong University of Science and Technology, China, in 1995 and 1997, respectively, and the PhD degree in electrical engineering from University of Rhode Island, Kingston, RI, in 2002. He is currently a professor in the Department of Computer and Information Sciences, Temple University, Philadelphia, PA. His research interests include computer architecture, data storage systems, virtualization, and high availability computing. Dr. He received the Ralph E. Powe Junior Faculty Enhancement Award in 2004 and the Sigma Xi Research Award (TTU Chapter) in 2005 and 2010. He is a senior member of the IEEE, a member of the IEEE Computer Society and USENIX.

Mathew Wolf is a Senior Scientist working in Scientific Data Group, Computer Science and Mathematics Division, Oak Ridge National Laboratory (ORNL), Oak Ridge, Tennessee, USA. He earned his Ph.D. degree at Georgia Institute of Technology, Atlanta, GA. His research targets high performance, scalable applications, particularly focused on I/O and adaptive event middlewares. Specific research topics include adaptive I/O interfaces, metadata-rich data services, creation of dynamic, semantic indexes for scientific data, and handling and fusion of heterogeneous data types.