Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters

Stephen Herbein
University of Delaware
Newark, DE, United States
sherbein@udel.edu

Dong H. Ahn
Lawrence Livermore National Laboratory
Livermore (CA), United States
ahn1@llnl.gov

Don Lipari
Lawrence Livermore National Laboratory
Livermore (CA), United States
lipari1@llnl.gov

Thomas R.W. Scogland
Lawrence Livermore National Laboratory
Livermore (CA), United States
scogland1@llnl.gov

Marc Stearman
Lawrence Livermore National Laboratory
Livermore (CA), United States
stearman2@llnl.gov

Mark Grondona
Lawrence Livermore National Laboratory
Livermore (CA), United States
grondona1@llnl.gov

Jim Garlick
Lawrence Livermore National Laboratory
Livermore (CA), United States
garlick1@llnl.gov

Becky Springmeyer
Lawrence Livermore National Laboratory
Livermore (CA), United States
springmeyer1@llnl.gov

Michela Taufer
University of Delaware
Newark, DE, Un/ited States
taufer@udel.edu

ABSTRACT

The economics of flash vs. disk storage is driving HPC centers to incorporate faster solid-state burst buffers into the storage hierarchy in exchange for smaller parallel file system (PFS) bandwidth. In systems with an underprovisioned PFS, avoiding I/O contention at the PFS level will become crucial to achieving high computational efficiency. In this paper, we propose novel batch job scheduling techniques that reduce such contention by integrating I/O awareness into scheduling policies such as EASY backfilling. We model the available bandwidth of links between each level of the storage hierarchy (i.e., burst buffers, I/O network, and PFS), and our I/O-aware schedulers use this model to avoid contention at any level in the hierarchy. We integrate our approach into Flux, a next-generation resource and job management framework, and evaluate the effectiveness and computational costs of our I/O-aware scheduling. Our results show that by reducing I/O contention for underprovisioned PFSes, our solution reduces job performance variability by up to 33% and decreases I/O-related utilization losses by up to 21%, which ultimately increases the amount of science performed by scientific workloads.

Keywords

High performance computing; Resource management; Scheduling algorithms; Nonvolatile memory

HPDC'16, May 31-June 04, 2016, Kyoto, Japan

© 2016 ACM. ISBN 978-1-4503-4314-5/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2907294.2907316

1. INTRODUCTION

High performance storage is critical to achieving computational efficiency on high performance computing (HPC) systems. For over a decade, HPC has met this need by separating compute systems from parallel file systems (PFS) [25, 14, 32] that are built from an array of disks for both capacity and bandwidth. As the capacity growth of disks continues to outpace increases in their bandwidth, disks will meet capacity demands but fail to deliver bandwidth cost-effectively [12, 5]. Large HPC centers have begun to face this challenge by using solid-state burst buffers [23, 31], a new storage media that is cost-effective for bandwidth while not yet viable for capacity, between the compute nodes and the PFS. As HPC applications alternate between computationally dominant and I/O-dominant execution phases, the burst buffers can absorb their bursty I/O requests and turn them into a constant I/O stream, as seen by the PFS. This approach can not only offer better performance for the applications but also reduce the requisite PFS bandwidth. The PFS no longer needs to be provisioned for the worse-case scenario where multiple jobs happen to enter their I/O-dominant phases simultaneously. This trend is already apparent in recent leadership-class system procurements [1]. Next-generation systems will deliver 7 to 10× higher peak floating-point performance with only 1 to 2× higher PFS bandwidth compared to previous generation systems [18, 28]. The underprovisioning level is expected to increase, as system software technologies (e.g., smart checkpoint staging) will more effectively exploit the burst buffers. Similarly, a desire for global mounts (i.e., mounting all of the PFSes on all of the clusters) [33, 19] will increasingly split the already underprovisioned bandwidth across multiple clusters.

This paper targets next-generation HPC centers characterized by large-scale systems with burst buffers, reduced I/O network bandwidth, and underprovisioned PFS bandwidth. These characteristics require scheduling techniques that use I/O awareness to make informed decisions at the batch scheduling level and reduce PFS contention caused by over-allocated I/O resources. Unfortunately, the dynamicity and uncertainty of batch jobs (i.e., which jobs happen to be scheduled and burst into I/O phases) as well as the sheer scale of the largest centers have long made the definition of effective, yet scalable, I/O-aware scheduling a hard problem to solve. As a result, I/O awareness within batch schedulers in use for today's large centers remains largely rudimentary (e.g., Moab [2] can only hold dependent jobs when a file system is marked as being down). Our work tackles the I/O-aware scheduling problem within batch schedulers and complements existing work that manages the I/O contention problem by coordinating and optimizing the I/O performed at runtime [9, 34, 10, 36]. While these runtime techniques have shown promising results at maximizing the use of the PFS and minimizing I/O contention, they are ultimately constrained by the I/O requests of the specific running applications. If many I/O-intensive jobs are running concurrently, these runtime techniques are unable to prevent a slowdown of the applications. On the other hand, because we tackle I/O contention at batch job scheduling time, our work prevents applications' slowdown by ensuring that many I/O-intensive jobs are not run concurrently.

Our I/O-aware scheduling takes advantage of two emerging technologies for next-generation systems: (1) burst buffers and (2) center-wide resource management and job scheduling frameworks. The burst-buffer layer absorbs random I/O bursts and thus makes the bandwidth requirement of a job constant over its lifetime. A center-wide resource and job management framework enables schedulers to model the global I/O subsystem and view jobs beyond cluster boundaries, enabling global scheduling decisions for shared file systems. We integrate the knowledge of the I/O subsystem including burst buffers and I/O-aware algorithms directly into Flux [4], one of the few resource and job managers for next-generation centers. Exploiting a global view in Flux, however, comes with multidimensional scale challenges. Our hierarchical representation of the system must cope with unprecedented numbers of jobs and individual resources. By using a Flux emulator, this paper explores the effectiveness and costs of this large-scale constrained scheduling. Our exploration includes the degree of resource utilization under the I/O-aware scheduling as well as the computational costs of the scheduling itself, as it must consider both compute nodes and available bandwidth at every level of the I/O hierarchy. Specifically, the paper makes the following contributions:

- Modeling of the hierarchical nature of real-world I/O subsystems using a resource description language;
- Scheduling algorithms that integrate I/O awareness as a driving scheduling factor into one of the most popular scheduling policies (i.e., EASY backfilling);
- Exploration of the cost-effectiveness space of I/O-aware scheduling relative to the I/O-ignorant baselines.

Our results show that by reducing I/O contention for underprovisioned PFSes, our solution reduces job performance variability by 33% and decreases I/O-related utilization losses by 21% on a system underprovisioned by 30%. In addition, our solution offers these benefits over I/O-ignorant scheduling policies with negligible scheduling cost, making deployment of our solution practical for next-generation environments.

The rest of this paper is organized as follows: Section 2 explains the need for I/O awareness; Section 3 presents our I/O subsystem model and I/O-aware scheduling algorithms; Section 4 describes the Flux emulator along with the system and workload models used in our tests; Section 5 presents the results of our tests and demonstrates the benefits of I/O-aware scheduling; Section 6 discusses related work; and Section 7 concludes this paper.

2. NEED FOR I/O AWARENESS

Due to the economics of flash vs. disk storage media, the HPC community has long predicted that flash-based burst buffers (BBs) must be included in the next-generation storage hierarchy, and recent high-end system procurements have attested to this prediction [1, 3, 23, 31, 24]. Initially, burst buffer enabled systems [3, 1] will still provision a reasonably high bandwidth PFS to ensure expedient draining of jobs' last checkpoints from the burst buffers to the PFS. This will allow a quick turnaround time even though, for simplicity, the system may require draining a job's burst buffers before scheduling the following job. Over time, smart staging, in conjunction with advancements in relevant system software, will remove this requirement. The system will stage the new job's data into the burst buffers while the previous job's last checkpoint is still being drained to the PFS, hiding the latency [21]. With the support of BB and smart staging, the PFS and the I/O links no longer need to be provisioned for worst-case scenarios (i.e., the I/O-dominant phases of many independent jobs overlap). Instead, the I/O links and the PFS will be provisioned to handle the average case (i.e., the average egress bandwidth of the burst buffers). Under this scenario, when multiple data-intensive applications run in concert on a cluster that has burst buffers and uses I/O-ignorant scheduling, the applications' cumulative average bandwidths can exceed the PFS bandwidth, causing I/O contention. In other words, the applications spend time blocking on I/O rather than performing computation.

With a significantly underprovisioned PFS, it will be crucial to avoid scheduling any combination of jobs that can create I/O contention at the lower storage levels. Fortunately, next-generation technologies present a practical opportunity to address this challenge at the batch job scheduling layer through I/O-aware scheduling.

3. I/O-AWARE JOB SCHEDULING

This section describes our approach. We begin with the enablers of our approach and progressively add core techniques until we can fully describe our methodology.

3.1 Constant Job-Lifetime I/O Bandwidth

Researchers have long observed that the common I/O patterns of HPC applications are bursty: they alternate between computationally dominant and I/O-dominant execution phases. Thus, the actual bandwidth used by a job during its lifetime is not a quantity that can easily be scheduled. Even if a batch system schedules jobs in accordance with their average bandwidth, the I/O dominant execution phases of these jobs can still often overlap, and utilization can oscillate widely, often exceeding the threshold. Burst buffers (BBs) naturally address this challenge: they turn the bursty I/O requests into a *constant* I/O stream. The BB layer absorbs applications' I/O bursts and the applications' I/O requests are then drained in the background into the parallel file system (PFS) from the BBs. As also proposed in [27], we assume that the BBs are placed either at the compute nodes (CNs) or on the same high-speed interconnect as the CNs, thus minimizing the probability of I/O contention occuring on resources in-between the CNs and the BBs. This architectural assumption allows us to focus our study on the I/O contention in-between the BBs and the PFS. In the rest of this paper, when we refer to I/O contention between the nodes and the PFS, we mean the CNs+BBs and the PFS.

The BB write-behind scheme is the first enabler for the design and implementation of practical I/O-aware scheduling. Batch job schedulers can use the draining rate of the BB as the overall bandwidth requirement of a job, a constant that machine learning algorithms trained on historical job records can provide. Work of McKenna and coworkers indicates that by using decision trees, we can predict, within 16MB, the I/O produced over the lifetime of a job, 80% of the time [26].

3.2 Global View

With a strong desire to mount a PFS to multiple clusters [33, 19], effective I/O-aware schemes also require a global view over all compute resources from which jobs can access the PFS. Without global knowledge, many jobs from multiple scheduler domains can request the shared resources concurrently, which precludes the scheduler from being able to manage bandwidth allocation reliably. Thus, practical schemes need advanced resource and job management software (RJMS) that can provide the scheduler with visibility into both jobs and resources across system boundaries. Fortunately, demand has grown to have an RJMS to schedule jobs at levels above these boundaries using hierarchical approaches. Flux [4] is among a few of the already available next-generation resource and job management software systems. Being developed at Lawrence Livermore National Laboratory (LLNL), Flux responds to the aforementioned need and presents an opportunity for our methodology to be developed.

3.3 I/O Subsystem Modeling

Using these enablers (i.e., burst buffers and Flux), we first model the full I/O subsystem including the BBs and I/O switches into the RJMS so that our scheduler can reason about key bandwidth constraints. We consider an I/O architecture based on an approach used by the Tri-Laboratory Linux Capacity Cluster (TLCC) [1]. A TLCC cluster is built on the notion of a scalable unit (SU). A large cluster can be built by scaling out and replicating SUs, each of which consists of a certain number of compute nodes, a few gateway nodes that can route I/O traffic to global parallel file system, and one or two login nodes. A multilevel system of switches connects nodes within a single SU and across SUs. For example, at LLNL, an SU is composed of 154 compute nodes, 6 gateway nodes, and 2 login nodes; one or multiple high level Infiniband switches connect multiple lower level Infiniband switches. Figure 1 shows an example of a TLCC cluster containing 12 SUs with its low and high level switches.

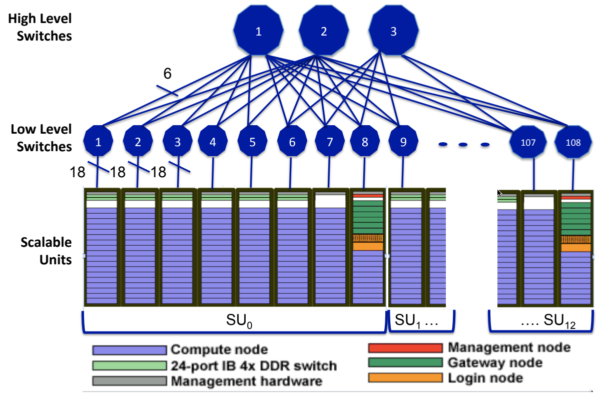


Figure 1: Example of 12 SU cluster with high and low level switches.

Any effective job scheduling is based on a rigorous but efficient model of the cluster resources and must include the resource relationships pertaining to the scheduler's capabilities. In the case of I/O-aware scheduling, a resource model must also capture the essential attributes of the I/O subsystem of clusters in addition to their compute resources (e.g., compute nodes and cores). We use the Resource Description Language (RDL) within Flux [4] to model the I/O subsystems as a hierarchy of network switches and gateway nodes leading to a parallel file system (PFS) and establish hierarchical relationships between them. Figure 2 shows a simple example of an RDL-based model with its compute nodes, each with a BB at the fringes of the hierarchy. Nodes are connected to their leaf switches before accessing the PFS through the gateway nodes (e.g., in the case of Lustre file system, LNET routers run across the gateway nodes). Since I/O traffic is routed round-robin across the high-level switches and the gateway nodes, we aggregate these resources into pools, where the BW of the pool is the sum of the individual resources' BWs. This model still matches well with the high-level architecture of a TLCC SU as shown in Figure 1, maintains a reasonable computational cost, and can easily be extended when multiple large clusters share a PFS.

When modeling how a job's I/O affects the I/O hierarchy and vice versa, we consider each level in the I/O hierarchy (i.e., PFS, switches, and compute nodes). When placing a job on the I/O hierarchy, BW is allocated at every level in the hierarchy. When modeling the impact of the I/O hierarchy on a job's I/O BW, we consider both the state of the I/O hierarchy and the job's position within the hierarchy using a contention model described in detail in Section 4.4. Figure 3 shows an example of I/O contention for the hierarchical resources considered in Figure 2 when two jobs with different BW requirements are executed. In Figure 3, Job1 runs on three nodes. Job1's required I/O rate is 192 MB/s per process, while the upper limit of the low level switches is only 256 MB/s (or 128 MB/s per child). If we assume that Job1's I/O pattern is highly synchronous, which is a dominant I/O pattern at HPC centers [23], Job1's overall I/O rate is limited to 128 MB/s, despite one of its processes

[1] Tri-Laboratory refers to the NNSA's (National Nuclear Security Agency) three national laboratories: Livermore, Sandia, and Los Alamos National Laboratory.

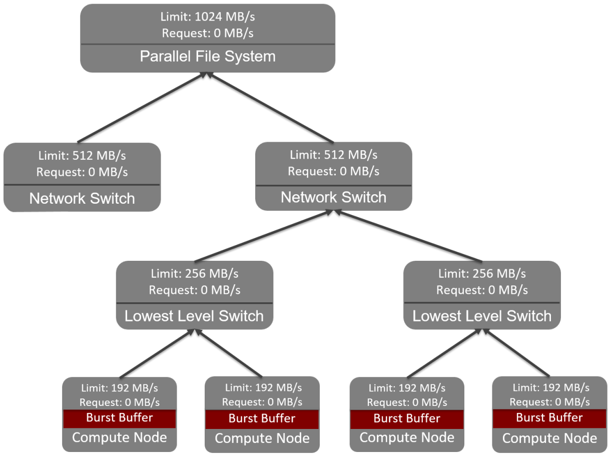


Figure 2: Example of an I/O-resource hierarchy in RDL.

having access to the full 192 MB/s. Job1 is over-utilizing bandwidth not only at the low level switches but at the higher level switches. If a new job, Job2 arrives and tries to use additional bandwidth, on the higher level switch, the job can further steal BW from Job1. Our tests inject job arrivals and their bandwidth requests into the models that represent real clusters at LLNL in a similar but larger scale than in the example presented above.

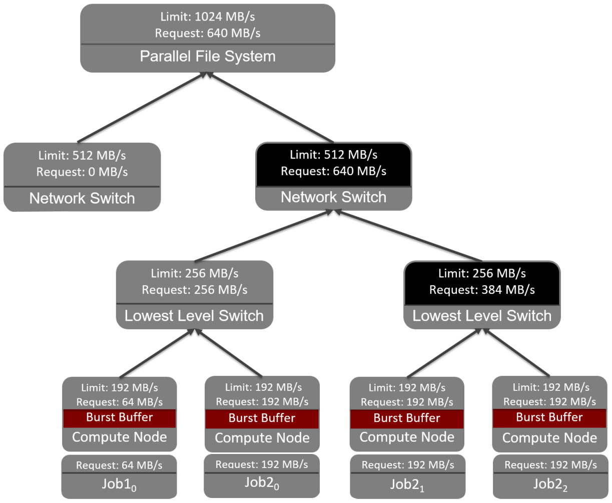


Figure 3: Example of I/O contention in action for our I/O-resource hierarchy.

3.4 Making a Scheduler I/O-Aware

We use our I/O subsystem and the integrated I/O contention model to extend popular batch job scheduling algorithms. We consider two existing scheduling policies: First-Come First-Served (FCFS) and EASY backfilling [22]. We select these policies because they represent algorithms with increasing complexity and expected efficiency in real resource managers [7]. The base scheduling schemes of the FCFS and EASY backfilling algorithms are currently without I/O awareness.

As part of the FCFS algorithm, jobs are scheduled in the order they are submitted. Thus, the algorithm may suffer from head-of-line blocking, where a large job can delay every other job in the queue from running. The EASY backfill algorithm is similar to FCFS except that if a job at the front of the queue cannot be scheduled, lower priority jobs are scheduled on resources reserved for the highest priority job - as long as doing so does not delay the projected start time of the highest priority job. We assume that jobs are ordered in the queue based on their submission time. Under this assumption, both FCFS and EASY backfilling will not delay high bandwidth jobs infinitely (i.e., starve). The problem of job starvation associated to different job sorting (e.g., priority-based) is addressed elsewhere [17].

Defining a scheduler as I/O-aware means that the I/O is used as an additional constraint when determining if a job can be scheduled. The I/O subsystem described in Section 3.3 allows us to add I/O awareness to the two schedulers. This additional I/O constraint means that if scheduling a job over-allocates any of the I/O resources available, the job should not be scheduled even though compute resources are idle. Our I/O-aware schedulers keep track of over-allocations by updating the state of the I/O model based on each job's I/O BW requirements and make scheduling decisions accordingly. Scenarios like the one in Figure 3 should never occur on a system that relies on our I/O aware scheduler.

To add I/O awareness to either of the two scheduling algorithms (i.e., FCFS and EASY backfilling), we extend the scheduler to consider both available compute nodes and the I/O BW available to the nodes themselves. Consequently, our I/O-aware scheduler assigns a job to a node only when the available BW at each level in the I/O switch hierarchy, from the PFS to the node itself, meets the BW requirements. For example, let's assume we have an empty system similar to the one modeled in Figure 2 and two jobs to be scheduled (i.e., Job 1 and Job 2 with Job 1 at the head of the queue). Job 1 only requires a single node and 64 MB/s of BW, and Job 2 requires three nodes, with each node requiring 128 MB/s of BW. The scheduler starts with Job 1 and attempts to schedule the job on Node 1. The scheduler also checks the feasibility of the job by ensuring that the node is free and that 64 MB/s of BW are available at Node 1 and Node 1's ancestors (i.e., Lowest Level Switch 1, Network Switch 2, and the PFS). Since there is enough bandwidth, Job 1 is scheduled on Node 1 and the BW is allocated at Node 1 and at all of the ancestors. The scheduler then attempts to schedule Job 2. The same process applied to Node 1 is repeated for Nodes 2 and 3, but this time, the scheduler must ensure that 128 MB/s are available. If both feasibility checks succeed, Nodes 2 and 3 are reserved for Job 2. Finally, the scheduler attempts to reserve Node 4 for Job 2. Node 4 is available, but there is not enough available BW at Lowest Level Switch 2 or Network Switch 2. Thus, the scheduler does not reserve Node 4 for Job 2. Because the scheduler now has no nodes left to consider in order to satisfy Job 2's requirements, it deems Job 2 unschedulable with the current state of the entire system and releases the reservations on Nodes 2 and 3. Figure 3 demonstrates the overallocation thats occur if Job 2 is scheduled. This does not mean that Job 2 cannot run; once Job 1 finishes, the scheduler is be able to schedule Job 2. If a job's requirement are not satisfiable even on an empty system, the job is deemed unsatisfiable and an error is returned to the user when they submit the job.

4. EVALUATION METHODOLOGY

This section describes our evaluation methodology including the modeling of an expected large next-generation system and of job workloads, as well as our metrics.

4.1 A Large Next-Gen. System Model

We test our approach against the model of a large, realistic system. We construct such a model by analyzing the request for proposal (RFP) for next-generation systems. Specifically, we build the node and I/O components of our modeled system after a large system that will be built as part of the Commodity Technology System 1 procurement (CTS-1) [20]. The CTS-1 cluster model consists of 3,888 compute nodes (24 SUs); 216 GB/s per edge IB switch (i.e., 4x EDR 36-port switch); and a 70 GB/s parallel file system (PFS) with perfect provisioning [2] as well as 432 GB/s core switch pool (i.e., the bandwidth available in routing I/O requests to the LNET routers). We assume that no significant I/O contention occurs when applications write to the burst-buffer (BB), which requires that the BB is either located at the compute node or close enough such that the probability of contention is minimal, as discussed in Section 3.3.

We define the bandwidths for the edge switch and core switch pool based on the requirements in the draft RFP and the latest IB technologies [20]. The bandwidth of the PFS is determined based on the checkpointing patterns captured in the CORAL RFP [1], as CTS-1 does not capture these requirements in the presence of the burst buffers. We assume that applications need to be able to write a checkpoint every hour with a per-node checkpoint size of 1/2 of the available node memory. We use the same frequency and size for the checkpointing on CTS-1 whose nodes are each assumed to have 128GB of memory. With this setting, a CTS-1 node will need to checkpoint at least 64GB/hour with an average I/O bandwidth of ∼18MB/s per compute node and ∼70GB/s for the entire system (i.e., 18MB/s × 3,888 nodes). We model the BW of the PFS either as perfectly provisioned (i.e., perfectly matching the applications' requirements) or as underprovisioned (i.e., less than the applications' requirements), as further described in Section 5.1.

4.2 Workload Model

As CTS-1 does not exist yet, we extrapolate the job traces from two other clusters at LLNL: the 1,200 compute-node Cab and the 2,740 compute-node Zin. We feed the same traces to both the I/O-aware and I/O-ignorant versions of our tests. We statistically generate the profiles of job workloads in terms of job submission times/rate, job request size per node, requested time limit and elapsed times by using real traces from LLNL's current resource and job management system: Slurm (cluster resource manager) and Moab (scheduler).

At LLNL, jobs are submitted to Moab which stores the submission times and queues the jobs up until they are ready to launch. When a set of jobs are ready to be executed, Moab sends these jobs to Slurm which then launches them on its cluster. Since Slurm only receives jobs when they are ready to launch, only Moab has the correct submit time in its database. Slurm records the start and end times of the jobs but does not return these times back to Moab. The net effect is that an individual database does not contain all of the needed information, and thus we relate the two data sets. Because these data sets do not have common unique identifiers, we relate them by generating statistical patterns from the two sets of data and then merge these statistics to get a holistic view of the data.

From these statistics, we derive representative job workloads for our tests. Specifically, we model the *arrival rates of jobs* as random events by using a Poisson distribution. There are two assumptions underlying the use of the Poisson distribution: the event occurrences are all independent and the events occur at a constant, average rate. Previous work by Dinh *et al.* [8] shows, however, that it is inaccurate to model job submission times directly as they are not independent events: a single user normally submits multiple jobs at the same time. Instead, we use a method proposed by Dinh *et al.*: modeling "user arrivals." In the existing traces, we find all the instances where a user submitted many jobs in quick succession (i.e., less than 10 seconds between each job) and bundle those job submissions up into one "user arrival." We also build a distribution that models the number of jobs that a user is expected to submit when they arrive. This binning into "user arrivals" makes our I/O profiles satisfy the independence assumption of Poisson distributions. However, this approach can still violate the constant, average rate assumption. It is well-known that user arrival is more common during the day on a weekday than at night on a weekend. To satisfy the constant, average rate requirement, we extend the previous work done by Dinh *et al.* as follows. We break the job data into four ranges: weekday day from 6:00am to 6:59pm; weekday night from 7:00pm to 5:59am; weekend day from 6:00am to 6:59pm; and weekend night from 7:00pm to 5:59am. These four ranges represent periods of time where the user arrivals are mostly constant. By chunking our data into these four ranges, we now satisfy the constant, average rate requirement of Poisson distributions as well. We record the lambda value for user arrival in each time range and build a Poisson distribution for each range.

We model the *job request size* in terms of the number of nodes requested by each job. These values are recorded by both Moab and Slurm, so either data set is sufficient. The analysis of the distribution of node request sizes shows a rapidly decreasing function that is heavily biased towards smaller sizes requests. We also notice that request sizes are biased towards powers-of-two values. To model the population distribution of node request sizes both accurately and automatically, we use a sampling distribution. We bin the data where the size of each bin is a monotonically increasing power of two and then record the number of jobs in each bin — much like a histogram. We then build a sampling distribution from these values.

We model the *time limit* and *elapsed time* in terms of the wall-time requested for a job and the job's actual execution time. These values are taken from the Slurm data set since it contains the execution time information. There is a correlation between the amount of time requested by the user and how long the job ran (i.e., elapsed time). Thus, instead of modeling the values separately, we model them together. We again build a sampling distribution of these values by collecting all of the unique tuples of the form (time limit, elapsed time) and counting the frequency at which they oc-

[2] We expect that the initial PFS BW provisioning of such a system will be much larger because not all needed system software including smart staging will have matured [21].

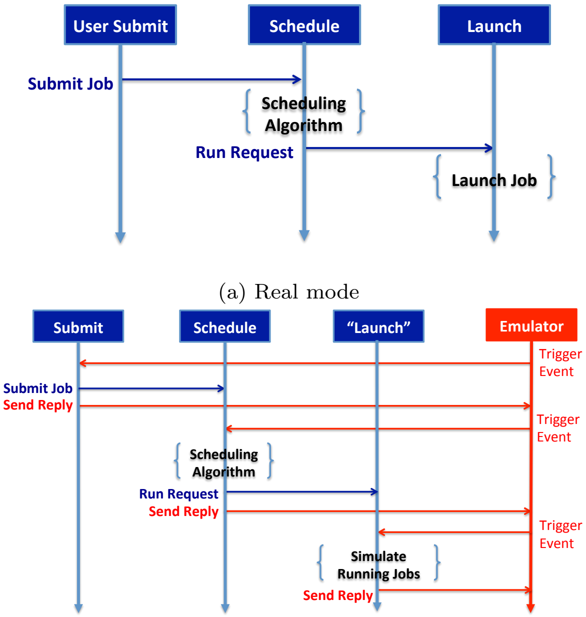
cur. From these frequencies, we calculate the probability density function for time limit and elapsed time.

Finally, we build our I/O workloads by sampling values from the aforementioned distributions. Jobs (i.e., the smallest unit of input for our tests) consist of the following information: (1) a monotonically increasing identifier starting from one; (2) the number of nodes requested by the user generating the job; (3) the number of cores requested by the user generating the job; (4) the time limit (i.e., how much time is requested by the user) sampled from the sampling distribution that is generated using the Slurm data; (5) submit time (i.e., when is the job submitted) sampled from the appropriate Poisson distribution (i.e., from weekday day, weekday night, weekend day, or weekend night traces); (6) elapsed time sampled simultaneously with time limit from the sampling distribution that is generated using the Slurm data; and (7) an average I/O Rate of 18MB/s per node in the job. We use 18MB/s as the average I/O rate per node under the assumption that all jobs will follow the checkpointing pattern outlined in Subsection 4.1. This means that the total I/O rate of a job is correlated with the number of nodes in the job. For future work, we are investigating the effects of a more diverse mix of job I/O rates.

4.3 Emulation Environment

To test our approach without having to launch real jobs, we developed a scheduling emulator within Flux. Figure 4.a depicts Flux in real use. Users submit jobs to the scheduler which launches them based on a defined scheduling policy. Our emulation mode entirely removes any user interaction and replaces it with an auto-submission module as shown in Figure 4.b. Moreover, the emulator does not rely on the direct execution of jobs but on the simulation of their execution. It can use the Slurm database of submitted jobs to generate the profile of simulated job executions. As real time is too slow for the testing, the emulator triggers events synthetically by extending Flux as shown in Figure 4.b. By using the emulator, we can rapidly study critical questions associated with the scheduler.

For the schedulers themselves, we take an incremental approach in implementing them into the Flux emulator for a fair comparison. We start with the simple FCFS scheduler and extend it to support EASY backfill scheduling. We adapt these schedulers to become I/O-aware by adding I/O as an additional constraint when determining if a job can be scheduled and executed. To this end, our adapted scheduler allocates and deallocates I/O bandwidth within the I/O hierarchy to work correctly within the I/O-aware emulator. When adding I/O to schedulers that use a priority function, we include the I/O requirements of a job in the scheduler's priority function.



(a) Real mode

(b) Emulation mode

Figure 4: Flux in real vs. emulation mode.

4.4 Contention Model

To quantify the effect of I/O contention, we model the contention and calculate the slowdown that applications experience when running on an over-allocated PFS.

We first model the contention that occurs at every resource within the I/O hierarchy (i.e., nodes, switches, and PFS). For each resource, we consider the amount of I/O BW that is being requested by its children in the hierarchy. The child requests ($ReqBW$) are sorted based on their BW size from least to greatest (i.e., given n children, $ReqBW_i \leq ReqBW_{i+1}$, for each $0 \leq i < n-1$). We calculate the actual bandwidth that each child receives as:

$$ActualBW_i = min(ReqBW_i, AvgRemainingBW_i)$$

where $AvgRemainingBW_i$ is a function of the parent's peak BW ($PeakBW$) and is defined as:

$$AvgRemainingBW_i := \frac{PeakBW - \sum_{j=0}^{i-1} ActualBW_i}{n-i}$$

Two cases can be observed. First, the sum of the requests BWs is smaller than the available. Thus, all children get their requested BW and any extra BW remains at the parent. Second, the sum of the requests BWs is greater than the available. Thus, the smaller requests are completely satisfied but for larger requests the remaining BW is distributed equally. Contention only occurs in the second case. In our contention model, we assume that the number of children and the degree of overallocation has no effect on the aggregate bandwidth. In reality, the aggregate bandwidth of a resource decreases w.r.t. the number of children and degree of overallocation due to increased cache thrashing, extra seeks at the PFS, and additional dropped packets. Our assumption is still valid for our tests because it minimizes the effects of contention, which is expected to occur only under I/O-ignorant scheduling. Thus, we do not introduce any positive bias towards our I/O-aware scheduling.

Each job comes with a required BW ($JobActualBW$) but receives an actual BW ($JobActualBW$). The actual job BW is the minimum of the actual BW of the resource the job is running on. We use the interference factor (I) as defined in [9] to determine the time an application spends performing computation or blocking on I/O, where the interference factor is defined as:

$$I = \begin{cases} 1, & \text{if } JobActualBW = JobReqBW \\ \frac{JobActualBW}{JobReqBW}, & \text{otherwise} \end{cases}$$

When a job's actual BW equals its requested BW, I is equal to one; otherwise, the job experiences contention and I ranges between zero and one. As the jobs running on the system change, the interference factor for each job also changes; we discretize time into intervals (Δ_i), during which each job's interference factor (I_i) can be considered constant. The time an application spends performing computation is defined by:

$$T(computing) = \sum_i (I_i \times \Delta_i) \tag{1}$$

and the time an application spends blocking on I/O is defined by:

$$\begin{aligned} T(I/O) &= \sum_i (\Delta_i - (I_i \times \Delta_i)) \\ &= (\sum_i \Delta_i) - T(computing) \end{aligned}$$

Under our model, jobs under I/O contention ($I < 1$) perform less computations than jobs not suffering from contention ($I = 1$) in the same period of time. For our tests in Section 5.1, we observe that the percentage of time that the interference factor is less than one ranges between 55% and 65% for underprovisioned PFSes with an I/O-ignorant scheduler; the percentage of time is 0% in the other cases.

5. RESULTS

In this section we provide empirical evidence that I/O-aware scheduling can increase the PFS efficiency and reduce job performance variability due to I/O contention.

5.1 Critical Questions and Test Settings

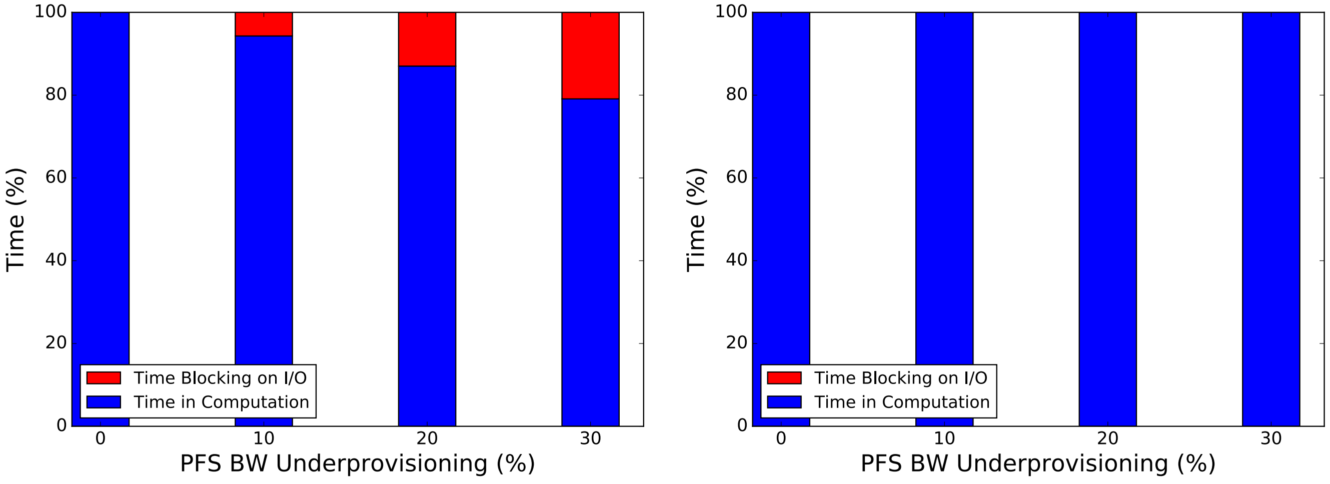
We address four critical questions: (1) Does I/O-aware scheduling impact the percentage of time that nodes spend in computation? (2) Does I/O-aware scheduling impact the variability of each individual job's performance? (3) Does I/O-aware scheduling affect the time to make a scheduling decision? and (4) What is the trade-off between system efficiency and turnaround time when comparing I/O-ignorant with I/O-aware schedulers? To answer these questions, we run tests with an EASY Backfilling scheduler on CTS-1 using four levels of PFS underprovisioning: no underprovisioning or 0% (70GB/s), an underprovisioning of 10% (63GB/s), an underprovisioning of 20% (56GB/s), and an underprovisioning of 30% (49GB/s). The sets of tests with the FCFS schedulers displayed similar trends as the EASY backfilling scheduler and thus are not shown in this section. Each test consists of 2500 jobs built from the workload model described in Section 4.2 and executed with the Flux emulator using the model of CTS-1 described in Section 4.1. For our tests, we assume that there are no external sources of I/O outside the control of the scheduler (e.g., interactive users). However if the system does have external sources of I/O, our scheduler can still work by reserving a fraction of the parallel filesystem (PFS) bandwidth (BW) for these external sources of I/O. Our results can support system administrators in deciding how much BW to reserve for these external sources. For example, a system with a perfectly provisioned PFS that reserves 30% of the PFS BW for extraneous I/O will produce the same results as a system with a PFS that has been underprovisioned by 30% and does not have extraneous sources of I/O, which is one of the scenarios that we show in our results.

5.2 Impact on Total Performance

To assess whether I/O-aware scheduling impacts the percentage of time that nodes spend in computation, we measure the total time that allocated nodes spend performing computation versus blocking on I/O. Figure 5.a refers to the set of tests in which the I/O-ignorant scheduler is used; Figure 5.b refers to the tests where the I/O-aware scheduler is used. For each level of underprovisioning, the figures report the percentage of time that nodes spend in computing (i.e., the blue bar) and in blocking on I/O (i.e., the red bar). As shown in Figure 5.a, under I/O-ignorant scheduling, nodes spend 100% of the jobs' time in computation only if the PFS is perfectly provisioned. However, as the PFS bandwidth decreases due to underprovisioning, the percentage of time allocated to computation also decreases. This is due to the fact that the reduced PFS bandwidth increases the probability that I/O contention occurs. For example, when the PFS is underprovisioned by 30%, only 79.1% of the nodes are executing jobs' computations; the rest are blocking on I/O. On the other hand, Figure 5.b shows that regardless of the PFS underprovisioning, I/O-aware scheduling keeps the nodes in computation 100% of the time. This is because jobs are scheduled only when sufficient resources (both CPU and I/O) are available. These results support the need for I/O-aware scheduling in order to prevent I/O contention when data-intensive jobs are simultaneously run on an underprovisioned PFS.

5.3 Impact on Individual Job Performance

To assess the impact of the I/O-aware scheduling on each individual job's runtime, we measure the performance variability of jobs launched on the model of CTS-1 under I/O-ignorant and I/O-aware EASY backfilling scheduling. Performance variability is measured as the percentage of time spent by each individual job doing computations versus blocking on I/O. In Figure 6, we present the variability of the 2500 jobs run under the four different levels of PFS underprovisioning (i.e., 0%, 10%, 20%, and 30%). Under I/O-ignorant scheduling in Figure 6.a, we observed that only when the PFS is perfectly provisioned do the jobs not exhibit any variability in performance . As the degree of PFS underprovisioning increases, the variability of job performance also increases. For example, when the PFS is underprovisioned by 30%, the amount of time that individual jobs spend in computation ranges from 66.7% to 100%. On the other hand, in Figure 6.b we do not observe any performance variability under I/O-aware scheduling. The conclusions are twofold. First, we observe that under an I/O-ignorant scheduling, there is no guarantee on job performance and each job's performance varies wildly. In contrast, under an I/O-aware scheduling, every job is guaranteed to receive the required I/O, thus resulting in no variability. Second, as the PFS is being increasingly underprovisioned, individual jobs' performance variability grows larger under I/O-ignorant scheduling but remains zero under I/O-aware scheduling. This consistency under I/O-aware scheduling means that users fully receive the requested resources and thus their jobs are not unexpectedly slowed down by other jobs on the system. In other words, I/O-aware scheduling



(a) I/O-ignorant

(b) I/O-aware

Figure 5: Percentage of total time spent by the entire CTS-1 cluster in computation and blocking on I/O.

resolves the variability in job performance due to I/O contention.

5.4 Impact on Scheduling Decision Time

The benefits of I/O-aware scheduling do not come without a cost. The additional checking that the I/O-aware scheduler performs when deciding which jobs to place on the system increases the time to make a scheduling decision versus an I/O-ignorant scheduler. The scheduling decision time is the number of seconds between when a job state change occurs and when the scheduler makes a decision based on that state change. Intuitively, a more computationally-expensive scheduling algorithm, such as our I/O-aware scheduling, can cause a longer scheduling decision time. The decision time of a scheduler becomes a major concern when it is longer than the time between state changes. In the job workloads described in Section 4.2 and used for our tests, the average time between state changes is 31.7 seconds.

Figures 7.a and 7.b summarize the observed additional cost in terms of the scheduler's decision time (in seconds) for EASY backfilling running on the emulated model of CTS-1 under the four different underprovisioning levels with I/O-ignorant and I/O-aware schedulers respectively. The number of sampled times in each scenario is on average 9,580 (roughly four times the number of jobs). This is expected since there are four state transitions that each job goes through, and each state transition causes the scheduler to run. Jobs that transition states together cause only one single invocation of the scheduler and are counted as a single sample time.

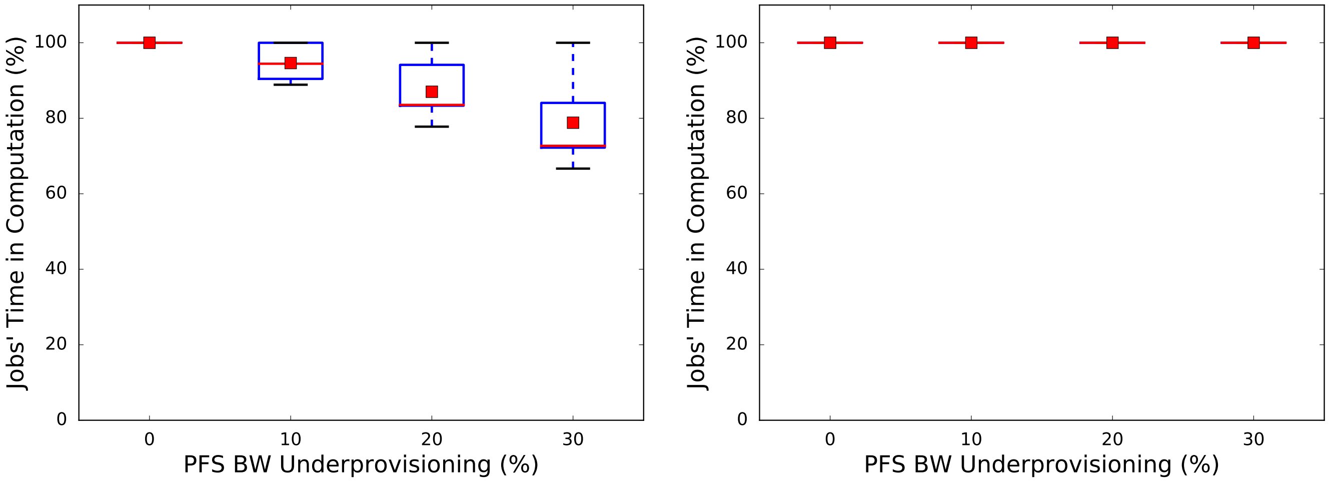
In Figures 7.a and 7.b, we represent the sampled times with box plots. Traditionally a box plot consists of six different pieces of information. The whiskers on the bottom extend from the 5th percentile to the top 95th percentile. The top, bottom, and line through the middle of the box correspond to the 75th percentile (top), 25th percentile (bottom), and 50th percentile (middle). A square indicates the arithmetic mean. Due to the distribution of the times in our tests, only the 75th and 95th percentiles are visible. With I/O-ignorant scheduling, 75% of the decision times are below 0.07 seconds and the 95th percentile times are ∼1.43 seconds. With I/O-aware scheduling, 75% of the decision times are below 0.12 seconds and the 95th percentile times range between 1.97 and 6.64 seconds.

The critical comparison of Figures 7.a and 7.b outlines the following three trends. First, when we consider the lowest 75% of the decision times for I/O-ignorant and I/O-aware scheduling, scheduler decision times differ by at most 0.04 seconds. Second, the variability of the decision times under I/O-ignorant scheduling remains constant irrespective of PFS BW underprovisioning. This is not the case for I/O-aware scheduling, for which we observed that the variability of the largest 25% decision times increases when the PFS is underprovisioned. Third, for both schedulers, the 95th percentile decision time is still shorter than the average time between state changes, which is on the order of 10s of seconds.

Preliminary analysis of the longest 25% of decision times under both schedulers suggests that these times occur after the completion of a job with either large compute or I/O requirements followed by the scheduling of many smaller jobs. The fact that I/O-aware scheduling with an underprovisioned PFS exhibits larger variability indicates that the problem is exacerbated by the additional I/O constraints. The additional variability can potentially be alleviated with the introduction of new mechanisms such as caching, preemptive scheduling, and hierarchical scheduling [13].

5.5 System Efficiency vs. Turnaround Time

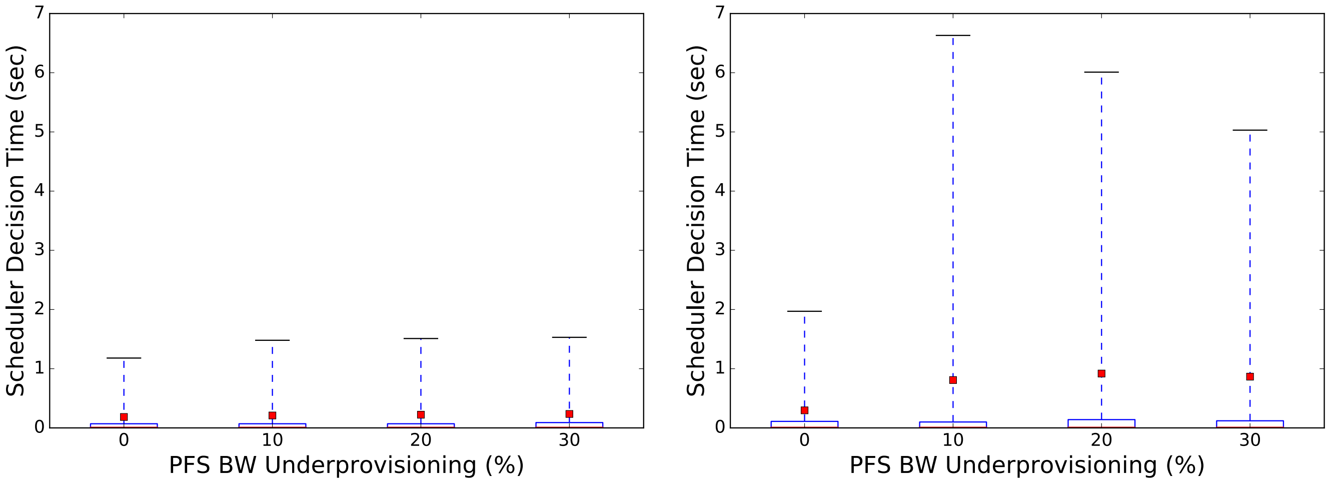
Users running applications want a system that maximizes the total useful computation performed on the allocated nodes (system efficiency) and minimizes the time between when a job is submitted and when the job completes (turnaround time). The turnaround time is the time the job spends in the scheduler's queue plus the time the job spends in execution. To ensure that jobs obtain their required I/O bandwidth, the I/O-aware scheduler may delay the execution of some jobs until more I/O resources become available. In other words, jobs obtain the exactly required resources, achieving 100% system efficiency, in exchange for a poten-



(a) I/O-ignorant

(b) I/O-aware

Figure 6: Variability of individual jobs' time spent in computation.



(a) I/O-ignorant

(b) I/O-aware

Figure 7: Scheduler decision time distributions.

tially longer time in the queue and consequently, a longer turnaround time. To quantify this trade-off, we measure the lower-bound ratio of I/O-aware system efficiency over I/O-ignorant system efficiency as well as the upper-bound ratio of I/O-aware turnaround time over I/O-ignorant turnaround time for the same CTS-1 tests as in Figures 5-7 when using EASY backfilling scheduling. For the system efficiency ratio, the higher, the better for I/O-aware scheduling; for the turnaround time ratio, the lower, the better for I/O-aware scheduling. The ratio of system efficiencies is defined as a lower-bound since we expect this ratio to be higher in real systems. This is because the contention model that we use for our tests is quite conservative and underpenalizes I/O contention, which only occurs in I/O-ignorant simulations, as we discussed in Section 4.4. A greater penalty on contention can further reduce the overall system efficiency under I/O-ignorant scheduling and consequently increase the system efficiency ratio. On the other hand, the turnaround time ratio is an upper-bound since we expect this ratio to be lower in real systems. This is because in real workloads we can observe a greater diversity in job I/O requirements; this diversity allows the I/O-aware scheduler to fill idle nodes with jobs that have a low I/O requirement, decreasing the turnaround time of jobs under I/O-aware scheduling, and consequently decreasing the ratio.

In Figure 8, we observe how in a perfectly provisioned PFS, I/O-ignorant and I/O-aware scheduling have the same system efficiency and turnaround time (i.e., the ratios are equal to one). As the level of PFS underprovisioning increases, the ratios increase. For example, in our tests we observed that at 30% underprovisioning, I/O-aware scheduling has, on average, a 1.29 times greater lower-bound system efficiency ratio and a 1.52 times greater upper-bound turnaround time ratio. This means that for real-world workloads, the system under I/O-aware scheduling can perform at least 29% more science (lower bound) as the nodes are fully utilized for computation but, at the same time, the turnaround time can be up to 52% longer (upper bound). These observations support our claim that I/O-aware scheduling boosts the amount of science performed by scientific workloads despite a longer turnaround time.

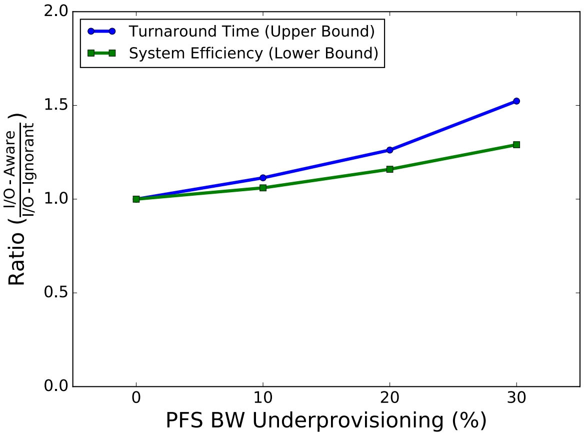


Figure 8: System efficiency versus job turnaround time.

6. RELATED WORK

I/O-aware scheduling is still in its infancy. First efforts have extended existing scheduling policies by using heuristics in application-specific domains. For example, the extended FCFS scheduling in [11] integrates heuristics to deal with irregular I/O-intensive jobs. The heuristics search for I/O parameter values among the parameter ranges using testing. The work builds upon and extends previous work [6]. In this paper, we move away from individual heuristics and applications.

More recent efforts on runtime scheduling of I/O contention include [35, 10, 9, 36]. Specifically, work in [35] addresses I/O contention with a PFS access controller: the controller provides a single application exclusive access to the PFS for a time window. Work in [9] analyzes the I/O contention between two applications and investigates the use of runtime application coordination in order to avoid congestion. Work in [10] analyzes the impact of congestion on applications' I/O bandwidth and assesses a variety of runtime techniques designed to either maximize system efficiency or minimize application slowdown. Finally, work in [36] presents an I/O-aware scheduling framework that coordinates I/O requests at runtime on petascale computing systems driven by either user-oriented metrics or system performance. Our work aligns with and complements these four runtime methods by targeting next-generation large-scale systems and defining a method that operates at batch schedule time to mitigate I/O contention.

IBM defines network-aware scheduling within their production resource manager and scheduler [15]. Our I/O-aware scheduling can directly apply to network-aware scheduling by extending our I/O subsystem resource model to include the full switch hierarchy and making schedulers consider the network bandwidth requirement of a parallel or distributed application.

When looking at resource allocations in a broader spectrum of systems, including grid and batch systems, many do not target I/O bandwidth as constraints, although they have begun to consider increasingly diverse resource types beyond compute nodes and cores. For example, work in [16] deals with multiple resource scheduling (MRS) algorithms aiming for the minimal execution schedule through efficient management of available grid resources (i.e., memory, disk and CPUs of a wide area distributed computing platform).

Finally, dynamic scheduling work such as [29, 30] can complete our approach by enabling the extension of our solutions to more traditional systems without burst buffers. Dynamic scheduling extends a batch system with dynamic allocation facilities to support on-the-fly resource allocation to elastic jobs; although dynamic scheduling does not currently target any non-traditional constraints, including I/O bandwidth.

7. CONCLUSIONS

The economics of flash vs. disk storage is increasingly motivating large HPC centers to underprovision parallel file system bandwidth. With such underprovisioned parallel file systems (PFSes), avoiding an I/O storm at the PFS level is critical to achieving computational efficiency and to avoiding any disruption of the entire HPC center.

In this paper we present a novel solution that allows us to meet these challenges at the batch job scheduler layer. Our technique reduces I/O contention by incorporating I/O-awareness directly into scheduling policies such as FCFS and EASY backfilling. We model the links between all levels in the storage hierarchy and use this model at schedule time to avoid I/O contention.

We explore the effectiveness and scalability of our method using schedulers and an emulator built on top of the Flux resource and job management framework. We show that I/O-aware scheduling eliminates all I/O contention on the system, regardless of the level of underprovisioning. In addition, it ensures that all jobs receive the I/O bandwidth they require. Furthermore, we observed that our solution reduces job performance variability by up to 33% and increases system utilization by up to 21%.

Our results suggest that I/O-aware scheduling can scale to handle the next-generation of HPC systems and ultimately improve the amount of science performed on these systems.

8. ACKNOWLEDGMENTS

The authors thank Jay Lofstead for his insights and feedback during the review process. Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-679271). The UD authors acknowledge the support of NSF grants CCF-1318445/1318417.

9. REFERENCES

[1] CORAL. Retrieved Jan 3, 2015 from https://asc.llnl.gov/CORAL/.

[2] The Moab workload manager. http://www.adaptivecomputing.com/resources/docs/mwm/help.htm#topics/0-intro/productOverview.htm. Retrieved Jan 3, 2015.

[3] Trinity and NERSC-8 Computing Platforms Draft Technical Requirements. Retrieved Jan 3, 2015 from http://www.lanl.gov/business/vendors/_assets/docs/Trinity-NERSC-8-DRAFT-technical-requirements.pdf.

[4] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz. Flux: A Next-Generation Resource Management Framework for Large HPC Centers. In *Proc. of the 10th International Workshop on Scheduling and Resource Management for Parallel and Distributed System (SRMPDS)*, Sep. 2014.

[5] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring. Jitter-free Co-Processing on a Prototype Exascale Storage Stack. In *Proc. of 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, April 2012.

[6] M. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage System. In *Proc. of the 2000 Mass Storage Conference*, March 2000.

[7] A. Chandio, C.-Z. Xu, N. Tziritas, K. Bilal, and S. Khan. A Comparative Study of Job Scheduling Strategies in Large-Scale Parallel Computational Systems. In *Proc. of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, July 2013.

[8] T. Dinh, L. Andrew, and P. Branch. Exploiting per User Information for Supercomputing Workload Prediction Requires Care. In *Proc. of the 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2013.

[9] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim. CALCioM: Mitigating I/O Interference in HPC Systems Through Cross-Application Coordination. In *Proc. of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, May 2014.

[10] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir. Scheduling the I/O of HPC Applications Under Congestion. In *Proc. of the 2015 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2015.

[11] L. F. Goes, P. Guerra, B. Coutinho, L. Rocha, W. Meira, R. Ferreira, D. Guedes, and W. Cirne. AnthillSched: A Scheduling Strategy for Irregular and Iterative I/O-Intensive Parallel Jobs. In *Job Scheduling Strategies for Parallel Processing*, volume 3834 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg, 2005.

[12] G. Grider. Speed Matching and What Economics Will Allow. In *HEC FSIO Research and Development Workshop*, August 2010.

[13] S. Herbein, D. H. Ahn, and M. Taufer. Poster: Exploring the Trade-Off Space of Hierarchical Scheduling for Very Large HPC Centers. In *Proc, of the 27th ACM/IEEE International Conference for High Performance Computing and Communications Conference (SC)*, November 2015.

[14] IBM. General Parallel File System. http://www-03.ibm.com/software/products/en/software. Retrieved 10 Nov, 2015.

[15] IBM. Network-aware scheduling. https://www-01.ibm.com/support/knowledgecenter/#!/SSETD4_9.1.2/lsf_admin/pe_network_aware_sched.dita. Retrieved 12 Nov, 2015.

[16] B. B. Khoo, B. Veeravalli, T. Hung, and C. S. See. A Multi-dimensional Scheduling Scheme in a Grid Computing Environment. *Journal of Parallel and Distributed Computing*, 67(6):659 – 673, 2007.

[17] D. Klusáček and Š. Tóth. On Interactions among Scheduling Policies: Finding Efficient Queue Setup Using High-Resolution Simulations. In *Proc. of the 2014 20th International Euro-Par Conference on Parallel Processing*, August 2014.

[18] Lawrence Livermore National Laboratory. Advanced Simulation and Computing Sequoia. https://asc.llnl.gov/computing_resources/sequoia. Retrieved 16 May, 2015.

[19] Lawrence Livermore National Laboratory. Linux @ Livermore. https://computing.llnl.gov/linux/projects.html. Retrieved 10 Nov, 2015.

[20] Lawrence Livermore National Laboratory. Tri-Laboratory Commodity Technology System 1. https://asc.llnl.gov/computers/cts1-rfp. Retrieved 11 Nov, 2015.

[21] Lawrence Livermore National Laboratory. Trinity / NERSC-8 Use Case Scenarios. https://www.nersc.gov/assets/Trinity--NERSC-8-RFP/Documents/trinity-NERSC8-use-case-v1.2a.pdf. Retrieved 10 Nov, 2015.

[22] D. A. Lifka. The ANL/IBM SP Scheduling System. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS)*, May 1995.

[23] 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 *Proc. of the 2012 IEEE Conference on Massive Data Storage (MDS)*, Apr. 2012.

[24] J. Lofstead, I. Jimenez, and C. Maltzahn. Consistency and Fault Tolerance Considerations for the Next Iteration of the DOE Fast Forward Storage and IO Project. In *Proc. of the 6th Workshop on Interfaces and Architectures for Scientific Data Storage (IASDS)*, Sep. 2014.

[25] Lustre. Lustre. http://lustre.org. Retrieved 10 Nov, 2015.

[26] R. Mckenna, T. Gamblin, A. Moody, and M. Taufer. Poster: Forecasting Storms in Parallel File Systems. In *Proc, of the 27th ACM/IEEE International Conference for High Performance Computing and Communications Conference (SC)*, November 2015.

[27] NERSC. Burst buffer architecture and software roadmap. http://www.nersc.gov/users/computational-systems/cori/burst-buffer/burst-buffer/. Retrieved 01 Apr, 2016.

[28] Oak Ridge National Laboratory. Introducing Titan - Advancing the Area of Accelerated Computing. https://www.olcf.ornl.gov/titan/. Retrieved 16 May, 2015.

[29] S. Prabhakaran, M. Iqbal, S. Rinke, C. Windisch, and F. Wolf. A Batch System with Fair Scheduling for Evolving Applications. In *Proc. of the 43rd International Conference on Parallel Processing (ICPP)*, pages 351–360, Sep. 2014.

[30] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kalé. A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications. In *Proc. of 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2015.

[31] K. Sato, K. Mohror, A. Moody, T. Gamblin, B. de Supinski, N. Maruyama, and S. Matsuoka. A User-Level InfiniBand-Based File System and Checkpoint Strategy for Burst Buffers. In *Proc. of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2014.

[32] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Jan 2002.

[33] G. M. Shipman, D. A. Dillow, S. Oral, and F. Wang. The Spider Center Wide File System: From Concept to Reality. In *Proc. of the Cray User Group (CUG) Conference*, 2009.

[34] S. Thapaliya, P. Bangalore, J. Lofstead, K. Mohror, and A. Moody. IO-Cop: Managing Concurrent Accesses to Shared Parallel File System. In *Proc. of the 6th Workshop on Interfaces and Architecture for Scientific Data Storage (IASDS)*, Sep. 2014.

[35] S. Thapaliya, P. Bangalore, J. Lofstead, K. Mohror, and A. Moody. IO-Cop: Managing Concurrent Accesses to Shared Parallel File System. In *Proc. of 2014 43rd International Conference on Parallel Processing Workshops (ICCPW)*, Sept 2014.

[36] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan. I/O-Aware Batch Scheduling for Petascale Computing Systems. In *Proc. of 2015 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2015.