I/O-Aware Batch Scheduling for Petascale Computing Systems

Zhou Zhou *, Xu Yang *, Dongfang Zhao *, Paul Rich †, Wei Tang †, Jia Wang *, Zhiling Lan *
* *Illinois Institute of Technology, Chicago, IL 60616, USA*
{*zzhou1, xyang56, dzhao8*}*@hawk.iit.edu, lan@iit.edu, jwang@ece.iit.edu*
† *Argonne National Laboratory, Argonne, IL 60439, USA*
**richp@alcf.anl.gov,* †*wtang@mcs.anl.gov*

***Abstract*—In Big Data era, the gap between the storage performance and application's I/O requirement is increasingly enlarged. I/O congestion caused by concurrent storage accesses from multiple applications is inevitable, and therefore severely harms the performance. Conventional approaches either focus on optimizing an application's access pattern individually or handle I/O requests on low-level storage layer without any knowledge from the upper-level applications. In this paper, we present a novel I/O-aware batch scheduling framework to coordinate ongoing I/O requests on petascale computing systems. The motivation behind this innovation is that the batch scheduler has a holistic view of both system state and jobs' activities and can control jobs' status on the fly during their execution. We treat a job's I/O requests as periodical sub-jobs within its lifecycle and transform the I/O congestion issue into a classical scheduling problem. We design two scheduling polices with different scheduling objectives either on user-oriented metrics or system performance. We conduct extensive trace-based simulations using real job traces and I/O traces from a production IBM Blue Gene/Q system. Experimental results demonstrate that our design can effectively improve job performance by more than 30% as well as system performance.**

I. Introduction

As we have already entered the age of petascale computing, the insatiable demand for more computing power continues to drive the deployment of ever-growing high-performance computing (HPC) systems [1]. Today's production systems already comprise hundreds of thousands processors [2][3], and are predicted to have millions at exascale computing by 2018 [4]. Along with the rapid evolution of micro-processors, the explosion of the amount of data should never be neglected. In this so-called "Big Data" era, the datasets generated by scientific applications are increasing exponentially in both volume and complexity [5][6].

While the computing systems can leverage more parallelism at exponential rate to gain computing performance improvement, the storage infrastructure performance is still improving at a significantly lower rate. For example, the IBM Blue Gene/Q (BG/Q) system *Mira* at Argonne Leadership Facility (ALCF) has a peak performance of 10 peta-flops which is 20 times as fast as its predecessor *Intrepid* (0.5 peta-flops), an IBM Blue Gene/P system [2][7][8][9]. But Mira's I/O throughput increases only 3 times comparing with Intrepid [9]. As shown in Figure 1, Mira's computing nodes have the capability to drive the I/O network at a full speed of 1536 GB/s while its storage system can only deliver 250 GB/s. This implies that one quarter of the computing system can saturate the bandwidth of its storage system and incur I/O congestion, leading to potential application performance degradation. Consequently, the increasingly enlarged gap between applications' I/O requirement and the storage performance makes big data processing one of the most challenging problems.

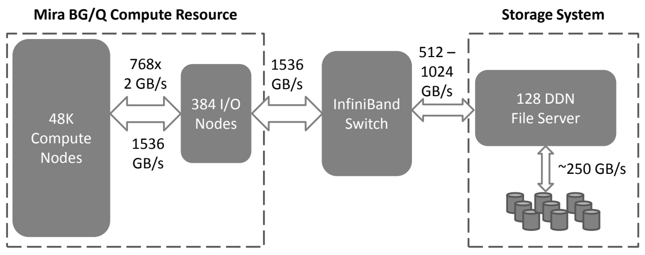


Figure 1. Storage system architecture of the 10-petaflop Mira at Argonne National Laboratory

In order to bridge this gap, numerous studies have been conducted from different perspectives. A well-known approach is to boost I/O performance through I/O middleware (e.g., PLFS [10], DataStager [11], Damaris [12], IOrchestrator [13]) or I/O library (e.g., HDF5 [14], NetCDF [15]). As SSD becomes more cost effective, memory-class storage is proposed to cache the temporary data from compute nodes so as to mitigate I/O bandwidth contention [16][17]. Additionally, researchers also propose new architectural changes to push the I/O handling closer to the compute resource by installing burst buffer on I/O nodes [18].

The aforementioned studies typically either focus on application-level optimization or are implemented on I/O nodes or file servers. From the application's perspective, some techniques are dedicated to individually optimizing this application's I/O access pattern. The I/O interference among multiple applications are not taken into account. On the other hand, from the system view, most approaches handle lower-level I/O requests in an uncoordinated manner without any knowledge from the upper-level applications. For instance, when processing I/O accesses, the storage server attempts to establish a fair share of throughput among multiple concurrent applications, which may lead to unexpected application performance degradation. Moreover, due to the lack of global view of all ongoing I/O activities from different applications, important system-wide performance metrics – such as average job turnaround time and system utilization – are often overlooked.

In this work, we intend to address the I/O congestion problem at the level of batch scheduling. To be more specific, this work aims to answer the following question: *if a batch scheduler is aware of ongoing I/O requests from multiple jobs, will it be able to mitigate the I/O congestion issue by coordinating different I/O requests?*

We argue that batch scheduler is a good candidate to handle I/O congestion due to the following reasons:

1) Batch scheduler is a global controller of all user jobs. It has a global and high-level knowledge of user jobs, and can initiate, suspend, terminate, or restart user jobs once they are submitted to the system.
2) Batch scheduler often contains a monitoring component to collect abundant information of system and user job status (e.g., node utilization, bandwidth usage, sensor data, etc.) from various sources [19][20][21][22].

In this paper we present a novel I/O-aware scheduling framework to coordinate concurrent I/O requests for petascale computing systems. The new batch scheduler not only selects queued user jobs for execution, but also coordinates job I/O activities during their execution. In case of I/O congestion, our batch scheduler suspends certain running jobs from conducting I/O. The decision is made based on a holistic view of all the running jobs and their respective I/O activities. More specifically, this paper makes the following contributions:

1) We present an I/O-aware batch scheduling framework for petascale computing that encompasses job abstraction, machine model, I/O congestion model, and new I/O-aware scheduling policies. In particular, two I/O-aware scheduling policies, namely *conservative* and *adaptive*, are designed for the framework, each for achieving different scheduling objectives (e.g., fairness, lower job slowdown, higher system utilization, etc.).
2) We conduct extensive trace-based simulations by using real job traces and I/O traces from the production 10-petaflop Mira system. Experimental results demonstrate that our design can effectively not only improve job performance by more than 30% and also improve system performance.

The remainder of this paper is organized as follows. Section II introduce background knowledge of the target platform. Section III describes our design of the I/O-aware batch scheduling. Section IV presents experimental results of the scheduling study. Section V discusses related work. Section VI concludes this paper and points out future work.

II. Background

A. Mira: The IBM Blue Gene/Q at Argonne

Mira is a 10 PFLOPS (peak) Blue Gene/Q system operated by Argonne National Laboratory for the U.S. Department of Energy [1]. It is a 48-rack system, arranged in three rows of sixteen racks. Each rack contains 1,024 sixteen-core nodes, for a total of 16,384 cores per rack. Mira has a hierarchical structure: nodes are grouped into midplanes, each midplane contains 512 nodes, and each rack has two such midplanes. Each node has 16 cores, giving a total of 786,432 cores. Mira was ranked 5th in the latest Top500 list [1]. Mira is a capability system, with single jobs frequently occupying substantial fractions of the system. The smallest production job on Mira occupies 512 nodes; 8192-node and 16384-node jobs are common on the system; larger jobs also occur frequently. Jobs up to the full size of Mira run without administrator assistance. Time on Mira is awarded primarily through the Innovative and Novel Computational Impact on theory and Experiment (INCITE) program [23] and the ASCR Leadership Computing Challenge (ALCC) program [24].

B. I/O Infrastructure of Mira

As shown in Figure 1, there are three major components in the IBM Blue Gene/Q system. The first component is the compute resources that comprise compute and I/O nodes. Compute nodes are only responsible for application execution and do not directly communicate to the outside. I/O nodes, on the other hand, are not involved in applications and only send/receive data to/from the second component—a large number of high-performance switches. The third and last component, which is connected by the second component of storage resources, comprises file servers where the application's data are persistently stored.

The I/O bandwidth between three components is also illustrated: both compute and storage resources could transfer more than 1 TB/S data via the high-performance InfiniBand switches. Yet, the aggregate disk I/O throughput within the file servers is only 250 GB/S, which is 4X slower than the outside I/O bandwidth. Therefore, it is the file server disk that throttles the overall performance of data-intensive applications.

C. Cobalt: Batch Scheduler on Mira

Our work is based on Cobalt [21][25], a production batch scheduling system used on Mira, as well as its previous generations of HPC systems at Argonne National Laboratory. It use a scheduling policy called "WFP" to order the jobs in the queue [25]. WFP favors large and old jobs, adjusting their priorities based on the ratio of their wait times to their requested runtimes. On Mira, Cobalt uses a partition-based resource allocation scheme which allocates computing resource to jobs in an exclusive manner [25]. Cobalt components correspond to pieces of functionality in resource management systems, such as scheduling, queue management, hardware resource management, and process management. Its component architecture allows easy replacement of key software functionality.

In this work, we present an I/O-aware framework for Cobalt extension (see Section III). While we target Mira in this work, the idea can be easily extended to other petascale systems and their batch schedulers.

III. Methodology

In this section, we present our methodology including problem formulation, I/O-aware scheduling framework, and two new I/O-aware scheduling policies.

A. Problem Formulation

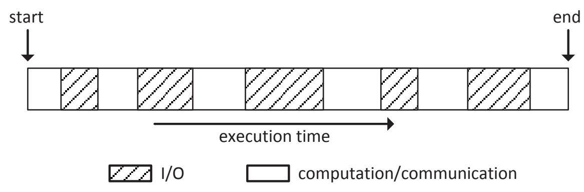
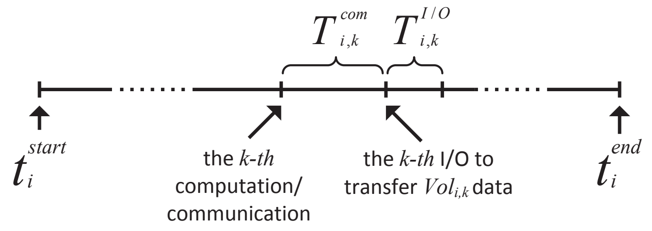


Figure 2. Lifecycle of a HPC scientific application

*1) **Job abstraction**:* A typical HPC job often contains three type of activities: computation, communication and I/O. Figure 2 shows the lifecycle of a sample job. The job's runtime is splitted into phases for different activities, which repeatedly run during the job lifecycle. The horizontal axis represents the execution time. The height usually represents the job's resource requirement (e.g., nodes, memory, bandwidth, etc.). This abstraction provides a high-level view of application behavior. For example, the I/O chunk in Figure 2 may contain several consecutive I/O calls (e.g., a loop of write accesses). In our job abstraction, this serial of I/O calls is regarded as a single I/O request. Notice that our target platform is the IBM BG/Q system which uses a partition-based resource allocation scheme [25]. The computation and network resource in a partition are dedicated to serve the job running on it. Hence, once a job is scheduled to run, the time consumed for computation and communication is fixed. As such, we unify computation and communication into one type of activity in Figure 2. The time for I/O activity is substantially variable due to potential I/O congestion.

More precisely, we depict the runtime phases of a job in Figure 3. Each running job J_i is associated with two basic attributes: *start time* as t_i^{start} and *size* (in nodes) as N_i. A job shows a periodical running pattern that computation/communication interleaves with I/O. We assume each computation/communication in followed by an I/O request that transfers a chunk of data from or to the storage system via the I/O network. So each job J_i consists of n_i computation/communication and n_i I/O activities. The k-th computation/communication activity of job J_i requires time



$T_{i,k}^{com}$
$T_{i,k}^{I/O}$
t_i^{start}
the *k-th* computation/ communication
the *k-th* I/O to transfer $Vol_{i,k}$ data
t_i^{end}

Figure 3. Runtime phases of the i-th job J_i requiring N_i nodes

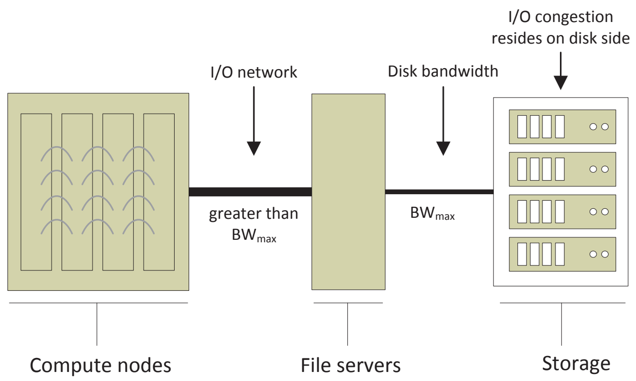


Figure 4. Simplified I/O infrastructure model of petascale HPC systems

$T_{i,k}^{com}$ to finish and the k-th I/O needs to transfer $Vol_{i,k}$ of data (in gigabytes).

*2) **System model**:* As shown in Figure 4, we assume a HPC system composed of N compute nodes, each installed with the same number of identical uniform-speed processors. Each compute node is connected to the I/O nodes and has the capability to transfer data to the storage system across the I/O network at maximum bandwidth b (in gigabytes per second). It is also guaranteed that the aggregate bandwidth of the whole system $b \times N$ is less than the maximum capacity of the I/O network. Inside the storage system, we assume a group of file serves connected to a centralized parallel file system with a maximum bandwidth BW_{max}. In many systems such as Intrepid and Mira, this maximum bandwidth BW_{max} is always less than the compute nodes' aggregate bandwidth $b \times N$ and is usually subject to the upper-bound maximum speed of hard drivers. More specifically, the rotation speed of disk-heads dominates the actual access speed of the storage system. Therefore, in this model we assume the I/O congestion takes place at the storage side if the aggregated bandwidth of all active compute nodes $b \times N_{active}$ exceeds the total bandwidth capacity BW_{max} of the storage system.

*3) **Scheduling Model**:* Figure 5 shows the scheduling model regarding I/O congestion based on the aformentioned job abstraction and the system model. The x-axis represents the execution time, the y-axis the aggregated bandwidth. Suppose there are 3 jobs J_1, J_2 and J_3 running concurrently on separated sets of nodes and each job has its own I/O

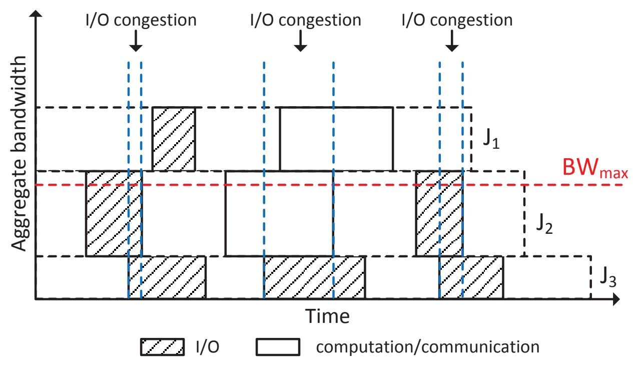


Figure 5. Scheduling model

pattern along with different bandwidth usage illustrated as the height of each rectangle. Naturally a job's I/O operations may overlap with those of others. At this point, multiple jobs are accessing the storage system simultaneously and inevitably they have to compete for I/O bandwidth with each other. If the aggregate bandwidth of these three jobs exceed the maximum bandwidth BW_{max} denoted as the red dashed line, I/O congestion occurs and consequently impact these jobs' I/O performance if no congestion control is taken.

This scheduling model is simple but it reflects the motivation of this work, that is, to utilize the global job scheduler to mitigate I/O congestion by monitoring and controlling jobs' I/O operations on the fly. After pulling information of all ongoing I/O operations, the job scheduler determines the bandwidth usage among jobs within a certain length of time-intervals. If we treat all I/O operations as sub-jobs inside a normal job, this I/O controlling issue can be transformed into a classical job scheduling problem which can be well solved using practical approaches.

*4) **I/O congestion model**:* In the system model of Figure 4, we assume the I/O network bandwidth is much larger than that of the storage system. Thus, the I/O congestion actually locates at the hard disk side, which is limited by the disk rotation speed. Lower-level scheduler in the storage server may apply a simply independent "first-in-first-out" policy on coming I/O requests or a more elaborated strategy, such as minimizing disk-head movements by aiming at better data locality. From the system view, the storage system is accessed in a fair sharing mode. We define a general I/O congestion model on HPC systems as: if $b \times N_{active} \geq B$, the actual bandwidth allocated to each compute node is $\frac{B}{N_{active}}$. Although extra overhead such as the time for disk-head movements should be considered, it is hard to precisely calculate the overhead in an analytical model. So in this work, we set up a fair sharing storage model with no extra overhead.

B. I/O-Aware Batch Scheduling

With the purpose to mitigate I/O congestion and improve system performance, we design a novel I/O-aware batch

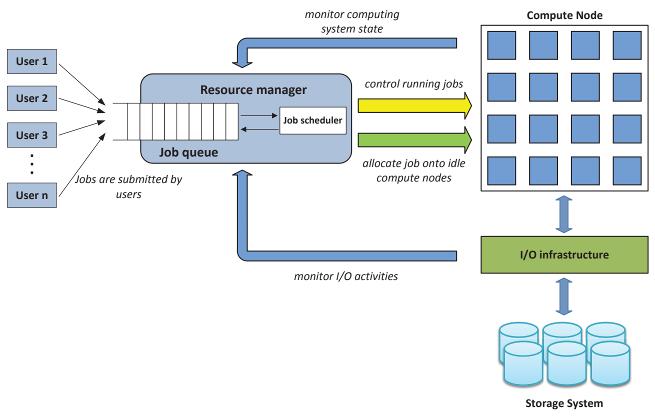


Figure 6. Our I/O-aware scheduling framework includes two new features: runtime I/O montoring (blue arrow) and dynamic control of running jobs (yellow arrow)

scheduling framework to coordinate concurrent I/O requests. As shown in Figure 6, the new scheduling framework extends the current HPC scheduling framework by integrating runtime monitoring of system state and I/O activities and runtime I/O control. The key idea of our solution works as follows:

1) Every time a new I/O request is forwarded to the I/O server, the batch scheduler monitors all I/O requests being processed and checks whether the aggregate bandwidth requirement exceeds the maximum value BW_{max}.
2) In case of I/O congestion, the I/O-aware batch scheduler dynamically acts on the runtime information to coordinate these concurrent I/O requests.
3) The runtime decision is made based on certain scheduling objectives which will be specified in the next subsection. The basic idea is to dynamically coordinate concurrent jobs' I/O acitivities (e.g., running or suspending) for avoiding I/O congestion.

C. I/O-Aware Scheduling Policies

Suppose there are K jobs in the system that are performing I/O or are ready to perform I/O, our scheduler selects a subset from the K jobs to perform I/O. Those jobs that are not selected will be suspended and wait until the next scheduling cycle. In this work, we propose two scheduling policies, each focusing on different objectives, either user-oriented or system-oriented. The selection of the subset of jobs is a process of searching for the optimal solution to maximize or minimize a certain objective.

1) Job performance quantification: First we present two user-oriented metrics to quantify job performance regarding I/O congestion.

- **InstantSlowdown (InstSld)**: Suppose at time t, job J_i is performing the k-th I/O operation. It has a total of $Vol_{i,k}$ data to transfer and has already transferred

$W_{i,k}$ data. Let $t_{i,k}^{I/O}$ be the start time of the k-th I/O operation. We define $InstSld_{(i,k)}(t)$, the instant slowdown of the k-th job J_i at time t as:

$$InstSld_{(i,k)}(t) = \frac{b \times N_i \times (t - t_{i,k}^{I/O})}{W_{i,k}} \quad (1)$$

where $t - t_{i,k}^{I/O}$ is the elapsed time from the start time of current I/O operation to now and $b \times N_i \times (t - t_{i,k}^{I/O})$ is the theoretically maximum total size of data job J_i could transfer by now within this I/O operation, which stands for an ideal case without any I/O congestion. The ratio of this maximum size of data to the already transferred $W_{i,k}$ data represents the slowdown of tranferring data caused by I/O congestion. It is obvious that $InstSld_{(i,k)}(t) \geq 1$ with $InstSld_{(i,k)}(t) = 1$ indicating the data transferring is not interfered by any I/O congestion.

- **AggregateSlowdown (AggrSld)**: Again, we assume job J_i is performing its k-th I/O operation. We define $AggrSld_{i,k}(t)$, the aggregate slowdown of job J_i at time t as:

$$AggrSld_{(i,k)}(t) = \frac{t - t_i^{start}}{\sum_{j \leq k} T_{i,j}^{com} + \sum_{j<k} T_{i,j}^{I/O}} \quad (2)$$

where $T_{i,j}^{com}$ is the time spent on the j-th computation/communication operation and $T_{i,j}^{I/O}$ the time of the j-th I/O operation without I/O congestion. $t - t_i^{start}$ equals the total elapsed time between the job start and the present time. $j \leq k$ is the number of computation/communication or I/O the job has executed at time t since the start time t_i^{start}. The $AggrSld$ reflects the extent of an application's delay due to I/O congestion.

2) Scheduling policies: Now we propose two types of scheduling policies as "*conservative*" and "*adaptive*":

Conservative: A conservative scheduling policy always obeys a basic principle – I/O congestion should be avoided as much as possible. Thus the scheduler only selects a subset of jobs whose aggregate bandwidth is no greater than the maximum storage bandwidth BW_{max}.

- *Cons-FCFS*: The *FCFS* (first-come-first-serve) policy works in the manner of what a traditional job scheduler does. It chooses jobs in chronological order based on the start time of concurrent I/O requests. Jobs performing I/O with earlier start time has higher priority and is favored by the scheduler.
- *Cons-MaxUtil*: The *MaxUtil* policy aims at maximizing the system utlization under the storage bandwidth constraint. This can be formalized to a classical optimization problem as: *to select a subset of jobs such that their aggregate bandwidth usage doesn't exceed the maximum bandwidth capacity BW_{max}, with the objective of maximizing the number of compute nodes allocated to these jobs*. In our previous work [26][27], we transform such problem into a standard 0-1 knapsack model which can be solved in pseudo-polynomial time by using dynamic programming method. The objective is system-oriented, as it seeks to achieve maximized utilization of compute resource.
- *Cons-MinInstSld*: The *MinInstSld* policy favors jobs with low $InstSld$ value. This method is close to *Cons-FCFS* which orders jobs before making scheduling decision.
- *Cons-MinAggrSld*: Similar to the *MinInstSld*, the *MinAggrSld* favor jobs with low $AggrSld$ value.

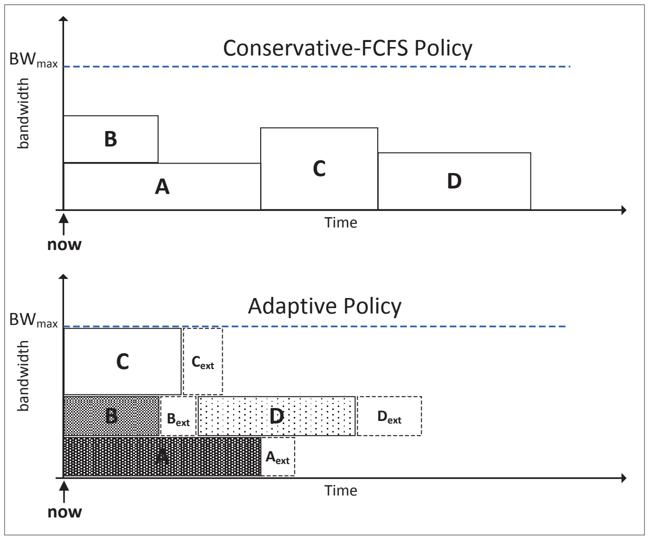


Figure 7. Using the adaptive scheduling policy can more effectively improve job performance, although I/O congestion may incur.

These four *conservative* polices are designed for different objectives. The *Cons-FCFS* focuses on preserving the user fairness and *Cons-MaxUtil* focuses on improving system utilization. Both *Cons-MinInstSld* and *Cons-MinAggrSld* aim to minimize the slowdown cause by I/O congestion.

Adaptive: While the above conservative scheduling makes effort to minimize I/O congestion, it may impact job performance. Hence we propose an adaptive scheduling policy which is not restricted to the I/O bandwidth. The adaptive scheduling policy allows more jobs to perform I/O operations, despite that this may break the I/O bandwidth bound.

Figure 7 illustrates the difference between adaptive policy and conservative policy. In this example, we assume that I/O requests *A* and *B* are already scheduled and requests *C* and *D* have just arrived. Both requests *C* and *D* require more bandwidth than what is currently available in the system. According to the *Cons-FCFS* policy, requests *C* and *D* will have to wait until enough bandwidth becomes available. This leads to wasted bandwidth and unnecessary performance degradation. Instead, the adaptive scheduling will calculate the overhead of immediately scheduling request *C* which shares the bandwidth with requests *A* and *B*. The overhead is expressed as the extension of time to finish the I/O request (i.e., square of dashed line labeled with "X_{ext}" in Figure 7). If the overhead is lower than suspending I/O request *C*, request *C* is allowed to start. Although this allows the job to performance I/O in contention with other jobs and may slow down their performance, it improves the job turnaround time and better utilize available bandwidth.

Algorithm 1 Adaptive Schedule

```
Input: $S$ as the set of jobs performing or ready to perform I/O
Output: $S_{opt}$ as the subset of jobs from set $S$ to be allowed to continue or start their I/O
 1: function AdaptiveSchedule($S$)
 2:     Prioritize jobs in $S$ based their I/O start time in FCFS fashion
 3:     $BW_{avail} \leftarrow BW_{max}$
 4:     $S_{opt} \leftarrow \varnothing$
 5:     for $J_i \in S$ do
 6:         $BW\_Req \leftarrow b \times N_i$
 7:         if $BW\_Req \leq BW_{avail}$ then
 8:             add $J_i$ to $S_{opt}$
 9:             $BW_{avail} \leftarrow BW_{avail} - BW\_Req$
10:         else
11:             Find the earliest time $T_i$ can start I/O if not schedule $J_i$ now
12:             Calculate $T_{FCFS}$ as the average time need to finish I/O for jobs in $S_{opt}$ plus $J_i$
13:             Calculate $T_{Adaptive}$ as the average time on I/O if $J_i$ is allowed to compete for bandwidth with jobs in $S_{opt}$
14:             if $T_{Adaptive} < T_{FCFS}$ then
15:                 add $J_i$ to $S_{opt}$
16:                 $BW_{avail} \leftarrow 0$
17:             end if
18:         end if
19:     end for
20:     return $S_{opt}$
21: end function
```

The pseudocode of the adaptive policy is shown in Algorithm 1. In case of I/O congestion, the procedure takes the set of jobs performing or ready to start I/O as input. It first sorts jobs based on their start time of current I/O requests. The earlier it starts, the higher priority it has (Line 2). Then the batch scheduler selects jobs in descending order of their priorities as long as the remaining available bandwidth can satisfy its requirement (Line 5-9). This step works exactly the same as the FCFS policy does. If the remaining available bandwidth is insufficient, the batch scheduler decides whether to still schedule this job. It calculates the average time needed to finish current I/O requests based on the choice of scheduling this job or not (Line 11-13). If this job is not scheduled, the earliest possible start time is calculated. Otherwise, this job will share the bandwidth with jobs already chosen into the subset S_{opt}. Obviously, other jobs' I/O will be impacted and thus need more time to finish. If the cost is acceptable (Line 14, this job is marked to for schedule (Line 15). This adaptive policy is a runtime optimization version resembling the Conservative backfilling.

IV. Evaluation

A. Qsim Simulator

Qsim is an event-driven scheduling simulator for Cobalt [21][25]. Taking the historical job trace as input, Qsim quickly replays the job scheduling and resource allocation behavior and generates a new sequence of scheduling events as an output log. Qsim uses the same scheduling and resource allocation code that is used by Cobalt and thus has been proved to provide accurate resource management and scheduling simulation [25][28][29]. Qsim is open source and available along with the Cobalt code releases [21].

B. Job Trace and I/O Trace

We evaluate our design by means of actual workload traces from the production Mira machine. We have collected a three-month job trace in 2014. The job trace provides rich information for our simulation including submission time, job size, duration and walltime.

To obtain information of I/O activities, we use the Darshan log collected from Mira [30]. Darshan is a user-level system tool to allow users to characterize the I/O behavior in an efficient and transparent manner at extreme scales. In essence, for each every application running on the systems, Darshan collects the I/O footprint and summarizes it in a compact and statistically reproducible manner. Because it only stores the statistical summary of many I/O operations, the space and I/O overhead when enabling Darshan logs is minimum. By default, Darshan is turned on for all applications on the Blue Gene systems at, and is completely transparent to the users; no application change is in need. The effectiveness of Darshan logs have been demonstrated at extreme scales—up to 64K nodes.

By combining the job trace and I/O trace via pairing, we have build a comprehensive workload which contains job information (e.g., submission, runtime, size, etc.) as well as their I/O characteristics (e.g., number of I/O calls, I/O time, read and write transfer size, etc.). We divide the 3-month workload trace into three single-month workload traces, and evaluate our design on each of them. This enables us to evaluate our design under workloads with different characteristics.

C. Evaluation Metrics

Three widely used metrics are taken for evaluation:

- *Average job wait time*. This metric denotes the average time elapsed between the moment a job is submitted and the moment it is allocated to run. It is commonly used to reflect the "efficiency" of a scheduling policy.
- *Average response time*. This metric denotes the average time elapsed between the moment a job is submitted and the moment it is completed. Similar to the above metric, it is often used to measure scheduling performance from the user's perspective.
- *System utilization*. System utilization rate is measured by the ratio of busy node-hours to the total node-hours during a given period of time [31] [32]. The utilization rate at the stabilized system status (excluding warm-up and cool-down phases of a workload) is an important metric of how well a system is utilized.

D. Results

In this work, we compare our design with a default scheduling policy as the baseline. This *BASE_LINE* policy allocates the I/O bandwidth among multiple applications in a fair sharing manner. In case of no I/O congestion, each application will have the maximal I/O bandwidth it needs; in case of I/O congestion, the *BASE_LINE* policy will evenly distribute the I/O bandwidth among the concurrent applications. This is similar to the current I/O scheduler using round-robin method in HPC systems [13][33].

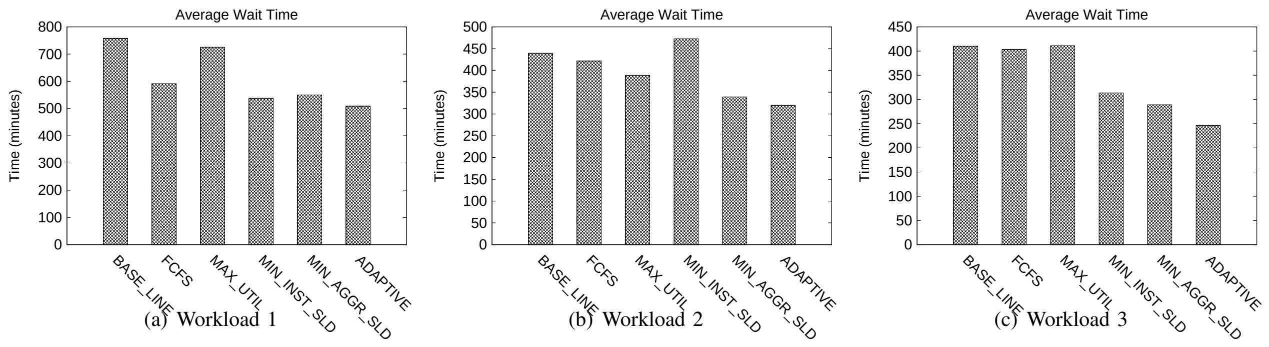
Figure 8 presents the average wait time of three workloads separately. We first observe that both our designated conservative and adaptive scheduling policies can reduce the wait time compared with the baseline result. The only exception occurs on workload 2 using *MIN_INST_SLD* which increases the wait time by less than 10%. This proves our motivation that a scheduling policy aware of I/O-congestion is much more efficient than the *BASE_LINE* policy which never controls I/O-congestion. Second, on all three workloads our *ADAPTIVE* policy have better results on reducing wait time than other scheduling polices. On both workload 1 and 2, the *ADAPTIVE* policy cuts back the wait time by at least 30%. These four conservative policies using strict I/O-congestion control lessens the flexibility of bandwidth utilization and thus are less efficient than the *ADAPTIVE* policy. Allowing a certain degree of I/O contention under the *ADAPTIVE* policy can sometimes provides shorter wait time as long as the overall job runtime extension is smaller than the cost to wait in the queue. Third, these three scheduling polices (*MIN_INST_SLD*, *MIN_AGGR_SLD* and *ADAPTIVE*) perform better than the other two policies (*FCFS* and *MAX_UTIL*). The latter two polices are either user fairness or system utilization oriented while the form three take account of the turnaround time when doing scheduling. Moreover, the *MIN_AGGR_SLD* outperforms the *MIN_INST_SLD* because the latter only focuses on the local optimum which is unable to guarantee a global optimal on wait time reduction.

Figure 9 presents the results on average response time. Similar to Figure 8 on wait time, first it is observed that both *MIN_AGGR_SLD* and *ADAPTIVE* have lower response times than the other four. The largest reduction on response time is achieved on workload 3 by more than 30% with the *ADAPTIVE*. And the *MIN_AGGR_SLD* can reduce the response time by more than 20%. On workload 2, we can see the *MIN_AGGR_SLD* works even better than the *ADAPTIVE*, both of which have improvement up to 15%. We also observe that using the *FCFS*, *MAX_UTIL* and *MIN_INST_SLD* can not always improve the response time or even bring negative impact on it. For example, on workload 2 we could notice that the *MAX_UTIL* increases the response time by nearly 10%. Especially, the *FCFS* and *MAX_UTIL* are ineffective on lowering the response time as they show nearly the same value as the baseline result. The same phenomena is noticed in Figure 8 as well. According to the definition, the job response time consists of wait time and job runtime. While the decrement of wait time benefit from our I/O-aware scheduling policies, a job may not be assigned with sufficient bandwidth. This causes expansion to its running time, which is likely to diminish the benefit on wait time.

Figure 10 shows the system utilization on three workloads. For convenience, we normalize the absolute utilization value based on the results of *BASE_LINE*. The *MAX_UTIL* is system utilization oriented so we can clearly see it improves the utilization on workload 1 and 3 up to 10%. This indicates that the *MAX_UTIL* is more efficient in optimizing the system performance than other scheduling polices. The largest utilization drop is seen using the *FCFS* and *MIN_INST_SLD* on workoad 2 where it decreases by nearly 10%. Other scheduling policies have negligible loss on utilization which is treated as acceptable cost. Taking all the results into consideration with Figure 8 and 9, the *ADAPTIVE* policy is the best trade-off and achieves good balance between user satisfaction and system utilization.

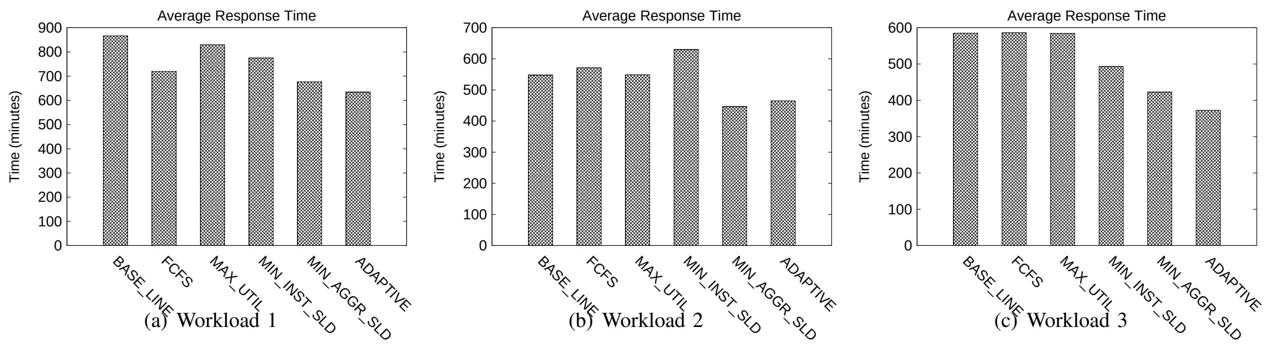
E. Sensitivity Study

The workloads used for the above simulations are from the real job trace and Darshan log collected from Mira. In order to assess the scheduling performance under different I/O intensiveness, we also conduct a sensitivity study by tuning job I/O time in the workload. For each simulation, we define an expansion factor (*EF*) as the change of I/O time. For example, *EF*=30% means each job's I/O time is compressed to 30%. Similarly, *EF*=150% increases each job's I/O time by 1.5 times, which implies it needs to transfer 50% more data. In this way, we have built various workloads that can have "heavy" or "light" I/O activities.



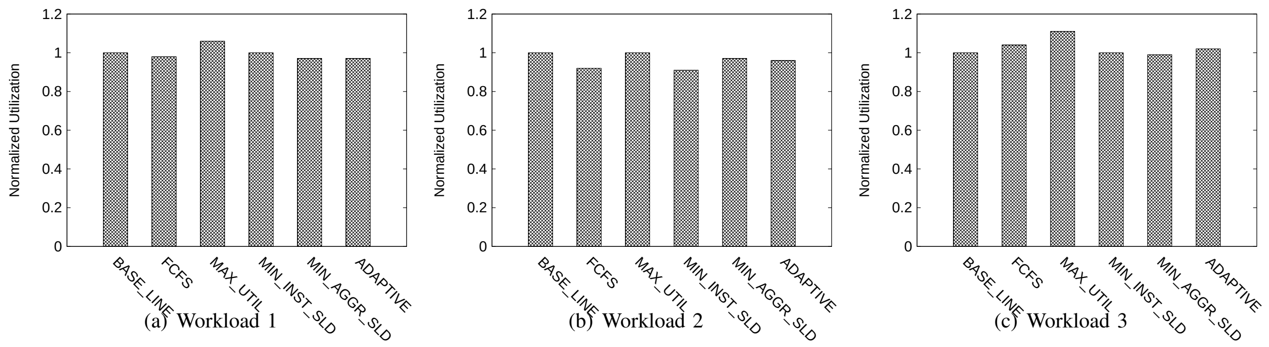
(a) Workload 1 (b) Workload 2 (c) Workload 3

Figure 8. Average wait time



(a) Workload 1 (b) Workload 2 (c) Workload 3

Figure 9. Average response time



(a) Workload 1 (b) Workload 2 (c) Workload 3

Figure 10. Normalized system utilization

Due to the space limitation, we only presents the results on average wait time in Figure 11. We set up six expansion factors to compress or enlarge the I/O time. It's apparent that the wait time goes up as the job spends more time on I/O. With low expansion factors (30% and 50%), there is no obvious improvement on wait time while FCFS even increases it. As the expansion factor becomes larger, we observe the *ADAPTIVE* and *MIN_AGGR_SLD* greatly surpass other policies in reducing the wait time. When the I/O time is expanded by 150%, the largest reduction on wait time is near 50%. Overall, the *ADAPTIVE* and *MIN_AGGR_SLD* policies should be favored to handle I/O-intensive workload due to their substantial performance improvement.

V. Related Work

One traditional approach to address the I/O bottleneck on extreme-scale systems is to employ high-level I/O libraries, such as HDF5 [14] and NetCDF [15]. These libraries stipulate the data format of the applications as well as their I/O interfaces. In other words, the applications need not assume the POSIX interface, but manipulate the data according to the customized programming API. One limitation of this approach is portability: Once the application assumes a particular API, it takes considerable amount of effort to migrate it to other platforms.

To improve the portability of the customized API discussed above, researchers proposed several loosely-coupled middleware solutions, such as PLFS [10], DataStager [11],

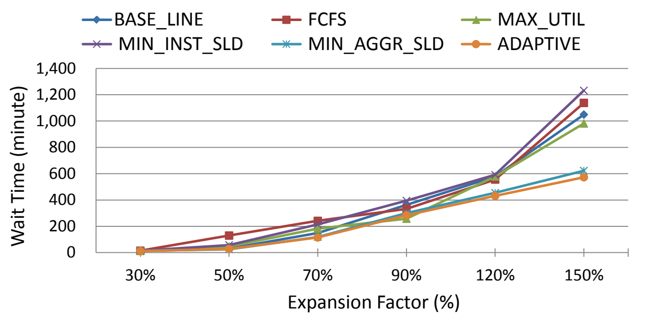


Figure 11. Impact of the I/O intensiveness over average wait time of all policies

Damaris [12]. These systems work independently of both the underlying storage systems and the upper-level applications, thus greatly generalize the applicability. Nevertheless, an obvious drawback of this approach is the potential overhead introduced by the middleware.

More recently, the trend of solving the I/O imbalance in extreme-scale systems is to move the data closer to the compute resource. Ning et al. [18] proposed to move many file handling to the I/O nodes to ameliorate the I/O pressure from the massive number of compute nodes.

I/O contention in HPC systems draws a lot of attension in the community because it is the root cause of parallel applications' performance variability [34][35][36]. Zhang et al. schedule each application's I/O request individually without a global view from system's perspective [36] [35]. Their solutions require supports from specific I/O management in the system level for better results. The solutions for the I/O contention between parallel applications have been studied in recent works [37][38][39]. Hashimoto et al. evaluate the performance variability of each job when they run concurrently on the same physical computing server[37]. They identify that network I/O sharing introduces most of the performance degradation. Xie et al. analyze the behavior and performance variability of Lustre, a parallel file system on supercomputer Jaguar [6]. They found that the shared filesystem between concurrently running parallel applications cause most of system performance degradation. Lebre et al. propose a new scheduling design for multi applications with the objective of better aggregating and reordering I/O requests without hurting the fairness across applications [40]. Dorier et al. analyze the I/O interference between two applications [41]. They make quantified study about performance improvement obtained by interrupting or delaying either one in order to avoid I/O contention. Gainaru et al. analyze the effects of interference on application I/O bandwidth and propose several scheduling techniques that apply to system I/O level to mitigate congestion [33].

VI. Conclusion

In this paper, we have presented an I/O-aware batch scheduling framework to alleviate the I/O congestion on petascale HPC systems. In our design, the I/O congestion scenario is formalized into a classical batch scheduling problem by treating I/O accesses as schedulable sub-jobs and the batch scheduler dynamically schedules these I/O accesses. Along with this, we have designed two types of I/O-aware scheduling policies, namely *conservative* and *adaptive*, each focusing on either user-oriented objectives or system performance. Our trace-based simulations clearly demonstrate the performance benefit obtained by these new scheduling polices. In particular the *ADAPTIVE* and *MIN_AGGR_SLD* policies have substantial advantage over other polices regarding user-oriented metrics. The *ADAPTIVE* has better scheduling performance than *MIN_AGGR_SLD*, whereas *MIN_AGGR_SLD* has lower time complexity. Furthermore, the *MAX_UTIL* policy should be favored if the optimization for system performance (i.e., system utilization) is given to priority. While this study targets Mira, our design is generally applicable to other HPC systems.

To the best of our knowledge, this is the first work on addressing I/O congestion in the batch scheduling. Several avenues remain open for our future work. One is to build a model to predict an application's I/O behavior based on its past I/O trace. In addition, we plan to expand this work with the aim of developing a smart resource management framework for better managing non-traditional resources including I/O and power consumption.

Acknowledgments

The work at Illinois Institute of Technology is supported in part by US National Science Foundation grants CNS-1320125 and CCF-1422009.

References

[1] "Top500 supercomputing web site." [Online]. Available: http://www.top500.org

[2] Mira at ANL, "http://www.alcf.anl.gov/mira," Accessed September 5, 2014.

[3] Titan at ORNL, "https://www.olcf.ornl.gov/titan/," Accessed September 5, 2014.

[4] Exascale computing predicted by 2018, "http://www.computerworld.com/article/2550451/computer-hardware/scientists–it-community-await-exascale-computers.html," Accessed September 5, 2014.

[5] P. A. Freeman, D. L. Crawford, S. Kim, and J. L. Munoz, "Cyberinfrastructure for science and engineering: Promises and challenges," *Proceedings of the IEEE*, vol. 93, no. 3, pp. 682–691, 2005.

[6] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing output bottlenecks in a supercomputer," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 8:1–8:11.

[7] R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski *et al.*, "The IBM Blue Gene/Q compute chip," *Micro, IEEE*, vol. 32, no. 2, pp. 48–60, 2012.

[8] "Overview of the IBM Blue Gene/P project," *IBM J. Res. Dev.*, vol. 52, no. 1/2, pp. 199–220, Jan. 2008.

[9] Parallel I/O on Mira, "http://www.alcf.anl.gov/files/parallell_io.pdf," Accessed September 5, 2014.

[10] 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*, 2009.

[11] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "Datastager: Scalable data staging services for petascale applications," in *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, 2009.

[12] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, Sept 2012, pp. 155–163.

[13] X. Zhang, K. Davis, and S. Jiang, "Iorchestrator: Improving the performance of multi-node I/O systems via inter-server coordination," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, Nov 2010, pp. 1–11.

[14] HDF5, "http://www.hdfgroup.org/hdf5/doc/index.html," Accessed September 5, 2014.

[15] NetCDF, "http://www.unidata.ucar.edu/software/netcdf," Accessed September 5, 2014.

[16] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the best use of solid state drives in high performance storage systems," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 22–32.

[17] X. Zhang, K. Davis, and S. Jiang, "iTransformer: Using SSD to improve disk scheduling for high-performance I/O," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 715–726.

[18] 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 *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, April 2012, pp. 1–11.

[19] M. A. Jette, A. B. Yoo, and M. Grondona, "Slurm: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.

[20] G. Staples, "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 8.

[21] Cobalt Resource Manager, "https://trac.mcs.anl.gov/projects/cobalt," Accessed September 5, 2014.

[22] Portable Batch System, "http://www.pbsworks.com/," Accessed September 5, 2014.

[23] "Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program." [Online]. Available: https://www.alcf.anl.gov/incite-program

[24] "ASCR Lleadership Computing Challenge (ALCC)." [Online]. Available: http://science.energy.gov/ascr/facilities/alcc/

[25] W. Tang, Z. Lan, N. Desai, and D. Buettner, "Fault-aware, utility-based job scheduling on Blue Gene/P systems," in *IEEE International Conference on Cluster Computing and Workshops, 2009, CLUSTER '09.*, 2009, pp. 1–10.

[26] Z. Zhou, Z. Lan, W. Tang, and N. Desai, "Reducing energy costs for IBM Blue Gene/P via power-aware job scheduling," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, 2014, pp. 96–115.

[27] X. Yang, Z. Zhou, S. Wallace, Z. Lan, W. Tang, S. Coghlan, and M. E. Papka, "Integrating dynamic pricing of electricity into energy aware scheduling for HPC systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 60:1–60:11.

[28] W. Tang, N. Desai, D. Buettner, and Z. Lan, "Analyzing and adjusting user runtime estimates to improve job scheduling on the Blue Gene/P," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, pp. 1–11.

[29] W. Tang, Z. Lan, N. Desai, D. Buettner, and Y. Yu, "Reducing fragmentation on torus-connected supercomputers," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 828–839.

[30] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, pp. 8:1–8:26, Oct. 2011.

[31] J. Jones and B. Nitzberg, "Scheduling for parallel supercomputing: A historical perspective of achievable utilization," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, 1999, vol. 1659, pp. 1–16.

[32] Y. Xu, Z. Zhou, W. Tang, X. Zheng, J. Wang, and Z. Lan, "Balancing job performance with system performance via locality-aware scheduling on torus-connected systems," in *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, Sept 2014, pp. 140–148.

[33] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC applications under congestion," LIP, Research Report RR-8519, Oct 2014.

[34] J. Lofstead and R. Ross, "Insights for exascale IO apis from building a petascale IO api," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 87:1–87:12.

[35] 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 *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, Nov 2010, pp. 1–12.

[36] X. Zhang, K. Davis, and S. Jiang, "Opportunistic data-driven execution of parallel programs for efficient I/O services," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, May 2012, pp. 330–341.

[37] Y. Hashimoto and K. Aida, "Evaluation of performance degradation in HPC applications with VM consolidation," in *Proceedings of the 2012 Third International Conference on Networking and Computing*, ser. ICNC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 273–277.

[38] D. Skinner and W. Kramer, "Understanding the causes of performance variability in HPC workloads," in *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, Oct 2005, pp. 137–149.

[39] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker, "Parallel I/O performance: From events to ensembles," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–11.

[40] A. Lebre, G. Huard, Y. Denneulin, and P. Sowa, "I/o scheduling service for multi-application clusters," in *Cluster Computing, 2006 IEEE International Conference on*, Sept 2006, pp. 1–10.

[41] 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.