ADVANCED SCHEDULERS FOR NEXT-GENERATION HPC SYSTEMS

by

Stephen Herbein

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Summer 2018

ADVANCED SCHEDULERS FOR

NEXT-GENERATION HPC SYSTEMS

by

Stephen Herbein

Approved: _____

Kathleen F. McCoy, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____

Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____

Douglas J. Doren, Ph.D.
Interim Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Michela Taufer, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Sunita Chandrasekaran, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Rui Zhang, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Martin Schulz, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Dean Hildebrand, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Dong H. Ahn, MS
Member of dissertation committee

ACKNOWLEDGEMENTS

This work would not have been possible without the mentorship and support of several intelligent and passionate colleagues. First, I would like to express my sincerest gratitude to my advisor Dr. Michela Taufer. Without her mentorship and encouragement, I would never have gone to graduate school, let alone completed a Ph.D. I have learned countless things under her mentorship, and I greatly appreciate all the time and energy she has invested in me. Second, I would like to thank Dong Ahn for being a part of my committee, mentoring me during my summers at Lawrence Livermore National Laboratory (LLNL), and collaborating with me during the school year. Third, I would like to thank the other members of my committee: Dr. Sunita Chandrasekaran, Dr. Rui Zhang, Dr. Martin Schulz, and Dr. Dean Hildebrand. Completion of this research would not have been possible without the guidance and mentorship of my committee.

My thanks go to LLNL and Livermore Computing for providing me with the funding and computing resources to perform this research. I am especially grateful to the Flux team members for developing a cutting-edge framework that I had the luck and privilege of building my research on top of: Dr. Tapasya Patki, Dr. Becky Springmeyer, Jim Garlick, Mark Grondona, Dr. Tom Scogland, Don Lipari, Chris Morrone, and Al Chu. I want to thank Dr. Bronis R. de Supinski, Dr. Tamara Dahlgren, and David Domyancic for their advice and collaboration on hierarchical scheduling; Michael Wyatt, Dr. Todd Gamblin, Adam Moody, and Ryan McKenna for helping me integrate their machine learning models; and Dr. Felix Wolf and Dr. Suraj Prabhakaran for their collaboration on dynamic hierarchical scheduling; and Dr. Jay Lofstead and Marc Stearman for their insights and feedback on the I/O-aware scheduling work. Many

thanks to the members of the Global Computing Lab for their stimulating discussions, valuable collaboration and priceless friendship.

Last but not least, I would like to thank my friends and family for always believing in me. When the progress slowed and the hours grew long, you were there to encourage and uplift me. I could not have achieved this without your unfailing love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

High performance computing (HPC) is undergoing many changes at both the system and workload levels. At the system level, data movement is becoming more costly in relation to computation and HPC centers are becoming increasingly power-constrained. In an effort to adapt to these trends, HPC systems are including new resources such as burst buffers and GPUs which makes the resource set larger and more diverse. At the workload level, new ensemble workloads, such as uncertainty quantification (UQ), are emerging within HPC, driving up the workload scale in terms of the number of jobs. Existing HPC scheduling models are unable to adapt to these changes, leading to degraded system efficiency and application performance.

In this thesis, we claim that new schedulers are needed to overcome the challenges mentioned above and efficiently manage the next-generation of HPC systems. To this end we design, implement, and evaluate three fundamental transformations to the existing scheduling models. First, we integrate I/O-awareness into existing scheduling policies and demonstrate that I/O-aware scheduling increase the efficiency of burst buffer-enabled HPC systems. Second, we expand our I/O-aware scheduler to incorporate the accurate knowledge of application I/O utilization patterns provided by machine learning models. Third, we design a prototype scheduler based on the fully hierarchical scheduling model and show that it reduces scheduler overhead and increases job throughput on synthetic and real-world ensemble workloads, such as UQ. Our work is the first step towards a new generation of scheduling models for HPC.

Chapter 1

THESIS OVERVIEW

1.1 Problems, Motivations, and Proposed Solutions

Due to the ever-growing computational needs of scientific simulation and modeling, compute resources continue to diversify and grow. Current architectural trends, such as the use of accelerators (e.g., GPUs) and other CPU designs that require managing massive numbers of hardware threads, present new challenges to schedulers. Furthermore, schedulers must account for the additional system constraints (e.g., I/O) that come from the introduction of burst buffers and underprovisioned parallel filesystems into next-generation systems. Schedulers that are ignorant of these increasingly diverse and constrained resources expose users to degraded system efficiency and additional application performance variability [46, 70, 89, 22].

The changes in HPC are not just confined to the diversity of resources. Today's workloads also stray from the traditional patterns. Scientific workloads are increasingly incorporating ensemble simulations, such as uncertainty-quantification (UQ). These ensemble simulations, which inject up to millions of short-running jobs into the system, result in a workload uncommon to HPC. The size of these workloads stresses existing schedulers beyond their limits, requiring users to build and maintain stop-gap solutions [44].

Despite these changes in HPC, batch job scheduling largely remains stuck in a decades-old paradigm. Existing batch schedulers lack the proper resource models and scheduling algorithms to effectively handle the increasingly complex resources and workloads. HPC batch job schedulers must change in three key ways to meet these new challenges. First, they must support I/O awareness to maintain high system

efficiency in the face of increasingly sophisticated and constrained I/O subsystems. Second, they must drive this I/O awareness with sources of I/O knowledge beyond HPC system users. Third, they must leverage hierarchical scheduling models to achieve the scalability required by emerging ensemble workloads. We will elaborate on these needs in Sections 1.1.1, 1.1.2, and 1.1.3, respectively.

1.1.1 I/O-Aware Scheduling

The first way that HPC schedulers must change is to incorporate I/O-awareness in their scheduling algorithms. For the past decade, HPC schedulers have only managed the compute systems, which are separate from the parallel file systems (PFSs) [75, 55, 98]. These PFSs are built from an array of disks and are provisioned for both capacity and bandwidth goals. As the growth of disk capacity continues to outpace increases in disk bandwidth, the community expects that PFSs will meet capacity goals but fail to deliver bandwidth cost-effectively [43, 19]. This trend is already apparent in recent leadership-class system procurements [1]. Next-generation systems will provide 7 to 10 times higher peak floating-point performance with only 1 to 2 times higher PFS bandwidth compared to previous generation systems [65, 86]. Large HPC centers have begun to face this challenge by using solid-state burst buffers [73, 97], a new storage medium that is cost-effective for bandwidth while not yet viable for capacity, as a layer 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 promises not only better and more consistent performance for users and their applications but also promises to reduce the necessary PFS bandwidth. If the burst buffer can be scheduled effectively, the PFS no longer needs to be provisioned for the worst-case scenario where multiple jobs happen to enter their I/O-dominant phases simultaneously.

Despite this rapidly changing environment, the resources models of existing schedulers remain simple and compute-focused. The resource models are incapable

of effectively expressing the connections between compute nodes, burst buffers, and parallel filesystems. Thus, the I/O awareness of today's batch schedulers remains rudimentary (e.g., Moab [28] can only hold dependent jobs when a file system is inoperative). While runtime I/O coordination and optimization techniques have shown promising results at maximizing the PFS utilization and minimizing I/O contention, they are ultimately constrained by the I/O requests of the specific running applications [32, 101, 38, 108]. If many I/O-intensive jobs are scheduled to run concurrently, these runtime techniques are unable to prevent a slowdown of the applications.

In this thesis, we tackle the I/O-aware scheduling problem using batch scheduling and complement existing work that manages the I/O contention problem at runtime [32, 101, 38, 108]. Our technique enables us to prevent application slowdown by ensuring that many I/O-intensive jobs are not scheduled to run concurrently. To this end, we integrate a model of the I/O subsystem and I/O-aware algorithms directly into Flux [15], one of the few resource and job managers for next-generation centers. By using the Flux emulator, we explore the effectiveness and cost of our scheduler on a burst buffer-enabled, large-scale, HPC system. 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.

1.1.2 I/O Knowledge Integration

The second way in which HPC schedulers must change is to begin driving their decisions with resource usage knowledge provided by resource prediction tools. As outlined in Section 1.1.1, I/O-aware scheduling is necessary for next-generation schedulers to effectively prevent contention on shared parallel filesystems and thus increase the amount of useful science performed (i.e., scientific goodput). To this end, our I/O-aware scheduler requires accurate knowledge about job resource usage patterns before the jobs run. Even in the case of simple resource usage patterns like job runtime, user-provided information is seldom accurate, with users tending to over-request job

runtimes [69, 81, 27]. In the case of more advanced resource usage patterns like I/O, users are not required to, and cannot be expected to, provide estimates. If our I/O-aware scheduler had access to accurate knowledge of job resource usage, it would better prevent system resource contention and improve scientific goodput.

In this thesis, we solve this information challenge by leveraging a state-of-the-art machine learning workflow (i.e., PRIONN) [104] to provide the job resource usage knowledge required by our next-generation, I/O-aware scheduler. We integrate PRIONN's resource usage predictions into our I/O-aware scheduler using Flux [15]. We evaluate the I/O-aware scheduler's ability to anticipate future system I/O usage.

1.1.3 Hierarchical Scheduling

The third way in which HPC schedulers must change is to support hierarchical scheduling models. Today's HPC workloads are straying from traditional execution patterns to leverage the increasing number and variety of resources that enable larger scale and higher fidelity simulations. Solving computational problems, whether they involve engineering safety-critical systems (e.g., bridges and aircraft) or studying real-world phenomena (e.g., climate [82], epidemics [34], molecular dynamics [91], and fusion [37]), typically involves executing many runs of the same application with varying inputs. These types of runs, referred to as *ensembles*, are a growing part of HPC workloads. Lawrence Livermore National Laboratory (LLNL) has seen a doubling in the number of ensembles executed over the last several years. Furthermore, the analysis of jobs in 2016 on a major cluster at LLNL shows 48.1% involved the submission of at least 100 identical jobs by the same user with 27.8% submitted within one minute of each other. Today's ensemble simulations employ 10^3 to 10^5 jobs, which pushes current resource management techniques to their limit, especially in terms of ingestion and scheduling [48]. This scale requires scientists and simulation tool developers to work around the limitations of today's schedulers [44, 30] with techniques that will simply not work for expected future studies with 10^8 jobs. The growing size and diversity of HPC workloads and resources require resource and job management software (RJMS)

to overcome three significant scheduling challenges: massive numbers of jobs and resources (*scalability*); diverse workloads (*workload specialization*); and enforcement of complex constraints at various system levels (*multi-level constraints*).

At present, there exist three common models for HPC batch job scheduling. First, the *centralized* model, which uses a single scheduler to maintain and track the full knowledge of resources and jobs. Second, the *decentralized* model, which uses multiple, interacting schedulers with no central leader responsible for global scheduling. Finally, the *(limited) hierarchical* model, which uses a meta-scheduler to maintain a global view of resources and jobs while delegating some job-related decision-making to its child schedulers. Under the centralized model (the most common), a single, global batch job scheduler makes all the decisions for ensemble workloads and, thus, becomes a performance bottleneck. The sheer number of jobs in ensemble workloads and the required coordination among those jobs can thrash a centralized scheduler. The decentralized and limited-hierarchical models have some advantages over the centralized model, including better scalability. However, these two models do not overcome the other two significant scheduling challenges: workload specialization and multi-level constraints. Specifically, the decentralized scheduling model cannot scalably guarantee strict multi-level resource constraints (e.g., power, network, or I/O bounds at the datacenter-, cluster-, rack-, and node-level). The limited hierarchical scheduling model can only provide workload specialization and multi-level constraints in certain configurations, and even then, its support is weaker than required for today's HPC systems and workloads. While tuning of the scheduling policies can improve the existing models [60], few studies have explored new, alternative HPC batch job scheduling models that overcome the three scheduling challenges.

In this thesis, we overcome all three scheduling challenges with *fully hierarchical* scheduling, which creates a scalable hierarchy of seamlessly integrated schedulers that each can be tailored to specific application workloads and system constraints. We prototype the scheduling model using Flux [15]. We evaluate the model's scalability on both synthetic and real-world ensemble workloads.

1.2 Thesis Statement

In this thesis, we claim that the batch job schedulers for the next-generation of HPC systems must support I/O awareness, drive the I/O awareness with external I/O knowledge, and leverage fully hierarchical models. To validate the thesis statement, we:

- Integrate an I/O subsystem model into a next-generation resource manager's scheduling policies to prevent contention within the I/O subsystem and evaluate the model's impact on system efficiency and scheduling cost
- Incorporate knowledge about job I/O usage patterns into the I/O-aware scheduler and evaluate the knowledge's effect on the scheduler's ability to accurately anticipate future system states
- Design a fully hierarchical scheduler to improve scheduler scalability, workload specialization, and multi-level constraint support and then evaluate the scheduler's job throughput and overhead on synthetic and real-world ensemble workloads

1.3 Contributions

From the perspective of I/O-aware scheduling, the contributions of this dissertation are as follows:

- Model the hierarchical nature of real-world I/O subsystems using a resource description language
- Incorporate I/O awareness as a driving scheduling factor into the scheduling policies of a next-generation resource manager (i.e., Flux)
- Evaluate the cost-effectiveness of I/O-aware scheduling when using perfect knowledge of application I/O patterns

From the perspective of I/O knowledge integration, the contributions of this dissertation are as follows:

- Integrate resource usage predictions made by a state-of-the-art machine learning workflow (i.e., PRIONN) into the I/O-aware scheduler
- Evaluate the scheduler's ability to estimate future system I/O usage

From the perspective of hierarchical scheduling, the contributions of this dissertation are as follows:

- Design a hierarchical scheduler using the fully hierarchical model
- Implement a working prototype of the scheduler on top of Flux
- Evaluate the prototype's scalability on both synthetic and real-world ensemble workloads

1.4 Organization

The remainder of this thesis is organized as follows: Chapter 3 presents the I/O subsystem model we developed and the impact of the model on system efficiency and application performance. Chapter 4 presents our integration of resource usage knowledge from a machine learning workflow into our I/O-aware scheduler and the knowledge's effect on the scheduler's ability to estimate future I/O usage and anticipate I/O bursts. Chapter 5 presents our fully hierarchical scheduler and its impact on scheduler scalability for synthetic and real-world ensemble workloads. Chapter 6 provides a summary of this thesis along with a description of future work.

Chapter 2

BACKGROUND

This chapter introduces the design of traditional resource and job management systems (RJMSes) as well as provides examples of how to use several real-world RJMSes. We focus on six RJMSes, in particular, based on their current relevance and impact in the field: SLURM, LSF, Moab/Torque, PBS, HTCondor, and Flux. These RJMSes represent a mixture of both open-source (i.e., SLURM, PBS, HTCondor, and Flux) and closed-source (i.e., LSF and Moab/Torque) projects developed by a wide variety of entities, including universities, national laboratories, and private companies. Figure 2.1 shows the general setup and flow of information of these RJMSes and serves as an outline for this chapter.

2.1 User Interaction

As depicted in Figure 2.1, users feed information into the RJMS via job submissions and cancellations as well as pull information out of the RJMS through job and

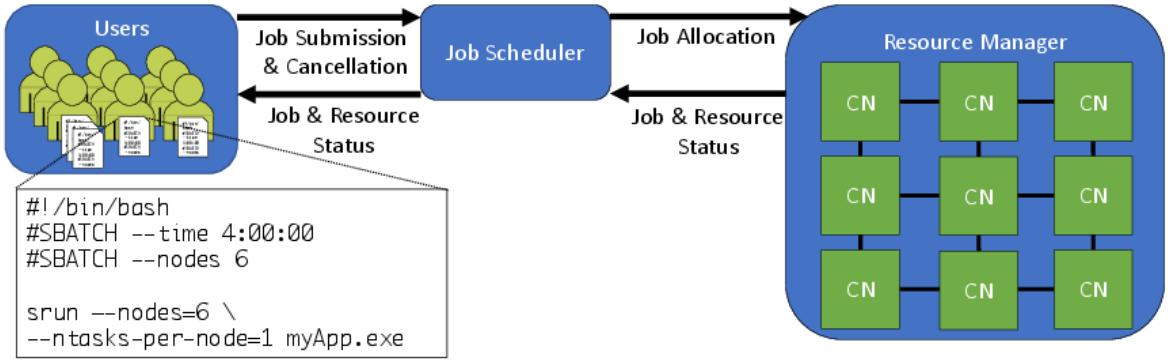


Figure 2.1: The setup and flow of information in a traditional resource and job management system.

resource status queries. When submitting a job, users are required to provide both job resource requirements and job tasks, and users are optionally able to provide hints for task mapping (i.e., how the system should map the job tasks on the job resources). These three pieces of information (i.e., job resources, tasks, and task mapping) are provided to the RJMS via a combination of job scripts and command-line interfaces. A minimal job script contains only the tasks to be executed, but a more complex job script also includes details about job resource requirements and task mapping. The example job script shown in Figure 2.1 contains all three pieces of information: the task to be run (i.e., myApp.exe), the overall resources required by the job (i.e., six nodes for four hours), and the task mapping (i.e., run one task on each of the six nodes). This job script is then submitted by the user to the scheduler via a command-line interface (CLI). The submission CLI for the six RJMSes that we focus on, along with the CLIs for canceling jobs, querying job and resource statuses, and other common operations, are shown in Table 2.1. It is important to note that all of these interfaces are compute-focused (i.e., number of nodes and number of cores), with little to no support for specifying job requirements on other resources like network and disk I/O.

	SLURM [7]	**LSF [11]**	**Moab [12]**	**PBS [13]**	**HTCondor [9]**	**Flux [10]**
Submit a Job	sbatch	lsub	msub	qsub	condor_submit	flux submit
Run a Job Step	srun	jsrun	mpiexec or mpirun	mpiexec or mpirun	openmpiscript, mp1script, or mpirun	flux wreckrun
Request an Interactive Session	salloc	bsub -I	msub -I	qsub -I	condor_submit -i	N/A
Cancel a Job	scancel	bkill	mjobctl -c	qdel	condor_rm	flux wreck cancel
Query Resource Status	sinfo	bhosts	showstate	pbsnodes	condor_status	N/A
Query Job Status	squeue	bjobs	showq	qstat	condor_q	flux wreck ls
Query Historical Job Usage	sacct	bhist	showstats	pbfs	condor_stats	N/A

Table 2.1: Summary of the command-line interfaces provided by various RJMSes

2.2 Job Scheduling

Once users submit their job, it is the responsibility of the scheduler to select and allocate resources on the cluster for the job. HPC schedulers select these resources to optimize for objectives like resource utilization and scientific goodput. Typical metrics used for evaluating schedulers are discussed further in the next section, Section 2.2.1. These scheduling objectives are not the only thing a scheduler must consider. Schedulers must also consider the constraints put in place by system administrators. For example, unlike in cloud computing where multiple applications from different users all run on the same resources, HPC systems traditionally provide jobs with exclusive access to resources. Given this exclusive access constraint, there are many times when an insufficient number of resources are available to execute a newly submitted job; in this scenario, the scheduler must add the job to a queue and execute it later, once enough resources have become idle. The algorithm used to determine how to process the queued jobs is called the *scheduling policy*, and the most common scheduling policies employed are discussed in more detail in Section 2.2.2.

Once a scheduler has determined which job to run and what resources to allocate to the job, the scheduler passes the job allocation information on to the resource manager for execution. The resource manager will notify the scheduler when the job completes as well as if any resources go on or offline. The scheduler uses these notifications to update its internal model of the system. An accurate internal model ensures that resources are quickly reallocated after a job completes and offline resources are not allocated to new jobs. In addition to an internal model of the current system state, many schedulers will keep databases to store historical job and resource information. This database is useful for administration, auditing, and enforcing fair usage. Users and administrators can query this database through several of the CLIs listed in Table 2.1.

2.2.1 Scheduling Metrics

There are three main metrics that HPC schedulers are evaluated and optimized for: resource utilization, goodput, and job throughput.

Resource Utilization is one of the most important metrics from an administrative perspective. After the significant financial investment required to procure a cluster, obtaining a return on investment through high utilization is essential. Given the compute-centric nature of schedulers and HPC in general, resource utilization is normally measured as the average percentage of compute nodes allocated over a given period of time. It is the scheduler's responsibility to ensure that resource utilization remains as close to 100% as possible for as long as possible. Unfortunately, in some cases, a trade-off exists between resource utilization and other important metrics, like scientific goodput.

Goodput is a metric typically used in computer networks, but it applies well to scheduling too. In computer networks, goodput is the rate at which *useful* data crosses a link, excluding protocol overhead and retransmissions. In scheduling, goodput is the amount of *useful* computation that is performed at any point in time, excluding time spent blocking on network and disk I/O. In networks, goodput decreases when a link becomes congested and retransmissions are common. Similarly, in scheduling, goodput decreases when shared resources on the system are overallocated, and jobs contend with one another for access to the overallocated resource. From this, it is easy to see how there is a trade-off between resource utilization and goodput. To achieve high utilization, the scheduler may need to run many resource-intensive jobs simultaneously, decreasing goodput. Inversely, to improve goodput, the scheduler may need to delay some resource-intensive jobs to prevent overallocation of shared resources. In Chapter 3, we discuss the specific case of increasing goodput by reducing contention on a parallel filesystem.

Job Throughput is the rate, in jobs per second, that a scheduler can ingest, schedule, and launch jobs on a system. In the context of workloads with many jobs (e.g., ensemble simulations), job throughput is an important metric. Schedulers with

a job throughput significantly lower than the job submission rate of a workload will become the bottleneck for that workload. With the growing emergence of ensemble simulation consisting of millions of jobs, it is important for HPC schedulers to support these workloads with increased job throughput. In Chapter 5, we discuss a model for improving the throughput of HPC schedulers.

2.2.2 Scheduling Policies

There are three sub-policies to any complete scheduling policy: an ordering policy, a resource selection policy, and a reservation policy.

Ordering Policy: this policy prioritizes jobs in the queue based on data such as submission time, user priority, historical usage, and the resources requested. Examples of ordering policies include First-Come-First-Served (FCFS), largest first, and fair share. The FCFS policy orders jobs based on their submission time, such that the oldest jobs are first in the queue. The largest first policy orders jobs based on the size of the resources requested, with the largest jobs at the front of the queue. The fair share policy prioritizes jobs based on a multitude of weighted factors that are unique to each computing site. The idea behind fair share scheduling is to maximize the appearance of fairness, which will differ across computing sites. As an example, at Lawrence Livermore National Laboratory (LLNL), a job's priority is based on the percentage of system time promised to a user and their historical usage (i.e., users that exceed their percentage of time are given a lower priority) [17]. Sometimes these policies are combined; for example, on select clusters at Oak Ridge National Laboratory (ORNL) and LLNL, fair share is merged with largest first by providing a priority increase to larger jobs [50, 17].

Resource Selection Policy: this sub-policy determines if enough resources exist to satisfy a job's requirements, and then selects the exact resources to allocate to or reserve for the job. The simplest resource selection policy only considers the number of compute nodes required by the job and selects the nodes to allocate arbitrarily. More complex, network-aware selection policies optimize locality by selecting nodes based

on a model of the network topology [62, 59]. In Chapter 3, we discuss an I/O-aware resource selection policy that eliminates contention on the parallel filesystem.

Reservation Policy: this policy combines the ordering and resource selection policies to determine whether to allocate resources now or to reserve resources in the future for a given job. The reservation policy considers queued jobs in the order provided by the ordering policy. For each job in the queue, the reservation policy uses the resource selection policy to determine if an allocation or reservation is necessary. If the resource selection policy determines that enough resources are available now, the reservation policy will allocate the resources immediately, otherwise, the reservation policy will create a reservation in the future. Example reservation policies are first-come, first-served (FCFS), conservative backfilling, and EASY backfilling. In the FCFS scheme, scheduling stops at the first job for which resources are currently not available (i.e., only allocations are made, no reservations are made). While FCFS perfectly preserves job order and fairness, it does so at great expense to resource utilization. A large job at the front of the queue will force most of the system to go idle before the job is scheduled. This is commonly known as head-of-line-blocking and can be avoided with backfilling. When backfilling is enabled, resources are tentatively reserved for jobs that cannot be currently executed, and jobs later in the queue will be scheduled if they do not interfere with any of the reservations. In conservative backfilling, a reservation is made for every job, in the order that they appear in the queue. In other words, a job cannot be backfilled if it would delay any of the jobs earlier in the queue. This greatly increases resource utilization with no impact on fairness. In EASY backfilling, a reservation is made only for the first job that needs it (i.e., the first job in the queue). In other words, jobs can be backfilled so long as they do not delay the first job in the queue. This increases resource utilization further at the cost of decreased fairness. Hybrids exist between conservative backfilling and EASY backfilling, where reservations are made for a finite number of jobs greater than one.

2.3 Resource Management

The final piece to a complete RJMS is the resource manager (RM). The RM is responsible for executing the jobs and monitoring the resources of the cluster. When the RM receives job allocation information from the scheduler, it begins setting up the job by running a job prologue, which is a script written by the system administrators to perform custom actions to prepare the allocation. Next, the RM begins running the user's job script on one node within the allocation. If within that jobs script, the user launches a parallel application, the RM helps bootstrap and launch the parallel application through interfaces like PMI [16]. As part of the bootstrapping process, the RM is responsible for mapping the various process of the parallel application to the physical resource within the job allocation and enabling inter-process communication across nodes. As part of the launching process, the RM is responsible for forwarding stdout/stderr/stdin and propagating signals (e.g., SIGINT). If a job exceeds its time limit, the RM is responsible for killing the job and all of its processes and then notifying the scheduler of the job's completion. Once a job completes (i.e., all processes have exited), the RM then runs a job epilogue to prepare the resources for the next allocation.

Chapter 3

I/O-AWARE SCHEDULING

In this chapter, we integrate the knowledge of the I/O subsystem and I/O-aware algorithms directly into Flux [15]. Using a Flux emulator, we explore the effectiveness and costs of I/O-aware scheduling.

The rest of this chapter is organized as follows. Section 3.1 explains the need for I/O awareness and the aspects of novelty in our work. Section 3.2 presents our I/O subsystem model and I/O-aware scheduling algorithms. Section 3.3 describes the Flux emulator along with the system and workload models used in our tests. Section 3.4 presents the results of our tests and demonstrates the benefits of I/O-aware scheduling.

3.1 Need for I/O Awareness and Aspects of Novelty

In recent years, a gap has grown between the amount of data an HPC system can produce and the amount of data the connected parallel file system (PFS) can absorb. Next-generation systems will deliver 7 to 10 times higher peak floating-point performance with only 1 to 2 times higher PFS bandwidth compared to previous generation systems [65, 86]. 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 storage hierarchy to fill in the bandwidth gap, and recent high-end system procurements have attested to this prediction [1, 2, 73, 97, 74]. Initially, burst buffer enabled systems [2, 1] will still require a reasonably high bandwidth PFS to ensure expedient draining of jobs' last checkpoints from the burst buffers to the PFS. The high bandwidth PFS 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 [68]. With the support of BB and smart staging, the PFS and the I/O links no longer need to be provisioned to handle the I/O-dominant phases of HPC applications. Instead, the I/O links and the PFS will be provisioned to handle 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, while BBs do add a high bandwidth layer to the storage hierarchy, they do not entirely solve the I/O bandwidth problem; the data in the BB must eventually be drained to the PFS. With a significantly underprovisioned PFS, it will be crucial to avoid scheduling any combination of jobs that can create I/O contention at the PFS.

The first efforts in managing I/O contention have extended existing scheduling policies by using heuristics in application-specific domains. For example, the extended FCFS scheduling by Góes et al. integrates heuristics to deal with irregular I/O-intensive jobs [41]. The heuristics search for I/O parameter values among the parameter ranges using testing. The work builds upon and extends previous work [21]. In this thesis, we move away from individual heuristics and applications.

More recent efforts on runtime scheduling of I/O contention include [101, 38, 32, 108]. Specifically, work by Thapaliya et al. addresses I/O contention with a PFS access controller: the controller provides a single application exclusive access to the PFS for a time window [101]. Work by Dorier et al. analyzes the I/O contention between two applications and investigates the use of runtime application coordination to avoid congestion [32]. Work by Gainaru et al. examines the impact of congestion on application I/O bandwidth and assesses a variety of runtime techniques designed to either maximize system efficiency or minimize application slowdown [38]. Finally, work by Zhou et al. 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 [108]. 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.

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, IBM defines network-aware scheduling within their production resource manager and scheduler [56]. 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. Other efforts have looked into simultaneously managing these diverse resource types. Specifically, work by Khoo et al. 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) [63].

3.2 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 explain our methodology.

3.2.1 Constant Job-Lifetime I/O Bandwidth

Researchers have long observed that the typical 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 based on 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 simplify this challenge: they turn the bursty I/O requests into a *constant* I/O stream. The BB layer absorbs application I/O bursts and the application

I/O requests are then drained in the background into the parallel file system (PFS) from the BBs. While a BB simplifies the problem, it is not sufficient to solve all I/O contention on an HPC system. I/O contention can still occur if multiple I/O-intensive applications whose combined average bandwidth exceeds the capabilities of the PFS are all run simultaneously. Thus, we leverage the BB write-behind scheme to enable the design and implementation of practical I/O-aware scheduling. Our I/O-aware batch job schedulers 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, it is possible to predict, within 16MB, the I/O produced over the lifetime of a job, 80% of the time [77]. In Chapter 4, we take a more in-depth look at using the I/O predictions of machine learning models as the job bandwidth requirement.

As is common in many burst buffer-enabled systems, we assume that the BBs are placed at the compute nodes (CNs), thus minimizing the probability of I/O contention occurring on resources in-between the CNs and the BBs [83]. 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 chapter, when we refer to I/O contention between the nodes and the PFS, we mean the CNs+BBs and the PFS.

3.2.2 Global View

The most straightforward system configuration for I/O-aware scheduling is one where each cluster has a dedicated parallel filesystem. In this configuration, each cluster can have its own scheduler which controls and manages all sources of I/O on the cluster's PFS. While this configuration makes managing I/O easier, it creates issues for users, who must manually transfer data between filesystems before they can change clusters.

The more common system configuration is one where a PFS is mounted on multiple clusters [99, 66]. In this one-to-many configuration, if each cluster has its own scheduler, then no single scheduler controls and manages all the sources of I/O on the

PFS. Many jobs from multiple scheduler domains can access the parallel filesystem concurrently, creating load on the filesystem that the scheduler has not accounted for, precluding the scheduler from managing bandwidth reliably.

To support configurations where a PFS is mounted on multiple clusters, effective I/O-aware methods also require a global view over all compute resources from which jobs can access the PFS. Thus, practical I/O-aware 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 [15] 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 need for scheduling beyond system boundaries and presents an opportunity to develop our methodology.

3.2.3 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] and Commodity Technology Systems (CTS). TLCC and CTS clusters are 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. Figure 3.1 shows an example TLCC cluster housed at LLNL, consisting of twelve SUs that are connected by three high-level InfiniBand switches. Each SU of this TLCC cluster is composed of 144 compute nodes - including six gateway nodes and two login

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

nodes - spread across eight racks. Each TLCC SU also contains twelve 24-port low-level switches which connect to both the compute nodes and the high-level switches. Newer cluster architectures have slightly different configurations. For example, TLCC2 clusters use 162 nodes per SU and 36-port low-level switches, and CTS-1 clusters use 192 nodes per SU and 48-port low-level switches [6].

The TLCCs used at LLNL mount one or more Lustre parallel filesystems (i.e., PFS) via the gateway nodes contained within each SU. The gateway nodes act as a bridge between the interconnect used within the TLCC and the interconnect used within the Lustre network (i.e., LNET). Each I/O request made by a compute node to the PFS is split into multiple packets which are spread round-robin across the gateway nodes. This gives each individual compute node full access to the bandwidth provided by the PFS and minimizes load imbalances at the gateway nodes.

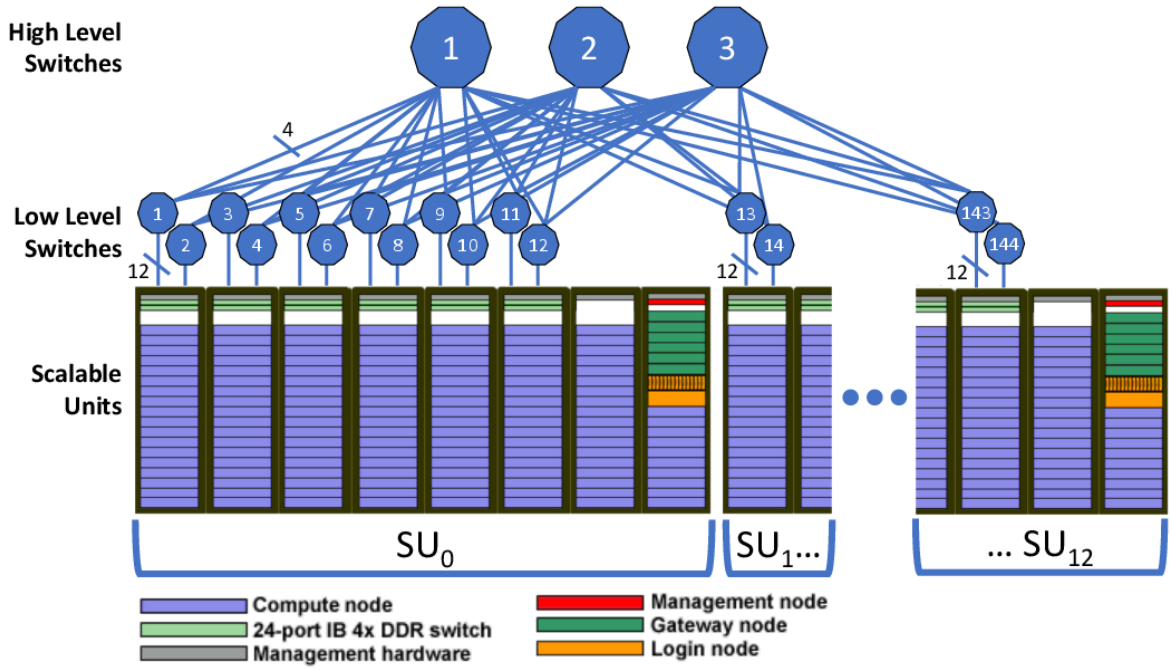


Figure 3.1: Example of a 12-SU TLCC cluster with high-level and low-level switches.

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 cluster's I/O subsystem in addition to its compute resources (e.g., compute nodes and cores). Flux provides a generalized resource model, known as the Resource Description Language (RDL), which allows the modeling of arbitrary resources and the hierarchical relationships between them [15]. Flux's RDL is implemented as a Lua library with a C interface, which provides interoperability with Flux's scheduler. Within Flux's RDL, each resource type is represented by a Lua class. To extend Flux to be I/O-aware, we define new Lua classes representing the resource types within our target I/O subsystem: gateway nodes, network switches, and parallel filesystems (PFS). Using these new resource types within Flux's RDL, we model the I/O subsystem as a hierarchy of network switches and gateway nodes leading to a parallel file system (PFS) and establish the hierarchical relationships between the resources. Each of these new resource classes contains an *I/O limit* attribute, which represents either the I/O bandwidth capacity of the resource itself (e.g., the PFS can read and write at a maximum of 1GB/s) or the bandwidth limit of the resource's connection to its parent (e.g., a compute node can communicate with its lowest level switch with a maximum bandwidth of 128MB/s).

Figure 3.2 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. 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 3.1, maintains a reasonable computational cost, and can easily be extended when multiple large clusters share a PFS. While in this work we focus on modeling a TLCC that is connected to a Lustre filesystem, modeling other configurations is possible. For example, a model of a cluster with a GPFS filesystem [98] that uses direct attached storage would not include the network switches of the cluster interconnect or gateways nodes. Instead, the model would include the resources within the storage area network. Alternatively, if the GPFS filesystem uses Network Shared

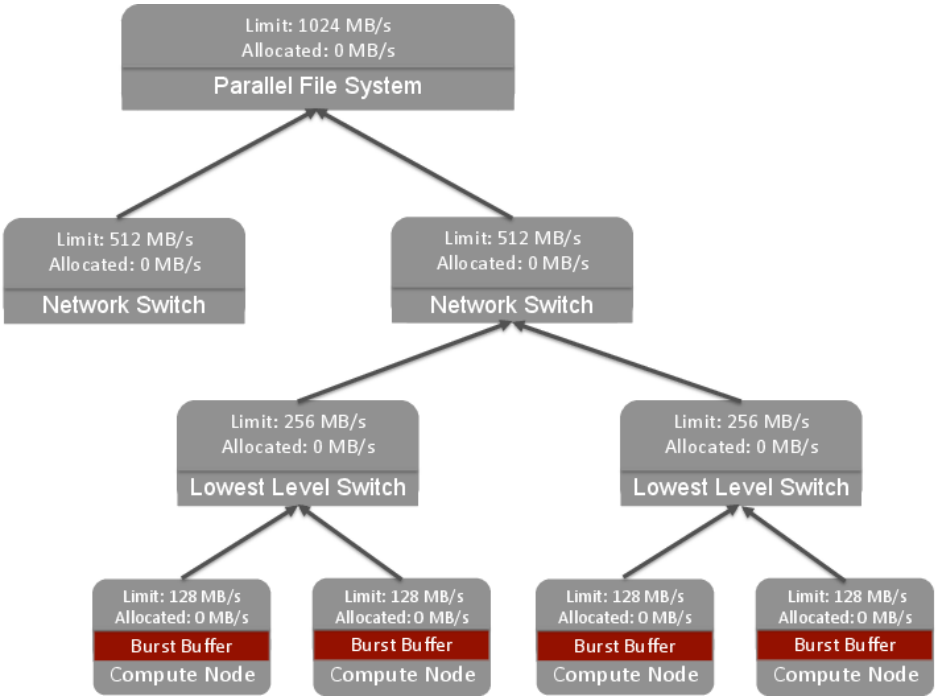


Figure 3.2: Example of an I/O-resource hierarchy in the RDL.

Disk servers [95], then the model would be the same as shown in Figure 3.1.

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 3.3.4. Figure 3.3 shows an example of I/O contention for the hierarchical resources considered in Figure 3.2 when two jobs with different BW requirements are executed. In Figure 3.3, 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 [73], Job1's overall I/O rate is limited to 128 MB/s, despite one of its processes having access to the full 192 MB/s. Job1 is over-utilizing bandwidth not only at the low-level switches but also at the higher level switches. If a new job, Job2 arrives and tries

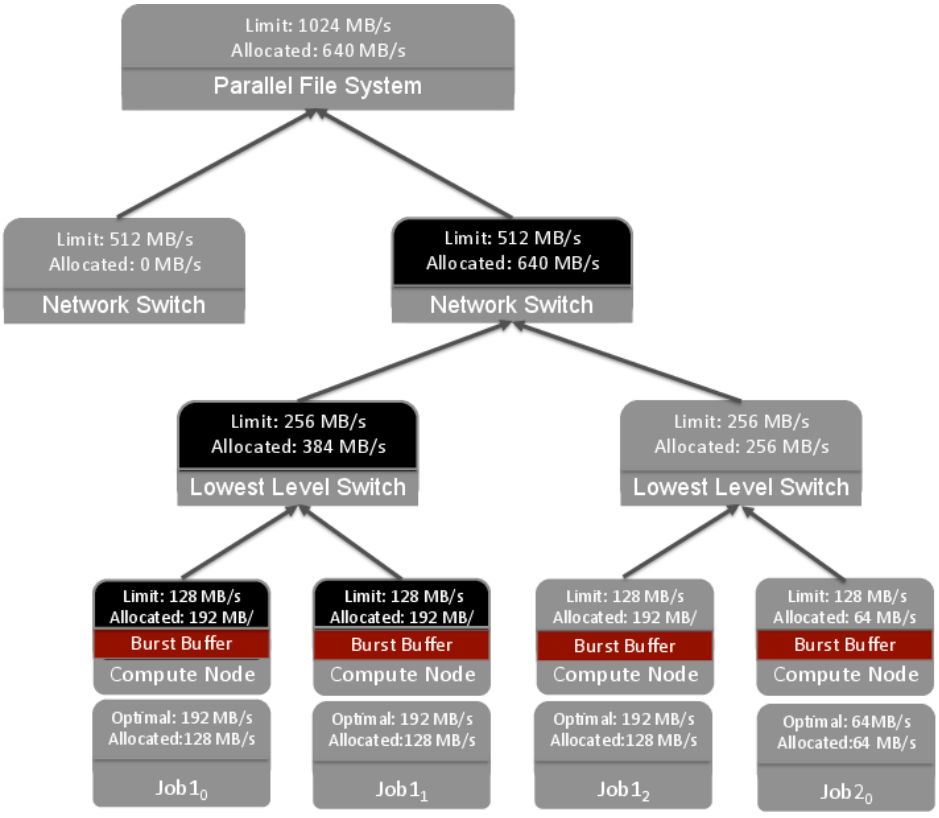


Figure 3.3: Example of I/O contention in action for our I/O-resource hierarchy.

to use additional bandwidth on the higher level switch, the job can further steal BW from Job1. To avoid I/O contention, the scheduler should place Job2 on a different part of the system; specifically, running Job2 under the other top-level network switch to avoid contention on the switches used by Job1. If there is nowhere else for Job2 to run without causing contention, the scheduler should delay the execution of Job2. Our tests inject jobs 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.

3.2.4 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 [71]. We select these policies because they represent algorithms with increasing complexity and expected efficiency

in real resource managers [25]. 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 with different job sorting (e.g., priority-based) is addressed in work by Klusáček et al. [64].

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 3.2.3 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-allocation 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 3.3 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 3.2 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 3.3 demonstrates the overallocation that occurs if Job 2 is scheduled. This does not mean that Job 2 cannot run; once Job 1 finishes, the scheduler will 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. Satisfiable jobs submitted to our I/O-aware scheduler are guaranteed to eventually execute assuming a starvation-free scheduling algorithm like FCFS or EASY backfilling is used.

3.3 Evaluation Environment and Models

This section describes our evaluation environment including our models of an expected large next-generation system and job workloads.

3.3.1 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 was built as part of the Commodity Technology System 1 procurement (CTS-1) [67]. The CTS-1 cluster model consists of 3,888 compute nodes (24 SUs); 216 GB/s per edge IB switch (i.e., 4× 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 3.2.3.

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 [67]. 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 3.4.1. Similar to the requirements stated in the CORAL RFP, we assume that the BB is large enough to store at least one full checkpoint (i.e., 64GB).

[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 [68].

3.3.2 Workload Model

At the time of this experimental design, the CTS-1 system has not yet been built. To generate job traces that are representative of workloads that will run on CTS-1, 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 to Moab. The net effect is that an individual database does not contain all the needed information, and thus we relate the two datasets. Because these datasets 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.* [31] 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.

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. Empirically, we observe that these four ranges represent periods of time where the user arrivals are mostly constant. This observation fits well with the well-known working patterns of the system users: more active during the day than at night and more active during weekdays than weekends. 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. Moab and SLURM both record these values, so either dataset 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 dataset 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 the unique tuples of the form (time limit, elapsed time) and counting the frequency at which they occur. From these frequencies, we calculate the probability density function for the time limit and elapsed time.

Finally, we build our I/O workloads by sampling values from the distributions described above. 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 3.3.1. 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.

3.3.3 Emulation Environment

To test our approach without having to launch real jobs, we developed a scheduling emulator within Flux. Figure 3.4a 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 3.4b. 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

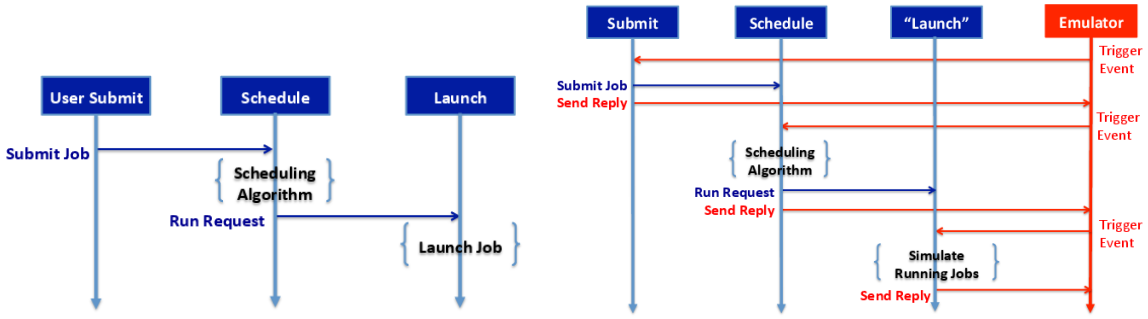
Flux as shown in Figure 3.4b. By using the emulator, we can rapidly study critical questions associated with the scheduler.

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.

3.3.4 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



(a) Real mode (b) Emulation mode

Figure 3.4: Flux in real vs. emulation mode.

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 requested 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 requested 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 does not effect 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 ($JobReqBW$) but receives an actual BW ($JobActualBW$). The actual job BW is the minimum of the actual BWs of the resources the job is running on. We use the interference factor (I) as defined by Dorier et al. [32] to determine the time an application spends performing computation or blocking on I/O, where the interference factor is defined as:

$$I = \frac{JobActualBW}{JobReqBW}$$

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. I represents the fraction of the time that a job spends in computation. For example, if a job is running on a contended system and thus has access to only three-quarters of the bandwidth required,

the job's interference factor (I) is 0.75. When a job's interference factor is 0.75, the job is spending only three-quarters of its time in computation, with the other quarter spent waiting on I/O. As the jobs running on the system change, the interference factor for each job also changes. For example, if an I/O-intensive job that is running on a contended system finishes, the contention in the system will decrease, allowing the other jobs to spend more time in computation. We discretize time into intervals (Δ_i), which represent the time during which the set of jobs running on the system remains unchanged. During these time intervals, since the set of running jobs remains constant, each job's interference factor for that period of time (I_i) also remains constant. The time a job spends performing computation over its lifetime is defined by:

$$T_{computing} = \sum_i (I_i \times \Delta_i)$$

where i ranges over the indices of valid time intervals (Δ_i) during which the job is running. The time a job spends blocking on I/O over its lifetime is defined by:

$$\begin{aligned} T_{IO} &= \sum_i (\Delta_i - (I_i \times \Delta_i)) \\ &= \left(\sum_i \Delta_i \right) - T_{computing} \\ &= T_{total} - T_{computing} \end{aligned}$$

where T_{total} is the total time the job spent on the system. Continuing our earlier example, a job with an $I = 0.75$ over a two hour time interval (Δ) has a $T_{computing} =$ equal to 1.5 hours and a T_{IO} equal to 0.5 hours. Thus, under our model, jobs under I/O contention ($I < 1$) perform fewer computations in the same period of time than jobs not suffering from contention ($I = 1$). For our tests in Section 3.4.1, we observe that the percentage of time that the interference factor is less than one is between 55% and 65% for underprovisioned PFSes with an I/O-ignorant scheduler; the percentage of time is 0% in the other cases.

3.4 Results

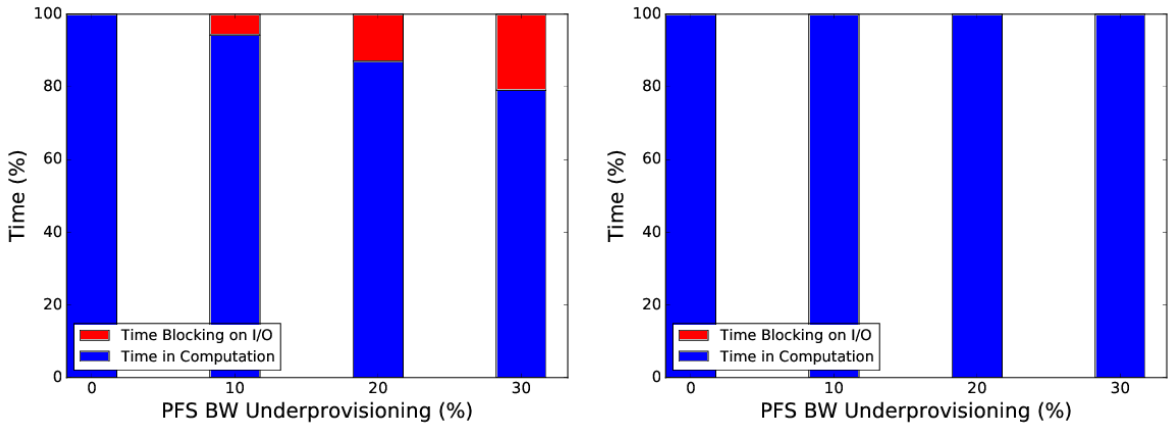
In this section, we provide empirical evidence that I/O-aware scheduling can increase the PFS efficiency and reduce job performance variability due to I/O contention.

3.4.1 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 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 3.3.2 and executed with the Flux emulator using the model of CTS-1 described in Section 3.3.1. 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.

3.4.2 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 3.5a refers to the set of tests in which the I/O-ignorant scheduler is used; Figure 3.5b 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).



(a) I/O-ignorant (b) I/O-aware

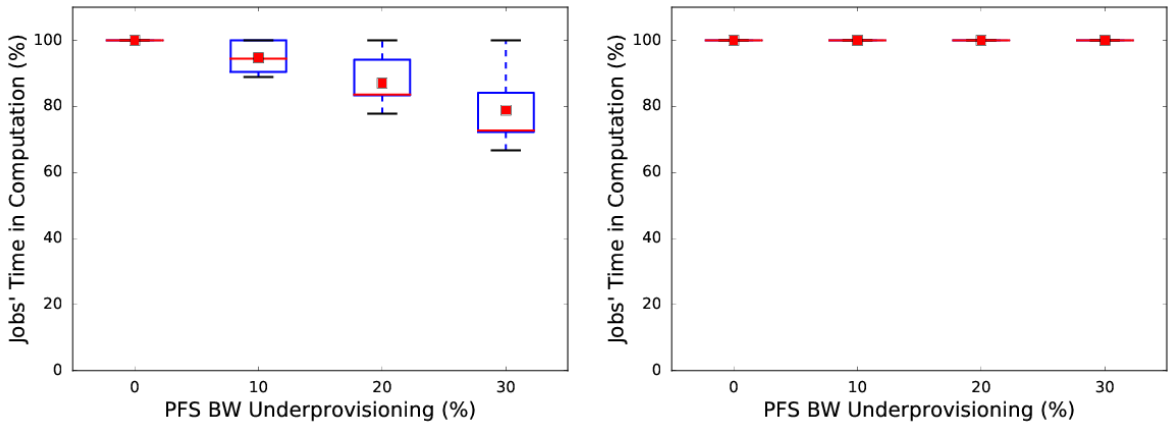
Figure 3.5: Percentage of total time spent by the entire CTS-1 cluster in computation and blocking on I/O.

As shown in Figure 3.5a, 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 because 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 of the nodes are blocking on I/O. On the other hand, Figure 3.5b 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 to prevent I/O contention when data-intensive jobs are simultaneously run on an underprovisioned PFS.

3.4.3 Impact on Individual Job Performance

To assess the impact of the I/O-aware scheduling on each 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 job doing computations versus blocking on I/O. In Figure 3.6, 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 3.6a, 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 3.6b 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 increasingly underprovisioned, individual job 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 resolves the variability in job performance due to I/O contention.



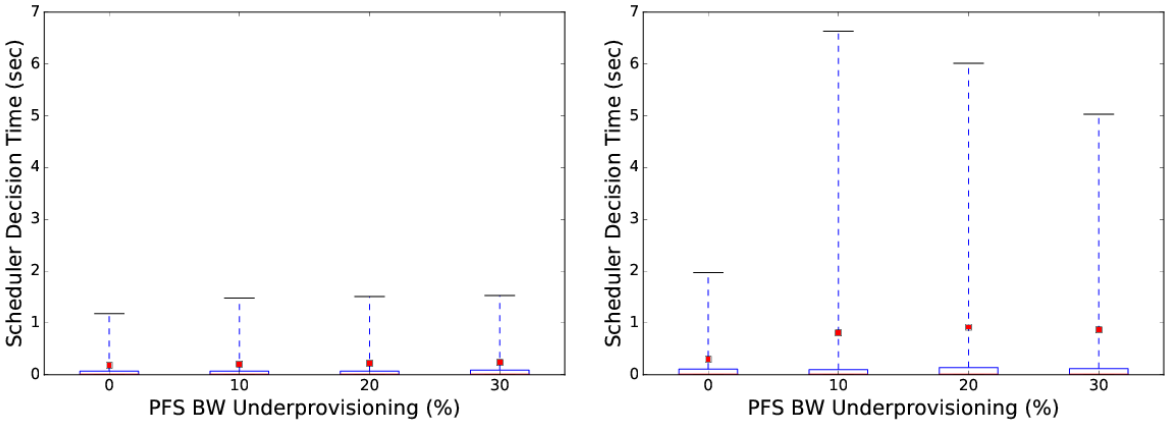
(a) I/O-ignorant (b) I/O-aware

Figure 3.6: Variability of individual jobs' time spent in computation.

3.4.4 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 3.3.2 and used for our tests, the average time between state changes is 31.7 seconds.

Figures 3.7a and 3.7b 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



(a) I/O-ignorant (b) I/O-aware

Figure 3.7: Scheduler decision time distributions.

invocation of the scheduler and are counted as a single sample time.

In Figures 3.7a and 3.7b, we represent the sampled times with boxplots. Traditionally a boxplot 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), the 25th percentile (bottom), and the 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 3.7a and 3.7b 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 [47].

3.4.5 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 potentially 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 3.5-3.7 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 3.3.4. A more significant 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 the 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.

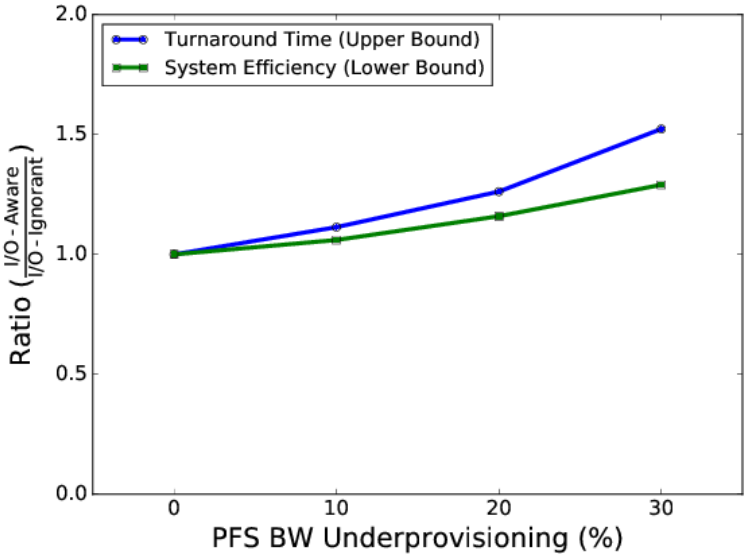


Figure 3.8: System efficiency versus job turnaround time.

In Figure 3.8, 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 (i.e., goodput) despite a longer turnaround time.

3.5 Summary

As the growth of disk capacity continues to outpace increases in disk bandwidth, parallel file systems (PFSes) are failing to meet bandwidth goals, leading to underprovisioned PFSes. With such underprovisioning, avoiding an I/O storm at the PFS level is critical to achieve computational efficiency and to preventing any disruption of the HPC center. We present a novel solution that allows us to meet these challenges on burst buffer-enabled systems 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 an I/O-aware scheduler, with perfect knowledge of every application's I/O requirements, 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 observe that our solution reduces job performance variability by up to 33% and increases system utilization by up to 21%.

Our results show that with perfect knowledge of job resource requirements, I/O-aware scheduling can handle the I/O challenges of next-generation HPC systems and ultimately increase the goodput of these systems. While the benefits of I/O-awareness are clear, it remains unknown whether the scheduler can have access to perfectly accurate knowledge of job resource requirements before the jobs' execution. What is clear is that we cannot delegate to users the specification of the job I/O requirements: user estimates can be inaccurate due to bad incentives from scheduler termination policies or because of poor understanding of job requirements. Before

I/O-aware scheduling can be deployed on real-world systems, we must first find and integrate a reliable source of job I/O knowledge into the scheduler.

Chapter 4

I/O KNOWLEDGE INTEGRATION

In this chapter, we incorporate the resource usage predictions made by a state-of-the-art resource prediction tool into our I/O-aware scheduler. Using the Flux emulator, we evaluate the scheduler's ability to anticipate both future system I/O usage and future I/O bursts.

The rest of this chapter is organized as follows. Section 4.1 explains the need for I/O knowledge integration and the aspects of novelty in our work. Section 4.2 describes our integration of runtime and I/O knowledge into our I/O-aware scheduler and how we use the knowledge to anticipate future system I/O usage. Section 4.3 describes the workload, resource knowledge, and metrics used in our tests. Section 4.4 presents the results of our tests.

4.1 Need for I/O Knowledge Integration and Aspects of Novelty

In Chapter 3, we showed how an I/O-aware scheduler with perfect knowledge can reduce job performance variability by up to 33% and increase system utilization by up to 21%. However, this assumption about perfect knowledge is unrealistic. In reality, a production I/O-aware scheduler will require a reliable source of knowledge for application I/O usage. Delegating the specification of resource usage to users is not a feasible solution. Previous analyses of job traces in HPC centers have shown how even the simple user-request runtimes are seldom accurate, achieving only 24% accuracy on average [78, 104]. Users are incentivized to overestimate job runtimes since current batch scheduler policies terminate jobs when they exceed the user-requested time [69]. Furthermore, when extending the types of resources to include I/O and other resources, users in scientific computing do not have an accurate understanding of job

requirements, and cannot be expected to learn how to estimate resource requirements accurately. Unable to rely on users as a source of knowledge, our I/O-aware scheduler requires a tool that can provide an accurate source of knowledge for application runtime and I/O usage.

Most of the previous work on integrating predictive tools with HPC schedulers has focused on directly predicting job queue times (i.e., the time between the submission and the start of execution of a job). Typically, these queue time predictions are then provided to users as hints for which system or which resource request size (e.g., number of nodes) will result in the shortest turnaround time (i.e., the time between a job's submission and completion). Downey developed a statistical model for predicting the queue time of a job based solely on the jobs already queued or running on an HPC system [33]. Nurmi, Brevik, and Wolksi developed a non-parametric method, QBETS, to provide bounds on wait times for jobs with quantifiable confidence levels [84]. QBETS uses clustering to find historical jobs that are similar to the current job and then calculates a bound on the job start time based on the related jobs. Nurmi et al. later extended QBETS to aid users in making probabilistic job reservations without cooperation from the target HPC system [85]. Murali and Vadhiyar created an adaptive framework, Qespera, that makes a queue time prediction based on both historical jobs and historical system states that are similar to the current job and the current state.

The remaining previous work on integrating predictive tools with HPC schedulers has focused on using runtime predictions to improve job backfilling. Tsafrir et al. modified a backfilling scheduler to integrate job runtimes predicted using a simple average of the past two previous jobs [102]. Their work still uses the user-provided walltime as the kill-time (i.e., the time after which the system kills the job) but uses the predicted runtime for everything else, including the backfill constraint checks. Gaussier et al. use the same underlying technique as Tsafrir et al., but they build a machine learning model specifically for the problem [40]. The more complex model is both more accurate and allows for tweaking the predictions to favor different types of jobs

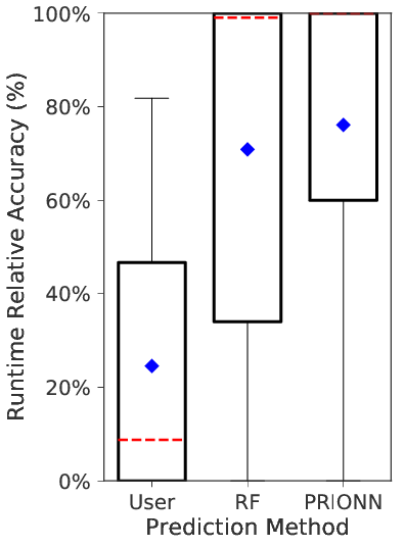


Figure 4.1: Boxplots of user, random forest, and PRIONN runtime prediction accuracies.

(e.g., short versus long-running) and different types of error (e.g., under- versus over-prediction). To the best of our knowledge, our work is the first method to leverage predictions other than runtime in a batch job scheduler.

4.2 Integration into I/O-Aware Scheduling

As described in Chapter 3, our I/O-aware scheduler relies on knowledge of job I/O behavior to schedule jobs such that I/O bandwidth contention is avoided. To this end, we use the resource prediction tool developed by Wyatt et al., Predicting Runtime and I/O using Neural Networks (PRIONN) [104], as the source of per-job runtime and I/O usage predictions for our I/O-aware scheduler. We chose to use PRIONN because it both outperforms existing methods in runtime prediction, as seen in Figure 4.1, and provides accurate I/O usage predictions, which no other existing techniques are capable of delivering. To mimic the evolution of a high-end HPC system, we use the simulator of the open-source, next-generation job scheduler Flux [46, 15].

Figure 4.2 shows the flow of data as we combine PRIONN's per-job runtime and I/O predictions with the Flux open-source resource management framework simulator and its I/O-aware scheduler to anticipate system I/O and I/O bursts. Figure 4.3

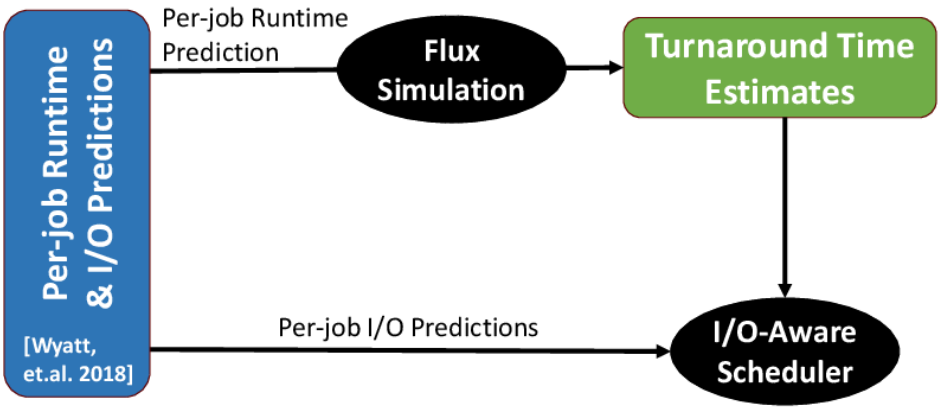


Figure 4.2: Overview of the application of runtime and I/O predictions to an I/O-aware scheduler to estimate system I/O and anticipate I/O bursts.

outlines the three main steps performed when the scheduler anticipates I/O bursts. In the zeroth step not shown explicitly in the figure, the scheduler creates an in-memory snapshot of the system, including currently running jobs and jobs in the queue. With this snapshot, the scheduler can now simulate the predicted future states of the system. In the first step, the scheduler queries PRIONN for the predicted runtime and I/O of each job in the snapshot. In the second step, the scheduler simulates the scheduling of all the jobs in the queue using the in-memory representation (i.e., no jobs are executed on the real system). The simulated start and stop times of the jobs are used as estimates for the jobs real start and stop times. In the third step, the scheduler sums the I/O of every job estimated to be running for each future point in time in the simulation. This I/O sum is used as the estimated overall system I/O. When this estimated system I/O exceeds the defined threshold for I/O bursts, the scheduler labels the time and amount of overall I/O as an anticipated I/O burst. With the timing and size of future I/O bursts, the I/O-aware scheduler now has the knowledge necessary to avoid I/O contention.

4.2.1 Estimating Turnaround Time

As Figure 4.2 shows, there are two data sources required to effectively estimate and manage future system I/O usage. The first required data source is accurate job

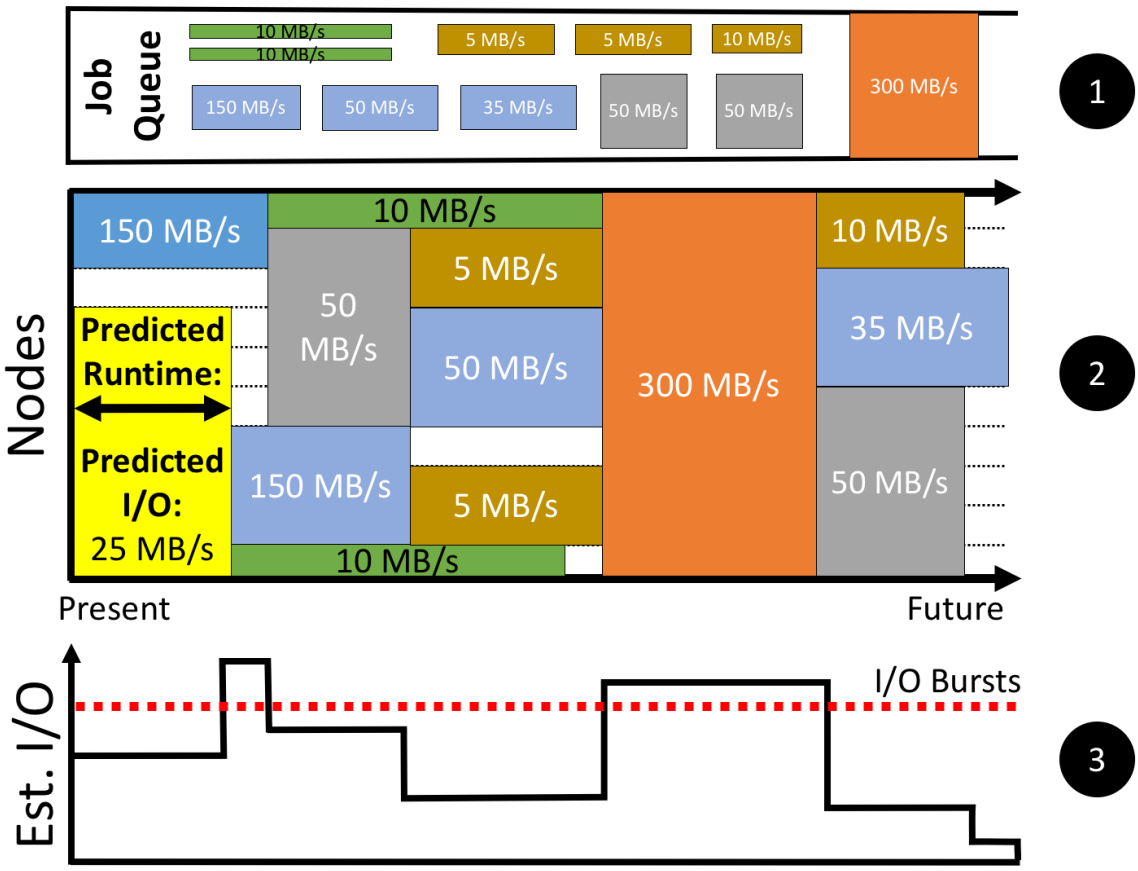


Figure 4.3: Three stages of how our I/O-aware scheduler uses PRIONN's runtime and I/O predictions to anticipate I/O bursts. 1) Query PRIONN for runtime and I/O predictions for each job in the queue. 2) Schedule jobs on an in-memory snapshot of the system to estimate when jobs will start and stop. 3) Sum the predicted I/O of each job run at a given time to estimate the overall system I/O and thus anticipate the high activity I/O bursts

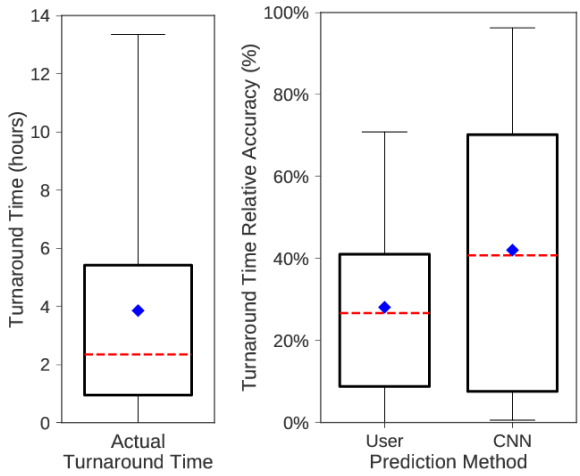
turnaround time estimates (i.e., the amount of time between when a job is first submitted to the scheduler and when the job completes). The turnaround time estimate for a given job depends on the predicted runtime of the jobs currently queued or executing. Therefore, inaccuracies in runtime predictions for individual jobs can accumulate into inaccurate turnaround time estimates. Relying on inaccurate runtime predictions, such as those based on user estimates, can result in very poor turnaround time estimates that are detrimental to an I/O-aware scheduler.

For our turnaround time estimates, we submit jobs to our simulated HPC system and record for each job both the simulated turnaround time and the estimated turnaround time. When a job is submitted to the system, a snapshot of the system is created. The snapshot creation is followed by four steps, depicted in step 2 of Figure 4.3. First, we copy the system state (i.e., allocated nodes, free nodes, simulated time, executing jobs, and queued jobs) in memory. Second, we replace the runtime of each queued and running job with the predicted job runtime. Third, we simulate the evolution of the system state from the snapshot until the submitted job has completed. Last, we record the difference between completion time and submission time of the job as our turnaround time estimate.

To quantify the turnaround time estimation accuracy, we sample five 10,000 job subsets from real job traces collected on the Cab cluster at Lawrence Livermore National Laboratory; the subsets were randomly selected across the span of the job traces. We run five simulations, one for each job subset, and estimate turnaround time as described above. Figure 4.4a shows the distribution of the simulated turnaround times. Figure 4.4b compares the resulting relative accuracy of turnaround time estimates when the simulated system uses user-requested runtime (left) and PRIONN's runtime (right). We observe that PRIONN improves the mean accuracy by 14.0% and the median accuracy by 14.1% over user-requested runtime. Our mean and median turnaround time accuracy are 42.1% and 40.8%. Additionally, we note that the 75th and 95th percentile accuracies are over 20% greater when using PRIONN's predictions compared to user-requested runtimes. This indicates that our scheduler is able to leverage the more accurate per-job runtime predictions from PRIONN to better model future system state and thus make better turnaround time estimates.

4.2.2 Estimating System I/O

The second data source required to effectively estimating and managing future I/O usage is system I/O estimates. To this end, we combine turnaround time estimates



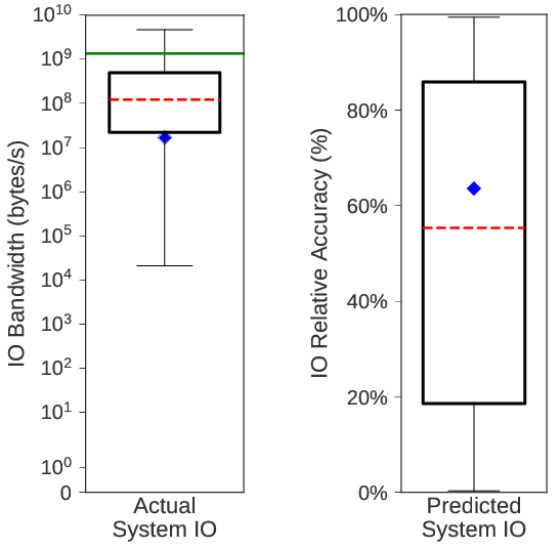
(a) (b)

Figure 4.4: Distribution of our simulated turnaround times (a) and relative accuracy of turnaround time estimates with user requested runtime and PRIONN's runtime (b).

from our simulated system with PRIONN's per-job I/O usage predictions (i.e., predicted read and write bandwidth), as shown in Figure 4.2. Specifically, to estimate the total system I/O in use at a given time, we first use the job turnaround time estimates to determine which jobs are running on the system at that time. Then, for the running jobs, we sum their predicted I/O usage, producing the estimated total system I/O.

4.3 Evaluation Configurations and Metrics

To quantify the accuracy of the estimated total system I/O, we perform two types of evaluations, each one using different sources of knowledge for runtime, I/O, and turnaround time. In the first evaluation, we use perfect knowledge of runtime and turnaround time (i.e., from real job traces) and PRIONN's predictions for per-job I/O usage. This evaluation isolates the accuracy of I/O predictions from the accuracy of runtime predictions and turnaround time estimates. In the second evaluation, we use the runtime and I/O usage predictions from PRIONN and the turnaround time estimates from our scheduler, as described in Section 4.2.1. This evaluation reflects how an I/O-aware scheduler can rely on runtime and I/O usage predictions in a real-world or production scenario.



(a) (b)

Figure 4.5: Actual aggregate I/O (a) and relative accuracy of the system's accumulate I/O estimates (b) using perfect turnaround time knowledge.

For each one of the two evaluations, we report two metrics. First, we report the relative accuracy of our estimated I/O behavior (i.e., system bandwidth over time). Second, we measure the precision and sensitivity for anticipating I/O bursts, or unusually high levels of I/O bandwidth, that occur in the system I/O behavior. I/O bursts are of particular importance to an I/O-aware scheduler because they are the most likely time for I/O contention to occur.

We define I/O bursts based on the actual system I/O bandwidth distribution shown in Figure 4.5a. The fact that the boxplot in the figure is symmetrical around the median suggests that the distribution of system I/O measurements forms a normal distribution on the logarithmic Y-axis. We fit a lognormal distribution to the actual system I/O and calculate the mean and standard deviation of this distribution. One standard deviation above the mean is marked with a green horizontal line at 1.35×10^9 bytes/s in Figure 4.5a. We define an I/O burst as any bandwidth measurement above this value.

When measuring the effectiveness of our I/O burst predictions, we look at each real I/O burst and determine if an equivalent I/O burst is also anticipated within

a given window of time. For example, with a three-minute window, we look for an anticipated burst one minute before the actual I/O burst, at the time of the real I/O burst, and one minute after the actual I/O burst. If a burst is anticipated in this window, we record a True Positive (TP). We record False Negatives (FN) (i.e., there is an I/O burst, but we do not anticipate an I/O burst) and False Positives (FP) (i.e., we anticipate an I/O burst when there is not an I/O burst) using this same window technique. We use these values (i.e., TP, FP, and FN) to calculate sensitivity and precision for our I/O burst anticipation. Sensitivity and precision have a range from 0% to 100%; larger values indicate better performance. Sensitivity is the ratio of correctly anticipated I/O bursts to actual I/O bursts (i.e., $\frac{TP}{TP+FN}$). Precision is the ratio of correctly anticipated I/O bursts to total anticipated I/O bursts (i.e., $\frac{TP}{TP+FP}$).

These metrics have direct connections back to the performance of our I/O-aware scheduler and the scheduler's effects on job and system performance. A False Negative (FN) I/O burst means that the scheduler is unaware of an occurring burst, resulting in I/O contention at the PFS. An abundance of FNs results in a schedule similar to the status-quo (i.e., a schedule which an I/O-ignorant scheduler would produce). Our first metric, sensitivity, is a ratio of the number of TPs and the number of FNs, and thus a high sensitivity means that our I/O-aware scheduler is aware of most of the occurring I/O bursts and can effectively avoid I/O contention. Alternatively, a False Positive (FP) I/O burst means the scheduler will delay I/O-intensive jobs in an attempt to reduce I/O utilization and avoid contention that would not occur anyway, resulting in an underutilized PFS. An abundance of FPs produces a schedule that excessively delays jobs predicted to be I/O-intensive. Our second metric, precision, is a ratio of the number of TPs and the number of FPs, and thus a high precision means that our I/O-aware scheduler can trust that anticipated I/O bursts will actually occur and thus will only delay jobs when absolutely necessary to avoid I/O contention.

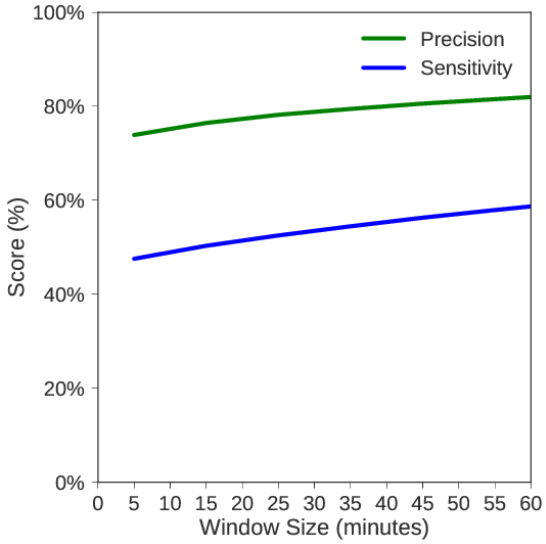
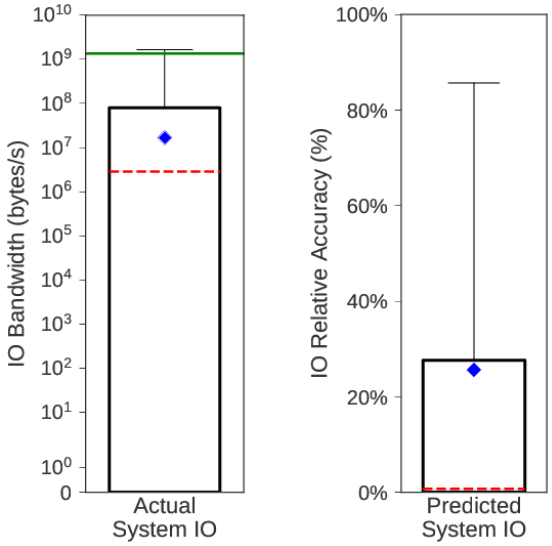


Figure 4.6: Sensitivity and precision of our I/O burst anticipation across windows ranging from 5 minutes to 60 minutes using perfect turnaround time knowledge.

4.4 Results

For our first evaluation of system I/O bandwidth estimates, we use perfect turnaround time knowledge for all jobs in our dataset and per-job I/O usage predictions. Figure 4.5b shows our first metric: the relative accuracy of each system I/O estimate. We achieve the mean and median accuracy of 63.6% and 55.3% respectively. Figure 4.6 shows our second metric: the sensitivity and precision of our I/O burst anticipation across windows ranging from 5 minutes to 60 minutes. We observe in the figure that we anticipate 47.5% of real I/O bursts within two minutes of their occurrence (i.e., sensitivity is 47.5% at a window size of 5 minutes). It also shows that a real IO burst accompanies 73.9% of anticipated IO bursts within two minutes of the burst being anticipated (i.e., precision is 73.9% at a window size of 5 minutes). Finally, we note that as the window size for anticipating I/O bursts increases, the sensitivity and precision also increase. From these results, we can see that when using the I/O predictions from PRIONN, our I/O-aware scheduler can accurately anticipate nearly 75% of I/O bursts.

For our second evaluation of system I/O bandwidth estimation, we use both



(a) (b)

Figure 4.7: Actual aggregate I/O (a) and relative accuracy of the system's accumulate I/O estimates (b) using our estimated turnaround time from our simulated system.

the estimated turnaround time and predicted per-job I/O usage for our five samples of 10,000 jobs. Figure 4.7a shows the distribution of simulated system I/O. We compare the distributions of system I/O for jobs used in our first evaluation, shown in Figure 4.5a, and jobs used in our second evaluation, shown in Figure 4.7a. We note that the distributions of system I/O are not identical, but have similar maximum, mean, and median values. Figure 4.7b shows our first metric: the relative accuracy of each system I/O estimate using estimated turnaround time and predicted per-job I/O usage. Comparing Figure 4.5b with Figure 4.7b shows that the system I/O estimate accuracy decreases when our turnaround time estimates are used in place of perfect turnaround time knowledge. This is an expected result of using less accurate turnaround time information. Despite the decreased average accuracy, we are still able to estimate several patterns in the I/O behavior in the simulation accurately, as indicated by the top whisker of the boxplot in Figure 4.7b.

Figure 4.8 shows our second metric: the sensitivity and precision of our I/O burst anticipation across windows ranging from 5 minutes to 60 minutes. We observe in Figure 4.8 that we anticipate 55.3% of real I/O bursts within two minutes of them

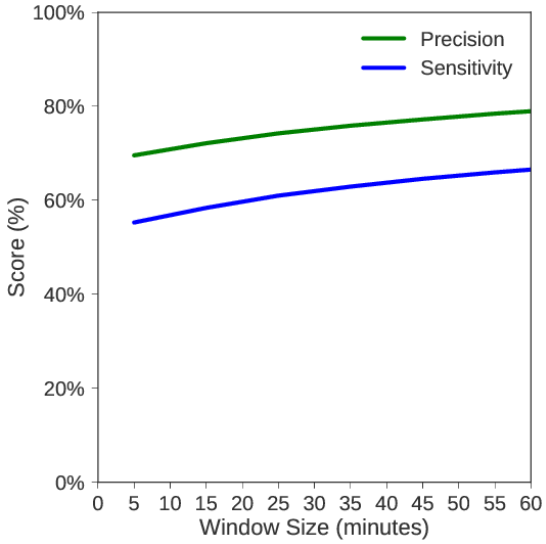


Figure 4.8: Sensitivity and precision of our I/O burst anticipation across windows ranging from 5 minutes to 60 minutes using our estimated turnaround time from our simulated system.

occurring (i.e., sensitivity is 55.3% at a window size of 5 minutes). It also shows that 70.0% of anticipated I/O bursts are accompanied by a real I/O burst within two minutes of it occurring (i.e., precision is 70.0% at a window size of 5 minutes). We compare sensitivity and precision of I/O burst anticipation in our first evaluation, shown in Figure 4.6, and our second evaluation, shown in Figure 4.8. We note that despite switching from perfect to estimated turnaround time, we achieve similar sensitivity and precision. Like Figure 4.6, we also observe that sensitivity and precision increase as window size increases in Figure 4.8. These results indicate that with the knowledge provided by PRIONN, our I/O-aware scheduler can correctly anticipate over 50% of I/O bursts. This is a massive improvement over being able to anticipate 0% of I/O bursts without the resource usage knowledge. Additionally, 70% of anticipated bursts actually occur, allowing the scheduler to trust the burst anticipations and schedule jobs accordingly without an excessive delay of I/O-intensive jobs.

4.5 Summary

I/O-aware scheduling represents a new, promising technique for mitigating contention on shared parallel filesystems and thus improving both individual job and overall system performance. A fundamental requirement of the I/O-aware scheduler that we present in Chapter 3 is access to perfectly accurate knowledge about job I/O requirements before the jobs run. Since perfect I/O knowledge does not exist, we must use alternatives such as users or predictive tools. Users have historically provided inaccurate resource requirements, and so we must look towards resource prediction tools like PRIONN which use state-of-the-art machine learning techniques to predict job resource requirements. We show that with perfect turnaround time information and PRIONN's job I/O requirement predictions, future system I/O can be estimated with $> 55\%$ accuracy. Additionally, when using our scheduler's turnaround time estimates and PRIONN's job I/O requirement predictions, we correctly anticipate $> 50\%$ of future I/O bursts. These results indicate that we can remove the assumption of perfect knowledge from our I/O-aware scheduler without involving system users. Instead, we can leverage runtime and I/O predictions from resource prediction tools, like PRIONN, to generate accurate estimates of HPC system I/O bandwidth and anticipate I/O bursts.

Our results show that our proposed I/O-aware scheduler mitigates the performance bottlenecks associated with I/O resources, which are playing an increasingly important role on next-generation systems. What remains to be assessed is whether an I/O-aware scheduler can remain effective during the shift in HPC workloads towards predominantly ensemble simulations. By dealing with ensemble jobs on an individual basis without regard to any similarities between jobs, centralized schedulers like our I/O-aware scheduler become a bottleneck, ultimately negatively impacting ensemble scalability. To scalably process these ensembles in an I/O-aware fashion, we need to switch to a new scheduling model that supports parallelism while retaining structured, hierarchical control over resources.

Chapter 5

HIERARCHICAL SCHEDULING

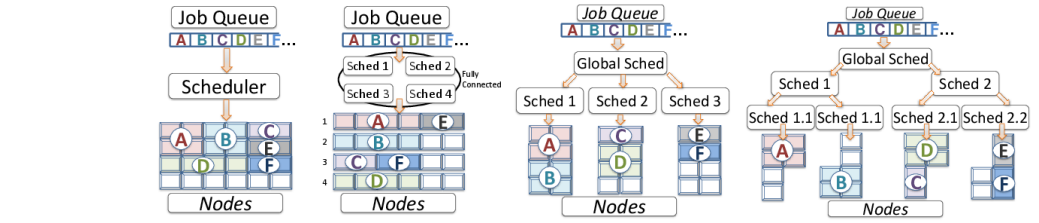
In this chapter, we leverage the fully hierarchical model to build a scalable hierarchy of seamlessly integrated schedulers. We develop a prototype based on Flux, an open-source, next-generation resource management framework. We evaluate our prototype and compare it against the centralized and limited hierarchical scheduling models.

The rest of the chapter is organized as follows. Section 5.1 provides evidence for the limitations of existing HPC scheduling models and presents aspects of novelty in our work. Section 5.2 introduces the fully hierarchical model with its defining principles and properties. Section 5.3 describes the details of our implementation. Section 5.4 presents our evaluation on synthetic and real-world workloads.

5.1 Need for Hierarchical Scheduling and Aspects of Novelty

HPC employs three distinct approaches for batch-job scheduling: centralized, decentralized and (partially or limited) hierarchical models [54, 88].

Each one of the three models is born out of necessity regarding the features, the number of resources managed, the number of jobs managed, and the domain in which resources are located. Centralized scheduling is the *de facto* state of the practice in HPC, in particular for managing a single cluster. Decentralized scheduling is the state of the art in theoretical and academic efforts, and limited hierarchical scheduling has emerged predominantly in grid and cloud computing, although some large HPC centers are using it for better scalability. Figure 5.1 depicts the three scheduling models



	Centralized	Decentralized	Limited Hierarchical	Fully Hierarchical
Scalability	Low	High	Medium	High
Workload Specialization	Low	Medium	Medium	High
Multi-Level Constraints	High	Low	Medium	High

Figure 5.1: The trade-off space of scheduling models

based on their theoretical scalability, workload specialization, and support for multi-level constraints. Figure 5.1 also compares the three existing models with the fully hierarchical scheduling model.

5.1.1 Centralized Model

The centralized model uses a single, global scheduler that maintains and tracks the full knowledge of jobs and resources to make scheduling decisions. Centralized schedulers can manage a variety of resources and enforce a uniform set of policies. The model is simple and effective for moderate-size clusters, making it the *de facto* state of the practice in most cloud and HPC centers today. Cloud schedulers such as Swarm [8] and Kubernetes [42] and HPC schedulers such as SLURM [106], MOAB [28], and LSF [57] use this model. In the example of centralized scheduling shown in Figure 5.1, jobs `A-F` all are allocated to nodes by the single centralized scheduler. While simple, the centralized model lacks the scalability and workload specialization necessary for today's HPC environments.

The Lawrence Livermore National Laboratory (LLNL) reported a real-world example that outlines the scalability problems of the centralized model. Their supercomputer Sequoia, which has 1.6 million cores, encountered several *scale-up* problems

in 2014 when users saturated the scheduler's components by trying to run an ensemble of 1,500 small uncertainty quantification (UQ) jobs (1-4 MPI tasks each). These problems included reaching the limits of sockets, user processes, and file descriptors as well as experiencing frequent timeouts and crashes in SLURM and IBM's control software. LLNL managed to expand the limits to about 20K simultaneously executing jobs after fine-tuning various configuration parameters within the operating system, SLURM, and IBM's control software. A temporary fix to scale even further was to create a limited scheduler hierarchy (described in detail in Section 5.1.3) by packing many small MPI jobs into a single large MPI job using a library, CRAM [44].

Clusters with many fewer cores than Sequoia can expose the scalability problems of centralized scheduling. Our two scaling tests, shown in Figure 5.2, demonstrate the problems at a much smaller scale. Figure 5.2a shows the results when we run variably sized synthetic ensembles on a 1,152-core (i.e., 32-node) cluster. Figure 5.2b shows the results of the second scaling test that we run for a synthetic ensemble of 131,072 jobs on variable numbers of cores. Each test uses single-core, `sleep 0` jobs that are submitted using repeated calls to SLURM's `sbatch` command. For each test, we measure both the makespan (i.e., the time difference between when the first job is submitted and when the last job finishes) and the number of job submissions that fail. Each test is run three times, and the figures show the minimum, maximum, and median of both the makespan times (i.e., the black line with y-axis on the left) and the number of submission failures (i.e., the red line with the y-axis on the right). Each node in the cluster is an Intel Xeon CPU E5-2695 node with 36 cores, hyper-threading enabled, and 128 GB of memory. The tests are run with SLURM 17.02.5, configured with the `select/cons_res` plug-in, the `CR_Core` plug-in argument, and used along with `munge` authentication. These and all other configuration values used are typical for SLURM-managed clusters. For each test, we run the SLURM control daemon `slurmctld` on a separate management node that is identical to the other compute nodes in the cluster. Each compute node in the cluster runs an instance of the SLURM compute node daemon `slurmd`. It is important to note that while SLURM has multiple compute node daemons running in parallel,

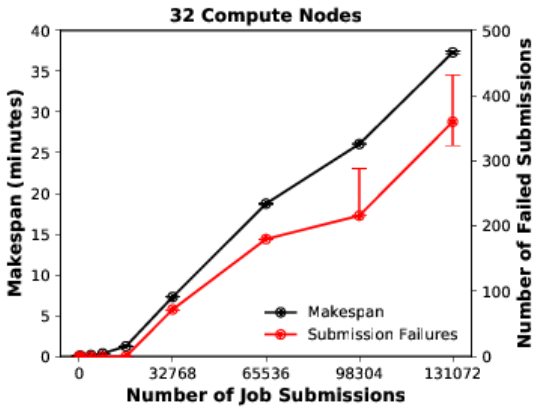
the critical function of scheduling jobs is performed sequentially by the single SLURM control daemon process.

In our first scaling test, which uses all 1,152 cores of our system, we progressively increase the number of jobs submitted to SLURM from a few jobs to 131K jobs. Figure 5.2a shows that the makespan increases with the increasing load on the scheduler's job queue (i.e., the number of job submissions). The linear makespan pattern, which is typical of the centralized model, is far from the ideal, constant pattern. Worse, as the number of submitted jobs increases, submission failures also increase: the first failures are already reported with 32k jobs. The following SLURM error is associated with the failures:

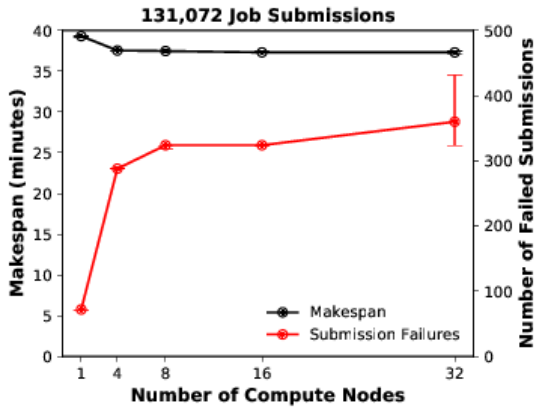
```
sbatch: error: Batch job submission failed:
Resource temporarily unavailable
```

This error suggests that problems occur when submitting a new job to the SLURM control daemon. The increase in failures can be addressed by re-engineering the application to use specific features of SLURM, for example, `job arrays` and `job steps`. However, these features have limited support for jobs with different sizes, time limits, and executables.

In our second scaling test, shown in Figure 5.2b, we fix the number of jobs at 131,072 and vary the number of cores in the cluster from 36 to 1,152. With a fixed number of jobs, we would expect the makespan to decrease as the number of cores increases, but the gains associated with making additional cores available to SLURM become insignificant already at 144 cores. Because we are running `sleep 0` jobs, the makespan times are entirely dominated by the sequential work of the scheduler ingesting and scheduling jobs. The constant makespan pattern is a consequence of the sequential work performed by the centralized scheduler and, once again, is far from the ideal makespan that halves with each doubling of compute cores. Additionally, we observe that as we increase the number of cores available to SLURM, the rate of submission failures also increases. The additional submission failures are caused by the



(a) Scalability test on 32 nodes with variable numbers of jobs



(b) Scalability test on 131,072 jobs with variable numbers of nodes

Figure 5.2: Example scaling tests with SLURM centralized scheduler (black lines show makespans, red lines show job submission failures; both increase with job and node counts)

overhead that managing more resources places on the scheduler. This test exhibits the same submission error messages as above.

Centralized schedulers also lack support for workload specialization. Currently, centralized schedulers can apply only one scheduling policy (e.g., fair-share) uniformly to all jobs. While alternative centralized schedulers might support workload specialization, the support would likely be ad hoc or convoluted. The co-existence of different workloads on the same platform, managed by the same scheduling policy results in degraded performance [96]. For example, when an ensemble workload of millions of jobs runs alongside traditional HPC workloads that consist of few, large jobs, the ensemble jobs starve the other applications and fairness is degraded. HPC facilities typically prevent this scenario with an ad hoc solution that caps the number of jobs that a single user can have in the queue [17]. At some HPC centers, the job cap is as low as 150 jobs per user [72]. The cap forces users to throttle their job submissions, which reduces ensemble workload throughput.

Centralized schedulers do support multi-level constraints. The single scheduler with a global view of all resources simplifies enforcement and guarantees of constraints

at all levels of an HPC system. Nonetheless, the centralized model entails a trade-off between multi-level constraints and scalability. For example, using a centralized scheduler to enforce IO bandwidth constraints at multiple levels in the IO subsystem can increase scheduling decision time by threefold, in the worst case [46].

5.1.2 Decentralized Model

Decentralized scheduling is the state-of-the-art in theoretical and academic efforts, but, contrary to centralized scheduling, it has not gained traction: to the best of our knowledge, this model is not in use in any production environment. Sparrow [87], in cloud computing, and SLURM++ [107], in HPC, are implemented with this model. In the decentralized model, multiple schedulers each manage a disjoint subset of jobs and resources. The schedulers are fully-connected and thus can communicate with every other scheduler. In this model, a scheduler communicates with other schedulers when performing work stealing and when allocating resources outside of its resource set (i.e., resources managed by another scheduler). The example in Figure 5.1 depicts a scenario with four schedulers (Sched 1, Sched 2, Sched 3, and Sched 4) handling six jobs (A, B, C, D, E, and F). Each scheduler is responsible for a set of resources, indicated in the figure by a row annotation. Despite providing higher levels of scalability, the decentralized model provides little to no support for workload specialization and multi-level constraints.

Decentralized schedulers do not have a central leader that is responsible for scheduling. Thus, they can scale to larger numbers of jobs and resources than centralized scheduling. Since the schedulers are fully connected, the decentralized model's communication grows superlinearly (i.e., $O(n^2)$) with the number of schedulers. Thus, despite the scalability improvement over the centralized model, the decentralized model's scalability is ultimately bounded by its communication. Existing studies in HPC environments have only demonstrated scalability up to 50 decentralized schedulers [107].

A homogeneous configuration of this model in which each scheduler executes the same scheduling policy does not support workload specialization. A heterogeneous

configuration could tune the scheduling policy of each scheduler for a specific workload type to support specialization, but no studies have demonstrated these configurations. Further, the number of schedulers would bound the number of different workload specializations. As previously stated, the number of schedulers cannot be arbitrarily large because of the model's superlinear communication scaling.

Finally, the decentralized model cannot scalably guarantee multi-level constraints. Since none of its schedulers has a global view of the resources, enforcement of strict policies and constraints that rely on knowledge at the global level, such as a strict system power bound, are particularly challenging for a decentralized scheduler. For global constraints to be enforced, each scheduler must constantly communicate with the other schedulers about their resource set, further bounding the number of schedulers supported by the model. While this model improves scalability over centralized scheduling, it provides limited mechanisms for workload specialization and enforcing multi-level constraints.

5.1.3 Limited Hierarchical Model

Limited hierarchical scheduling has emerged predominantly in grid and cloud computing. This model uses a fixed-depth scheduler hierarchy that typically consists of two levels. The scheduling levels consist of independent scheduling frameworks stacked together, relying on custom-made interfaces to make them interoperable. Figure 5.1 shows a high-level representation of this model, with a global scheduler (Global Sched) managing three local schedulers (Sched 1, Sched 2, and Sched 3). In the example, the jobs A and B are allocated by Sched 1; the jobs C and D are allocated by Sched 2; and the jobs E and F are allocated by Sched 3. Examples of this model's implementations include the cloud computing schedulers Mesos [49] and YARN [103] as well as the grid schedulers Globus [35] and Legion [26]. Efforts to achieve better scalability in HPC have resulted in this model's implementation in some large HPC centers. For example, at LLNL multiple clusters are managed by a limited hierarchical scheduler that uses the MOAB grid scheduler on top of several SLURM schedulers, each of which manages a

single cluster [17]. While this solution increases scalability over centralized scheduling, it provides only limited support for workload specialization and multi-level constraints.

The limited hierarchical model's finite hierarchy depth limits its scalability. Adding layers to increase the depth, while feasible, requires the creation of interfaces between each additional layer. These interfaces cause runtime performance overhead when translating job and resource information between layers. Additionally, engineering effort is required to create and maintain these interfaces [4].

The limited hierarchical model only supports workload specialization and multi-level constraints in specific circumstances. For workload specialization, the model requires a heterogeneous configuration in which lower-layer schedulers execute different scheduling policies. Such static specialization is only useful if it matches the workload requirements. The model can only support multi-level constraints in scenarios where all schedulers in the hierarchy support the constraints and no information is lost at the interfaces between layers. For example, if the root scheduler supports constraints on IO bandwidth, but the lower-layer schedulers, or the interface between the scheduler layers, do not include support for IO bandwidth, then the model cannot strictly guarantee the IO bandwidth in use across the hierarchy.

5.1.4 Impact on Emerging HPC Workloads

The scheduling issues that manifest due to the limitations of today's models have already led to a proliferation of ad hoc solutions on HPC systems. These solutions may be implemented by end-users or in special-purpose tools that manage the execution of ensembles of simulations, such as the Uncertainty Quantification Pipeline (UQP) [30].

Table 5.1 lists some common workarounds and their side effects for overcoming scheduling limitations. For example, many ensemble management systems create an ad hoc limited hierarchical scheduler (using manually developed tools or public libraries such as CRAM) to aggregate thousands of individual runs into a single job. This approach avoids the system job cap but can increase the total runtime as each aggregate is dependent on the longest running simulation [47]. A more critical issue may be

Schedulers' Limitations	Workaround	Side Effects
Cap on jobs in the queue	Throttle job submissions	Decreased job throughput
Limited job throughput	Aggregate jobs	Increased workload runtime [47]
Lack of ensemble status & control API	Track individual jobs' status through files	IO bottleneck [45]

Table 5.1: Ad hoc workarounds for scheduling limitations

the cost of creating and maintaining user-level solutions. Without a better model for system-level scheduling, the burden of implementing and maintaining m different solutions for n environments quickly becomes mxn effort, which is an inefficient use of human resources.

5.2 Fully Hierarchical Scheduling

We now introduce a new model for HPC batch job scheduling: the fully hierarchical scheduling model. Overcoming the challenges with scheduling diverse workloads on large HPC systems drives the design of the fully hierarchical model in which schedulers use a *common framework* and are arranged into *a hierarchy of seamless levels*. The common framework ensures that the schedulers share the same communication channel and resource model, creating a seamless interface between schedulers at different levels. Unlike interfaces used in the limited hierarchical model described in Section 5.1.3, our seamless interface eliminates possible incompatibilities between schedulers, allowing schedulers to communicate without loss of information and without the need for costly resource translation.

In the fully hierarchical model, schedulers that are higher in the hierarchy maintain a global, coarse-grained view of the jobs and resources. Child schedulers maintain a finer granularity view of their allocated jobs and resources than their parent. This parent-child scheduler relationship can recurse down an arbitrary number of levels. Three distinct principles define the fully hierarchical model:

- **Hierarchical Bounding Principle**: A parent scheduler grants the job and resource allocations to its children.

- **Instance Effectiveness Principle**: Each scheduler is solely responsible for the most effective use of its resources (that is, the resource set allocated to it) for its workload.
- **Arbitrary Recursion Principle**: The above principles can recurse from the entire HPC center down to an arbitrarily small subset of resources.

Batch job scheduling software that complies with these three principles is highly scalable and supports workload specialization and multi-level constraints. The scalability is ensured by the *Arbitrary Recursion Principle*, which enables unlimited levels within the scheduling hierarchy. A hierarchy with unlimited depth enables the parallelism necessary to mitigate the bottlenecks present in the centralized and limited hierarchical models and thus ensure the scalability required for large ensemble workloads. Since the schedulers are connected hierarchically, they avoid the superlinear communication scaling of the decentralized model.

Workload specialization is guaranteed by the *Instance Effectiveness Principle*, which empowers workload-specific child schedulers to execute specialized scheduling policies (such as high throughput computing (HTC)-oriented or IO-aware) within their various resource allocations. When combined with the *Arbitrary Recursion Principle*, the amount of specialization that can occur in a single hierarchy is unbounded.

Support for multi-level constraints is ensured by the *Hierarchical Bounding Principle*. For example, a parent scheduler enforces a power constraint by applying the *Hierarchical Bounding Principle* to bind workload-specific child schedulers to power allocations whose sum is less than the parent's power allocation. Since all schedulers in the hierarchy use a common framework, they avoid the burdensome interfaces and loss of information that plague the limited hierarchical model, as described in Section 5.1.3. Additionally, since the scheduler at the root of the hierarchy maintains a coarse-grained global view of the resources, the fully hierarchical model can efficiently enforce global constraints, unlike the decentralized model.

The fully hierarchical model can overcome all three of the limitations of existing schedulers, as listed in Table 5.1. The first limitation of existing schedulers is that they

require a cap on the number of jobs a user can submit to the queue at any one time [17]. This usually results in users or workload management systems throttling the rate of their job submissions. The fully hierarchical model overcomes this by distributing the jobs across the scheduler hierarchy and thus reducing the number of jobs in any single scheduler's queue. This allows users to submit more significant numbers of jobs at once without creating a bottleneck. The second limitation of existing schedulers is their limited job throughput, especially when managing large numbers of jobs and resources. This is often ameliorated by aggregating the many jobs into fewer, larger jobs. Under the fully hierarchical model, users will not have to rely on this ad hoc technique, as the cost of scheduling and launching the workload's jobs is spread across multiple schedulers, thereby increasing job throughput performance. This allows users to avoid aggregating jobs and instead submit their jobs at the finest granularity.

The third limitation of existing schedulers is their lack of ensemble control. Existing schedulers allow users to interact with ensembles on a job-by-job basis but not at coarser granularity. (e.g., job-group or the ensemble as whole). The fully hierarchical model allows users to query and control the ensemble at varying granularities by communicating with different schedulers within the hierarchy. The higher the scheduler in the hierarchy, the coarser the granularity. For example, to query the overall status of the ensemble, the user can send a single query to the root scheduler, as opposed to a query for every job in the ensemble.

5.3 From Model to Practical Strategies

The practical contribution of this chapter has three components: the definition of our implementation of the fully hierarchical model, the implementation's integration into an open-source resource management framework, and the model's application to two types of ensemble workloads (i.e., a synthetically built workload and a real-world uncertainty quantification workload).

5.3.1 Model Implementation

Our implementation consists of three main design points: the scheduler hierarchy, the resource assignment, and the job distribution. For the scheduler hierarchy, our implementation supports a hierarchy of schedulers with a fixed size and shape. Ensemble workload managers or users specify the exact hierarchy size and shape using JSON, which our implementation parses and uses to automatically launch the corresponding scheduler hierarchy. For the resource assignment, by default, our implementation assigns a static, uniform number of resources to schedulers at each level in the hierarchy (e.g., all of the leaf schedulers are allocated the same number of cores). Additional assignments of resources are possible, but as described later, require careful consideration when distributing jobs. For the job distribution, our implementation, by default, uses round-robin to distribute jobs uniformly across the scheduler hierarchy, but other distribution policies are supported and can be implemented by users. On homogeneous ensembles (i.e., each job is identical in resource requirements), a round-robin policy is very performant; it has a low overhead while also ensuring that work is balanced across the hierarchy. The details of how parent/child communication is implemented are included in Section 5.3.2. Each job in the ensemble is submitted individually at runtime to the root scheduler instance, and then, the jobs are distributed across the hierarchy. In this configuration, it may seem that the root instance will become a bottleneck, but the work required to map and send a job to a child scheduler is significantly less than the work required to schedule and launch a job. After a job is submitted, the root instance in the hierarchy must only consider tens to hundreds of children, while a centralized scheduler must consider thousands of cores as well as all other jobs in the queue. Additionally, the job distribution at the root instance can overlap with the scheduling and launching of jobs at the leaf instances.

Figure 5.3 shows an example of our implementation of the fully hierarchical model for a set of five jobs (Jobs A, B, C, D, and E) belonging to two different users ($user_0$ and $user_1$). For comparison, we also present the allocation of the jobs using a centralized scheduling model. The centralized scheduler has all five jobs in a single

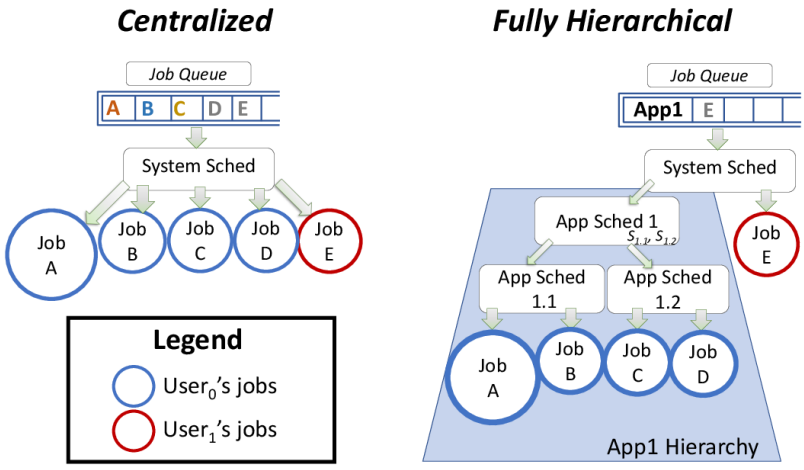


Figure 5.3: Fully hierarchical vs. centralized model implementations

queue, while the fully hierarchical scheduler distributes the jobs across all the schedulers' queues. It is important to note that this implementation represents only one strategy of many options available under the fully hierarchical model.

In Figure 5.3, the system scheduler sees Jobs A, B, C, and D belonging to $user_0$ as a single entity (App1) in its job queue; the sole job of $user_1$ is scheduled by itself by the same system scheduler. The system scheduler is now responsible for allocating resources for the hierarchy of $user_0$ by launching the top-level ensemble scheduler (App Sched 1). Within the hierarchy of $user_0$, App Sched 1 takes over the allocation of the jobs and is responsible for starting $user_0$'s predefined scheduler hierarchy by launching its children (that is, the two ensemble schedulers App Sched 2.1 and App Sched 2.2 in the figure). In our example, $user_0$'s hierarchy is limited to two levels with a branching factor of two, but our implementation does not limit the number of levels, the branching factor, or the number of schedulers that can be included in a hierarchy. Thus, it satisfies the *Arbitrary Recursion Principle*.

Given a distribution policy, the suitable number of schedulers in the hierarchy is driven by the number of cores available for execution as well as the number of jobs in the ensemble and their type (e.g., number of cores and time required). In the case of

the default distribution policy (i.e., round-robin) and a uniform assignment of resources across schedulers, the number of schedulers in the hierarchy is bounded by the number of cores required by the largest job. If the largest job requires m cores of the n cores available for execution, with $m < n$, then the number of schedulers in the hierarchy is bound to $\lfloor n/m \rfloor$. This is because with the default distribution policy any job in the ensemble can end up at any leaf scheduler, and thus every leaf scheduler must have enough cores to execute the largest job. We further explore suitable numbers of schedulers for two homogeneous ensembles in Sections 5.4.3 and 5.4.4. A more sophisticated resource assignment policy would take into account the resources required by each job, enabling specialization based on the workload being run. For example, some children could be given large allocations of resources to support traditional HPC jobs (i.e., large, long-running applications) while other children could be given small allocations of resources to support HTC jobs (i.e., small, short running applications). To avoid the situation where a scheduler is given too few resources to run some of the jobs it receives, the job distribution policy must be aware of the resources assigned to each scheduler and the resources required by each job. For example, a job distribution policy should not send a multi-node job to a scheduler assigned only a single node.

As mentioned previously, different job distributions policies may be preferable to the default round-robin distribution policy. For example, using the default round-robin distribution on a heterogeneous workload (i.e., each job has different resource and time requirements) will not be optimal. Under this policy, the largest jobs may all be distributed to the same child, creating a major load imbalance. A better distribution policy, for example, weighted round-robin, would take into account the requirements of each job to ensure that no one child gets saddled with the largest, most expensive jobs. Sometimes, even a more intelligent distribution policy is not sufficient. If the estimated time requirements of jobs (provided either by the user or ML models) are inaccurate, then a load imbalance amongst child schedulers will occur. In this scenario, a runtime re-balancing of either jobs or resources is required to ensure that the workload completes as quickly as possible and that resources do not go idle. We discuss in more

detail the dynamic exchange of resources in Section 6.2.3.

5.3.2 Integration into Flux

The fully hierarchical model represents a significant departure from existing batch job scheduling models. None of the existing production solutions are designed for use in a hierarchy, let alone an arbitrarily deep hierarchy. Thus, our approach is to prototype the fully hierarchical model by extending an open-source resource management software framework that supports configurable communication between schedulers, Flux [15].

The main foundation of the Flux framework is an overlay network powered by a message broker that supports various messaging idioms (such as publish-subscribe and remote procedure calls) and asynchronous event handling. Figure 5.4 shows an example of a two-level hierarchy, including a parent instance and a child instance, built using the Flux overlay network. In general, each Flux instance creates one process on each managed compute node. In the specific example in Figure 5.4, Sched 1 creates twelve processes, one on each managed node, and Scheds 1.1 and 1.2 create six processes each, again one on each managed node. This design not only allows remote execution of jobs on the managed nodes, but it also allows multiple jobs to be launched in parallel and without blocking the scheduler process.

Each of the Flux instances in the hierarchy has a set of dynamically loaded service modules that provide additional services such as a Key-Value Store (KVS), remote execution, and a scheduling framework. Flux's scheduling framework service module is, by default, loaded on the first process of each Flux instance and is designed to support scheduling policy plug-ins. This design enables a different policy plug-in to be loaded per scheduler instance in the hierarchy for scheduling specialization based on resource and workload types. Further, Flux also provides built-in plug-ins that implement known policies while allowing the instance owner to improve its performance with tunable scheduling parameters (such as limiting the number of jobs to consider per scheduling loop and delaying scheduling to process individual scheduling events

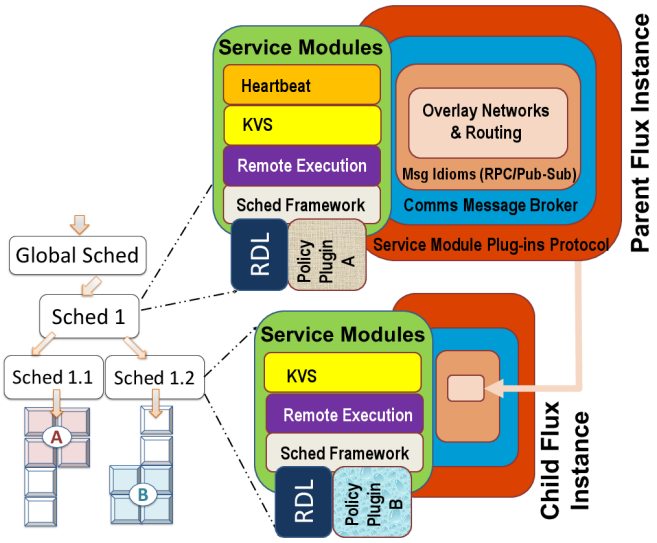


Figure 5.4: Flux framework managing two scheduler instances from our fully hierarchical implementation

in batches). Finally, the Flux resource description language (RDL) provides versatile support for defining resource types as well as their associations, enabling scheduling decisions based on many types of resources [15]. Other work has shown how Flux's RDL can be used to enforce IO bandwidth constraints on HPC systems [46].

The parent-child relationship of Flux instances must abide by the three main principles of the fully hierarchical model outlined in Section 5.2. To this end, we leverage four specific features of Flux: its communication overlay, its resource description language (RDL), its remote execution service, and its scheduling plug-in architecture. First, we use Flux's communication broker and overlay network to establish point-to-point communication channels between parent and child instances, which is necessary for routing job information, as described in Section 5.3.1. Second, we use Flux's RDL and remote execution service to ensure that the *Hierarchical Bounding Principle* is followed. When a parent instance spawns a child instance, it first allocates a subset of its resources and then sends a description of those resources, provided by Flux's RDL, to the remote execution service. The remote execution service parses the resource set,

again using the RDL, and then applies containment mechanisms (e.g., processor affinity, cgroups, network QOS, and IO bandwidth limits), before passing the description of the resources to the child. Third, we use the Flux scheduling service's plug-in architecture to help children follow the *Instance Effectiveness Principle*. Once the child receives the description of its allocated resources, the plug-in architecture enables the child scheduler to optimize its resource selection policies and scheduling decisions based on its workload and the resources given by its parent. For example, a child given a tight power budget by its parent can use a power-aware scheduling plug-in to ensure the best use of its most limited resource.

5.4 Results

Our model is designed to overcome the three major scheduling challenges facing next-generation schedulers: scalability, multi-level constraints, and workload specialization. Scalability is a continuous, quantitative property while workload specialization and multi-level constraints are more qualitative (i.e., the models either support the properties in some capacity or they do not the properties at all). In Section 5.2, we detail the principles that enable the fully hierarchical model to support these qualitative properties. To facilitate a true comparison of the fully hierarchical model against existing models which do not support these additional properties, we consider scalability in isolation from the other two properties. To this end, in this section, we quantify the fully hierarchical model's scheduling scalability; specifically, we tackle three key questions:

1. What is the impact of the fully hierarchical model on scheduling overhead and scalability, if any?
2. What is the suitable number of hierarchical levels in the scheduling hierarchy for differently sized resource allocations (that is, the total number of cores)?
3. What is the suitable number of hierarchical levels in the scheduling hierarchy for differently sized ensemble workloads (that is, the total number of jobs)?

5.4.1 Experimental Setup

All workloads' makespans (i.e., the time between the first job's launch and the last job's completion) include three components: job scheduling time; job launch time; and job execution time. The first two components represent scheduler performance while job execution time represents the cost inherent to the application. We address the above key questions through two analyses. One isolates the job scheduling and job launch costs while the other includes all three costs. The first analysis focuses on measuring only scheduling performance without any interference. We create a *stress test ensemble workload* in which each job immediately exits after it launches and so has zero runtime. Our second analysis focuses on measuring the scheduling performance on real-world ensemble workloads, in which we apply the fully hierarchical model to an uncertainty quantification (UQ) workload and consider all workload costs.

As described in Section 5.1, ensemble simulations present many challenges for existing scheduling models. To show that the fully hierarchical model succeeds where others fail, we generate a real-world ensemble workload with the Uncertainty Quantification Pipeline (UQP) [30]. The UQP, which runs ensembles of simulations and performs UQ analyses on the results, was used to run over 20,000 ensembles on HPC machines in 2017 [29]. Our UQ ensemble simulates a semi-analytical inertial confinement fusion (ICF) stagnation model that predicts the results of full ICF simulations, which is a major mission of the National Ignition Facility [37, 36, 5]. UQ ensembles with this semi-analytical model typically consist of tens of thousands of runs, but the scientists' goal is to execute millions of jobs.

The idling of resources due to jobs being submitted too slowly over time may adversely affect our scheduler makespan measurements. To avoid this overhead when evaluating the UQ workload, in which the UQP generates at runtime all of the files and directories necessary for the jobs' execution, we do not begin timing the workload makespan until all jobs have been generated. Only once the jobs are generated do we start to time the makespan and to distribute the jobs across the scheduler hierarchy. While this feature is useful for our evaluation, the fully hierarchical model and our

implementation do not preclude the submission of jobs over time or overlapping job submission and scheduling.

In our evaluation, we construct and compare three scheduler hierarchies with different depths: depth-1, depth-2, and depth-3 [1]. The depth-1 hierarchy only has a single scheduler instance in the workload's allocation that schedules every job in the queue, similar to the centralized scheduling model. For the depth-2 hierarchy, we create a root scheduler with one child scheduler for every node allocated to the workload, and we distribute the jobs equally among the lowest level of schedulers (i.e., the leaf schedulers). For the depth-3 hierarchy, we extend the hierarchy by adding one scheduler for every core allocated to the workload, and as with the previous hierarchy, we distribute the workload's jobs equally among the leaf schedulers.

Our evaluations on both workloads use a 728-node Intel Xeon E5-2695v4 cluster (with a system-imposed limit of 64 nodes per test), each node with 36 physical cores and 128 GB of memory. Our metrics of evaluation include job throughput, scheduler makespan, and submission failures. The latter metric is not shown in the paper because we did not encounter any submission failures in our evaluations of the hierarchical model built on Flux.

As part of a workload specialization investigation, we ran the stress test workload with all three hierarchy configurations (i.e., depth-1, depth-2, and depth-3) using two different scheduling policies: first-come, first-served (FCFS) and conservative backfilling. Regardless of the hierarchy configuration, we saw a consistent trend: FCFS achieves a higher job throughput and a lower scheduler overhead on the stress test workload. When running over one million jobs with the depth-3 hierarchy, FCFS can schedule jobs 41x faster than conservative backfilling, finishing the workload in under 35 minutes as opposed to 24 hours. Given this result, we continue using the FCFS scheduling policy rather than backfilling for the remaining tests in the paper.

---

[1] Our implementation supports additional levels. In our evaluation, we use a one-to-one mapping between hardware and scheduler levels. We are actively exploring mapping multiple scheduling levels to a single hardware level.

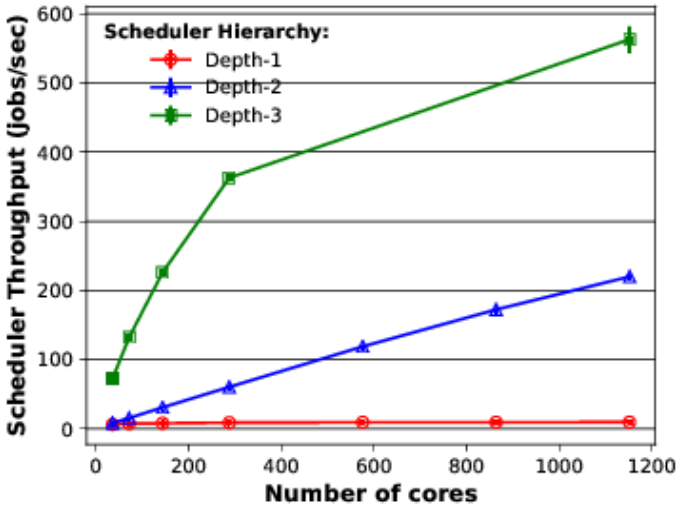


Figure 5.5: Scheduler throughput (jobs/sec) for the depth-1, depth-2, and depth-3 hierarchies scheduling our stress test workload

5.4.2 Quantifying Impact on Scheduler Scalability

To answer our first question about the impact of the fully hierarchical model on scheduler scalability, we measure the scheduler throughput (jobs/sec), which is a proxy for scheduler performance and scalability. We measure throughput as the number of jobs scheduled and launched per second (the higher, the better). We schedule workloads with each of the three hierarchies: depth-1, depth-2, and depth-3. We fix the number of jobs at each leaf scheduler in the hierarchy to 1,024. We vary the number of cores in the allocation for each type of scheduling model from 1 to 1,152 cores. Figure 5.5 shows the observed scheduler throughput on our stress test workload.

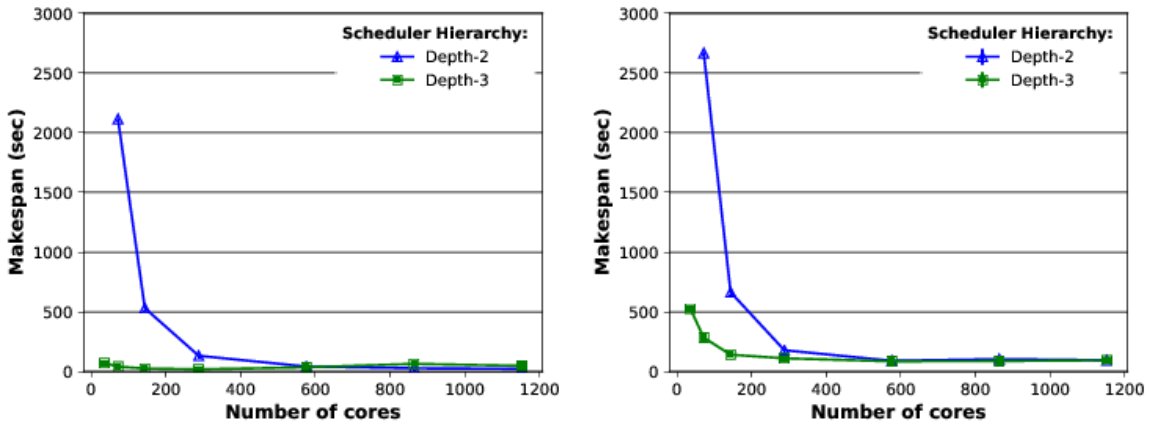
The figure shows that, when using the depth-3 hierarchy, we achieve an average scheduler throughput that is up to 59X greater than the depth-1 hierarchy on the cluster's 1,152 cores. With the depth-1 hierarchy, there is no scheduler parallelism because all 1,024 jobs go through a single scheduler. As we allocate more nodes to the workload, the single scheduler shows no increased scheduler throughput. This behavior highlights the fundamental limitation of single, centralized schedulers: their performance does not scale with the number of computational resources. This indicates

that using a single scheduler limits workload scalability, especially when using larger numbers of cores. The increase of scheduler parallelism in the depth-2 hierarchy results in a linear increase in throughput with respect to the number of cores with almost no variability. For the largest allocation, we reach up to 23X greater scheduler throughput over the depth-1 hierarchy. When we stretch the scheduler hierarchy down a level and use our depth-3 hierarchy on 1,152 cores, we achieve up to 59X greater throughput over the depth-1 hierarchy, as reported above, and up to 2.6X greater over the depth-2 hierarchy. When using our depth-3 hierarchy on only 36 cores, we also see a throughput improvement over the depth-1 and depth-2 hierarchies. The increased throughput is from the increased parallelism afforded by the depth-3 hierarchy. In the 36 core (i.e., one node) configuration, the depth-3 hierarchy (i.e., one scheduler per core) has 36 leaf schedulers while the depth-2 (i.e., one scheduler per node) and the depth-1 (i.e., one scheduler per workload) hierarchies only have one leaf scheduler. The results in Figure 5.5 shows that deeper hierarchies, with their additional scheduler parallelism, increase scheduler throughput and scalability on the stress test workload.

5.4.3 Impact on Resource Allocation Scaling

To answer our second question and find the suitable scheduling hierarchy for the targeted workloads with a different number of cores, we measure the makespan of both workloads (in seconds) on allocations up to 1,152 cores. In these tests, we use the same definition of makespan as defined in Section 5.4.1. The lowest makespan identifies the best scheduler hierarchy for a given core allocation. Figure 5.6 shows the results for the two scheduler hierarchies that completed successfully within 3 hours: depth-2 and depth-3. Figure 5.6a shows the results on the stress test workload, and Figure 5.6b shows the results on the UQ ensemble workload. In the tests, we fix the number of jobs per workload to 8,192 for both workloads. We vary the number of cores in the workload's allocation for each scheduling hierarchy up to 1,152 cores for both workloads.

Figure 5.6a outlines how the higher level of parallelism in the depth-3 hierarchy



(a) Stress test workload

(b) UQ workload

Figure 5.6: Workload makespan (in seconds) for the depth-2 and depth-3 scheduling hierarchies for a different number of cores allocated and 8,192 jobs. The depth-1 failed to schedule all jobs in under 3 hours, and thus is not included.

is beneficial for the performance until there are too few jobs per scheduler instance to saturate the schedulers. In this scenario, this happens at 576 cores. At higher node counts (i.e., > 576), the depth-2 hierarchy becomes faster due to its lower scheduler instance launching overhead and its better balance between scheduler parallelism and the number of jobs per instance. The results in Figure 5.6a indicates that, when keeping the total number of jobs constant, there is a threshold in terms of the number of cores in the allocation at which the cost of launching additional schedulers outweighs the benefit of additional parallelism.

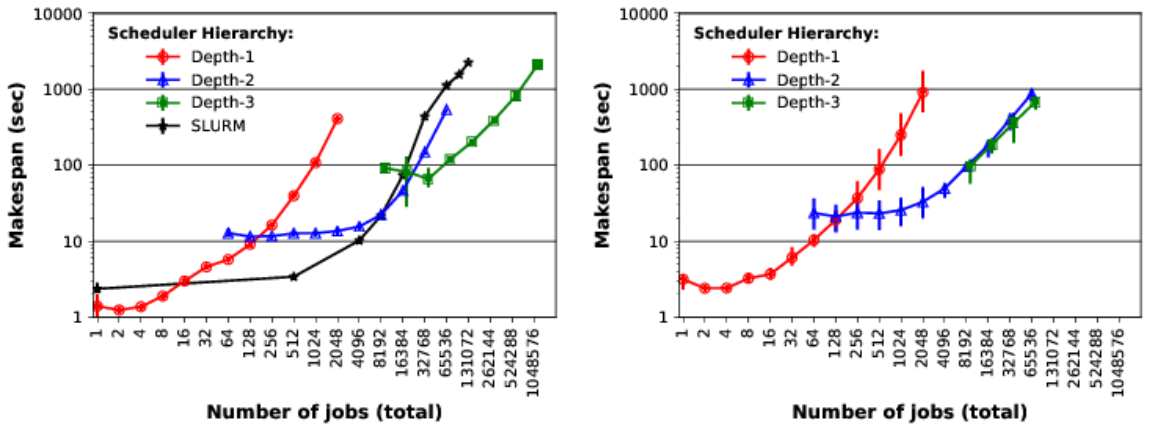
Figure 5.6b exhibits many of the same trends as Figure 5.6a. As we increase the number of cores in the allocation, and thus the number of schedulers, the makespan of the workload decreases. Also similar is the failure of the depth-1 hierarchy to schedule the workload in under 3 hours successfully. The main difference between the two evaluations is that the parallelism in the depth-3 hierarchy always provides the UQ workload with a better or equivalent makespan to the depth-2 hierarchy. The additional parallelism of the depth-3 hierarchy is beneficial in the smaller allocations and is not detrimental in the larger allocations. Figure 5.6b indicates that the suitable scheduler hierarchy depends not only on the allocation size but also the runtime of the jobs

within the workload.

5.4.4 Impact on Job Count Scaling

We look at the same question about the suitable scheduling hierarchy from a different perspective. Contrary to the case described above, we fix the number of cores in the allocation and vary the number of jobs per scheduler instance. The results in Figure 5.7 are generated on an allocation with 1,152 cores for both the synthetic workload and the UQ workload, a variable number of jobs per instance that ranges from 1 to 2,048, and a total number of jobs ranging from 1 to 1,048,576. In Figure 5.7a, we present for each of the hierarchy configurations (i.e., depth-1, depth-2, and depth-3) the maximum, minimum, and average makespan over three runs of the stress test workload. For comparison with a production monolithic scheduler, we also include the SLURM stress test results previously shown in Figure 5.2. In Figure 5.7b, we present for each of the hierarchy configurations the maximum, minimum, and average makespan over three runs of the UQ workload. We do not include SLURM given its high error rates and poor performance on the stress test workload. In both figures, the super-linear makespan curves is an artifact of the single threaded performance of the scheduler. Because the scheduling instances receive all the jobs immediately after they start, the schedulers must iterate over every job in the queue during every scheduling loop, resulting in the observed pattern.

Figure 5.7a shows that there are defined thresholds in terms of the total number of jobs in the workload for which moving from one scheduling hierarchy depth to the next results in better performance. For small workloads with less than 256 jobs, it is not beneficial to use a deep hierarchy, and thus the simple depth-1 hierarchy has the shortest makespan. For workload sizes between 256 jobs and 16,384 jobs, the depth-2 hierarchy outperforms both the depth-1 hierarchy and the depth-3 hierarchy. The reason is that the benefits of parallelism kick in causing the depth-2 scheduler to outperform the depth-1. In the case of the depth-3 hierarchy, the cost to launch the scheduler hierarchy is the penalizing factor. It is at the largest scales in terms of



(a) Stress test workload

(b) UQ workload

Figure 5.7: Workload makespan (in seconds, on a logarithmic scale) for the depth-1, depth-2, and depth-3 scheduler hierarchies for fixed-size clusters and differing numbers of total jobs (on a logarithmic scale)

numbers of jobs (that is, for workloads with at least 32,768 jobs) that we see the real benefit of a depth-3 hierarchy. In less than 410 seconds, the fully hierarchical model (i.e., depth-3) is able to schedule and launch 144X more synthetic ensemble jobs than the centralized model (i.e. depth-1): 294,912 and 2,048 jobs, respectively. In less than 2,240 seconds, the fully hierarchical model is able to schedule and launch 9X more synthetic ensemble jobs than SLURM: 1,179,648 and 131,072 jobs, respectively. Additionally, the fully hierarchical model is able to schedule 131,072 jobs 11X faster than SLURM (i.e., 202 seconds and 2,240 seconds, respectively). SLURM can outperform the depth-1 and depth-2 hierarchies in certain scenarios (i.e., between 16 and 8192 jobs) because it is a mature scheduler that has over a decade of optimizations built-in. On the other hand Flux, the foundation for our depth-1 and depth-2 hierarchies, is still a young scheduler that is not yet fully optimized. We outperform SLURM despite the lack of optimization in Flux only because of our fully hierarchical model, exemplified by the depth-3 hierarchy. Further optimizations in Flux, such as reducing the cost of launching Flux instances, can improve the performance of hierarchical scheduling.

Figure 5.7b, exhibits many of the same trends as Figure 5.7a. As we increase the number of jobs in the workload, there are defined points at which the best scheduler

hierarchy changes. For small workloads with less than 256 jobs, the simple depth-1 hierarchy has the lowest overhead and thus the shortest makespan. For workloads between 256 and 16,384 jobs, the depth-2 hierarchy has the right amount of scheduler parallelism to produce the lowest makespan. For workloads with more than 16,384 jobs, the increased scheduler parallelism of the depth-3 hierarchy produces the lowest makespan. In the same amount of time (i.e., 909 seconds), the fully hierarchical model is able to schedule and launch 36X more real-world ensemble jobs than the centralized model: 73,728 and 2,048, respectively. Figure 5.7b shows that the suitable scheduler hierarchy depends not only on the workload size but also the runtime of the jobs within the workload.

We can see from the stress test workload results, shown in Figure 5.7a, the dramatic effect that hierarchical scheduling has on the stress test ensemble's makespan. Our depth-3 hierarchy, given the same amount of time as the depth-1 hierarchy and SLURM, schedules 144X and 9X more jobs, respectively. In Figure 5.7b, we see these benefits carry over to real-world ensembles, where our depth-3 hierarchy schedules 36X more jobs than the centralized model (i.e., depth-1). By eliminating the critical performance bottleneck in existing schedulers, the fully hierarchical model opens up new possibilities for ensemble studies on HPC systems.

5.5 Summary

Existing production HPC schedulers largely rely on the decades-old centralized scheduling model. These schedulers fail to meet the challenges of future HPC systems: scalability, workload specialization, and multi-level constraints. Alternative approaches based on the decentralized and limited hierarchical scheduling models offer improved scalability over the centralized model, but they still fall short of addressing both the depth and breadth of the emerging challenges. Decentralized scheduling, for example, would make scalably implementing our I/O-aware scheduler presented in Chapters 3 and 4 challenging given its limited support for multi-level constraints. In this chapter, we implement a prototype that embodies the defining principles and properties of the

fully hierarchical model and use it to evaluate the model's scalability on two ensemble workloads: a stress test ensemble workload and an uncertainty quantification (UQ) workload. Our results show how the fully hierarchical model, when applied to ensemble workloads on HPC systems, can achieve a 59X increase in job throughput performance over the depth-1 scheduler and an 11X increase over SLURM for the stress test ensemble workload. When applied to the real-world UQ workload, the fully hierarchical model schedules and launches 36X more ensemble jobs than centralized scheduling in the same amount of time.

Taken together, our results light a promising path where our fully hierarchical scheduler can play a crucial role in enabling extreme-scale ensemble simulations. While the benefits of hierarchical scheduling on homogeneous ensembles (i.e., ensembles with jobs that have uniform resource requirements) are clear, what remains to be seen is hierarchical scheduling's effectiveness while managing heterogeneous ensembles (i.e., ensembles with jobs that have variable resource requirements). Will the siloing of resources across the scheduler hierarchy via static resource allocations create the potential for resources to go underutilized? Before hierarchical scheduling can be successfully applied to heterogeneous ensembles, we must investigate the scheduler's effects on resource utilization as well as potential mitigation strategies, such as switching to dynamic resource allocations.

Chapter 6

THESIS SUMMARY AND FUTURE WORK

6.1 Thesis Summary

The diversification of HPC resources and workloads present new challenges to batch job schedulers. HPC schedulers must now manage new I/O constraints imposed by the addition of burst buffers while simultaneously supporting large ensemble workloads, which are becoming increasingly important in scientific workflows. Existing batch jobs schedulers lack the resource models and scalability to effectively handle these new challenges, resulting in degraded system and workload performance. We tackle the problems from emerging I/O resource constraints and ensemble workloads in three fundamental ways.

Specifically, in Chapter 3 we tackle the problem of increasingly constrained I/O resources by developing an I/O-aware scheduler. To incorporate I/O-awareness, we integrate a model of the I/O subsystem into the scheduler, which is used in the scheduler's constraint checks. The model consists of the links between all levels of the I/O subsystem (e.g., compute nodes, switches, and the parallel filesystem). We show that our I/O-aware scheduler, when fed perfect knowledge of job I/O requirements, reduces job performance variability by 33% and increases system utilization by up to 21%. Assuming we have an accurate source of job I/O requirements, our I/O-aware scheduler can improve application performance and, ultimately, the amount of science performed on the system.

Second, in Chapter 4 we remove the assumption that our I/O scheduler has access to perfect I/O knowledge. We replace the perfect job I/O knowledge used in Chapter 3 with job I/O predictions provided by the resource prediction tool PRIONN.

This knowledge allows our I/O-aware scheduler to anticipate future system I/O usage and I/O bursts. We show that we can anticipate future I/O bursts with $> 50\%$ accuracy.

Finally, in Chapter 5 we address the scheduling scalability challenges associated with the increasing presence of ensemble workloads by developing a fully hierarchical scheduler. We implement a prototype scheduler based on the fully hierarchical model and evaluate it on synthetic and real-world ensembles. We show that the extra scheduling parallelism afforded by the fully hierarchical model results in increased job throughput and reduced ensemble makespans. We show that our depth-3 hierarchical scheduler can schedule and launch jobs 11 times faster than the state-of-the-practice scheduler SLURM. Additionally, we show that for real-world ensembles our depth-3 scheduler can launch 36 times as many jobs as a depth-1 scheduler in the same amount of time.

6.2 Future Work

While this thesis represents the first step towards next-generation batch job scheduling, more work is required before I/O-aware and hierarchical scheduling can be the default for all users and workloads running on production HPC systems.

6.2.1 Metadata-Aware Scheduling

As we demonstrate in Chapter 3, our I/O-aware scheduler is capable of scheduling the I/O bandwidth that occurs between the compute nodes and the parallel filesystem (PFS) as applications write data out to files. The I/O operations that our scheduler does not consider are the metadata operations, such as file creates and deletes. These metadata operations play a key role in many workflows. Workflows create files for many reasons, including checkpointing, logging, and synchronization. Metadata operations can be prohibitively expensive in ensemble workloads, where multiple small files are typically created for each job in the ensemble [45]. Large numbers of metadata operations are particularly troublesome for PFSes with centralized metadata servers (MDSes), like

Lustre, where the metadata processing is concentrated at a single location and shared amongst all users. With a centralized MDS, a single user can degrade the PFS performance for all other users, like in the case with I/O bandwidth. Thus, metadata-aware scheduling is necessary to achieve high and consistent scientific goodput.

One technique for avoiding metadata contention is to manage and schedule metadata operations with the batch job scheduler. Initially, this seems like a simple extension of the bandwidth-centric I/O-aware scheduling presented in Chapter 3. In the case of a centralized MDS, the resource model would only need to be extended to include one additional resource (i.e., the MDS itself). Upon further investigation, we see that scheduling metadata operations presents three unique challenges. First, not all metadata operations are equally expensive, with directory creations being twice as costly as file creation [58]. Second, not all sources of metadata operations are under the control of the scheduler (e.g., a user on a login node running the `ls` command). Third, the performance of an MDS is highly dependent on its cache, and thus performance drops as the working set grows (i.e., more files are operated on simultaneously) [3].

Below we propose a metadata-aware scheduler that is designed to addresses all three challenges. To address the first challenge of variable metadata costs, our proposed scheduler normalizes the cost of each metadata operation relative to the cost of other metadata operations to produce a `normalized metadata operation` (NMO). For example, if file creates are one NMO then directory creations are two NMOs. Armed with a standard measure, the scheduler then balances the required NMOs of the running jobs against the NMO-capacity of the MDS. As with bandwidth-centric I/O-aware scheduling, our proposed metadata-aware scheduler uses machine learning to predict the NMOs each job will perform. To address the second challenge of external sources of metadata operations, our metadata-aware scheduler will reserve a percentage of NMOs for external sources, which is easily configurable by administrators. To address the third challenge of variable MDS performance due to caching, our proposed scheduler imposes a limit on the working set of each job such that the cache of the MDS is not overwhelmed. Despite sharing some similarities to the bandwidth-centric I/O-aware

scheduler, this proposed metadata-aware scheduler is complex in its own right. More research is required to investigate the feasibility, efficiency, and cost associated with metadata-aware scheduling.

If research into metadata-aware proves successful, the next logical step is to merge metadata-aware and bandwidth-centric I/O-aware scheduling to form a comprehensive I/O-aware scheduler. Under a PFS where the metadata and bandwidth operations are performed on separate hardware and thus do not contend with one another, like Lustre and certain configurations of GPFS, combining bandwidth and metadata scheduling is a relatively straightforward engineering effort. The main requirement is that both of the underlying resource models are combined and their constraints are enforced. With a tool like Flux's resource matching service [14], this is as simple as creating two resource hierarchies that are joined by their common elements (i.e., the compute nodes). The main unknown and focus for research is how the multiple objectives will impact the resource utilization and goodput of a system running under these scheduling policies.

6.2.2 Hierarchical Scheduling

Before hierarchical scheduling can become a production-ready solution for all HPC users, it must include a technique to determine the most performant hierarchy automatically for a given workload. As we demonstrate in Chapter 5, the fully hierarchical approach can increase the job throughput and decrease the makespan of ensemble workloads, assuming the proper hierarchy configuration is chosen. For a given ensemble, if too shallow a hierarchy is chosen, there will be insufficient parallelism to reap the benefits of hierarchical scheduling fully, but if too deep a hierarchy is chosen, the startup costs of the hierarchy will cut into the benefits provided by the additional parallelism. Finding the correct hierarchy is key to workload performance under hierarchical scheduling, but not all users can be expected to find the best hierarchy for every one of their workloads. Thus, it is necessary to create an automatic technique for determining the most performant hierarchy for each workload. While there are

many possible solutions to this problem, three techniques, in particular, are especially promising: model-driven configuration, configuration auto-tuning, and adaptive runtime reconfiguration.

The first technique for relieving users of the burden of providing hierarchy configurations is to use a performance model to provide defaults based on each workload's characteristics. If necessary, users can then fine tune and tweak these default hierarchies to their liking. A performance model hand-built for hierarchical scheduling starts with a cost function that estimates a workload's makespan based on several variables: the configuration of the hierarchy, the startup cost of a single scheduler instances, the job throughput of a single scheduler instance, the runtime of applications in the workload, and the runtime of each application in the workload. Once we have a cost function that approximates the workload's makespan given these variables, we can then use that cost function in combination with optimization methods (e.g., integer linear programming and gradient descent) to find the hierarchy configuration that produces the lowest cost (i.e., shortest workload makespan). This performance model requires extensive effort to develop and validate the cost function but also provides users with a quick and inexpensive method for determining the best hierarchy configuration for their workload.

In contrast to a hand-built performance model, the hierarchical configuration can be auto-tuned, which requires significantly less human intervention. The literature on auto-tuning in HPC is extensive and breaks down into three main groups: machine learning [90, 105, 20], statistical models [61, 76], and meta-heuristics [18]. Any one of these auto-tuning techniques can be applied to hierarchical scheduling with little to no effort on the part of the user. The downside is that the auto-tuning techniques learn on-the-fly and require some amount of failure (i.e., hierarchy configurations with poor performance) to learn and improve.

The final method, which complements the previous two methods, is to use adaptive runtime reconfiguration of the hierarchy to adapt to the needs of workloads on-the-fly. The decision to either launch additional schedulers or tear down existing ones can

be driven by specific information pulled from the monitoring of Flux instances within the hierarchy. For example, if resources under the hierarchy's control are sitting idle and the schedulers within the hierarchy are busy, this is a signal that the schedulers are overwhelmed and cannot saturate the compute resources. In this scenario, additional schedulers should be launched and added to the hierarchy. In an alternative scenario, if all resources under the hierarchy's control are allocated and the schedulers within the hierarchy are idle, this is a signal that there are too many schedulers. In this second scenario, some schedulers within the hierarchy should be terminated, freeing up resources used by the idle schedulers and reducing the performance perturbation caused by the hierarchy. Additional considerations include determining what jobs and resources new schedulers should gain control of when added to a scheduler hierarchy at runtime. One way to introduce a new scheduler into the hierarchy is by widening the hierarchy. For example, whenever a scheduler becomes overwhelmed, its parent spawns a new child scheduler, which then becomes a sibling to the overwhelmed scheduler and takes control of the overwhelmed scheduler's idle resources as well as a portion of the jobs still in the queue. This exchange of jobs and resources would be accomplished using the technique discussed next in Section 6.2.3. Alternatively, a second way that schedulers can be added to the hierarchy is by increasing the hierarchy depth. For example, whenever a leaf scheduler becomes overwhelmed, it spawns multiple child schedulers and then distributes its resources and jobs amongst its new children.

6.2.3 Dynamic Scheduling

Before hierarchical scheduling can become a production-ready solution for all HPC workloads, it must also support dynamic scheduling. As we demonstrate in Chapter 5, the fully hierarchical approach can increase the job throughput and decrease the makespan of homogeneous ensemble workloads by employing a hierarchy of schedulers to schedule and execute jobs simultaneously across different sets of resources. Unfortunately, the advantages of hierarchical scheduling may come at the cost of resource fragmentation and resource utilization when scheduling heterogeneous ensembles (i.e.,

ensembles with jobs that have variable resource requirements) [24, 51, 52]. Resource fragmentation can occur whenever a resource is partitioned. For example, if an application is allocated an entire compute node with 16 cores but only utilizes 12 cores, the four unused cores are fragmented off from the rest of the system and go unutilized. The fragmentation problem worsens as the resources are hierarchically partitioned into multiple levels under hierarchical scheduling. With resource fragmentation, even if a system, as a whole, has a sufficient number of available resources to run a job, the job will not be scheduled if the resources are siloed behind different hierarchical scheduler boundaries. Thus, resource fragmentation leaves idle resources on the table, ultimately resulting in decreased system resource utilization. In this section, we present preliminary results demonstrating the problem of resource fragmentation under hierarchical scheduling as well as our proposed solution: dynamic hierarchical scheduling.

To evaluate the amount of resource fragmentation and reduction in resource utilization, we used the Flux emulator to emulate the execution of heterogeneous ensembles under a centralized (i.e., depth-1) scheduler and a hierarchical (i.e., depth-3) scheduler. The Flux emulator is used to schedule all the jobs, analyze the performance of the schedulers, and record the resource fragmentation. The jobs that we use in our evaluation come from a historical trace of jobs run on Cab, a 1,200 node cluster at Lawrence Livermore National Laboratory. The traces cover a 26-hour period and contain 3,000 jobs. Figure 6.1 shows the results of our evaluation. Centralized scheduling, with its global view, can make effective scheduling decisions that maximize resource utilization. Meanwhile, hierarchical scheduling trades a global view of the system for increased parallelism and scalability, resulting in a decrease of 28% resource utilization over centralized scheduling. These results make clear that effective applications of the fully hierarchical model will require a solution to the problem of resource fragmentation.

The main efforts for managing resource fragmentation in centralized scheduling consist of runtime mechanisms for dynamically adjusting jobs' resource allocations [93, 94, 92]. These mechanisms allow the centralized scheduler to respond to an individual

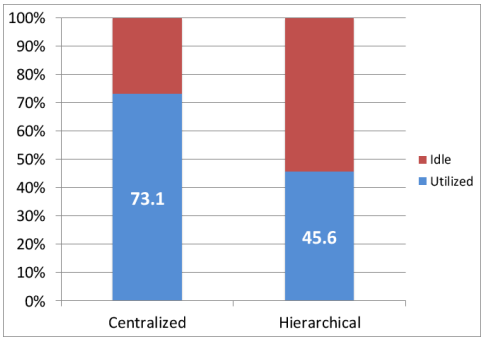


Figure 6.1: System resource utilization under centralized and hierarchical scheduling

job's requests for additional resources while remaining fair to queued jobs. The main efforts in tackling resource fragmentation in limited hierarchal scheduling have adapted the policies of grid schedulers to reduce the probability of fragmentation occurring [24, 51, 52]. To the best of our knowledge, no previous work exists that addresses the problem of resource fragmentation in hierarchical scheduling.

We propose a technique, dynamic hierarchical scheduling, to overcome the resource fragmentation and resource underutilization challenges posed by fully hierarchical scheduling. Under dynamic hierarchical scheduling, schedulers within the scheduler hierarchy detect temporarily idle resources that cannot be used to run any of their jobs and loans these resources to other schedulers within the scheduler hierarchy that can use them. Additionally, scheduler instances are also responsible for determining if borrowing additional resources would beneficial. Schedulers send to their parent both their surplus, idle resource available for lending and any needed additional resources. As demonstrated in Figure 6.2, if a scheduler has a four node job in the queue but only three idle nodes, it will inform its parent that it can utilize an additional node, if available. The parent scheduler then attempts to fulfill the request with idle nodes in its own resource set or its children's resource sets. The parent informs the child of a match to the request, and the child receives the idle resources from its sibling. To minimize the overhead incurred by dynamic hierarchical scheduling, a scheduler instance will only communicate its required and idle resources if it has been in a stable state for a user-configurable amount of time.

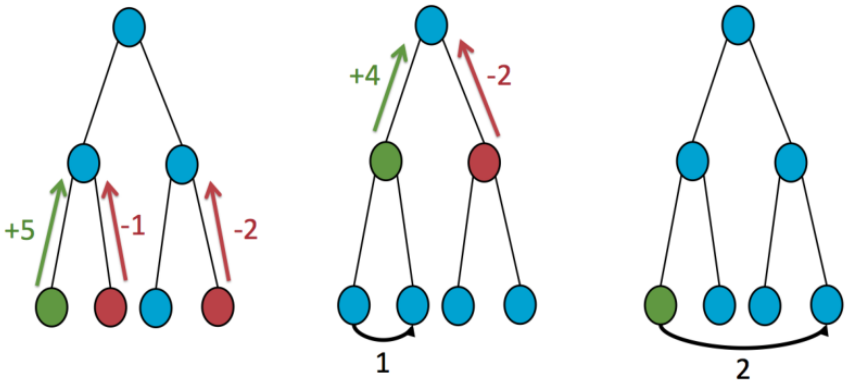


Figure 6.2: Example of a resource transfer in dynamic scheduling. Schedulers exchange surpluses and needs via their parents, who perform the matching. Once a match is made, child schedulers transfer the resources.

Fully hierarchical scheduling provides major scalability and makespan benefits to homogeneous ensembles, but the same is not necessarily true for heterogeneous ensembles. Our results demonstrate that when running heterogenous ensembles, fully hierarchical scheduling can cause resource fragmentation, thereby reducing resource utilization by up to 28% over centralized scheduling. In this chapter, we propose a strategy, dynamic hierarchical scheduling, that we expect to mitigate the problems of resource fragmentation in hierarchically scheduling systems. We leave for future work the implementation and evaluation of dynamic hierarchical scheduling's effects on resource fragmentation, resource utilization, and scheduler overhead.

6.2.4 Interoperability of Techniques

The three major scheduling techniques presented in this thesis (i.e., I/O-aware scheduling, I/O knowledge integration, and hierarchical scheduling) each require major changes to Flux and its components. While most of these changes are interoperable,

some of the changes still require additional work before they are fully compatible. In this section, we give a high-level description of the components that make up Flux, the modifications to the components necessary for the scheduling techniques, and the work still required to make the techniques fully interoperable.

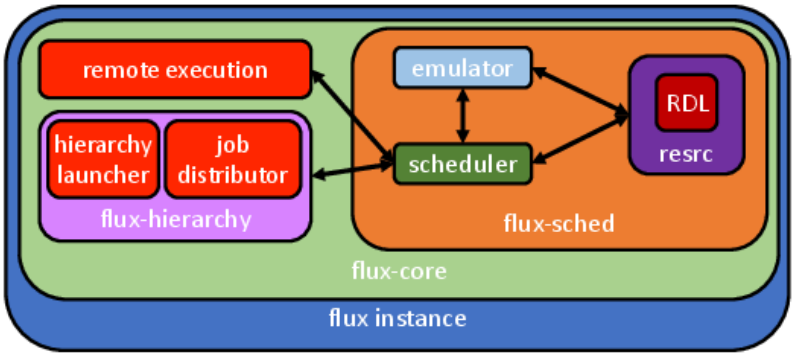


Figure 6.3: The architecture and components of a single Flux instance.

As Figure 6.3 shows, a single Flux instance consists of many components that all operate together to form a functioning resource manager and scheduler. The *flux-core* project represents the resource manager and communication backbone of Flux. By default, it provides communication services like point-to-point and event publish-subscribe as well as support for dynamically loadable modules which can provide additional functionality. Existing modules include the remote execution service, the scheduler, and the emulator. The remote execution service allows processes to be spawned on any of the resources managed by the Flux instance. The scheduler enables batch jobs to be queued and launched according to HPC scheduling policies like EASY and conservative backfilling. The emulator module, as described in detail in Section 3.3.3, emulates user actions (e.g., job submission) and allows job traces to be executed on a simulated system. The scheduler and emulator modules fall under the *flux-sched* project, which also includes the *resrc* and *RDL* libraries. The RDL represents Flux's first prototype resource model. It is implemented in Lua and provides a C interface. For performance purposes, Flux also contains a C reimplementation of the RDL, known as

the resrc. I/O-aware scheduling, I/O knowledge integration, and hierarchical scheduling each represent major changes to several of these components. These changes are highlighted in Figure 6.4, which presents a timeline of the changes made to the Flux project over the course of this thesis work.

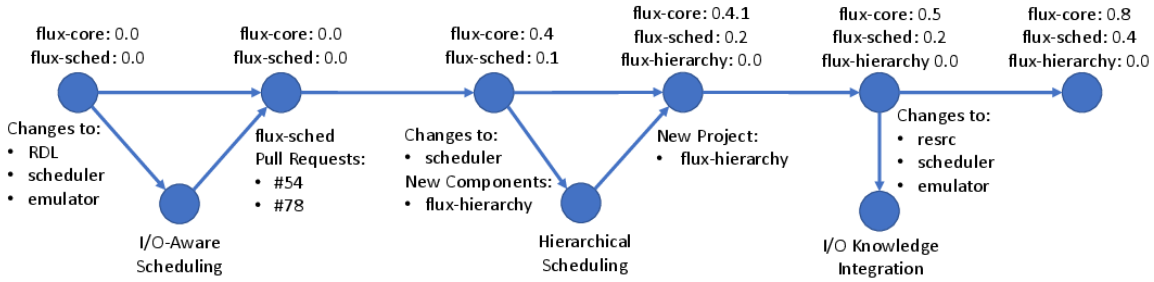


Figure 6.4: The timeline of changes made to Flux as part of this thesis.

The I/O-aware scheduler presented in Chapter 3 requires changes to three Flux components: the RDL, the scheduler, and the emulator. First, to model the I/O subsystem resources, we expand the RDL to include additional I/O resource class: gateway nodes, network switches, and the parallel filesystem. Second, to integrate I/O-awareness into the scheduler, we modify the scheduler to consider the I/O bandwidth attribute of these new resources as part of the constraint checks. Finally, to simulate the execution of applications on a contended system, we enhance the emulator to calculate I/O contention and slow down applications based on the interference factor described in Section 3.3.4. These changes are included in Flux since the pre-alpha (i.e., v0.0.0) of both flux-core and flux-sched.

The I/O knowledge integration described in Chapter 4 requires changes to three Flux components: the emulator, the scheduler, and the resrc. First, to feed into Flux the predictions made by the resource prediction tool, we augment the job submission tool in the emulator to include the runtime and I/O predictions when submitting jobs to the scheduler. Second, we modify the scheduler to make turnaround time predictions, as outlined in Section 4.2.1. Finally, we enhance the resrc to support the system state snapshots required for turnaround time predictions. Since these changes occurred after

the merge of the I/O-aware scheduling changes, they are fully interoperable with the I/O-aware scheduler. These changes are still in a separate fork from the master flux repository as we continue development of the runtime API between the scheduler and resource prediction tool. Once the resource prediction API of the scheduler is finalized and is proven to work outside of the emulator, we will merge the changes into the master branch of Flux.

The hierarchical scheduler described in Chapter 5 requires changes to the Flux scheduler and the addition of a new component, *flux-hierarchy*. First, to ensure that multiple Flux instances running on the same node do not contend for compute resources, we modify the scheduler to pin each child Flux instance to a separate core. Second, we develop a new component for Flux, flux-hierarchy, that launches hierarchies of Flux instances and manages the distribution of jobs across the instances. The new component has been added as a separate repository to the Flux codebase and is compatible with flux-core v0.4.1 and flux-sched v0.2. While the flux-hierarchy component was developed after the completion of the I/O-aware scheduler, they are not yet completely interoperable. The root Flux instance in a hierarchy can load and run the I/O-aware scheduler. This ensures that the child instances are created with an I/O allocation, but the child Flux instances are not yet fully capable of interpreting and leveraging information about their I/O allocation. Specifically, the child instances are unable to merge the resource information from their parent with the resource information they obtain from *hwloc* [23]. Once child instances can combine these two sources of resource information, the hierarchical and IO-aware schedulers will be fully interoperable.

BIBLIOGRAPHY

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

[2] 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.

[3] Architecting a high performance storage system. Technical report, High Performance Data Division, Intel, Jan 2014.

[4] Moab-SLURM integration guide. http://docs.adaptivecomputing.com/9-1-0/MWM/Content/topics/moabWorkloadManager/topics/appendices/slurmintegration.html, 2016. Retrieved March 27, 2018.

[5] Inertial confinement fusion. https://en.wikipedia.org/wiki/Inertial_confinement_fusion, 2017. Retrieved August 22, 2017.

[6] Intel Technologies Advance Supercomputing at National Laboratories for U.S. Security. Technical report, Intel, 2017.

[7] SLURM command/options summary. https://slurm.schedmd.com/pdfs/summary.pdf, 2017. Retrieved July 16, 2018.

[8] Swarm: a Docker-native clustering system. https://github.com/docker/swarm, 2017. Retrieved April 03, 2017.

[9] Command reference manual (man pages). http://research.cs.wisc.edu/htcondor/manual/v8.6/11_Command_Reference.html, 2018. Retrieved July 16, 2018.

[10] Flux-core documents (man pages). https://github.com/flux-framework/flux-core/tree/v0.9.0/doc/man1, May 2018. Retrieved July 16, 2018.

[11] LSF user manual. https://hpc.llnl.gov/banks-jobs/running-jobs/lsf-user-manual, 2018. Retrieved July 16, 2018.

[12] Moab workload manager 9.1.2 administrator guide. http://docs.adaptivecomputing.com/9-1-2/MWM/moab.htm, 2018. Retrieved July 16, 2018.

[13] PBS professional 18.2 reference guide. https://pbsworks.com/pdfs/PBSRefGuide18.2.pdf, Jun 2018. Retrieved July 16, 2018.

[14] Dong H. Ahn. Flux resource matching service. https://github.com/flux-framework/flux-sched/tree/master/resource, 2018. Retrieved July 12, 2018.

[15] Dong H. Ahn, Jim Garlick, Mark Grondona, Don Lipari, Becky Springmeyer, and Martin 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.

[16] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Jayesh Krishna, Ewing Lusk, and Rajeev Thakur. Pmi: A scalable parallel process-management interface for extreme-scale systems. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, EuroMPI, 2010.

[17] Blaise Barney. Slurm and moab. https://computing.llnl.gov/tutorials/moab, August 2017. Retrieved August 22, 2017.

[18] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. Taming parallel i/o complexity with auto-tuning. In *Proc. of the 25th ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, SC, Nov 2013.

[19] John Bent, Sorin Faibish, Jim Ahrens, Gary Grider, John Patchett, Percy Tzelnic, and Jon 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.

[20] James Bergstra, Nicolas Pinto, and David Cox. Machine learning for predictive auto-tuning with boosted regression trees. In *Proc. of Innovative Parallel Computing*, InPar, May 2012.

[21] Michael Beynon, Renato Ferreira, Tahsin Kurc, Alan Sussman, and Joel Saltz. DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage System. In *Proc. of the 2000 Mass Storage Conference*, March 2000.

[22] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. There goes the neighborhood: Performance degradation due to nearby jobs. In *Proc. of the 25th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2013.

[23] François Broquedis, Jérôme Clet-Ortega, Stépanie Moreaud, Nathlie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Proc. of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP, Feb 2010.

[24] Claris Castillo. *On the Design of Efficient Resource Allocation Mechanisms for Grids*. PhD thesis, North Carolina State University, 2008.

[25] Aftab Ahmed Chandio, Cheng-Zhong Xu, Nikos Tziritas, Kashif Bilal, and Samee U. Khan. A Comparative Study of Job Scheduling Strategies in Large-Scale Parallel Computational Systems. In *Proc. of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, July 2013.

[26] Steve J. Chapin, Dimitrios Katramatos, John F. Karpovich, and Andrew S. Grimshaw. The Legion Resource Management System. In *Proc. of the Job Scheduling Strategies for Parallel Processing*, JSSPP '99, pages 162–178. Springer-Verlag, 1999.

[27] Walfredo Cirne and Francine Berman. A comprehensive model of the supercomputer workload. In *Proc. of the IEEE International Workshop on Workload Characterization*, WWC, 2001.

[28] Adaptive Computing. The Moab Workload Manager. http://www.adaptivecomputing.com/, 2017. Retrieved April 03, 2017.

[29] Tamara Dahlgren. Personal Communication, Jan 2018.

[30] Tamara Dahlgren, David Domyancic, Scott Brandon, Todd Gamblin, John Gyllenhaal, Rao Nimmakayala, and Richard Klein. Poster: Scaling Uncertainty Quantification Studies to Millions of Jobs. In *Proc. of the 27th ACM/IEEE International Conference for High Performance Computing and Communications Conference (SC)*, November 2015.

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

[32] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O Interference in HPC Systems Through Cross-Application Coordination. In *Proc. of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2014.

[33] Allen B. Downey. Using queue time predictions for processor allocation. In *Workshop on Job Scheduling Strategies for Parallel Processing*, JSSPP, pages 35–57, 1997.

[34] Henrik. Eriksson, Massimiliano Raciti, Maurizio Basile, Alessandro Cunsolo, Anders Fröberg, Ola Leifler, Joakim Ekberg, and Toomas Timpka. A cloud-based simulation architecture for pandemic influenza simulation. *AMIA Annu Symp Proc*, 2011:364–373, 2011.

[35] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of High Performance Computing Applications*, 11(2):115–128, June 1997.

[36] Jim Gaffney, Dan Casey, Debbie Callahan, Ed Hartouni, Tammy Ma, and Brian Spears. Data driven models of the performance and repeatability of NIF high foot implosions. *Bulletin of the American Physical Society.*, November 2015.

[37] Jim Gaffney, Paul Springer, and Gilbert Collins. Thermodynamic modeling of uncertainties in NIF ICF implosions due to underlying microphysics models. *Bulletin of the American Physical Society.*, October 2014.

[38] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the I/O of HPC Applications Under Congestion. In *Proc. of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2015.

[39] Todd Gamblin, Matthew P. LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and W. Scott Futral. The Spack package manager: Bringing order to HPC software chaos. In *Supercomputing 2015 (SC'15)*, Austin, Texas, November 15-20 2015. LLNL-CONF-669890.

[40] Eric Gaussier, David Glesser, Valentin Reis, and Denis Trystram. Improving backfilling by using machine learning to predict running times. In *Proc. of the 27th ACM/IEEE International Conference for High Performance Computing and Communications Conference*, SC, 2015.

[41] Luís Fabrício Góes, Pedro Guerra, Bruno Coutinho, Leonardo Rocha, Wagner Meira, Renato Ferreira, Dorgival Guedes, and Walfredo Cirne. AnthillSched: A Scheduling Strategy for Irregular and Iterative I/O-Intensive Parallel Jobs. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 3834 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg, 2005.

[42] Google. Kubernetes by Google. http://kubernetes.io, 2017. Retrieved April 03, 2017.

[43] Gary Grider. Speed Matching and What Economics Will Allow. In *Proc. of the HEC FSIO Research and Development Workshop*, August 2010.

[44] John Gyllenhaal, Todd Gamblin, Adam Bertsch, and Roy Musselman. Enabling High Job Throughput for Uncertainty Quantification on BG/Q. In *IBM HPC Systems Scientific Computing User Group*, ScicomP'14, Chicago, IL, 2014.

[45] Stephen Hebrein, Tapasya Patki, Dong H. Ahn, Don Lipari, Tamara Dahlgren, David Domyancic, and Michela Taufer. Poster: Fully hierarchical scheduling: Paving the way to exascale workloads. In *Proceedings of the 29th ACM/IEEE International Conference for High Performance Computing and Communications Conference (SC)*, November 2017.

[46] Stephen Herbein, Dong H. Ahn, Don Lipari, Thomas R.W. Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters. In *Proc. of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 69–80, New York, NY, USA, 2016. ACM.

[47] Stephen Herbein, Dong H. Ahn, and Michela 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.

[48] Dave Higdon, Richard Klein, Mark Anderson, Mark Berliner, Curt Covey, Omar Ghattas, Carlo Graziani, Salman Habib, Mark Seager, Joseph Sefcik, Philip Stark, and James Stewart. Uncertainty quantification and error analysis. Technical report, U.S. Department of Energy, Office of National Nuclear Security Administration, and the Office of Advanced Scientific Computing Research, Jan 2010.

[49] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[50] Jonathon Hines. Moab scheduling tweak tightens titan's workload. https://www.olcf.ornl.gov/2015/10/13/moab-scheduling-tweak-tightens-titans-workload/, Oct 2015. Retrieved July 18, 2018.

[51] Kuo-Chan Huang. On effects of resource fragmentation on job scheduling performance in computing grids. In *Proc. of the 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pages 701–705, Dec 2009.

[52] Kuo-Chan Huang and Kuan-Po Lai. Processor allocation policies for reducing resource fragmentation in multi-cluster grid and cloud environments. In *Proc. of the International Computer Symposium (ICS)*, pages 971–976, December 2010.

[53] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[54] Hameed Hussain, Saif Ur Rehman Malik, Abdul Hameed, Samee Ullah Khan, Gage Bickler, Nasro Min-Allah, Muhammad Bilal Qureshi, Limin Zhang, Wang Yongji, Nasir Ghani, Joanna Kolodziej, Albert Y. Zomaya, Cheng-Zhong Xu, Pavan Balaji, Abhinav Vishnu, Fredric Pinel, Johnatan E. Pecero, Dzmitry Kliazovich, Pascal Bouvry, Hongxiang Li, Lizhe Wang, Dan Chen, and Ammar Rayes. A Survey on Resource Allocation in High Performance Distributed Computing Systems. *Parallel Comput.*, 39(11):709–736, November 2013.

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

[56] 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.

[57] IBM. IBM Spectrum LSF. https://www.ibm.com/, 2017. Retrieved April 03, 2017.

[58] Shuichi Ihara. Lustre metadata fundamental benchmark and performance. https://www.eofs.eu/_media/events/lad14/03_shuichi_ihara_lustre_metadata_lad14.pdf, Sept 2014. Retrieved July 12, 2018.

[59] Nikhil Jain, Abhinav Bhatele, Xiang Ni, Todd Gamblin, and Laxmikant V. Kale. Partitioning low-diameter networks to eliminate inter-job interference. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, IPDPS, May 2017.

[60] Morris Jette and Danny Auble. High throughput computing with SLURM. In *SLURM User Group Meeting*, October 2012.

[61] Travis Johnston, Connor Zanin, and Michela Taufer. Hyppo: A hybrid, piecewise polynomial modeling technique for non-smooth surfaces. In *Proc. of the 28th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD, Oct 2016.

[62] Ana Jokanovic, Jose Carlos Sancho, German Rodriguez, Alejandro Lucero, Cyriel Minkenberg, and Jesus Labarta. Quiet neighborhoods: Key to protect job performance predictability. In *International Parallel and Distributed Processing Symposium*, May 2015.

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

[64] Dalibor Klusáček and Šimon 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.

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

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

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

[68] 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.

[69] Cynthia Bailey Lee and Allan Snavely. On the user-scheduler dialogue: Studies of user-provided runtime estimates and utility functions. *International Journal of High Performance Computing Applications*, 20:495–506, 2006.

[70] Edgar A. León, Ian Karlin, Abhinav Bhatele, Steven H. Langer, Chris Chambreau, Louis H. Howell, Trent D'Hooge, and Matthew L. Leininger. Characterizing Parallel Scientific Applications on Commodity Clusters: An Empirical Study of a Tapered Fat-tree. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 78:1–78:12, Piscataway, NJ, USA, 2016. IEEE Press.

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

[72] Don Lipari. Performance issues with the moab workload manger. https://computing.llnl.gov/jobs/crm/GoodMoabPractices.pdf, August 2008. Retrieved January 22, 2018.

[73] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos 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.

[74] Jay Lofstead, Ivo Jimenez, and Carlos 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.

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

[76] Michael Matheny, Stephen Herbein, Norbert Podhorszki, Scott Klasky, and Michela Taufer. Using surrogate-based modeling to predict optimal i/o parameters of applications at the extreme scale. In *Proc. of the 20th IEEE International Conference on Parallel and Distributed Systems*, ICPADS, Dec 2014.

[77] Ryan Mckenna, Todd Gamblin, Adam Moody, and Michela 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.

[78] Ryan McKenna, Stephen Herbein, Adam Moody, Todd Gamblin, and Michela Taufer. Machine learning predictions of runtime and io traffic on high-end clusters. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2016.

[79] Wes McKinney. pandas: a foundational python library for dataanalysis and statistics. In *1st Workshop on Python for High Performance and Scientific Computing*, PyHPC, Nov 2011.

[80] Robert McLay, Karl W. Schulz, William L. Barth, and Tommy Minyard. Best practices for the deployment and management of production HPC clusters. In *State of the Practice Reports*, SC '11, pages 9:1–9:11, New York, NY, USA, 2011. ACM.

[81] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, Jun 2001.

[82] James M. Murphy, David M.H. Sexton, David N. Barnett, Gareth S. Jones, Mark J. Webb, Matthew Collins, and David A. Stainforth. Quantification of modelling uncertainties in a large ensemble of climate change simulations. *Nature*, 430:768–772, August 2004.

[83] NERSC. Burst Buffer Architecture and Software Roadmap. http://www.nersc.gov/users/computational-systems/cori/burst-buffer/burst-buffer/. Retrieved 01 Apr, 2016.

[84] Daniel Nurmi, John Brevik, and Rich Wolski. Qbets: queue bounds estimation from time series. In *Workshop on Job Scheduling Strategies for Parallel Processing*, JSSPP, pages 76–101, 2007.

[85] Daniel Nurmi, Rich Wolski, and John Brevik. Probabilistic advanced reservations for batch-scheduled parallel machines. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2008.

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

[87] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proc. of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[88] Marco Pasquali, Ranieri Baraglia, Gabriele Capannini, Laura Ricci, and Domenico Laforenza. A Multi-level Scheduler for Batch Jobs on Grids. *J. Supercomput.*, 57(1):81–98, July 2011.

[89] Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. Exploring Hardware Overprovisioning in Power-constrained, High Performance Computing. In *Proc. of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 173–182, New York, NY, USA, 2013. ACM.

[90] Simone Pellegrini, Thomas Fahringer, Herbert Jordan, and Hans Moritsch. Automatic tuning of mpi runtime parameter settings by using machine learning. In *Proc. of the 7th ACM International Conference on Computing Frontiers*, CF, 2010.

[91] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[92] Suraj Prabhakaran. *Dynamic Resource Management and Job Scheduling for High Performance Computing*. PhD thesis, Technische Universität Darmstadt, Darmstadt, August 2016.

[93] Suraj Prabhakaran, Mohsin Iqbal, Sebastian Rinke, Christian Windisch, and Felix 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.

[94] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V. Kalé. A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2015.

[95] Dino Quintero, Luis Bolinches, Puneet Chaudhary, Willard Davis, Steve Duersch, Carlos Henrique Fachim, Andrei Socoliuc, and Olaf Weiser. *IBM Spectrum Scale (formerly GPFS)*. IBM Redbooks, 2 edition, 11 2017.

[96] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys, 2016.

[97] Kento Sato, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama, and Satoshi 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.

[98] Frank Schmuck and Roger 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.

[99] Galen M. Shipman, David A. Dillow, Sarp Oral, and Feiyi Wang. The Spider Center Wide File System: From Concept to Reality. In *Proc. of the Cray User Group (CUG) Conference*, 2009.

[100] Ole Tange. *GNU Parallel 2018*. Ole Tange, April 2018.

[101] Sagar Thapaliya, Purushotham Bangalore, Jay Lofstead, Kathryn Mohror, and Adam 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.

[102] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, June 2007.

[103] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[104] Michael R. Wyatt II, Stephen Herbein, Todd Gamblin, Adam Moody, Dong H. Ahn, and Michela Taufer. PRIONN: Predicting Runtime and IO using Neural Networks. In *Proc. of the 47th International Conference on Parallel Processing (ICPP)*, 2018.

[105] Nezih Yigitbasi, Theodore L. Willke, Guangdeng Liao, and Dick Epema. Towards machine learning-based auto-tuning of mapreduce. In *Proc. of the 21st IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS, Aug. 2013.

[106] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple linux utility for resource management. In *Proceedings of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, June 2003.

[107] Xiaobing Zhou, Hao Chen, Ke Wang, Michael Lang, and Ioan Raicu. Exploring Distributed Resource Allocation Techniques in the SLURM Job Management System. Technical report, Illinois Institute of Technology, Department of Computer Science, 2013.

[108] Zhou Zhou, Xu Yang, Dongfang Zhao, Paul Rich, Wei Tang, Jia Wang, and Zhiling Lan. I/O-Aware Batch Scheduling for Petascale Computing Systems. In *Proc. of 2015 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2015.

Appendix

OPEN-SOURCE SOFTWARE CREDITS

This research and dissertation would not be possible without numerous open-source software projects. Most of the data in this dissertation were analyzed and plotted with the python packages Pandas [79] and Matplotlib [53]. The preprocessing of the data was frequently parallelized with the help of GNU Parallel [100]. All of the core research in this dissertation was built on top of the Flux Framework [15]. The complex testing environment required by these experiments was managed with a combination of Spack [39] and Lmod [80]. This dissertation is typeset with Latex. All of the code, text, and notes written during the course of my Ph.D. was written with Emacs.

Appendix

PERMISSIONS

Chapter 3 is based on my publication in the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing [46]. Based on the ACM's copyright policy [1], I, the original author, have the right to "reuse of any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media."

[1] https://www.acm.org/publications/policies/copyright-policy#permanent%20rights