AI4IO: A SUITE OF AI-BASED TOOLS FOR IO-AWARE HPC RESOURCE MANAGEMENT

by

Michael R. Wyatt II

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

Spring 2020

AI4IO: A SUITE OF AI-BASED TOOLS FOR IO-AWARE HPC RESOURCE MANAGEMENT

by

Michael R. Wyatt II

Approved: _____
Kathleen McCoy, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____
Levi T. Thompson, Ph.D.
Dean of the College of Engineering

Approved: _____
Douglas J. Doren, Ph.D.
Interim Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

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: _____

Li Liao, 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: _____

Todd Gamblin, 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: _____

Trilce Estrada, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

First, I would like to thank my advisor Dr. Michela Taufer and my other committee members: Dr. Sunita Chandrasekaran, Dr. Li Liao, Dr. Trilce Estrada, and Dr. Todd Gamblin.

Special thanks go to Adam Moody, Dong H. Ahn, Dr. Kathleen Shoga, and Dr. Stephen Herbein for their expertise and guidance. I wish to express my deepest gratitude to each member of the Glocal Computing Lab (former and current) for their collaboration and friendship. Especially to Dylan Chapp and Paula Olaya for supporting me and being true friends for so many years.

I would also like to acknowledge the love and support from my parents, family, and friends. This work would not have been possible without all of them.

The work in this dissertation is performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Users submit their simulations to High Performance Computing (HPC) clusters through batch systems which allocate cluster resources to user jobs. While some resource managers and job schedulers, such as Slurm, have a generalized resource model, they end up monitoring and managing only computing resources (i.e., nodes) in nearly all modern HPC systems. Other resources, such as parallel file systems, are also important to job execution but resource managers and job schedulers remain blind to their impact on the overall cluster utilization and job performance. For example, contention for IO resources increases job runtime and delays execution. Furthermore, we observe the trend of an increasing gap between compute power and IO bandwidth, meaning that the bandwidth to file systems is outpaced by the rate of data production for IO-intensive applications. These problems can be addressed with IO-aware schedulers. Unfortunately schedulers lack automatic, scalable, and general tools that support and enable IO-awareness by generating knowledge that the schedulers can leverage to prevent and mitigate IO contention while dealing with IO bandwidth constraints.

To address the problems, in this thesis we propose AI4IO, a suite of Artificial Intelligence (AI) based tools that enable resource awareness on HPC systems. AI4IO consists of two tools: PRIONN and CanarIO. PRIONN automates predictions about user-submitted job resource usage; CanarIO detects, in real-time, the presence of IO contention on HPC systems and predicts which jobs are affected by that contention. By working in concert, the AI4IO tools predict the a priori knowledge necessary to prevent and mitigate IO contention with IO-aware scheduling. We leverage the Flux simulator to implement a realistic simulation of a HPC environment and integrate AI4IO in the Flux simulation. We first evaluate PRIONN and CanarIO separately and show that they improve performance with the prevention and mitigation of IO contention. We

then use the two A4IO tools in concert to produce greater improvements in performance: we observe up to 6.2% improvement in makespan of real HPC job workloads, which amounts to more than 18,000 node-hours saved per week on a production-size cluster.

Chapter 1

THESIS OVERVIEW

1.1 Problem, Motivation, and Proposed Solution

Production resource managers, such as SLURM, manage nodes and on-node resources, like processors and GPUs, but fail to consider the wider set of shared resources in HPC systems (e.g., IO and network). Without the efficient management of shared resources, jobs can contend for those resources and experience performance loss due to the contention itself. In other words, jobs can take longer to run and may fail to complete within their allocated time. In the best case, jobs waste scarce node-hours by slowing down on an expensive resource; in the worst case, their entire allocation is wasted when they do not complete.

In this thesis we tackle IO contention. There is an intrinsic difficulty to preventing and mitigating IO contention on HPC systems. IO pipelines are complex and local or global bottlenecks may manifest for many reasons, making IO contention difficult to detect and treat. For example, two common causes of IO contention are bursts of IO demand from concurrently running jobs and Parallel File System (PFS) performance degradation. IO-aware resource management has the potential to prevent and mitigate the effects of IO contention. Through scheduling decisions generated with a priori knowledge of job resource usage, IO-sensitivity, and contention detection, jobs can be executed in a way that prevents and mitigates IO contention. Unfortunately the a priori knowledge about the jobs and system cannot be easily obtained until after job execution.

The community is in need of tools that provide schedulers in production resource managers with a priori knowledge for managing the wider set of shared HPC

resources in general and IO in particular. These tools must be able to: (1) identify and exploit temporal, system-specific patterns in resource management; (2) augment resource managers to be resource-aware; and (3) avoid the need to rely on the users' feedback on resource utilization. To remove the user from the picture and still gather reliable information on the resources, the community must leverage Artificial Intelligence (AI). AI methods can capture the trends unique to individual systems and resources, and then leverage this for making accurate predictions. AI predictions can be pipelined into HPC resource managers to make schedulers resource-aware.

In this thesis we address this community need by developing AI4IO, a suite of AI-based tools that capture trends in historical HPC data and make predictions that schedulers can use to prevent and mitigate IO contention, ultimately improving the system resource utilization. Our suite currently consists of two tools: PRIONN and CanarIO. These two tools approach the problem in an orthogonal way by providing complimentary prediction services. PRIONN prevents IO contention by providing resource managers with accurate predictions of individual job runtime and IO usage before executions. CanarIO detects and mitigates IO contention by predicting the occurrence of IO degradation in real-time and predicting which jobs are IO-sensitive (i.e., affected by IO contention). We show how their use in concert to support existing production resource managers, such as Flux, can improve performance in the tested workloads.

1.1.1 PRIONN

PRIONN (Predicting Runtime and IO using Neural Networks) targets the prevention of IO contention on HPC systems. When submitting jobs, users submit job scripts to a batch system with requests for compute resources (i.e., number of nodes and cores) for a period of time. Consequently, current production batch schedulers have access to and use only information on the number of nodes and time for their job allocation decision, omitting the fact that jobs may still contend for other resources.

For example, co-scheduling many IO-intensive jobs can cause IO contention and under-utilization of other resources, ultimately degrading performance as the jobs compete for access to the PFS. Because users omit information of shared resources, such as IO, current resource usage requests (i.e., number of nodes and time) are insufficiently equipped to achieve resource-aware scheduling. Delegating the specification of resource usage to users is not a feasible solution. Analysis of job traces in HPC centers shows how user-requested runtimes are often over-estimates. For example, user-requested runtimes for nearly 300,000 jobs on the Cab cluster at Lawrence Livermore National Laboratory in 2016 had a mean error of 172 minutes. Furthermore, when extending the type of resources considered in scheduling decisions to include IO and other resources, users in scientific computing do not have an accurate understanding of job requirements for a given HPC system, and cannot be expected to learn how to estimate the associated resource usage accurately.

PRIONN aims to accurately predict per-job runtime and IO bandwidth before the jobs are allocated to resources, preventing IO contention from occurring by avoidance of co-scheduling IO-intensive jobs. The novelty of PRIONN lies in the automatic, general, and scalable methods for obtaining accurate resource usage predictions. These features allow us to integrate PRIONN's predictions into real HPC schedulers. PRIONN reads job scripts from recently completed jobs and concurrently maps the text of each job script into an image-like data representation. Then, PRIONN trains a deep learning model, specifically a 2D-CNN, by feeding these representations into the model. PRIONN uses the trained CNN model to predict per-job runtime and IO resources of newly submitted jobs in the system queue.

1.1.2 CanarIO

CanarIO targets the mitigation of IO contention effects during jobs executions. While per-job predictions from PRIONN can be used to prevent IO contention caused by co-scheduling IO-intensive jobs, additional sources of contention, like PFS downtime, persist. Current batch schedulers enable sites to address this issue manually. Users can

specify needed file systems in their job submission script, and facility staff can mark particular file systems as unavailable. In many cases, facilities can know when major outages happen in the file system by looking at facility monitoring tools, but these tools do not help us understand which jobs can be run in an outage.

CanarIO leverages fine-grained metric data from the Lightweight Distributed Metric Service (LDMS) and "canary" jobs to monitor IO degradation and quantify the sensitivity of jobs to IO degradation. CanarIO's name is derived from the canaries used by coal miners to detect toxic gas in mine tunnels; if a canary job is killed when it runs out of allocation (or has increased runtime), it can be an indicator of degraded performance for the entire system. CanarIO comprises three parts: (1) the monitoring of periodic IO-probing jobs (i.e., canary jobs) to identify when IO performance degradation occur; (2) identification and labeling of historical jobs that are either sensitive or resistant to IO performance degradation; and (3) training of machine learning models to detect real-time IO degradation and predict which jobs are sensitive to that IO degradation.

The novelty of CanarIO comes from the canary jobs and our application of AI models to detect IO contention and IO-sensitive jobs. Canary jobs are used as indicators to identify historical jobs that are IO-sensitive or IO-resistant to IO degradation on the system. Using the dataset of historical labeled jobs, we train two supervised learning models to: detect IO-sensitive and IO-resistant jobs; and detect real-time IO degradation on the system. The first model (i.e., job classifier) is a CNN adapted from our first AI4IO tool, PRIONN. The second model (i.e., IO degradation detector) is a k-Nearest Neighbors model.

1.1.3 IO-Aware Scheduling with AI4IO

It is in the combined use of the two tools in a production resource manager such as Flux where we get the best out of AI4IO. We extend the Flux simulation environment to mimic the effects of contention and integrate AI4IO with the Flux scheduler to enable IO-aware scheduling.

PRIONN and CanarIO provide Flux with the a priori information about jobs and IO degradation detection needed to prevent and mitigate IO contention with IO-aware scheduling. We extend the Flux scheduler and simulator to incorporate PRIONN and CanarIO in a realistic HPC environment that captures the complexity of interactions among systems, users, and IO contention. To this end, we introduce IO resources and IO contention in the Flux environment, mimic real world consequence of IO contention (e.g., resubmitting jobs that exceed allocated time due to contention), and integrate AI4IO into the Flux scheduling logic to enable IO-awareness.

Using the Flux environment, we evaluate the IO-aware scheduling with each AI4IO tool individually. Our evaluation uses 10 datasets of real HPC jobs on a scaled version of the Quartz cluster at LLNL. We first evaluate our ability to prevent IO contention with PRIONN predictions and IO-aware scheduling. Then, we evaluate our ability to mitigate the effects of IO contention with CanarIO predictions and IO-aware scheduling. Finally, we evaluate the performance of our AI4IO tools in concert to prevent and mitigate IO contention. We observe that individually our tools improve performance of the workloads and together, they provide the largest improvement to performance.

1.2 Thesis Statement

In this thesis, we claim that AI-based tools can transform HPC resource management to be aware of shared resource (i.e., IO). By doing so, we introduce IO-awareness in batch schedulers by augmenting the schedulers with predictions to prevent and mitigate IO contention. To validate this statement, we:

- Develop a tool called PRIONN for predicting the runtime and IO usage of jobs and, by doing so, provide a priori knowledge to prevent IO contention
- Develop a tool called CanarIO for real-time monitoring and prediction job IO-sensitivity, that provide the knowledge to mitigate IO contention
- Combine our prevention and mitigation tools to work in concert with real HPC data and improve scheduling decisions based on IO awareness.

1.3 Thesis Contributions

From the perspective of preventing IO contention, the contributions of this dissertation are:

- We design the components of a workflow that leverages a neural network to interpret whole job scripts and predict per-job runtime and IO resource in HPC systems;
- We evaluate the ability of our workflow to outperform traditional machine learning techniques in accurately predicting per-job runtime and IO resources;
- We use predictions from our workflow (i.e., per-job runtime and IO resources) to capture system IO and IO bursts for an IO-aware scheduler

From the perspective of mitigating the impact of IO contention, the contributions of this dissertation are:

- We develop a method for identifying the occurrence and severity of IO-driven performance degradation events
- We develop a method for post-execution identification of jobs sensitive to IO degradation
- We design ML pipelines for the prediction of IO-sensitive jobs and real-time detection of IO degradation

From the perspective of applying A4IO to make schedulers IO aware, the contributions of this dissertation are:

- We extend the scope of the Flux simulator to incorporate IO contention and IO-aware scheduling logic
- We develop logic pipelines for enabling IO-aware scheduling with our tools
- We apply AI4IO for resource-aware scheduling to evaluate their effectiveness at preventing and mitigating IO contention

1.4 Organization

The remainder of this thesis is organized as follows: Chapter 2 presents background on HPC IO contention and applying AI methods for resource prediction, Chapter 3 presents the work for developing our IO contention prevention tool, PRIONN, Chapter 4 presents our IO contention mitigation tool, CanarIO, and Chapter 5 presents our evaluation of our AI4IO tools to prevent and mitigate IO contention in a realistic simulation of an HPC system and scheduler. Chapter 6 provides a summary of this thesis along with a description of future work.

Chapter 2

BACKGROUND AND EXISTING APPROACHES

In this chapter, we provide an overview of IO contention on HPC machines, previous work to quantify and address resource contention, and previous efforts on AI-based tools for HPC resource usage prediction and resource contention.

2.1 Defining IO Contention in HPC

When more than one job uses a shared HPC system resource, such as memory, network, and IO, resource contention can occur [38]. The result of resource contention is performance degradation for the user jobs. In turn, this causes increased runtimes, jobs exceeding their time limit which lead to wasted resources, and decreased job throughput across the HPC system. Among the shared resources that are contended for on HPC systems today, IO is especially problematic. Figure 2.1 shows current trends for compute and IO capacity in recent years. We see that the growth in system compute power is outpacing the growth in system IO bandwidth. IO-intensive applications currently cause IO contention on HPC systems, and we can expect this problem to worsen [30, 41]. For this reason, addressing IO contention is important for avoiding wasted resources and improving job throughput on HPC systems.

Put simply, IO contention is the result of aggregate job IO demand exceeding the bandwidth of the IO pipeline between compute nodes and storage. Previous work has demonstrated the numerous ways that IO contention can manifest. For example, in [24] and [19], it is shown that IO contention can occur during bursts of system IO activity, where IO demand temporarily exceeds IO capacity. Several works show that system network topology and IO nodes can cause bottlenecks in the IO pipeline that lead to localized IO contention [16, 28, 29].

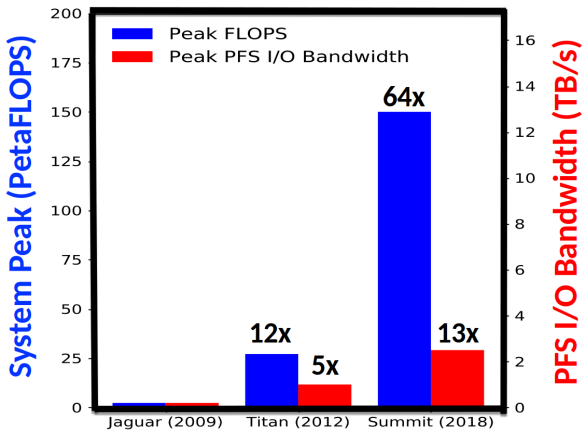


Figure 2.1: The gap between compute power and IO capacity for three flagship machines in the past decade shows an increasing disparity between the capacity to generate data and the capacity to store it.

Another important source of IO contention is PFS performance degradation. PFS typically experience IO degradation due to one of two causes. The first common cause is system failures, which can be partial or total failures and can occur in the software or hardware. System failures include network, hard drive, and storage server failures, as well as OS crashes [9]. A combination of hardware and software techniques can help mitigate these failures, but in some cases these mitigation techniques can themselves become sources of degradation and failure. For example, while RAID protects against data loss after hard drive failures, it greatly decreases drive performance during parity rebuilds. Additionally, storage servers can typically failover their drives to a neighboring server, but this failover can sometimes hang or fail, leaving the drives of both servers unavailable. Further complicating our understanding, it has been observed that file system performance can go through long and short trends of stability and instability for a variety of non-trivial reasons [18].

2.2 Existing Efforts to Addressing Resource Contention

Many efforts have been made to quantify and address shared resource contention on HPC systems. The need to understand resource contention has been addressed in several papers, such as [29] and [22], where HPC interconnects are studied, modeled,

and simulated to understand network performance and contention.Mohammad et al. take a similar approach in [27] where they extensively model and simulate components of HPC systems, including network contention. Similarly, Dietze, Hofmann, and Runger analyzed and modeled contention to improve runtime estimates in [10].

In addition to the work toward understanding and modeling the contention, solutions have been proposed which generally fall into two broad categories: application-centric and resource-centric. For example, an application-centric solution adjusts applications to match the available resources. This is evident in work from [11] where they focus on application coordination to avoid resource contention. Similarly, coordination and load balancing of application IO is explored in [31]. Blagodurov and Fedorova also explore the use of virtualization software to allow jobs to be migrated across a system when resource contention occurs [5].

In contrast to the application-centric solutions, resource-centric solutions work to modify the resources themselves to avoid contention. For example, Neuwirth et al. apply load balancing of IO across the PFS storage nodes in [30]. Similarly, in [45, 46] the virtualization of PFS resources for the purpose of load balancing is explored. The virtualization of HPC nodes themselves to accommodate resource-aware scheduling was also explored in [4].

While many of these proposed solutions have been shown to benefit resource utilization and effectively mitigate resource contention, few focus on the problem of IO resource contention. Further, these solutions often involve extensive modification to the HPC environment. For example, in [5], application must be run with virtualization to enable the migration of jobs to address issues of contention. Because these solutions could not be easily integrated with current batch schedulers, their chance of adoption is lesser. We address these short comings in this thesis by focusing on IO-aware scheduling that is easily integrated with popular batch schedulers, like Slurm, and automate the decisions with our AI-based tools. In the next session, we review the related work for our AI4IO tools.

2.3 State of the Art in HPC Job Prediction

The need for accurate job information is evident when considering how an IO-aware scheduler could address contention. The scheduler requires information about jobs, such as resource usage and IO-sensitivity, to adjust the schedule such that IO contention is prevented and mitigated. In this section, we discuss the extensive work that has been done to provide these predictions in the HPC environment and demonstrate how our AI4IO tools provide unique benefits.

In previous work, traditional machine learning models (e.g., Decision Tree and k-Nearest Neighbors) were primarily used to predict runtime [8, 17, 39, 40, 42] and, in some cases, more general resource usage [23, 24, 35]. This previous work has shown that machine learning models can provide more accurate runtime estimates than users [40].

Smith, Foster, and Taylor used historical HPC job data to predict runtime in [39, 40]. For each job runtime prediction, they train a linear regression model based on historically similar jobs. Job similarity is calculated with several extracted features, including user and job queue. Other runtime prediction papers have utilized similar methods of manually extracted features and machine learning models as Smith, Foster, and Taylor. For example, Krishnaswamy, Loke, and Zaslavsky develop similarity templates for predicting job runtime based on extracted features in [17]. Cunha et al. use a kNN model to predict runtime and turnaround time for HPC jobs in [8]. This work utilizes extracted features from job scripts augmented with data from the scheduler, such as the number of jobs in the queue at job submission time. Chen, Lu, and Pattabiraman present a method for predicting runtimes of jobs being executed using features extracted from log files and a hidden markov model [7]. Tsafrir, Etsion, and Feitelson predict job runtimes based on a user-centric model in [42]. The average users' previous job runtimes and use this as an estimation for the runtime of the next job submitted by a user. Downey developed a statistical model for predicting the queue time of a job based on jobs already running on an HPC system in [12]. Similarly, in the work of Nurmi, Brevik, and Wolksi a statistical method is developed, QBETS, to predict wait times for jobs [32]. Both of these works make predictions based on wait

times of jobs currently running on a system and do not build a prediction model based on extracted features.

A smaller body of work has also focused on other job resources, such as IO. Lofstead et al. outline challenges of dealing with IO contention of parallel file systems and the benefits for preventing contention that come with knowing job IO behavior [20]. Other works present methods for using extracted features to predict IO among other resources to prevent resource contention. McKenna et al. test several machine learning methods for predicting runtime and IO usage of HPC jobs in [24]. They test kNN, DT, and RF models with manually extracted features from job scripts and job logs. Rodrigues et al. predict job runtime, wait time, and memory usage with an ensemble of machine learning algorithms, including kNN and RF, in [35]. Their method extracted features from log files and batch scheduler logs. Matsunaga and Fortes investigate the prediction of many job features, such as CPU, memory, and IO usage with several machine learning algorithms using extracted features in [23].

A common characteristic of this previous work is the requirement to develop and maintain parsers to extract features from diverse source of data, such as job scripts. When relying on this parsing technique, these previous efforts have failed to generalize (e.g., different types of scripts require different parsers). Moreover, any effort to write general parsers incurs the cost of truncating and removing unique information present only in subsets of job scripts. In other words, not all information from job scripts can be captured by a parser. We address this problem in our AI4IO tools and highlight the novelty of our work in Sections 3.1 and 4.1.

Chapter 3

PREVENTING IO CONTENTION WITH PRIONN

In this chapter, we describe our first AI4IO tool, PRIONN, that provides resource usage predictions of jobs to schedulers for preventing IO contention.

3.1 Limitations of Traditional Machine Leaning Approaches

The use of machine learning for resource predictions is not novel. The replication of some of this work can outline limitations and opportunities. To this end, we replicate previous resource prediction work to highlight its limitations. Specifically, we replicate work from Smith and co-authors in [39, 40] and apply the methodologies to our own data. We use predictions for runtime and IO usage of HPC jobs in this case study for comparison to PRIONN.

Traditional machine learning models for resource predictions such as Random Forest (RF), Decision Tree (DT), or k-Nearest Neighbor (kNN) rely on manual feature extraction from job scripts: specific features (lines) in job scripts must be identified, parsed, and transformed into data usable with machine learning models [24, 35, 40]. Consequently, a priori knowledge of the job scripts' structure (e.g., number and type of lines) and which features are indeed useful for resource usage prediction are necessary for traditional machine learning models. This feature extraction also requires development and maintenance of parsers to extract features from diverse sets of jobs scripts. Therefore, when relying on simple parsers, the previous efforts have failed to generalize (i.e., different types of scripts require different parsers). Moreover, any effort to write general parsers incurs the cost of truncating and removing unique information present only in subsets of job scripts. In other words, not all information from job scripts can be captured by a parser. Additionally, features containing string data (e.g., user and

Feature Name	Feature Description
Requested Time	User-requested runtime in hours
Requested Nodes	User-requested number of computation nodes
Requested Tasks	User-requested number of tasks
User	User login ID
Group	User login group
Account	User account (i.e., bank)
Job Name	User specified job name
Working Directory	User specified directory for job execution
Submission Directory	Directory from which user submitted job

Table 3.1: List of manually extracted features from job scripts used with traditional machine learning models for comparison to our tool, PRIONN. These features are based on those described in the paper from the Smith and co-authors in [39, 40].

application name) must be further processed into numerical data using methods such as bag of words or label encoder. Features containing numerical data may also require some sort of processing (e.g., date values must be converted to epoch-seconds).

For the sake of a fair comparison, we manually extract features from our dataset of job scripts and use the extracted information with three traditional machine learning models (i.e., RF, DT, and kNN), which have been used for resource predictions in other work [24, 35, 40, 42]. To this end, we replicate the manual feature extraction in [39, 40]. We create a custom parsing script for our dataset of job scripts which captures features. This task proved difficult due to inconsistencies in job script format, demonstrating that this method is not ideal for deployment on real HPC systems. The complete list of parsed features is in Table 3.1. We use a label encoder to transform each parsed feature into a numerical value in which we assign a unique integer to each unique string value.

In general, the manually extracted features in Table 3.1 are suitable for the many traditional machine learning models from previous resource prediction work . From an implementation point of view, in our work we use regression versions of kNN, DT, and RF available from the *scikit-learn* Python library [34]. Each of these models uses the manually extracted features as inputs, and the outputs are predictions for

runtime and resource usage of individual jobs.

We predict the runtime of each job in our dataset with the manually extracted features in Table 3.1 and the three traditional machine learning models (i.e., kNN, DT, and RF), representative of previous resource usage prediction. We observe that each one of the three machine learning models has similar prediction accuracy; the RF slightly outperforms the other two with an average relative accuracy of 2% and 3% higher than DT and kNN respectively. We can speculate that the increased complexity of the RF explains why it performs better than the DT. Additionally, the method in which categorical features are encoded to numerical values is a likely explanation for the poor performance of the kNN, which relies on measuring Euclidean distance between jobs. The time required to extract job script features and train each model is less than one second for 500 jobs (i.e., the training data size for each batch of predictions). Thus, we identify the RF to be the best performing model among traditional methods. We further access RF's accuracy towards previous work in [40] by replicating the work in that paper with our RF implementation and their datasets. Table 3.2 shows the accuracy from [40] and our RF. We achieve similar or better accuracy than reported in [40] for both datasets. Results in the table confirm that (1) the RF is the best machine learning model for extracted features and (2) the RF achieves similar or better runtime prediction accuracy compared to previous runtime prediction work. We use the RF as a representative of previous methods for comparison to our PRIONN tool in the remainder of this chapter.

Dataset	**Data Size**	**Runtime MAE (minutes)**	
		Smith, et al. [40]	**Our Replication**
SDSC95	76,840	59.65	35.95
SDSC96	32,100	74.56	76.69

Table 3.2: Mean Absolute Error (MAE) for runtime from the Smith and co-workers' paper and from the RF model considered in this chapter as the best traditional method because of its accuracy.

In our replication of this previous resource prediction work, we note several

disadvantages. First, manual parsers require time and effort to create. Second, parsing job scripts reduces the total amount of information available to a machine learning model, which ultimately reduces the prediction accuracy. Third, parsers are job script specific and do not lend to scalable or general methods of predicting resource usage from HPC job scripts. These lessons learned guide us in the design and implementation of PRIONN.

3.2 Predicting Resource Usage

We describe PRIONN and how PRIONN components (e.g., the data mapping technique and deep learning model) are combined to create accurate job runtime and IO resource predictions without manually extracting features from job scripts. Figure 3.1 shows the steps of predicting jobs' resource usage with PRIONN (top) and, for the sake of comparison, with traditional machine learning methods (bottom). The PRIONN tool comprises of three steps: (1) data processing, (2) machine learning model, and (3) training and prediction. PRIONN executes the three steps on a single dedicated node of the Surface cluster at the Lawrence Livermore National Laboratory. Each node on Surface has 16 computational cores and two NVIDIA Tesla K40 GPUs. The data processing is performed on a single core; the training and prediction are performed on the node's two K40 GPUs. These overall steps are performed asynchronously to the scheduling of jobs on the production cluster.

3.2.1 Job Script Data Processing

Data processing is needed to transform raw text from job scripts into data that can be input into machine learning models. A novel component of our tool is the mapping of job scripts to an image-like data representation (i.e., one image-like representation per job script). Each image-like representation of a job script is composed of pixels which are mapped from text characters in the job script. Because of our mapping, whole job scripts can be input into deep learning models. Job scripts must be cropped and padded to a fixed size because deep learning models require a constant

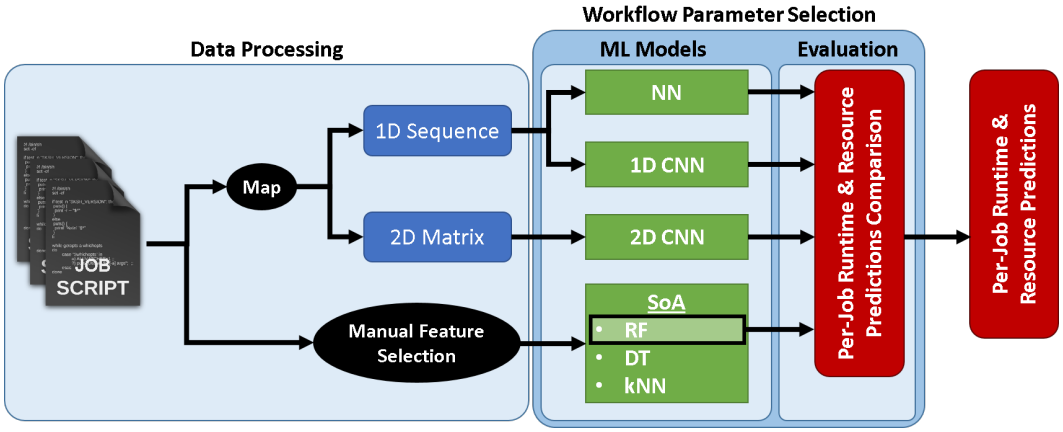


Figure 3.1: Overview of the PRIONN tool for obtaining jobs' resource usage predictions from HPC job scripts.

size input. We describe this process in Section 3.2.4. We consider two types of mapping: we map each job script into either a 1D sequence or a 2D matrix of pixels (i.e., numerical values). Each pixel is mapped from a character in the job script. When mapping job scripts to a 1D sequence, the job script is flattened such that all lines of the text are concatenated into a single line before mapping the characters to pixels. Mapping to a 2D matrix, on the other hand, preserves the structure of the job script text (i.e., the location of characters in the job script are not altered).

The mapping of characters to pixels is performed on a separate node from the batch scheduler to avoid interference with the critical path of the resource scheduler. The mapping can be performed with four different transformations: binary, simple, one-hot, and word2vec. The *binary character transformation* method is a lossy transformation that converts each character to a binary value. All space characters (i.e., space, tab) are assigned the value "0" and all non-space characters are assigned the value "1". The *simple character transformation* method is a lossless transformation that converts each unique character to a unique value. We convert a standard ASCII character from the job script text file to a unique integer value. The *one-hot character transformation* method is a widely used lossless transformation that converts each unique character to a unique 128 value vector. Each vector has exactly 1 scalar with

value "1" and the remaining are "0". The *word2vec transformation* method is a lossless transformation that converts each unique character to a unique 8 value vector. We use Google's word2vec method to obtain this transformation [25]. This method examines the context of a character (i.e., surrounding characters) to embed information about that character in a multidimensional vector.

Independent from the method to map characters into pixels, our data mapping gives PRIONN three advantages over manual feature extraction found in previous resource prediction work [7, 8, 17, 24, 35, 39, 40]. First, PRIONN is automatic and can be deployed as-is on any HPC system. Second, our tool is scalable for the increasing number of HPC jobs and HPC systems in that it does not require ongoing maintenance as job scripts and HPC environments change. Third, our tool is general to any HPC system. Any type of job script can be processed with our mapping to produce data which can be used with a deep learning model.

3.2.2 Machine Learning Models

The selection of a machine learning model for PRIONN is driven by the trade-off between prediction accuracy and prediction costs (i.e., training time). These properties are orthogonal: if a prediction method is highly accurate but slow to produce predictions, delays will make the prediction useless for the batch scheduler. We define our tool to be as accurate as possible without impacting the performance of the scheduler. We assess a diverse set of machine learning models for our image-like data mapping and for the manually extracted features. For our mapped data, we assess three deep learning models into which we can ingest our image-like representation of the job scripts: fully connected Neural Network (NN), 1D Convolutional Neural Network (1D-CNN), and 2D Convolutional Neural Network (2D-CNN). For the manually extracted features, we test three low cost, high-impact machine learning models: Random Forest (RF), Decision Tree (DT), and k-Nearest Neighbors (kNN).

With the image-like representation of our job scripts, we train deep learning models for predicting runtime and resource usage. With deep learning, the many hidden

layers of neurons are able to automatically detect important features and patterns from sets of characters in job scripts. Simpler machine learning algorithms, such as kNN, DT, and RF, are not able to build features from sets of characters and are not suited to analyze our mapped data. The NN uses a 1D sequence of the mapped data as input and contains many fully connected hidden layers. The 1D-CNN also uses a 1D sequence of the mapped data job scripts as input and contains several 1D convolutional layers followed by several fully connected hidden layers. The 2D-CNN uses a 2D matrix of the mapped data job scripts as input and contains several 2D convolutional layers followed by several fully connected hidden layers. The deep learning models are classifiers and each node in the final output layer is associated with a value or range of values predicted for resource usage (e.g., for runtime predictions, the output layer is 960 nodes in size where each node is associated with a runtime in minutes between 0 and 960 minutes).

3.2.3 Training and Evaluation

To substantiate the selection of the best deep learning model for PRIONN and quantify the prediction accuracy to cost trade-off, we mimic a scheduling system in which jobs are submitted to a batch scheduler's queue with the same frequency found on a high-end cluster. Specifically, in this work, we simulate the Cab cluster at Lawrence Livermore National Laboratory. We perform training and prediction at the time of job submissions. The submission, start, and end times of real HPC jobs are used to replicate the queuing and execution of jobs on the HPC cluster with its SLURM batch scheduler. Historical jobs' data (i.e., jobs that have already executed) are used to train each machine learning model. The trained model is then used to predict the resource usage of jobs as they are submitted to the batch scheduler. In our emulation of the real scheduling system, prediction of job runtime and IO resource occurs at the same time as the job submission to the batch scheduler. After every 100 job submissions, models are retrained with the newest historical job data (i.e., jobs that have recently completed). We train each model on the 500 most recently completed jobs. Our empirical evaluation of training with data from 50 up to 5,000 jobs indicated that

there is only a minor improvement of prediction accuracy and higher cost to train beyond 500 jobs for PRIONN. The low training size is unusual for most deep learning tasks, but PRIONN's models are retrained rather than re-initialized after each 100 submitted jobs. Learned parameters in the model are passed to subsequent models, thus knowledge is retained across several training events. This characteristic of deep learning models is not present in traditional machine learning models.

When evaluating prediction models, we first consider the accuracy of the predicted resource usage. Then, we consider the time needed to obtain the resource usage prediction. We compare predicted resource usage to the actual resource usage for each job with relative accuracy. Relative accuracy is preferred over absolute error because it mitigates the negative impact of small prediction error for jobs with high resource usage. For example, a runtime prediction error of 30 minutes is far worse for a one-hour job than for a twelve-hour job. Equation 3.1 shows how we calculate relative accuracy for each resource usage prediction, where *true* is the actual value and *pred* is the predicted value. The ϵ value in the denominator (i.e., machine epsilon) prevents division by zero when both *true* and *pred* are 0. We use the maximum value between *true* and *pred* in the denominator of Equation 3.1 for two reasons: (1) to maintain a range of $[0, 1]$ for the metric and (2) to penalize under-prediction more than over-prediction. We do this because under-predicted resource usage (e.g., we predict IO of 10 MB/s for a job that uses 25 MB/s) fed to an IO-aware scheduler results in resource contention.

$$relativeAccuracy = 1 - \frac{|true - pred|}{max(true, pred) + \epsilon} \tag{3.1}$$

We use a dataset of real HPC jobs to train and evaluate resource usage predictions. The dataset contains information for 295,077 jobs from Lawrence Livermore National Laboratory. The jobs were executed on the Cab supercomputer, which has 1,296 nodes and a maximum runtime of 16 hours; it is connected to a Lustre parallel file system. The dataset consists of job scripts, execution data, and resource usage data for each job. From the nearly 300,000 jobs in our dataset, 111,596 jobs are unique

(i.e., the job script is unique). The dataset exhibits a wide range of resource usage for both runtime and IO of jobs. A total of 29,291 jobs were either canceled by the user or removed from the system before executing. We exclude these jobs from our analysis, giving a total of 265,786 jobs and 97,361 unique job scripts.

3.2.4 Optimal Prediction Parameters

We define an optimal set of components (i.e., data mapping method and deep learning model) for our tool based on the trade-off between accuracy and performance described in Section 3.2.2. We also determine the best traditional machine learning model for comparison to our tool. For each machine learning model, we use the techniques described in Section 3.2.3 to train and evaluate runtime predictions. We use the data described in Section 3.2.3 for the evaluation of our resource usage predictions. We predict runtime down to one minute (i.e., real and predicted runtime are rounded to the nearest minute)

We find the optimal settings for PRIONN with a comparison of the four transformations of our data mapping method (i.e. binary, simple, one-hot, and word2vec) and a comparison of the three deep learning models (i.e., NN, 1D-CNN, and 2D-CNN) described in Sections 3.2.1 and 3.2.2. As discussed in Section 3.2.1, deep learning models require a constant size input. Motivated by the need to keep the data processing and training time low, we fix the job scripts to a standard size of 64 rows and 64 columns of characters before mapping the data to our image-like representation. Job scripts with less than 64 rows or columns of characters are extended to this size with space characters. Jobs larger than 64 rows or columns of characters are cropped to the correct size. Standardizing the size of job scripts incurs a small amount of data loss at the very end of the impacted scripts. Note that only 9.9% of jobs scripts contain more than 64 lines of text and 13.8% of text lines contain more than 64 characters.

We evaluate the time and accuracy of prediction for each data mapping method. Figure 3.2 shows the time necessary to process 500 job scripts (i.e., the number of jobs used each time we train a machine learning model) for each data mapping type. The

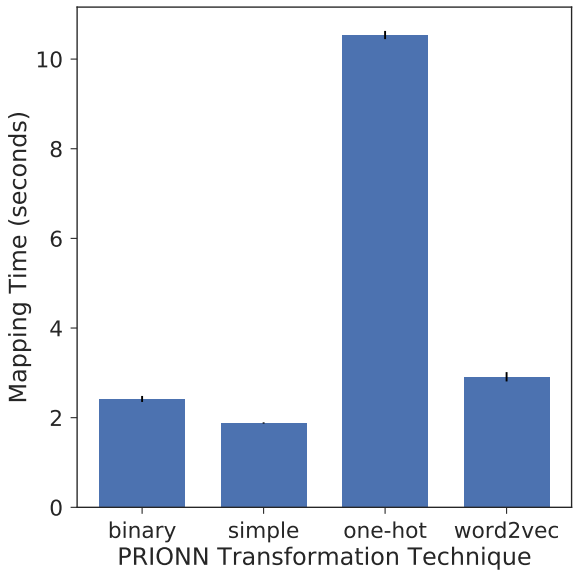


Figure 3.2: Time in seconds needed to transform 500 job scripts to 500 pixel-like representations by the four different transformations (i.e., binary, simple, one-Hot, and word2vect). The pixel-like representations are used for training the deep learning models.

one-hot transformation requires the most time and the three other transformations require less than three seconds for 500 jobs. Each transformation maps characters to values that are either scalars or vectors and the vectors can vary in size. As a result, the time to train a deep learning model with data from each transformation method is different. Figure 3.3 shows the time to train a 2D-CNN for 10 epochs on 500 jobs using data from the four transformation methods. Our results indicate that one-hot requires the most amount of time for training, while the three other transformations take much less time. Figure 3.4 shows the accuracy for each transformation method and a 2D-CNN model, where word2vec outperforms the other three transformation types. The word2vec transformation provides the best combination of processing and training time with high prediction accuracy.

We evaluate the time and accuracy of prediction for each deep learning model considered for PRIONN. Figure 3.5 shows the time to train each of the deep learning models with the word2vec data mapping. The 2D-CNN is trained in less time than the NN and more time than the 1D-CNN. Figure 3.6 shows the runtime prediction

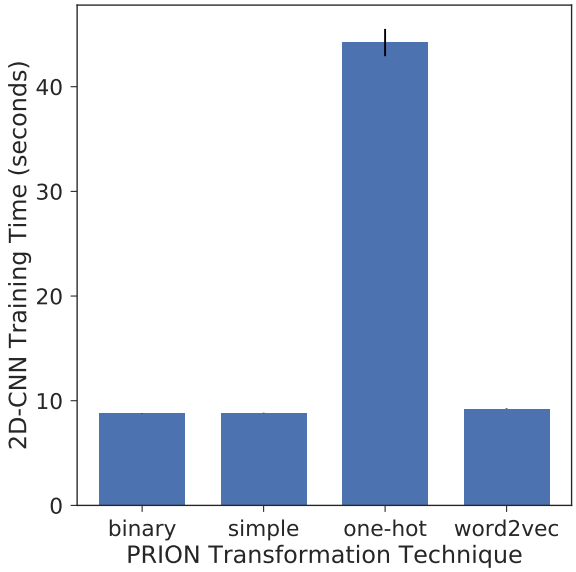


Figure 3.3: Time in seconds needed to train a 2D-CNN with each type of transformed data from our data mapping method.

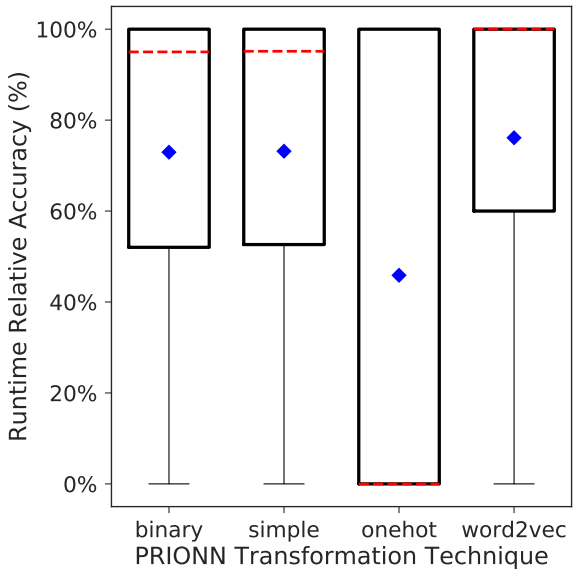


Figure 3.4: Distributions of relative accuracy for runtime predictions with each type of transformed data and a 2D-CNN.

accuracy using the word2vec transformation and our three deep learning models. The NN and 2D-CNN produce runtime predictions with higher accuracy than the 1D-CNN. These results indicate that the word2vec transformation and 2D-CNN model

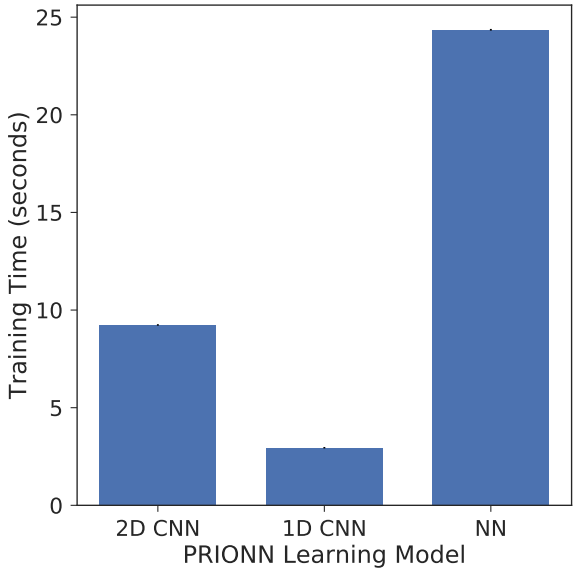


Figure 3.5: Time in seconds needed to train each of the tested deep learning models using the word2vec data mapping method.

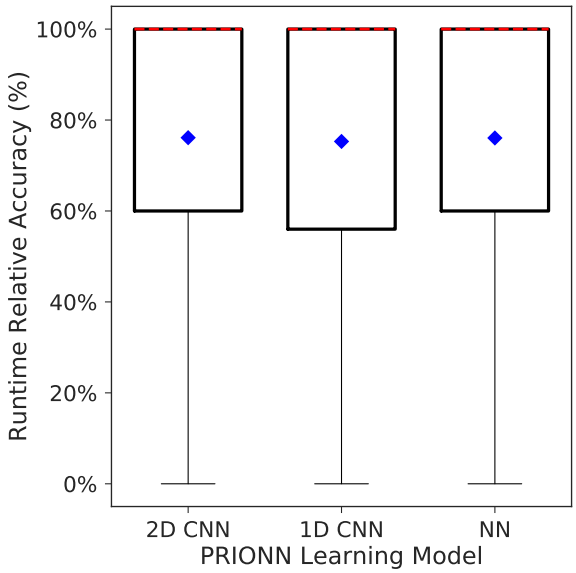


Figure 3.6: Distributions of relative accuracy for runtime predictions with each type of deep learning model and the word2vec data mapping.

give the best results for our resource prediction workflow and are also fast enough to be used with online training. We can reason that word2vec provided the best accuracy because each character is encoded with information about the context of that character.

This information likely decreases the epochs needed to converge on an accurate deep learning model. Similarly, we can hypothesize that the 2D-CNN performs best because the convolutions on the 2D matrix of characters provides an efficient method to build features around patterns of jobs scripts' code (i.e., patterns among subsequent lines of code). Based on our analyses in this section, we use the word2vec transformation and the 2D-CNN deep learning model for our PRIONN tool.

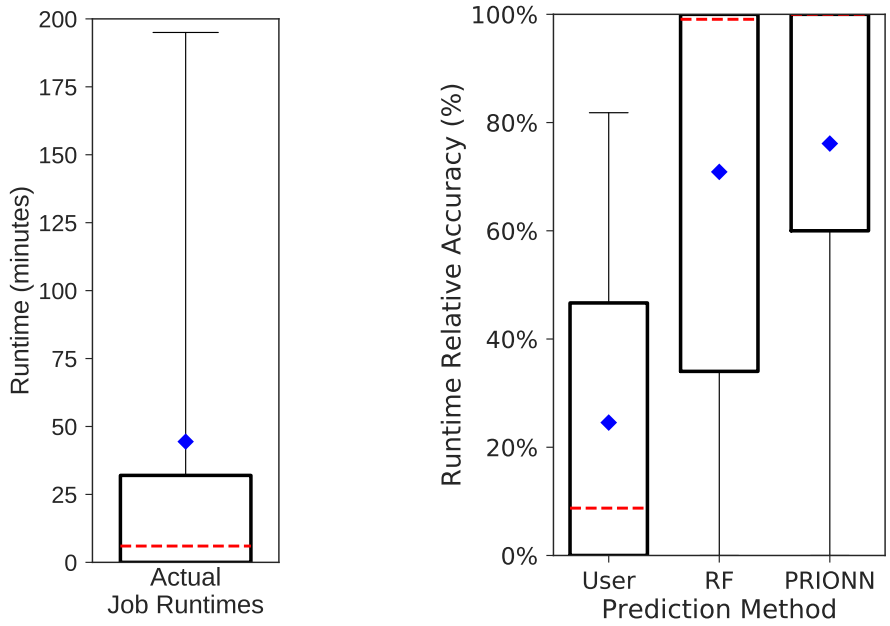
3.3 Evaluation of Resource Predictions

We evaluate the data mapping and deep learning model of PRIONN with predictions for runtime and IO resources of real HPC jobs. To this end, we compare our predictions to the best traditional machine learning model predictions as well as user predictions (when applicable). Our results show how PRIONN provides better per-job prediction accuracy and thus is better suited for augmenting schedulers with the information necessary for IO-aware scheduling.

3.3.1 Per-Job Runtime Predictions

Our dataset of 295,077 jobs comes from real traces whose jobs were executed on the Cab cluster at Lawrence Livermore National Laboratory during 2016. Figure 3.7a describes our dataset in terms of the distribution of actual runtimes. Nearly half of the jobs have a runtime between zero and sixty minutes. The mean job runtime is 44 minutes and a small percentage of jobs have runtimes over three hours. Figure 3.7b shows the boxplots describing the relative accuracy for predicted job runtimes when using: (1) user requested time, (2) the best traditional machine learning model (i.e., RF), and (3) PRIONN. We observe how our method has a mean accuracy of 76.1%, an increase of 6.0% over RF. The median accuracy for our predictions is 100%, indicating that for over half of the jobs in our dataset, we correctly predict the runtime. High accuracy for runtime prediction is important for accurately predicting which jobs will be running at a future time on an HPC system [12]. Therefore, the increase in mean and median accuracy with PRIONN over the previous methods is substantial for prediction

of system IO and IO bursts, which we demonstrate in Section 3.4. Note how the user predictions are substantially outperformed by PRIONN and the RF.



(a) (b)

Figure 3.7: Distribution of actual runtimes for our data (a) and the relative accuracy for predicted job runtimes of user requested time, RF, and our method (b).

3.3.2 Per-Job IO Predictions

For the IO resource predictions, we use the same dataset as in the section above, and we predict total bytes read and total bytes written for each job. Because bandwidth is what an IO-aware scheduler uses [15], we then compute the read and write bandwidth from the predicted total bytes read and total bytes written. We deal with a diverse dataset of jobs in terms of their actual read and write bandwidth. Figure 3.8a shows the distribution of actual read and write bandwidth for our dataset. In the figure we observe how the mean bandwidth for read and write is orders of magnitude larger than the median, indicating a handful of jobs in our dataset have extremely large IO bandwidth compared to the majority of the jobs. We first predict the total bytes read and total bytes written for each job in our dataset with PRIONN and the

RF. Note that users are not providing this information in their job scripts and thus, a comparison with the user predictions is not possible in this case. We then compute the bandwidth by dividing the total bytes read and written with the predicted runtimes of jobs. Figures 3.8b and 3.8c show the boxplots of the relative accuracy for predicted read and write bandwidth with RF and our tool respectively. The cross-comparison of these two figures outlines how our method outperforms RF for both bandwidth values (i.e., read and write). Specifically, PRIONN predictions have a mean accuracy of 80.2% and 75.6% for read and write bandwidth, which is 12.1% and 9.6% higher than the RF predictions.

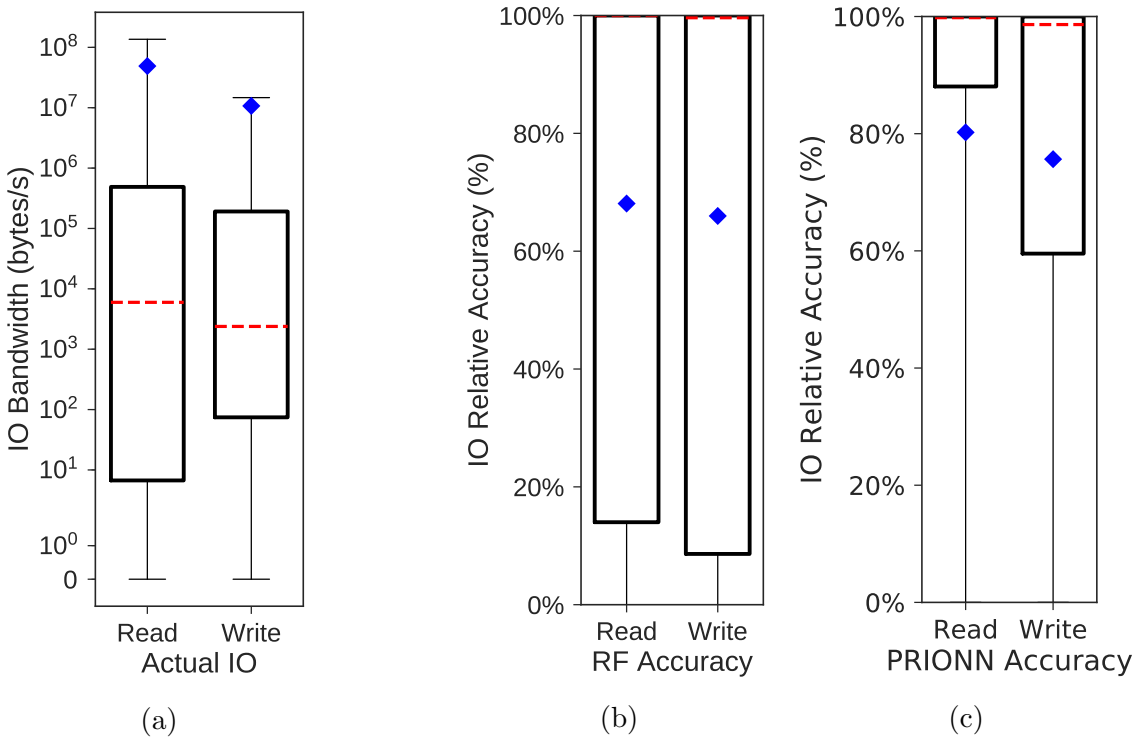


Figure 3.8: Distribution of read and write bandwidth for our dataset (a) and relative accuracy for predicted read and write bandwidth with RF (b) and PRIONN (c).

3.4 Application to Real HPC Systems

We demonstrate the value of per-job runtime and IO predictions from PRIONN to an IO-aware scheduler as part of the second phase of our workflow. Specifically,

Figure 3.9 shows how we use our per-job runtime and IO predictions with the Flux open-source resource management framework simulator and its IO-aware scheduler [2] to predict system IO and IO bursts.

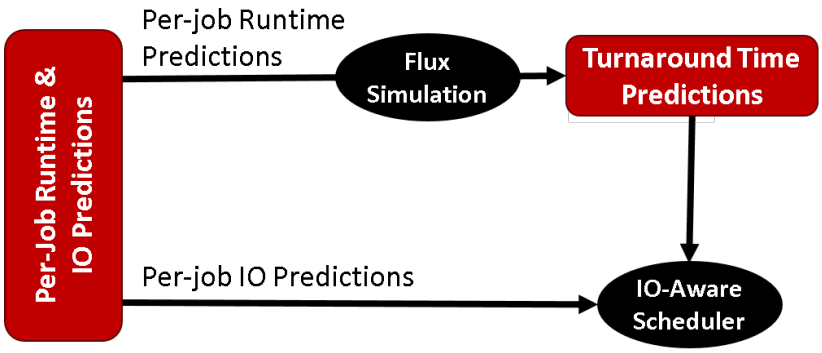


Figure 3.9: Overview showing the application of our runtime and IO predictions to an IO-aware scheduler to predict system IO and IO bursts.

3.4.1 IO-aware Scheduler

An IO-aware scheduler relies on knowledge of IO behavior and potential IO contention to schedule jobs such that IO bandwidth contention is avoided. We leverage per-job runtime and IO usage predictions from PRIONN to build this knowledge and drive the IO-aware scheduler in its decision. To this end, we use the simulator of the open-source, next-generation job scheduler Flux to mimic the evolution of a high-end HPC system [2, 15]. We modify the simulator to use job runtimes that are predicted either by PRIONN, defined in Section 3.2, or by the user. Specifically, the scheduler uses our runtime predictions to estimate when a job starts and completes; it also combines the estimated job's start time with our IO predictions (made for the same job) to estimate the job's impact on the system IO. By incorporating the sum of all jobs' IO impacts we can, ultimately, estimate future system IO (e.g., patterns including IO bursts).

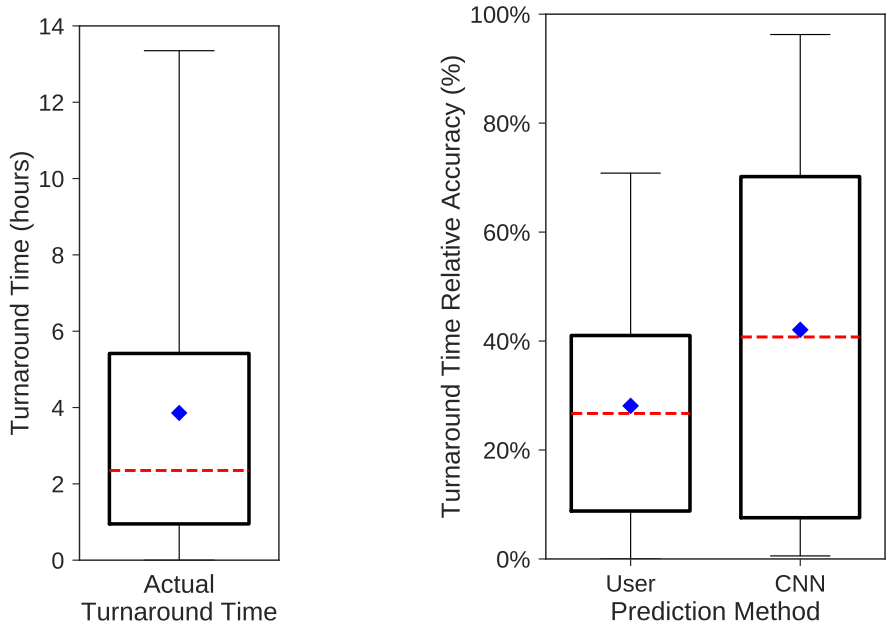
3.4.2 Turnaround Time Prediction

The first step to an effective IO-aware scheduler is accurate turnaround time prediction (i.e., the amount of time between when a job is first submitted to the scheduler and when the job completes), as shown in Figure 3.9. Turnaround time is necessary because it provides insight to which jobs will be executing on the HPC system at future times. The turnaround time prediction for a given job depends on the predicted runtime of the currently queued or in execution jobs. Therefore, inaccuracies of runtime predictions for individual jobs can accumulate into inaccurate turnaround time predictions. Relying on inaccurate runtime predictions, such as those based on user estimates, can result in very poor turnaround time predictions that are detrimental to an IO-aware scheduler.

To predict turnaround time, we submit jobs to our simulated HPC system and record both the simulated turnaround time of each job and the job's execution schedule. When a job is submitted to the system, a snapshot of the system is created. The snapshot creation is followed by four steps. 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 job in execution and in the queue 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 prediction.

To quantify the turnaround time prediction accuracy, we sample five 10,000 job subsets from our original data; the subsets were randomly selected across the span of the job traces. We run five simulations, one for each job subset, and predict turnaround time as described above. Figure 3.10a shows the distribution of the simulations' turnaround times (i.e., the turnaround time observed in the simulation). Figure 3.10b compares the resulting relative accuracy of turnaround time predictions when the simulated system uses user-requested runtime (left) and our framework's runtime (right). We observe that our framework 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 with our predictions compared to user-requested runtimes. This indicates that using per-job runtime predictions from PRIONN achieves greater than 70% accuracy for turnaround time predictions for the upper-quartile jobs and better turnaround time predictions than users for all jobs.



(a) (b)

Figure 3.10: Distribution of our simulations' turnaround times (a) and relative accuracy of turnaround time predictions with user requested runtime and our method's runtime (b).

3.4.3 System IO Prediction

The next step to an IO-aware scheduler is to obtain system IO predictions. To this end, we combine turnaround time predictions from our simulated system with per-job IO usage predictions from PRIONN (i.e., predicted read and write bandwidth), as shown in Figure 3.9. Specifically, to predict the total system IO in use at a given time, we first use the job turnaround time predictions to determine which jobs are running on the system at that time. Then, for the running jobs, we sum their predicted IO usage, producing the estimated total system IO.

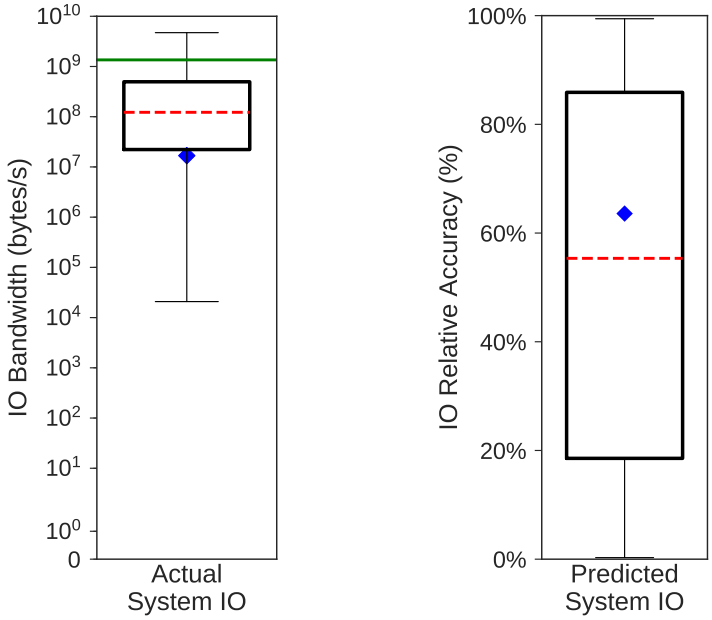
To quantify the accuracy of the estimated total system IO, we perform two types of evaluations, each one using different sources for runtime and turnaround time. In the first evaluation, we use perfect knowledge of runtime and turnaround time (i.e., from real job trace) and our predictions for per-job IO usage. This evaluation isolates the accuracy of our IO predictions from the accuracy of our runtime and turnaround time predictions to demonstrates the strength of PRIONN's per-job IO predictions alone. In the second evaluation, we use our runtime, turnaround time, and IO usage predictions (i.e., from Sections 3.3 and 3.4.2). This evaluation reflects how an IO-aware scheduler can rely on runtime and IO usage predictions from PRIONN in a real-world or production scenario.

For each one of the two evaluations, we report two metrics. First, we report the relative accuracy of our predicted IO behavior (i.e., system bandwidth over time). Second, we measure the precision and sensitivity (i.e., recall) for predicting IO bursts, or unusually high levels of IO bandwidth, that occur in the system IO behavior. IO bursts are of particular importance to an IO-aware scheduler because they are the most likely time for IO contention to occur. We define IO bursts based on the actual system IO bandwidth distribution shown in Figure 3.11a. We 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 3.11a. We define an IO burst as any bandwidth measurement above this value. For each real IO burst, we determine if an equivalent IO burst is also predicted within a given window of time. For example, with a three-minute window, we look for a predicted burst one minute before the actual IO burst, at the time of the real IO burst, and one minute after the actual IO burst. If a burst is predicted in this window, we record a True Positive (TP). We record False Positives (FP) (i.e., we predict an IO burst when there is not an IO burst) and False Negatives (FN) (i.e., there is an IO burst but we do not predict an IO burst) using this same window technique. We use these values (i.e., TP, FP, and FN) to calculate sensitivity and precision for our IO burst predictions. Sensitivity and precision have a range from 0% to 100%; larger values indicate better performance. Sensitivity is the

ratio of correctly predicted IO bursts to actual IO bursts (i.e., $\frac{TP}{TP+FN}$). Precision is the ratio of correctly predicted IO bursts to total predicted IO bursts (i.e., $\frac{TP}{TP+FP}$).

For our first evaluation of system IO bandwidth prediction, we use perfect turnaround time knowledge for all jobs in our dataset and per-job IO usage predictions from PRIONN. Figure 3.11b shows our first metric: the relative accuracy of each system IO prediction. We achieve the mean and median accuracy of 63.6% and 55.3% respectively. Figure 3.12 shows our second metric: the sensitivity and precision of our IO burst prediction across windows ranging from 5 minutes to 60 minutes. We observe in the figure that we predict 47.5% of real IO bursts within two minutes of their occurrence (i.e., sensitivity is 47.5% at a window size of 5 minutes). It also shows that 73.9% of predicted IO bursts correctly predict a real IO burst within two minutes of it occurring (i.e., precision is 73.9% at a window size of 5 minutes). Finally, we note that as the window size for predicting IO bursts increases, the sensitivity and precision also increase. From these results, we can see that IO predictions from PRIONN are sufficiently accurate to predict nearly 75% of IO bursts.

For our second evaluation of system IO bandwidth prediction we use predicted turnaround time and IO usage for our five samples of 10,000 jobs. Figure 3.13a shows the distribution of simulated system IO. We compare the distributions of system IO for jobs used in our first evaluation (i.e., all jobs), shown in Figure 3.11a, and jobs used in our second evaluation (i.e., sampled jobs), shown in Figure 3.13a. We note that the distributions of system IO are not identical, but have similar maximum, mean, and median values. This indicates that the randomly chosen sample of jobs is a good representation of all jobs. Figure 3.13b shows our first metric: the relative accuracy of each system IO prediction using predicted turnaround time and predicted IO usage. Comparing Figure 3.11b with Figure 3.13b shows that the prediction accuracy for system IO decreases when our turnaround time predictions 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 accurately predict several patterns in the IO behavior in the simulation, as indicated



(a) (b)

Figure 3.11: Actual aggregate IO (a) and relative accuracy of the system's accumulate IO predictions (b) using perfect turnaround time knowledge.

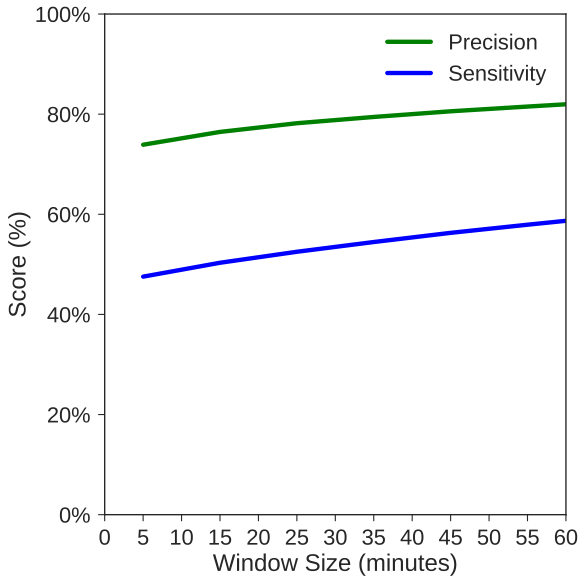
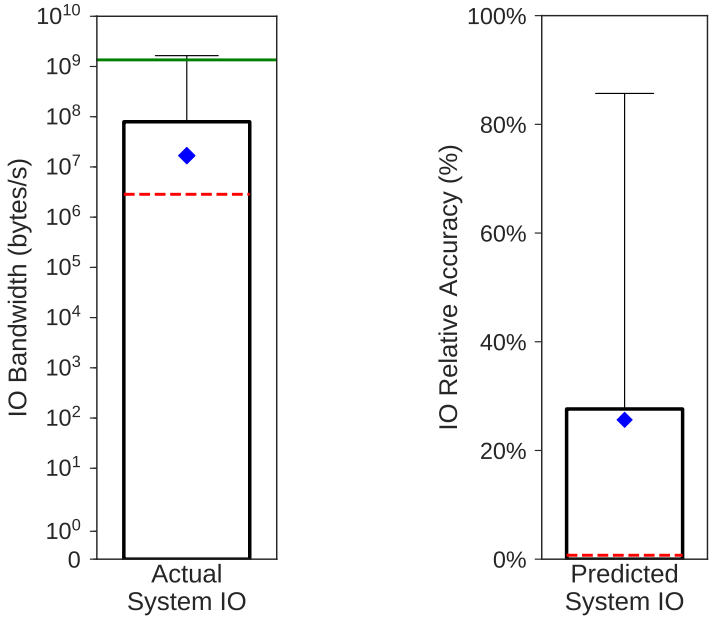


Figure 3.12: Sensitivity and precision of our IO burst predictions across windows ranging from 5 minutes to 60 minutes using perfect turnaround time knowledge.

by the top whisker of the boxplot in Figure 3.13b.

Figure 3.14 shows our second metric: the sensitivity and precision of our IO



(a) (b)

Figure 3.13: Actual aggregate IO (a) and relative accuracy of the system's accumulate IO predictions (b) using our predicted turnaround time from our simulated system.

burst prediction across windows ranging from 5 minutes to 60 minutes. We observe in Figure 3.14 that we predict 55.3% of real IO bursts within two minutes of them occurring (i.e., sensitivity is 55.3% at a window size of 5 minutes). It also shows that 70.0% of predicted IO bursts correctly predict a real IO 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 IO burst prediction in our first evaluation, shown in Figure 3.12, and our second evaluation, shown in Figure 3.14. We note that despite switching from perfect to predicted turnaround time, we achieve similar sensitivity and precision. Like Figure 3.12, we also observe that sensitivity and precision increase as window size increases in Figure 3.14. These results indicate that per-job runtime and IO usage predictions from PRIONN provide the IO-aware scheduler with enough information to correctly predict over 50% of IO bursts. This is a huge improvement over being able to predict 0% of IO bursts without PRIONN.

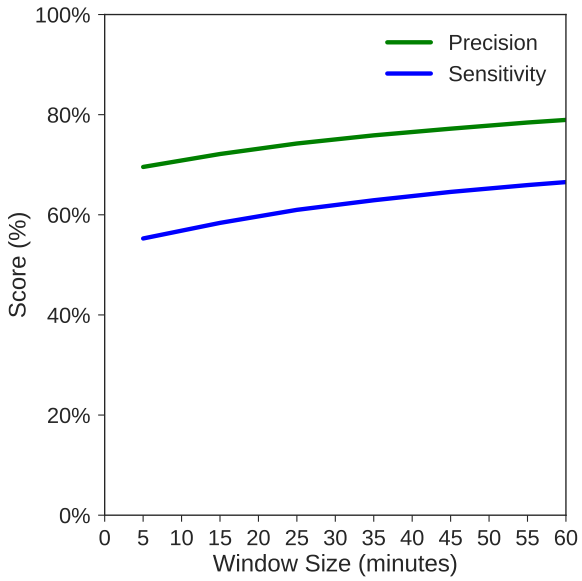


Figure 3.14: Sensitivity and precision of our IO burst prediction across windows ranging from 5 minutes to 60 minutes using our predicted turnaround time from our simulated system.

Our results in this section indicate that runtime and IO predictions from PRIONN can be used to generate accurate predictions of HPC system IO bandwidth and IO bursts. We show that with up to 57.9% mean turnaround prediction error, future IO bursts can be predicted with ¿50% accuracy. Additionally, with up to 36.4% mean system IO prediction error, future IO bursts can be predicted with ¿50% accuracy.

3.5 Summary

We show the benefits of using PRIONN, our tool to Predict Runtime and IO using Neural Networks, for an IO-aware scheduler over previous methods. Our tool relies on our methods for mapping the text of whole job scripts into image-like representations. PRIONN utilizes the image-like representations to train a 2D-CNN model for accurate per-job runtime and IO resource prediction. We achieve accuracies for runtime and IO resource predictions that exceed previous work. Specifically, we predict runtime and IO resources with over 75% mean and 98% median accuracy across

nearly 300,000 jobs from a large cluster at LLNL. We apply the predictions from PRIONN to an HPC system simulator to accurately predict system IO and IO bursts. We correctly predict over 70% of future IO bursts (i.e., times of IO contention). Our work demonstrates that PRIONN can augment schedulers with the per-job resource usage knowledge necessary to enable IO-aware scheduling.

Chapter 4

MITIGATING IO CONTENTION WITH CANARIO

We describe our second AI4IO tool, CanarIO, that provides job IO-sensitivity system IO contention predictions to schedulers for mitigating the effects IO contention.

4.1 Limitations of IO Contention Mitigation

Previous work on mitigation of resource contention has relied on methods for identifying when and how contention occurrs and making corrective actions to fix the issues it causes. In this section we review these methods for measuring performance of resources and mitigating contention. We then demonstrate how CanarIO is novel in its approach to mitigating IO contention

Parallel File Systems (PFS) performance work has relied heavily on the development and use of benchmark tools, such as IOR [37] [36] [18]. These benchmarks mimic a variety of real HPC workloads and can measure different aspects of the file system performance [6], but often they are run only once per day or week. This limited sampling does not provide the ability to detect hour-to-hour fluctuations that can occur due to system failure and IO contention that user-submitted jobs regularly experience. In addition, existing production monitoring tools focus primarily on detecting hardware failures and system software crashes as opposed to inter-application IO contention [33]. This ultimately leads to PFS with performance variability that is mostly invisible to system administrators.

Many studies have also directly address IO resource contention. We reviewed the work on HPC resource contention in Section 2.2. In addition to these works several studies have focused on HPC IO contention and understanding the complexities of IO performance behavior. These range from understanding IO for general HPC

workloads [21] to more specialized workloads and applications [13] [14]. In addition, the specific root causes of IO performance fluctuations have been investigated [43]. While understanding the cause of IO performance is important, there is a lack of work that focus on the ability to provide relevant information to a scheduler for mitigating negative effects from IO performance degradation.

With CanarIO, we have a novel reactive approach to mitigating IO performance loss, in that CanarIO is capable of predicting whether a job is IO-sensitive (i.e., is affected by IO contention) and giving realtime predictions of IO contention occurring on the system. CanarIO addressed the weaknesses of previous work described in this section. Specifically, we adapt the deep learning models from PRIONN, described in Chapter 3 to predict job IO-sensitivity. In addition, we adapt previous work on PFS performance measurements so that CanarIO can provide realtime system IO contention predictions. Our IO detection is novel in that we probe the system for performance data at a much higher frequency than previous work and use the mined data to train CanarIO's prediction models. In addition we collect metric data from our canary jobs. Specifically, we collect IO usage data for an HPC system in 1-second intervals as part of the LDMS, an emerging tool to collect large volumes of per-node system data with little or no overhead [1]. While LDMS is widely used, its detailed data often goes unused due to lack of sufficient analysis techniques. In the remainder of this chapter, we describe our tool CanarIO, show that it addresses weaknesses of previous work, and demonstrate its applicability mitigating IO contention with IO-aware scheduling.

4.2 Identifying IO Degradation

The task of identifying IO-related slowdown in jobs (also called IO-related performance degradation) is non-trivial. We leverage a set of "Canary" jobs (i.e., IO-proxy applications that are sensitive to IO performance) that run at regular intervals on the system. We cross-analyze the IO usage patterns of those jobs with PFS log files to validate the IO causality and understand the IO problem at the root of a slowdown.

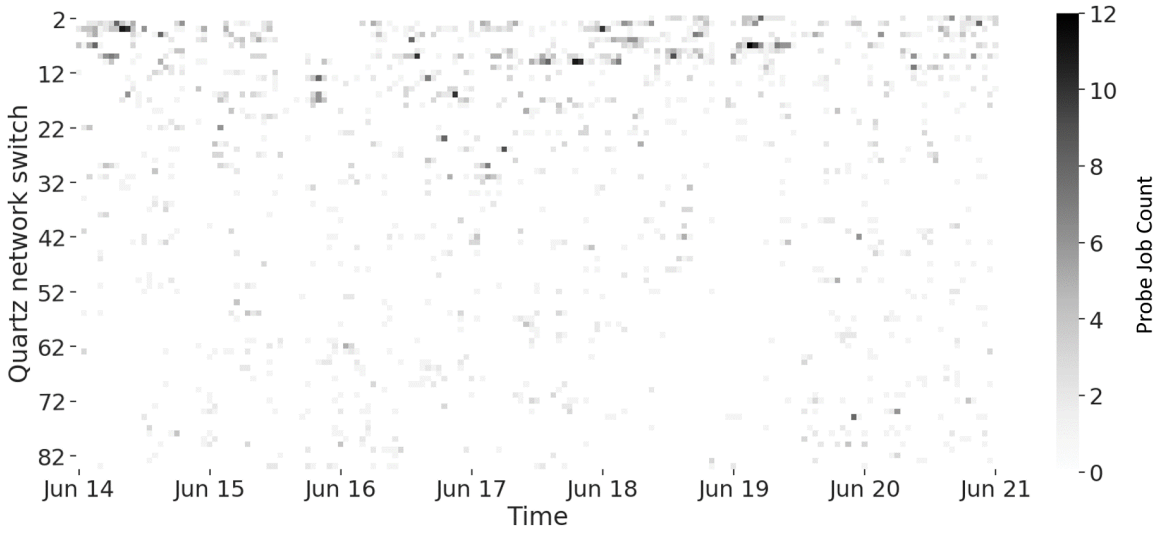


Figure 4.1: A heatmap showing the time of execution and network switch used for our probe jobs over an average week.

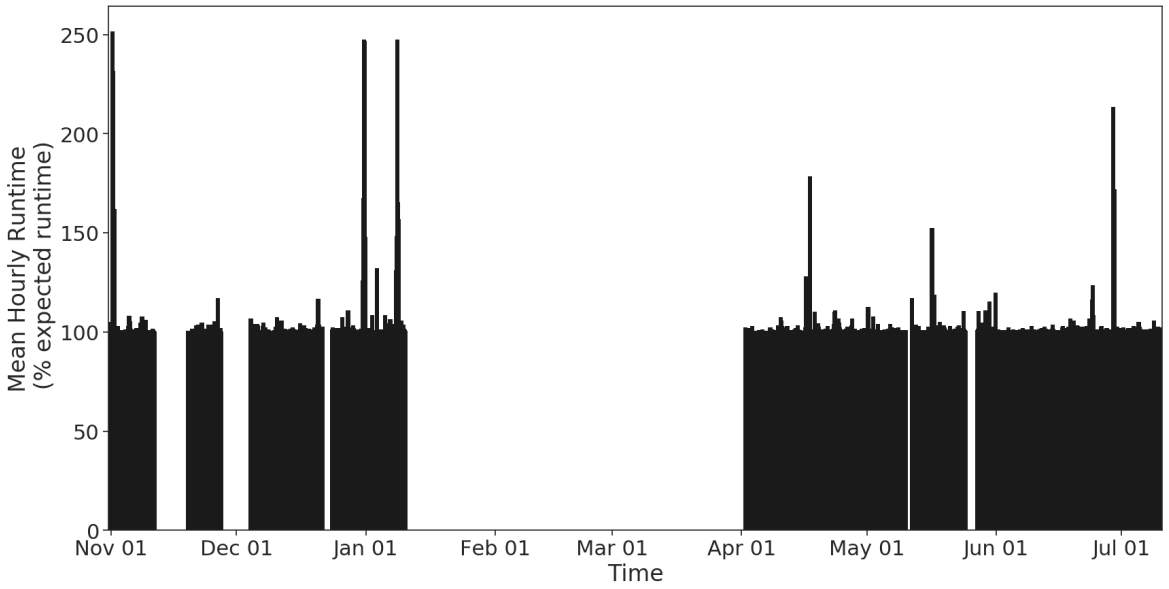


Figure 4.2: Runtime of our "Canary" probe jobs over time. The spikes above 100% indicate a longer runtime than expected and the detection of a possible IO-related degradation event.

4.2.1 IO Probing Jobs

We use a set of Multi-purpose, Application-Centric, Scalable I/O Proxy Application (MACSio) jobs, run at regular intervals, to collect high-granularity IO-related performance data from the system [26]. It is common practice to run benchmarking applications, such as MACSio, on HPC systems to collect information about the performance of system resources; however, these benchmarks are commonly run, at most, once per day. We model five common IO patterns in five MACSio probes. Each probe is a single node, 36 process execution of MACSio. We modify parameters for MACSio that affect the total IO, number of output files per process, and number of Lustre Object Storage Targets (OSTs). The five probes and these details are listed in Table 4.1.

Table 4.1: The 5 probe jobs that we run to detect IO degradation events on the Quartz system at LLNL model common IO access patterns.

Benchmark	Total Write IO (GB)	Files Per Process	Expected Runtime (s)	Description
Probe-1	100	10	1396	Few, large outputs
Probe-2	100	100	1023	Many, small outputs
Probe-3	2	100	519	Many, tiny outputs
Probe-4	100	10	521	1, w/ Max LSF striping
Probe-5	100	100	1408	2, w/ Max LSF striping

Each probe is submitted to the system queue as a single-node job. We maintain five jobs for each probe in the system queue at all times to collect IO performance data from the system on a regular and frequent basis. Additionally, we allow the Slurm system scheduler to randomly place our jobs across system nodes, ensuring more complete sampling of nodes and network switches on the system. Figure 4.1 shows a heatmap indicating where and when probe jobs ran during a week of monitoring. We achieve good coverage of the system topology and maintain a running probe job during most time intervals.

The overhead for running our probe jobs is minimal for the machine we tested on. For the jobs shown in Figure 4.1, a total of 571 node-hours were used over 7 days

to run our probe jobs. This amounts to just 0.11% of node-hours available on the machine. The one week time frame used in Figure 4.1 represents a week where more probe jobs are run, as compared to a typical week. If we expand the time window to the entire 8 months of data collection, only 0.04% of all cluster node-hours were used to run CanarIO probes. In addition to using negligible machine time, our probe jobs are often scheduled as back-fill jobs. Given their short runtime and capability to fit into small back-fill windows, they have limited impact on the scheduling and execution of other jobs on the machine. Put another way, our probe jobs often use resources that would remain idle while larger jobs wait for all allocated nodes. Overall, as we demonstrate in Section 4.6, the potential for saved node-hours is far larger than the node-hours needed to run the CanarIO probe jobs.

4.2.2 Job Performance Degradation

We define any job as being performance degraded base on the comparison of observed job runtime and expected job runtime. Each of our probes has an expected execution time, shown in Table 4.1. The expected runtime is calculated by taking the median runtime of the first 100 executions for each probe. We use a threshold runtime of 105% the expected runtime to define performance degraded jobs (i.e., jobs exceeding the threshold runtime are performance degraded). Figure 4.2 shows the runtime of over 20,000 probe jobs, averaged by hour, over 8 months ranging from November 2018 to July 2019. The figure shows small daily and hourly fluctuations in observed probe runtime with several large, isolated spikes that last several hours or days. Taller and wider spikes represent more severe performance degradation. We aim to detect both small and large performance degradation events with CanarIO. The gaps in the data are due to the temporary downtime of our data collection pipeline.

With the collected data, we investigate the cause of observed slowdowns in probe executions. We observe that none of the five MACSio probes are computationally, memory, or network intensive. Therefore, when a slowdown is observed it is assumed due to file system IO performance. We investigate the validity of our assumption

by analyzing the IO usage data for each probe job and matching probe execution slowdowns with Lustre PFS log files and empirically supporting the claim.

We collect LDMS data for probe jobs and analyze the IO usage patterns of each executions. Figure 4.3 shows an example of read and write IO bandwidth usage for two Probe-4 jobs. The top plot of Figure 4.3 shows the IO usage over execution time for a Probe-4 that runs for the expected runtime while the bottom plot of Figure 4.3 shows the IO usage over execution for a Probe-4 that runs during the first large performance degradation spike in Figure 4.2. Expected executions have regular intervals of computation and IO activity, indicated by the regular large spikes in write bandwidth usage. Performance degraded executions, on the other hand, have regular intervals of computation, but irregular intervals of IO activity. This is indicated by the smaller and irregular spikes in write IO during this execution. The IO degradation experienced by the job in the bottom plot of Figure 4.2 caused more than a 3X increase in runtime and the job was ultimately killed for exceeding the allocated runtime.

The log files from the attached Lustre PFS are analyzed to support the IO related cause of our probe slowdowns. Figure 4.4 shows the status of each MDS and OST of the Lustre file system attached to the system (bottom) and the performance degradation of probe jobs (top) during a 12-hour time interval on November 1, 2018. In the top plot of Figure 4.4, we observe a large spike in the execution time of our probe jobs that occurs suddenly and subsides several hours later. In the bottom plot, we observe a correlated crash of nearly every node on the Lustre PFS at the same time as the spike in probe execution time. After the crash, many nodes go into extended periods of the recovery phase and some Lustre nodes crash again. During this time, we see that our probe job performance can be either degraded or normal (i.e., having expected runtime). This indicates that Lustre logs alone are not reliable for determining IO performance degradation.

We observe further evidence that Lustre logs are unreliable for detecting IO performance degradation in Figure 4.5, which shows the IO degradation detected by CanarIO probe jobs (top) and the status of MDS and OST of the Lustre file system

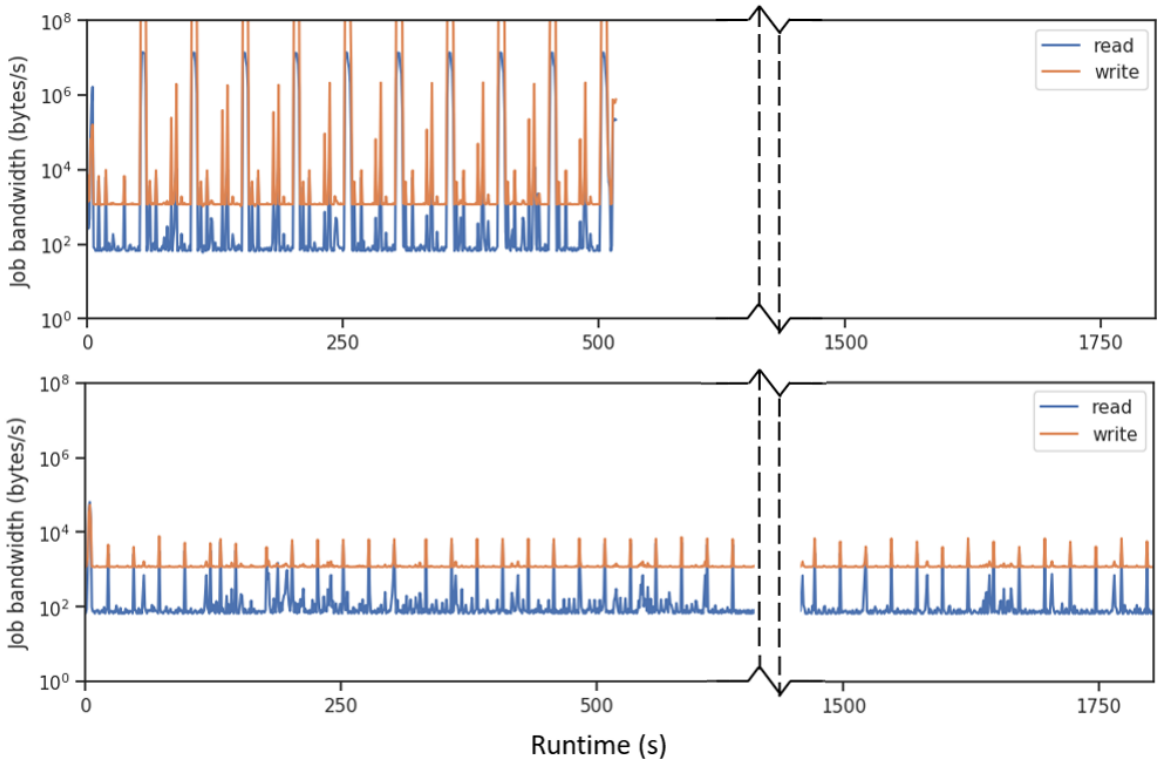


Figure 4.3: Comparison of IO bandwidth usage for two Probe-4 jobs. The expected runtime execution (top) and performance degraded execution (bottom) show distinctly different IO patterns that indicate IO-related performance degradation.

(bottom) for several weeks during May and June of 2019. The performance degradation events detected by our probe jobs are not detected in the Lustre log files. In addition to the CanarIO probe jobs detecting more IO performance degradation, they also provide the ability to quantify the severity of performance degradation.

From the analysis of LDMS IO usage data for our probe jobs and Lustre log data during performance degradation events, we gain confidence in the CanarIO probe jobs to detect and quantify the severity of IO-related performance degradation. We confirm that performance degradation of probe Jobs is related to IO and that our jobs can detect IO performance degradation that is unseen in Lustre logs. Our analysis also demonstrates that the sensitivity to IO and high periodicity of sampling for CanarIO probes are valuable for correctly labeling historical job and system data and ultimately building accurate prediction models in Sections 4.4 4.5.

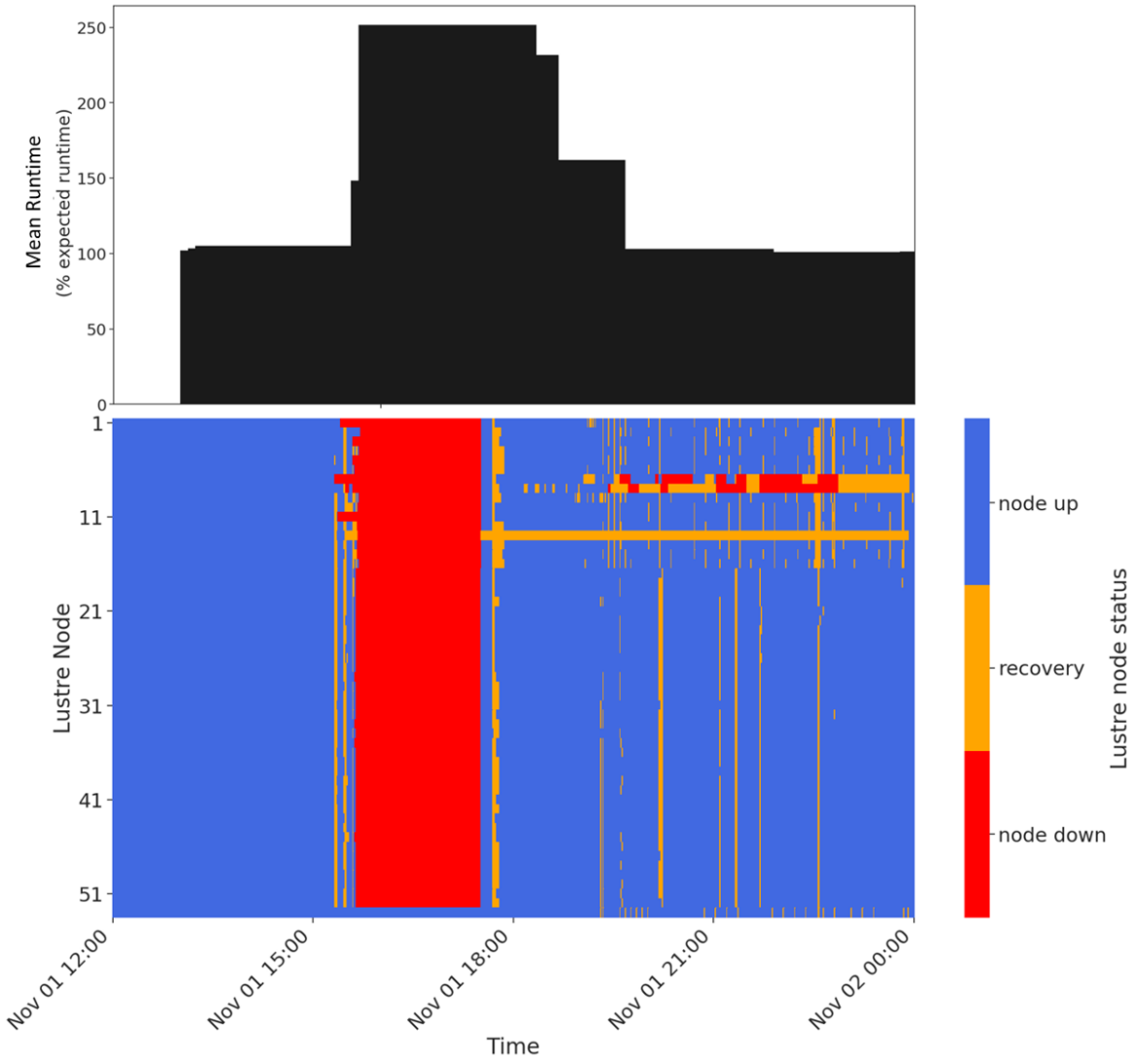


Figure 4.4: The measured runtimes of our probe jobs (top) and Lustre node status (bottom) for a 12-hour window show the correlation between performance degradation of our probe jobs and a Lustre crash.

4.3 Identifying IO Sensitive Jobs

Generating actionable data with CanarIO that enable schedulers reaction to the problems associated with IO degradation events means also identifying the user-submitted jobs that are affected by these events. Because we do not have access to the application codes and input decks of the user-submitted jobs, determining the IO sensitivity of a job is not trivial. Furthermore, the IO data we collect from LDMS is

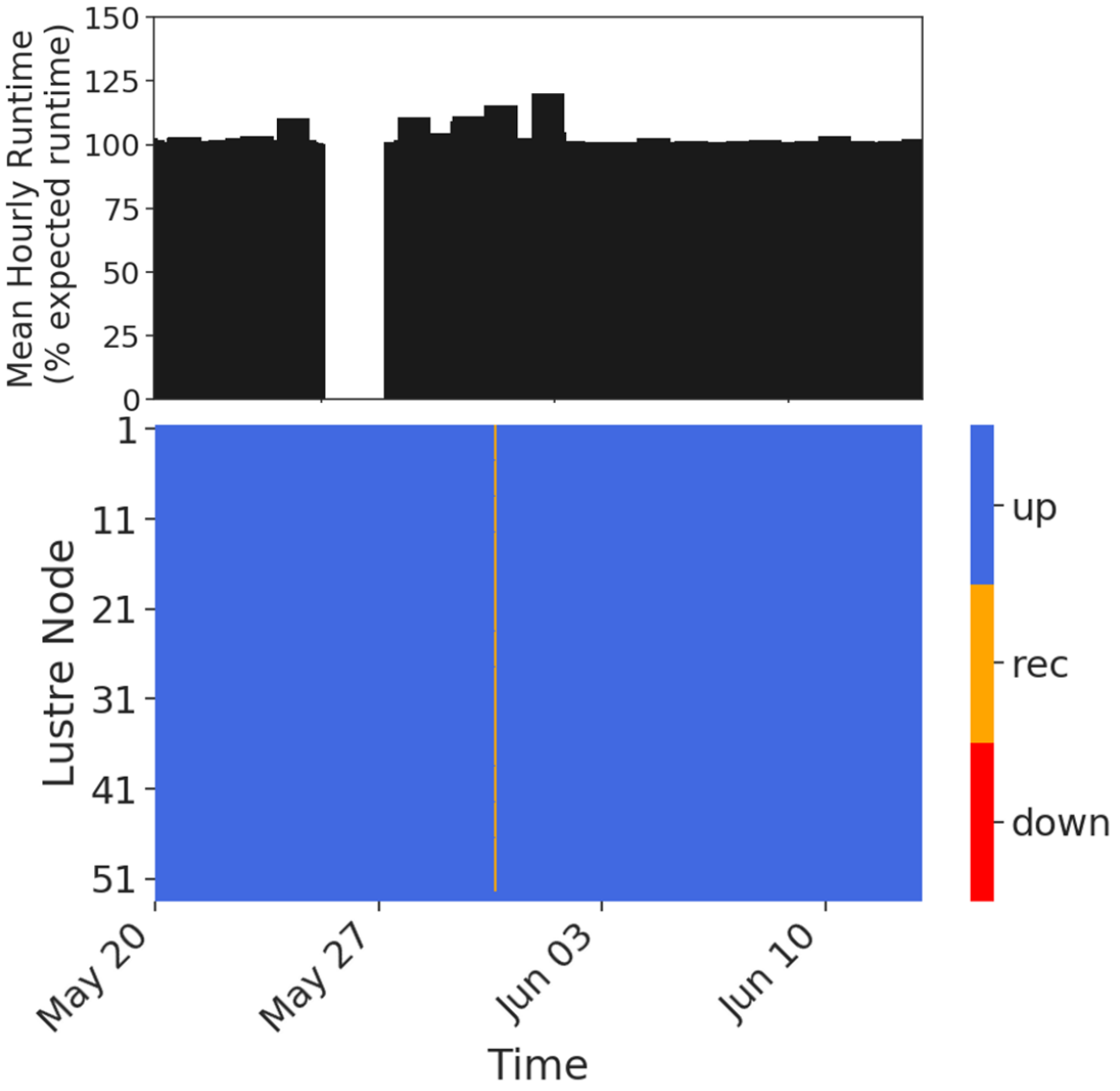


Figure 4.5: Lustre log files do not always capture performance degradation events. The top plot shows several instances of IO degradation events from increased CanarIO probe runtimes. The bottom plot shows the status for Lustre OST and MDS nodes. We note that only a small band of recovery status is seen in the log files.

a total count of bytes sent or received by a node, including network activity. Lustre specific LDMS counters are implemented in LDMS v3, but that is not supported by the Lustre system attached to the Quartz cluster at LLNL. As a result, we define a method for identifying IO-sensitive jobs that is a set of nested filters that rely on job metadata, data from our probe jobs described in Section 4.2, and LDMS IO usage

data. We apply our method to a dataset of jobs collected from the Quartz cluster at Lawrence Livermore National Laboratory (LLNL). Our method can be broadly applied to create datasets with two classes of jobs: those that are IO-sensitive and those that are IO-resistant.

We initially consider a dataset of 767,999 jobs run between November 2018 and July 2019 on Quartz. For each job, we collected the job script text, user ID, job ID, job name, start time, end time, requested runtime, and node count. We identify and remove in our datasets all the jobs that terminate at their initialization. These jobs are identified by runtimes of less than 1 minute. Additionally, we remove jobs that occupy the system for less than 30 node-minutes. We reason that these low resource jobs have negligible impact on the IO degradation we study. This results in a reduced-in-size dataset of 658,229 jobs. Using the job script, user ID, job name, and node count as the identifier of a job, we count a total of 126,310 unique jobs in the dataset.

Following the same approach applied to our probe jobs in Section 4.2, we calculate the expected runtime of jobs in the dataset. For each job we compare the observed runtime to the expected. Because we do not have the application code and input decks for the user-submitted jobs in our dataset, we must infer the expected runtime based on observations of jobs that are run multiple times. We remove jobs that were run only once from our dataset and analyze the remaining 18,538 unique and 390,726 total jobs. We calculate the expected runtime for each unique job as the minimum observed execution time.

Real user-submitted jobs may experience performance degradation for a number of non IO-related reasons. We reduce the probability of selecting these jobs with non IO-related performance degradation using information collected from the probe jobs in Section 4.2. Because our probe jobs are only IO-intensive, we can deduce user-submitted jobs with IO sensitivity (or resilience), if they are executed at the same time as our probe jobs. By considering only user-submitted jobs that overlap in execution times with one or more of our probe jobs, we further reduce our dataset to 7,557 unique jobs and 69,582 total jobs.

We label our dataset of user-submitted jobs as IO-sensitive or IO-resistant based on the expected runtimes and information from our probe jobs. If the runtime of a job is 5% larger than the expected runtime, we label it as performance degraded. For each unique job that has at least one execution that demonstrated performance degradation, we assert that the performance degraded executions must overlap with a probe job that also exhibited performance degradation. These jobs are labeled as IO-sensitive. Similarly, we define IO-resistant jobs as jobs that show no performance degradation but also have one or more executions that overlapped with a performance degraded probe job. We identify a total of 6,358 unique jobs and 47,969 total jobs that match our criteria for IO-sensitive and IO-resistant jobs.

As a final validation, we consider the IO usage data for each job in our dataset. We calculate the total read and write IO usage of each job and compare these values for executions of each unique job. For IO-sensitive jobs, the total IO of expected runtime executions must be equal to or greater than the total IO of performance degraded executions. This ensures that runtimes greater than the expected runtime are not caused by parameter changes in job input decks that generated more IO activity. The resulting dataset contains 4,906 unique IO-resistant jobs and 1,293 IO-sensitive jobs.

4.4 Predicting IO Sensitive Jobs

Our method for defining (i.e., labeling) IO-sensitive jobs in the previous section provides us the dataset to build the first of two predictive model components in CanarIO. In this section, we build a machine learning model for predicting whether a job is IO-sensitive or IO-resistant. We use the real queue, start, and end times of the jobs in our dataset to replicate the queuing and execution of jobs on the HPC cluster. Data from historical jobs (i.e., jobs that have already executed) are used to train the machine learning model. The trained model is then used to predict the IO sensitivity of jobs as they are submitted to the cluster (i.e., before execution). Figure 4.6 shows our complete workflow for online training and predicting of IO-sensitive and IO-resistant jobs.

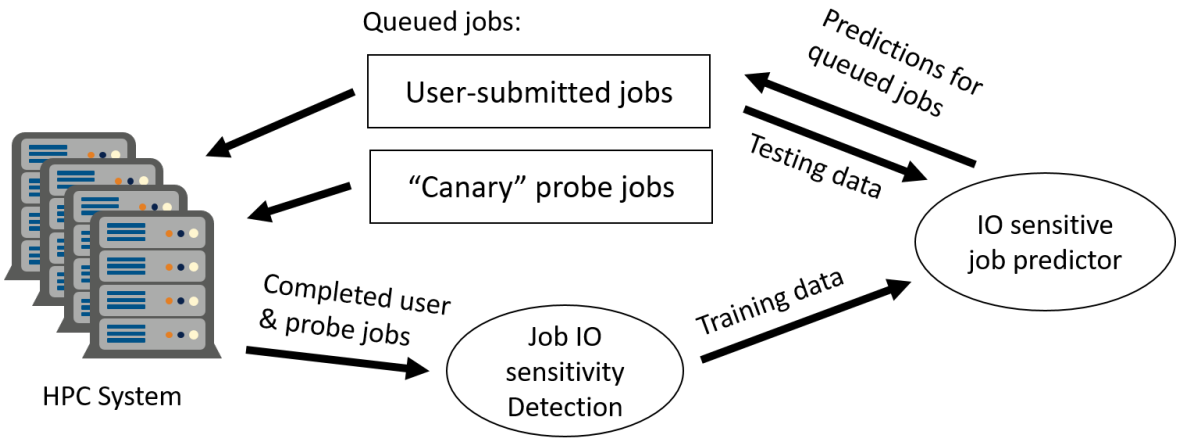


Figure 4.6: The dataflow for training and evaluating the job IO sensitivity model of CanarIO. Completed probe and user-submitted jobs are used to detect IO-sensitive jobs. These jobs are used to train the ML model and provide predictions for queued jobs.

We incorporate the deep Convolutional Neural Network (CNN) from the PRIONN tool [44] into CanarIO for the purpose of predicting IO-sensitive and IO-resistant jobs. This model is ideal for its ability to use all the information available in user-submitted job scripts to make accurate predictions about job resource usage. We maintain the CNN layer structure described in the PRIONN paper and adapt the output layer of the model to make the binary prediction for a job being IO-sensitive or IO-resistant from only job scripts.

We use an emulator to replay the submission and execution of real HPC jobs on the system to simulate how this model would make predictions when used online with a scheduler. Figure 4.7 shows how training data and testing data are chosen during the replay of HPC jobs on the system. We run the emulation of job submission and execution until 100 unique jobs are available as training data. Here, we use our method described in Section 4.3 to identify jobs that are either sensitive or resistant to IO. We then train the model and make predictions for newly submitted jobs. As jobs complete execution, we label them using our methods from Section 4.3, the training data grows in size, and the CNN is periodically retrained.

Our dataset of jobs collected from Quartz is used to evaluate the prediction of

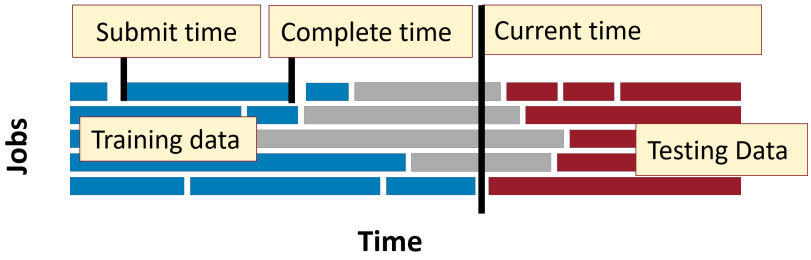


Figure 4.7: Emulation of the system for model evaluation involves replaying the execution of jobs. Training data for the model is selected from jobs that have completed execution and testing data comes from queued jobs.

job IO sensitivity with CanarIO. Because our methods require the intersection of three independently curated datasets, our evaluation is limited to two weeks of system time split between 2 time intervals. Figure 4.8 shows the cumulative True Positive % (i.e., percent of correctly predicted IO-sensitive jobs) and True Negative % (i.e., percent of correctly predicted IO resistant jobs) for the two evaluation time ranges. We observe that we maintain a prediction accuracy >90% for both IO-sensitive and IO-resistant jobs.

We shift our analysis to understand the potential impact of CanarIO for avoiding wasted resources during IO degradation events. Figure 4.9 shows the confusion matrix for all predictions in the evaluation period of Figure 4.8. The 7,687 correctly predicted IO sensitive jobs account for a total of 9,895 node-hours. Considering the case of an IO degradation event occurring, the predictions from CanarIO can be used to swap IO-sensitive jobs for IO-resistant jobs on the system and theoretically prevent the waste of nearly 10,000 node-hours. We further evaluate CanarIO's potential resource savings in Section 4.6.

4.5 Predicting IO Degradation

Making the predictions of job IO sensitivity from the previous section actionable (e.g., replacing IO-sensitive jobs with IO-resistant jobs during IO degradation) requires also knowing, in real-time, when IO degradation events are occurring. To this end, we develop the second predictive model of CanarIO for detecting real-time IO degradation

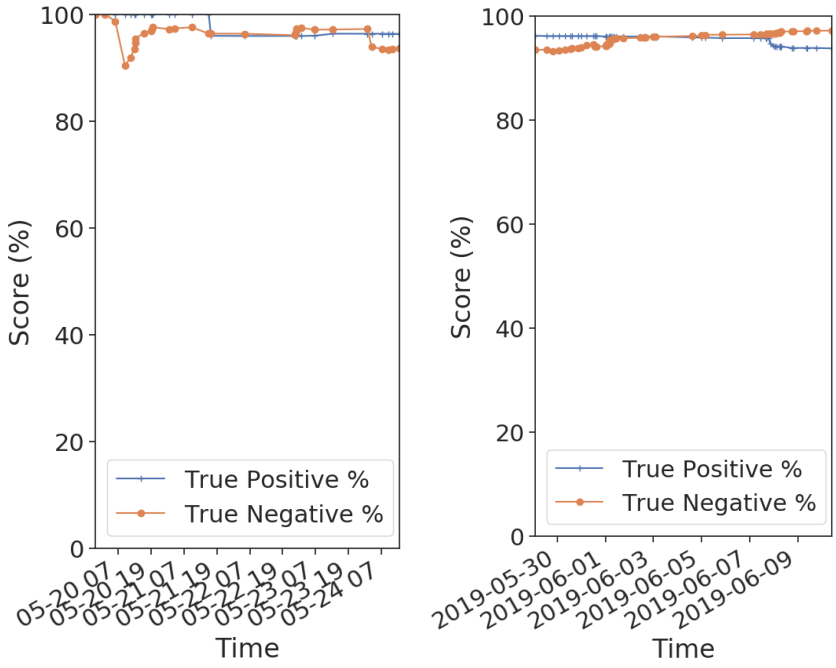


Figure 4.8: True positive and true negative prediction rates for the CanarIO job IO sensitivity prediction model for two evaluation periods.

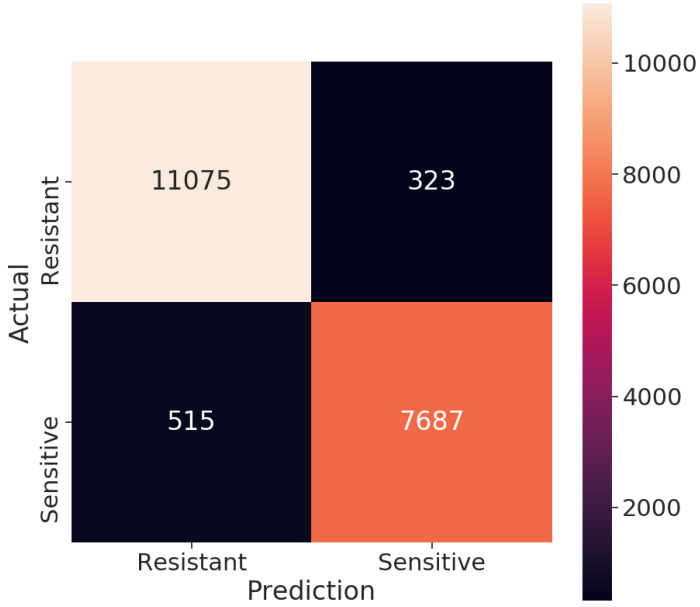


Figure 4.9: Confusion matrix showing the IO sensitivity prediction results for all jobs in our evaluation dataset.

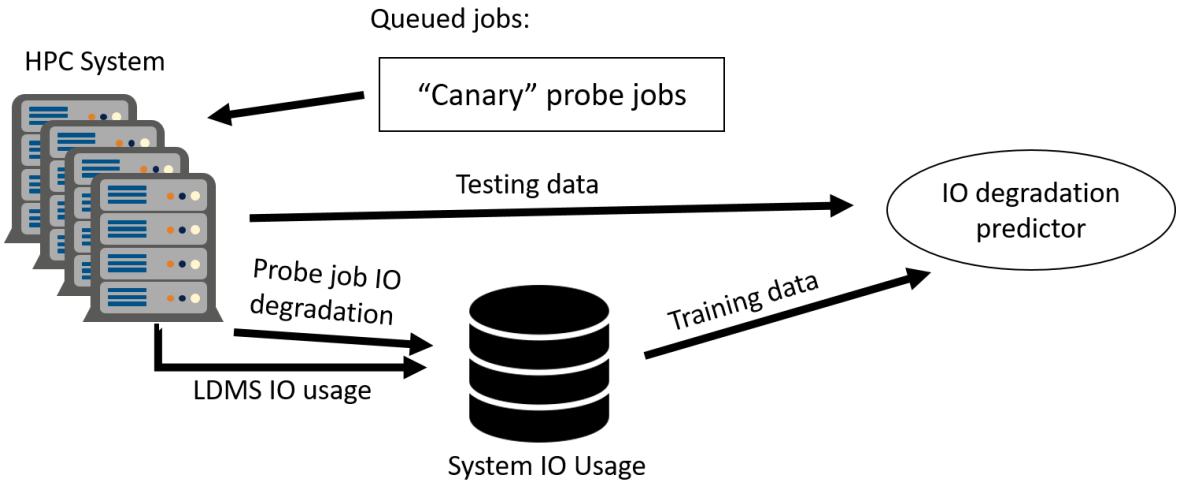


Figure 4.10: The dataflow for training and evaluating the IO degradation model of CanarIO. Probe jobs are run and the IO degradation events are recorded with LDMS system IO usage data in 1-minute observation windows. A bank of training data is collected and used to train the predictor. Predictions are made on new system IO observation windows.

events. We leverage a supervised machine learning model, LDMS IO usage data, and data from our probe jobs in Section 4.2. Similar to Section 4.4, We emulate the execution of real HPC jobs on a system for training and evaluating the model. Figure 4.10 shows our complete workflow for online training and predicting of IO degradation events.

The CanarIO prediction pipeline in this section is motivated by the observed correlation between system IO usage data and probe job performance degradation. In the example from Figure 4.4, we noted that when the file system nodes go offline and there is a spike in probe job execution time, the system IO drops and fluctuates significantly. We also observe that probe jobs provide accurate measurements for IO-related performance degradation only when their execution completes. However when there is IO-related performance degradation, probe runtime is extended and there is significant delay in detecting performance degradation from the probes alone. Under such circumstances, we must leverage system IO usage to accurately detect real-time IO-related slowdowns.

The LDMS IO usage collects system IO by node, in one second intervals (i.e., each node records a value for read and write usage every second). We define an *observation window* of one minute and split the stream of IO usage data into one minute intervals. Each observation window is a time series of data that describes the IO patterns occurring on the system. The minimum, maximum, median, mean, variation, and standard deviation of read and write IO usage for each observation window are calculated at the time the observation window of data is received. These values are used as a feature vector to describe the IO usage of each observation window. The feature vectors of past observation windows are labeled by the observed performance degradation of completed probe jobs and used to train a machine learning (ML) model to predict IO degradation for the current observation window.

We emulate Quartz system execution using the collected LDMS IO usage data and data from our probe jobs for a four-week time period. We accumulate an initial training dataset containing 24-hours of system IO. The observation windows are labeled based on the performance degradation observed from our probe jobs. For an observation window, we calculate the mean runtime of all the probe jobs executing in that observation window. If the runtime is greater than a *threshold* % of the expected runtimes, we label the observation window as being an IO degradation event. We make predictions for each new observation window and then add that observation window to our training data. The model is retrained every 10 minutes of system time.

We test three ML models for this prediction task: Random Forest (RF), Decision Tree (DT), and k-Nearest Neighbors (k-NN). We measure the F1, Precision, and Recall of predictions over the entire dataset. Figure 4.11 shows these metrics for each ML model. The F1 score is a weighted average of Precision and Recall and is superior for measuring prediction accuracy with unbalanced data, such as our data (i.e., many more samples of no IO degradation). We observe that k-NN has the highest F1 and Recall scores, and select k-NN as the predictive model for system IO degradation in CanarIO.

The F1, Precision, and Recall scores of our k-NN model is measure with different

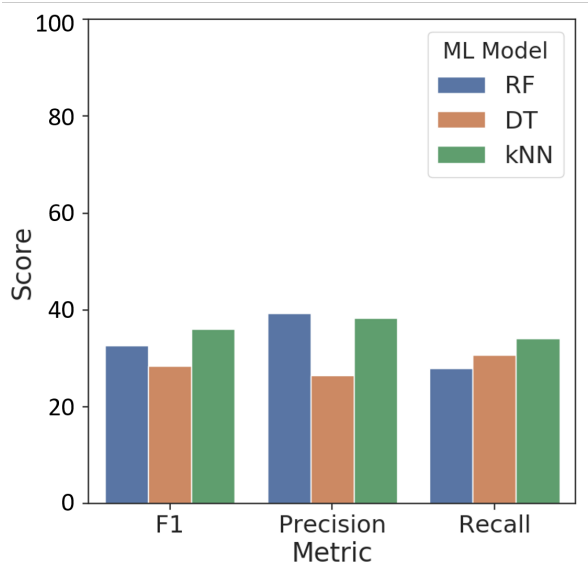


Figure 4.11: Comparison of machine learning models (i.e., Random Forest, Decision Tree, and k-Nearest Neighbors) for predicting IO degradation using 3 metrics (higher is better).

threshold definitions of IO degradation, as described above. Figure 4.12 shows these metrics for IO degradation defined by >0% up to >100% the expected runtime of our CanarIO probes. Values greater than 5% show consistent performance.

Figure 4.13 shows the True Positive % (i.e., percent of correct degraded observation window predictions), False Positive % (i.e., percent of incorrect non-degraded window predictions), True Negative % (i.e., percent of correct non-degraded observation window predictions), and False Negative % (i.e., percent of incorrect degraded observation window predictions) at different threshold values for our defined IO degradation events. As in Figure 4.12, we observe that at IO degradation thresholds larger than 5% (i.e., an observed probe runtime greater than 105% the expected runtime) our predictions are consistent. We achieve >95% accuracy for non-degraded observation windows and up to 37.5% for degraded observation windows. At 0% threshold value, we see that the true positive rate is very high, but that the true negative rate drops significantly and the false negative rate rises significantly. Misidentification of non-degraded time windows for degraded time windows is severely detrimental in the

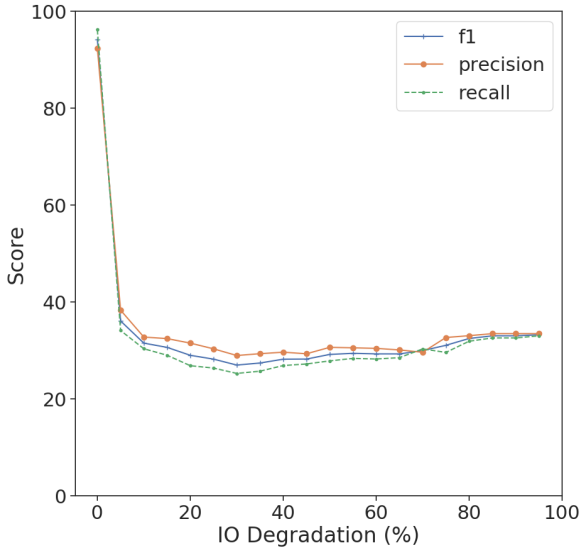


Figure 4.12: F1, precision, and recall scores for detection of IO degradation events using k-NN for different threshold values of defining an IO degradation event.

context of making scheduling decisions with CanarIO predictions . This indicates that using the 0% threshold would produce worse results for CanarIO (despite the large F1, precision, and recall scores).

Figure 4.14 shows the performance degradation % for all predictions on degraded observation windows with a threshold value of 5%. The whiskers of the box-plots extend to the 5 and 95 percentiles. We observe that the degraded observation windows which our model incorrectly predicted as non-degraded (i.e., false negatives) have more small degradation events than the correctly prediction observation windows. This indicates that CanarIO detects severe IO degradation events more reliably then minor degradation in our dataset.

4.6 CanarIO In Action

To evaluate the effectiveness of each CanarIO component workings together to recover valuable lost node-hours from IO performance degradation, we simulate a 10-day window of the system and use CanarIO to assist scheduling decisions. This 10-day window was chosen based on the data available to us and because it includes both minor and major performance degradation events. Our simulation is built on a simple

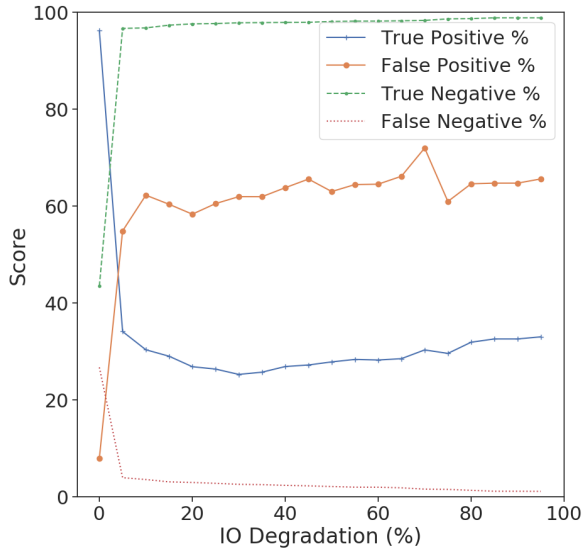


Figure 4.13: True positive, false positive, true negative, and false negative prediction rates for detection of IO degradation events using k-NN for different threshold values of defining an IO degradation event.

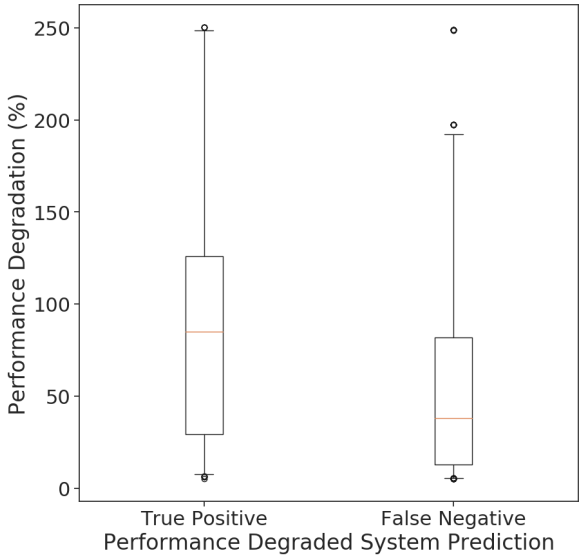


Figure 4.14: Comparison of the size of IO degradation events for correctly predicted (true positive) and incorrectly predicted (false negative) IO degradation events. We correctly detect most large IO degradation events.

discrete event simulator, which uses job queue, start, and end times to simulate the queuing and running of jobs on the system. The IO-sensitive job predictor, described in Section 4.4, and the system IO degradation predictor, described in Section 4.5, are trained and used to make predictions about the system during this 10-day simulation.

We modify the system job schedule using CanarIO predictions. When IO degradation is predicted by CanarIO, we use the CanarIO job IO sensitivity predictions to determine which currently running jobs are IO-sensitive and replace them with IO-resistant jobs from the queue. In doing so, we avoid wasting system node-hours on jobs that cannot make progress due to IO performance degradation. We recover the wasted node-hours by placing IO-resistant jobs, which do not rely on the PFS IO performance, on the machine. We use a greedy approach for replacing IO-sensitive jobs. We replace the ones with the smallest amount of used node-hours first to minimize the impact of false negative predictions.

We track two metrics: node-hours recovered and cumulative node-hours spent. The node-hours recovered metric is a calculation of the number of node-hours that are saved from using CanarIO to manipulate the job schedule. For this metric, we consider three impacting factors: 1) The node-hours recovered by running IO-resistant jobs during the performance degradation event; 2) The impact of false positive predictions (i.e., removing IO-resistant jobs from the system prematurely); and 3) The impact of false negative predictions (i.e., replacing an IO-sensitive job with another IO-sensitive job). Considering these three factors, we calculate the amount of system node-hours saved (or lost) using CanarIO (i.e., $\#1-(\#2+\#3)$). The cumulative node-hours spent metric measures the cost of running CanarIO probe jobs on the system. We track the cumulative node-hours used by our probe jobs to provide CanarIO the necessary information for making predictions.

Figure 4.15 shows the two metrics described above and our probe job performance degradation over a 10-day time window. We observe that initially, there are no large performance degradation events and the node-hours spent on probe jobs is greater than the node-hours saved. On June 29, 2019, we observe a large performance

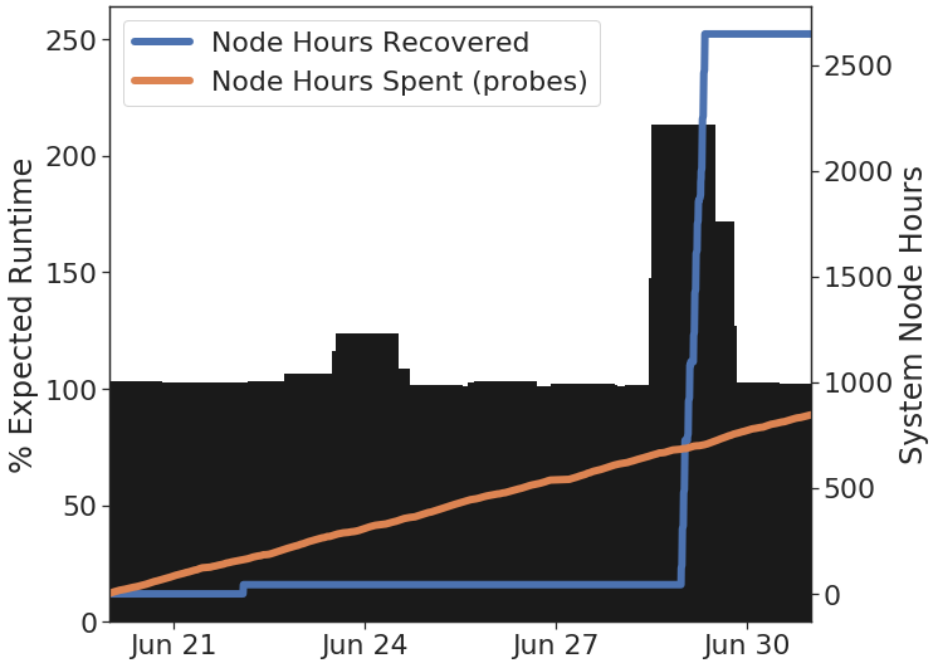


Figure 4.15: The results of a 10-day system simulation that uses CanarIO to modify the job schedule during performance degradation events. The left y-axis displays the performance degradation measured by CanarIO probe jobs (black). The right y-axis displays the the system node-hours spent on executing probes (orange) and saved from using CanarIO (blue).

degradation event. During this time, the CanarIO predictions are used to successfully recover over 2500 node-hours of wasted system resources. The amount of recovered node-hours is more than double the amount of cumulative node-hours spent on CanarIO probe jobs. This evaluation indicates how CanarIO can be successfully used to recover lost system resources due to IO performance degradation events. With further refinement for the application of CanarIO predictions to scheduling decisions, we believe system administrators can obtain much larger amounts of resources saved, such as the nearly 10,000 node-hours of potentially saved resources described in Section 4.4.

4.7 Summary

We show the benefits of using CanarIO, our tool for predicting job IO-sensitivity and realtime detection of IO contention. Our tool adapts the novel methods of training 2D-CNN models from entire jobs scripts, described in Chapter 3, and a novel data

collection method using canary probe jobs. We achieve an accuracy of $> 90\%$ for predicting IO-sensitive jobs and correctly detect 37.5% of IO degradation events in our evaluation. Additionally, we show that CanarIO can be used with IO-aware scheduling to recover more than 1,500 wasted node-hours in a 10-day span on a 2,600 node system by mitigating the effects of IO contention. Our work demonstrates that CanarIO can augment schedulers and enable effecting IO-aware scheduling.

Chapter 5

IO-AWARE SCHEDULING WITH AI4IO

In this chapter,

we evaluate the use of PRIONN and CanarIO in concert to provide IO-awareness individually and show that together they generate the greatest improvements in performance.

5.1 Extending the Flux Simulator

We use the validated open source Flux [3] scheduler and simulator as a starting point for building a realistic HPC environment for testing the use of PRIONN and CanarIO together. To this end, we extend the Flux simulator to capture the impact of IO contention, enable IO-aware scheduling, and integrate our AI4IO tools. These extensions to the Flux simulator provide the means for comparing traditional scheduling with AI4IO IO-aware scheduling in an HPC environment.

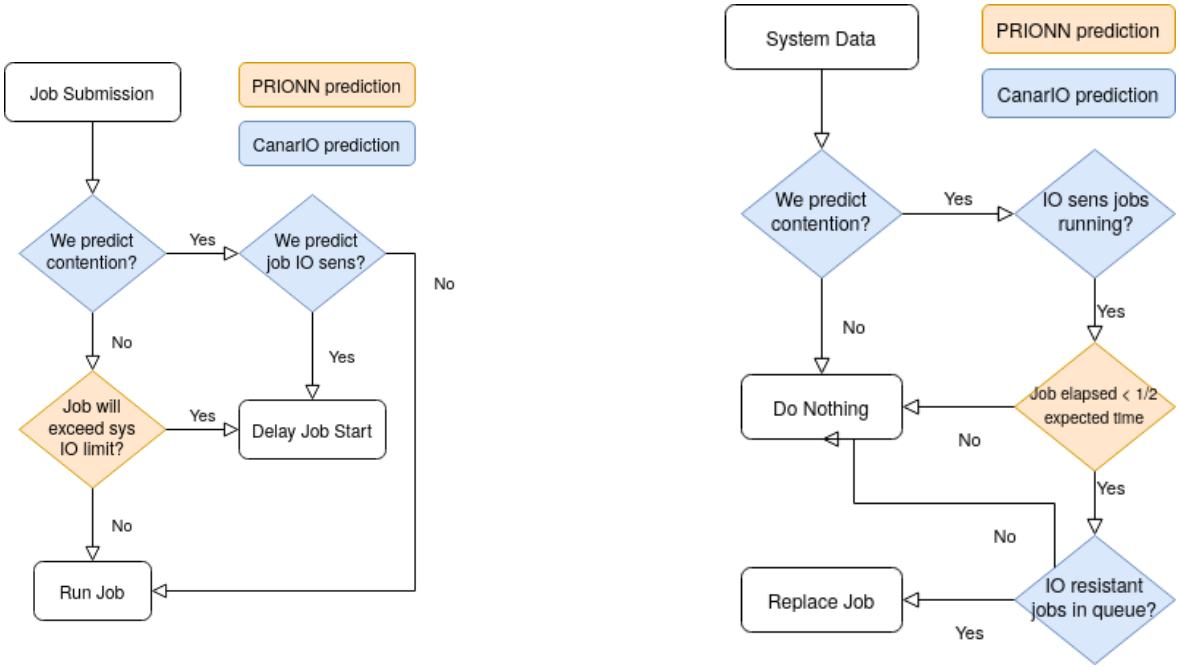
The available Flux simulator code is capable of simulating the scheduling and execution of jobs from an input containing the job resource usage (i.e., node count and runtime), submission time and timelimit. The Flux simulator creates a virtual HPC machine with a set of homogeneous nodes and an instance of the Flux scheduler that provides an interface to the virtual HPC machine. We extend the Flux simulator in three ways: (1) we add Parallel File System (PFS) IO as a simulated resource and develop an IO contention model that affects job execution; (2) we introduce mechanisms that mimic user response to increased job runtime; and (3) we integrate IO-aware scheduling with AI4IO tools.

Our first addition to the Flux simulator is the tracking of PFS IO and an IO contention model. We include the average IO bandwidth usage during execution for

each job and track the aggregate IO bandwidth demand of running jobs. We assume that all IO on the system is directed to an attached PFS. Therefore, we implement a limit to the IO bandwidth available to the entire system to mimic the maximum IO bandwidth we observe on LLNL systems and their associated Lustre PFSs. When the IO limit of the machine is less than the aggregate IO demand of executing jobs, IO contention occurs. IO contention is implemented in the Flux simulator, such that job runtimes of IO-sensitive jobs are increased. This contention model is based on previous contention modeling work in [10, 22, 27] and our observations in Chapters 3 and 4. Greater values of system IO demand above the IO limit create more severe IO contention. Contention ends when the IO demand drops below the IO limit. Additionally, IO contention events can be directly injected into the Flux simulator. This allows us to simulate the other sources of IO contention, such as partial outages of PFS resources. During the simulation, we inject IO contention modeled from IO contention events we observed in data collection from Chapter 4.

The second extension to the Flux simulator adds mechanisms to mimic user response to IO contention effects on jobs. Specifically, we emulate the action of users to rerun their jobs when they are cancelled for exceeding the timelimit. When IO contention extends job runtime, there will be jobs for which this causes the runtime to exceed the allocated timelimit. As a result, the entire job run or part of the job run may be wasted. Consequently, users will resubmit these jobs to complete the work. We observe this user behavior in our data collected at LLNL in Chapters 3 and 4. In the simulation, when a job exceeds its timelimit, it will be resubmitted, with a longer timelimit.

The third modification to the Flux simulator integrates IO-aware scheduling and our AI4IO predictions. To enable IO-aware scheduling, we develop decision logic that allows us to transform AI4IO predictions into job scheduling actions that prevent and mitigate IO contention. Figure 5.1 demonstrates the two pieces of integrated logic that generate IO-aware scheduling from PRIONN and CanarIO predictions. The added logic depicted in Figure 5.1a shows how we decide if a job should be delayed



(a) (b)

Figure 5.1: Flow charts showing the logic implemented in the Flux simulator to integrate PRIONN and CanarIO predictions for IO-Aware scheduling. In (a) we show how it is decided whether a job should start or be delayed and in (b) we show how it is decided when to replace IO-sensitive jobs with IO-resistant jobs when there is IO contention.

from executing. We first query CanarIO for information about IO contention on the machine and the IO-sensitivity of the job at the top of the queue. If we predict that an IO-sensitive job would be negatively impacted by IO contention, we delay the start and mitigate the impact of contention. In addition, we check whether the predicted IO usage of the job would cause the system IO to exceed the IO limit. If it does, we prevent contention by delaying the job start. When a job is delayed, it allows other jobs in the queue to be considered for running. Figure 5.1b shows the logic implemented to take action when CanarIO detects IO contention on the system. We query CanarIO for job IO-sensitivity predictions of jobs on the machine. If these jobs have run for less than half their expected runtime and there are eligible IO-resistance jobs, we replace the IO-sensitive jobs with IO-resistant jobs to mitigate the impact of IO contention.

In addition to implementing the IO-aware scheduling, we also integrate the predictions of our AI4IO tools with the Flux simulator. We implement an "AI4IO Oracle" that generates PRIONN and CanarIO predictions from a given desired prediction accuracy. This is beneficial over using the actual tools for two reasons: (1) we do not have all of the necessary inputs from the simulation to properly train PRIONN and CanarIO (e.g., LDMS IO usage). Modeling the hardware and counters necessary to provide this data in the simulation is outside the scope of this work; and (2) an Oracle allows us to precisely test a wide range of AI4IO prediction accuracy repeatedly over many data sets. The Oracle predictions use the real job values (e.g., IO and runtime) and desired prediction accuracy to generate the PRIONN and CanarIO predictions. Predictions are generated from a normal distribution of accuracy values centered at the desired overall accuracy (i.e., mean accuracy). The spread of these distributions is determined by fitting a normal curve to the PRIONN and CanarIO prediction distributions of real HPC data, measured in Chapters 3 and 4.

5.2 Simulated HPC Environment

Our modified Flux simulator has several parameters that impact our simulated HPC environment. In this section, we define the parameter values used for evaluation and show the changes to simulation contributed by each of the three extensions to Flux, described in Section 5.1. The HPC system modeled in our simulation is Quartz at LLNL. We simulate a reduced scale version of the Quartz machine, with 100 16-core nodes. We collect 10 sets of 1000-job workloads, sampled from real HPC job data. The job submission times are modified to be uniformly spread across a 12-hour window at the start of our simulation. IO contention caused by excessive IO demand causes IO-sensitive jobs to progress 10-20% slower. We inject a single IO contention event that slows IO-sensitive job progress by 80-90%. This contention event is modeled from the contention event observed on Quartz on November 1, 2018, shown in Figure 4.4. Jobs that exceed their allocated times are rerun with 50% probability. We use two settings for the Oracle prediction accuracies. We model the prediction accuracies from those

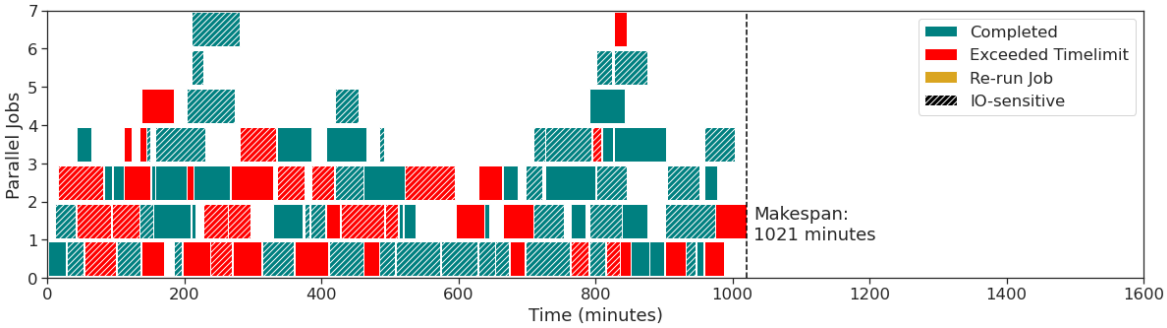
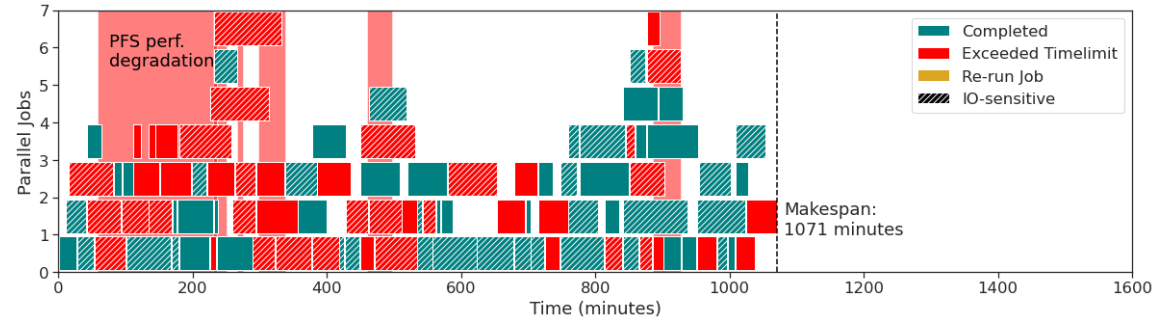


Figure 5.2: 100-job workload of HPC jobs on a 16-node cluster, simulated with the standard Flux simulator. This simulated workload execution gives a lower bound of 1,021 minutes for the makespan of this workload.

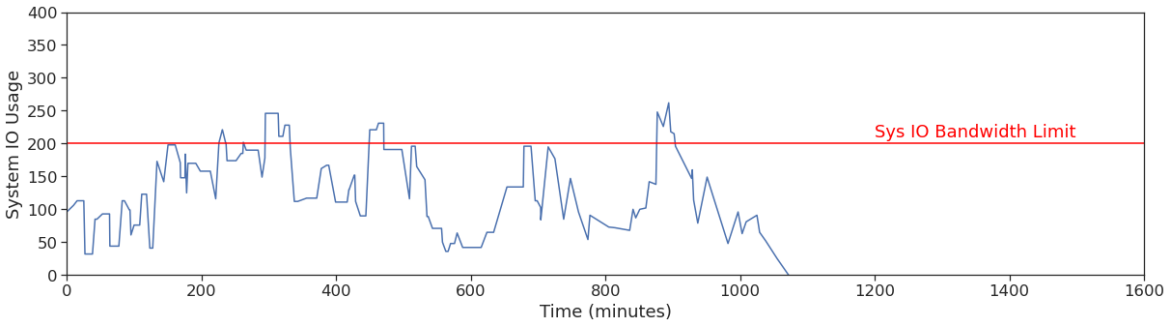
measured on real datasets in Chapters 3 and 4 and also test 100% accurate predictions.

To demonstrate the HPC environment with these parameter settings and our extensions to the Flux simulator, we visualize the execution of a small 100-job workload on a 16-node machine. We first show the base Flux simulation and add each Flux simulator extension described in Section 5.1. Figure 5.2 shows the execution of the 100-job workload using the standard Flux simulator (i.e., none of our extensions). The figure shows job parallel job executions for the entire workload. Each job is represented by a rectangle in the figure, where the width of the rectangle indicates the job execution time. We label important information about the jobs: Teal is normal executions, red is a job that exceeded its time limit, yellow is a rerun of a job that exceeded the time limit, and hatches indicate an IO-sensitive job. We observe in Figure 5.2, that a number of jobs exceed their time limits but are not rerun. The total makespan of this workload is a 1,021 minutes.

We next introduce IO resources and IO contention to the Flux simulator, as described in Section 5.1. Figure 5.3 shows the execution of the same set of 100 jobs and the system IO bandwidth demand. Similar to Figure 5.2, Figure 5.3a shows the execution of jobs in the simulation. Behind the rectangles representing jobs are pale red rectangles that show the occurrence of IO contention on the system. The first IO contention event in Figure 5.3a, labeled "PFS perf. degradation" is the injected contention event described at the beginning of this section. We observe that the other



(a)



(b)

Figure 5.3: 100-job workload simulated on the modified Flux simulator, tracking IO resources and adding an IO contention model. (a) shows the execution of jobs and the contention caused by excessive IO demand shown in (b).

occurrences of IO contention are caused by system IO demand exceeding the IO limit, as depicted in Figure 5.3b. The presence of IO contention causes IO-sensitive jobs to increase in runtime and many exceed their timelimit, increasing the total makespan of the workload to 1,071 minutes.

We show the addition of simulating user response to jobs exceeding their time limits in Figure 5.4. In this figure, 50% of the jobs that exceed their runtime are resubmitted to the queue. We observe that the combination of IO contention causing more IO-sensitive jobs to exceed time limits and the need to rerun these jobs extends the makespan to 1,414 minutes. Figure 5.4 shows that unchecked IO contention can inflate the makespan of a workload by more than 38%.

Our final extension to the Flux simulator shows how AI4IO IO-aware scheduling works to prevent IO contention and mitigate its impact. Figure 5.5 shows the same

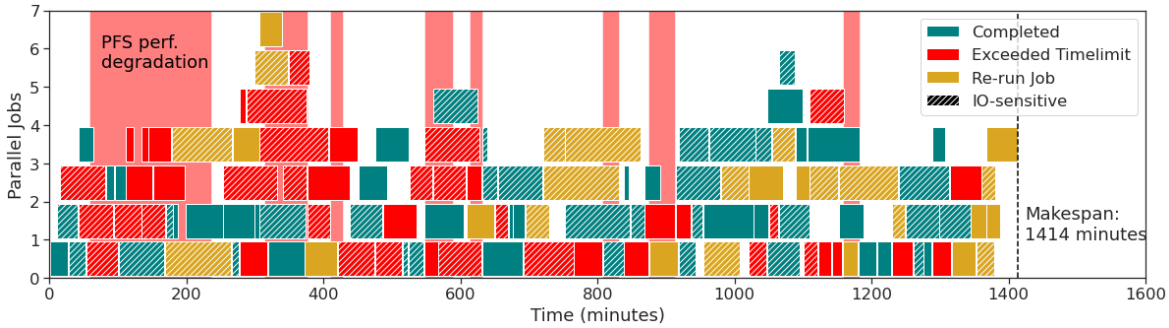


Figure 5.4: 100-job workload simulated on the modified Flux simulator, adding user-based actions for jobs that exceed their runtime. These jobs are resubmitted and rerun to mimic observed user behavior.

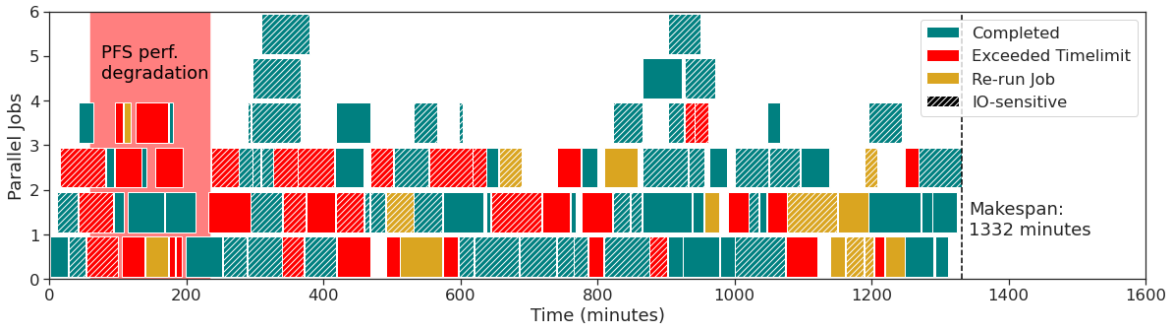


Figure 5.5: 100-job workload simulated on the modified Flux simulator, with IO-aware scheduling enabled by our AI4IO tools. The IO-aware scheduling using predictions from PRIONN and CanarIO to prevent IO contention and mitigate its effects on IO-sensitive jobs.

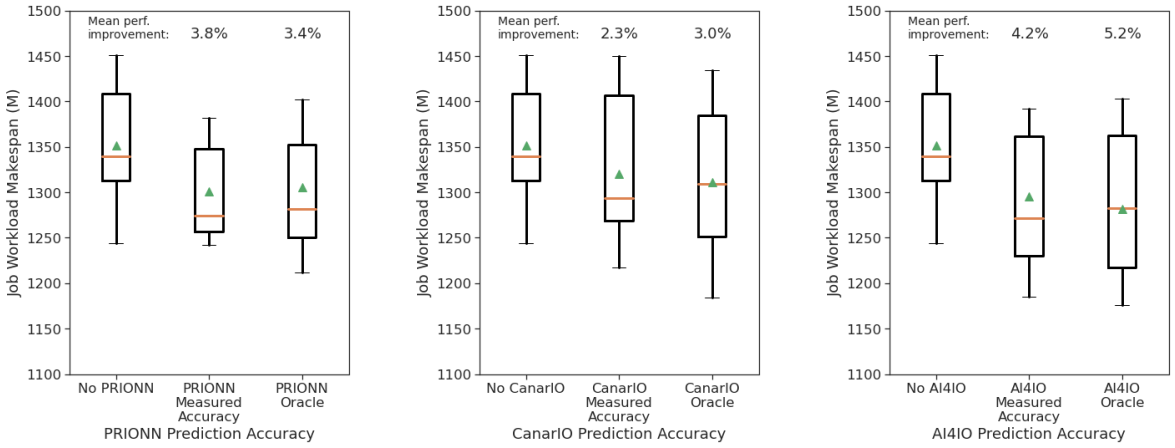
100-job workload with IO contention, user actions, and IO-aware scheduling with 100% accurate predictions from our AI4IO Oracle. We observe the the IO-aware scheduling is working in two ways: (1) IO contention is prevented by keeping system IO demand below the IO limit and (2) during the contention caused by the PFS, less IO-sensitive jobs are run to mitigate the impact of that IO contention. As a result, the makespan of this workload shrinks by 6% to 1,332 minutes. Our visualization of the simulation in this section validates the changes to the Flux simulator described in Section 5.1.

5.3 IO-Aware Scheduling Evaluation

In this section, we use our modified Flux simulator to evaluate the effectiveness of IO-aware scheduling enabled by AI4IO. We compare the makespan of 10 1000-job workloads of real HPC jobs simulated on a scaled Quartz system with and without IO-aware scheduling. We test IO-aware scheduling with 100% accurate predictions and prediction accuracies matching those of PRIONN and CanarIO in Chapters 3 and 4.

Figure 5.6 shows the results of our 10 simulated workload executions with and without IO-aware scheduling for several arrangements of the AI4IO tools. Specifically, Figure 5.6a shows the distribution of makespans for these 10 workloads without PRIONN, with PRIONN at accuracies matching those reported in Chapter 3, and with 100% accurate PRIONN predictions. We observe that enabling the PRIONN-based IO-aware scheduling logic in Figure 5.1a reduces the mean makespan by $> 3\%$ compared to the IO-unaware scheduler. Figure 5.6b shows the distribution of makespans for the 10 workloads without CanarIO, with CanarIO at accuracies matching those reported in Chapter 4, and with 100% accurate CanarIO predictions. We observe that enabling the CanarIO-based IO-aware scheduling logic in Figures 5.1a and 5.1b reduces the mean makespan by $> 2\%$ compared to the IO-unaware scheduler.

Figure 5.6c shows the distribution of makespans for the 10 workloads without AI4IO (i.e., PRIONN and CanarIO), with AI4IO accuracies matching those reported in Chapters 3 and 4, and with 100% accurate AI4IO predictions. We observe that enabling IO-aware scheduling with both our AI4IO tools improves the mean makespan by 4.2% with our experimentally observed accuracies and by $> 5\%$ with 100% accurate predictions. The maximum makespan improvement for our experimentally measure AI4IO prediction accuracy was 6.4%. Figure 5.6c also shows that the combination of preventing and mitigating IO contention, with PRIONN and CanarIO, gives the greatest reduction in makespan.



(a) (b) (c)

Figure 5.6: Simulated evaluation results comparing IO-aware scheduling with IO-unaware scheduling decisions. In (a) we measure the impact of only PRIONN-based scheduling decisions, in (b) we measure the impact of only CanarIO-based scheduling decisions. (c) shows that PRIONN and CanarIO work together to produce the greatest improvement in workload performance.

5.4 Summary

Our extensive evaluation of the AI4IO tools with the modified Flux scheduler and simulator validates the initial evaluations of the individual tools in Chapters 3 and 4. In this chapter, we demonstrated that in this more advance simulation environment PRIONN and CanarIO are capable of improving job throughput (i.e., workload makespan) via IO-aware scheduling. Further, we show that these tools work to improve makespan individually, but that together they provide the largest benefit. Using the accuracies of PRIONN and CanarIO predictions we observed from real HPC data in Chapters 3 and 4, we observed a maximum of 6.4% and a mean of 4.2% reduction in makespan compared to IO-unaware scheduling. Scaling this result to a large cluster, like Quartz with 2600 nodes, this amounts to recovering more than 18,000 node-hours per week of operation.

Chapter 6

SUMMARY AND FUTURE WORK

In this chapter, we summarize the contributions and lessons learned in this thesis. We also describe the future work of the AI4IO tool suite.

6.1 Thesis Summary

IO contention is a problem on HPC systems that increases job runtime and wastes valuable system resources. As HPC machines approach exascale, we observe that there is a growing IO gap where the compute capabilities of systems are growing more rapidly than their IO bandwidth to the Parallel File System (PFS), indicating that the problem will persist and become worse. Yet popular batch schedulers, like Slurm, remain blind to shared resources (i.e., IO) and are incapable of preventing or mitigating IO contention. We tackle this problem by enabling IO-awareness with our suite of AI-based tools, AI4IO. Our suite of tools were developed to integrate with existing schedulers, in that they provide automatic and accurate predictions that enable the prevention and mitigation of IO contention.

Specifically, in Chapter 3 we develop our tool, PRIONN, for preventing IO contention. PRIONN is able to predict runtime and IO usage of jobs before they execute. Our tool relies on a novel transformation of job script text into image-like data that is used to train 2D-CNN models. We demonstrate that we can predict job runtime and IO usage for real HPC jobs with 75% mean and 98% median accuracy. These predictions are used by IO-aware schedulers to anticipate and avoid IO bursts that cause IO contention and slowdown.

In Chapter 4 we develop our tool, CanarIO, for mitigating IO contention. CanarIO is able to predict job IO-sensitivity and detect, in real-time, the occurrence of

IO contention. Our tool relies on the use of canary jobs to frequently probe the system for PFS performance and IO contention. In turn, the data from these jobs and high-granularity system IO data are analysed and used to train an adapted PRIONN 2D-CNN and kNN models. We demonstrate that we can correctly predict $> 90\%$ of IO-sensitive jobs and detect 37.5% of IO degradation events. These predictions are used by IO-aware schedulers to mitigate the effects of IO contention on IO-sensitive jobs.

Finally, in Chapter 5 we extend the Flux scheduler and simulator to model contention and its effects on job execution. Furthermore, we develop IO-aware scheduling logic and integrate it and our AI4IO tools into the scheduler. We individually test IO-aware scheduling with PRIONN and CanarIO to demonstrate that our tools enable prevention and mitigation of IO contention, respectively. Our tools working in concert to prevent and mitigate IO contention offer the largest improvement in resource utilization and performance as we decrease the makespan of 10 real HPC job workloads by an average of 4.2% with a maximum performance increase of 6.4%. The results of our simulated IO-aware scheduling evaluation demonstrate that AI4IO tools are capable of preventing and mitigating IO contention through IO-aware scheduling.

6.2 Future Work

While this thesis represents the first steps towards IO-aware job scheduling enabled by AI4IO tools, additional work is needed before resource-aware scheduling can become a production-ready solution to shared resource contention.

6.2.1 AI4IO Tools

We demonstrated that our AI4IO tools are capable of making predictions that enable IO-aware scheduling, but there exists territory for exploration of new tools for other shared resources and optimizations for our tools. In Chapter 2 we reviewed the work that has been done on shared resource contention. These shared resources include IO and other resources, like interconnects. Further exploration of applying PRIONN

and CanarIO to other shared resources or developing new resource-specific AI4IO tools will enable more general resource-aware scheduling. In turn, the better management of many resources, especially those that are intrinsically linked (e.g., PFS IO is sent over the HPC network) could further improve the prevention and mitigation of resource contention.

The parameter space of our AI4IO tools could also be explored to find the most optimized settings. For example, CanarIO relies on canary jobs that are frequently run to collect IO performance information about the system. These jobs spend resources and are a cost for using CanarIO. Analysis of how often canary jobs should be run in order to maintain CanarIO prediction accuracy would provide an optimization that reduces the resources used to enable IO-aware scheduling. Additional optimizations could further improve the AI4IO tool suite.

6.2.2 IO-Aware Scheduling

In Chapter 5, demonstrated that AI4IO tools enable performance improvements (i.e., decreased workload makespans) with IO-aware scheduling. The IO-aware scheduling logic that we implemented in Figure 5.1 works to enable IO-aware decisions based on the AI4IO tools. However additional complexities and exploration of the decision parameter space should be explored. Implementing and modifying the logic could provide more optimal decisions that further improve the performance benefits of IO-aware scheduling.

Additionally, in evaluation of the IO-aware scheduling logic, we must consider a wider set of metrics. In Chapter 5, we report on the workload makespan improvements. However, implementing IO-aware scheduling in production schedulers will require metrics that capture the HPC user and admin experience. For example, while the total makespan of a job workload may improve with IO-aware scheduling, it comes at the cost increasing the makespan of jobs that are delayed. Exploring the tradeoffs between population and individual benefits and detriments is necessary to ready IO-aware scheduling on production HPC systems.

BIBLIOGRAPHY

[1] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, et al. The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165. IEEE, 2014.

[2] 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 *10th International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, pages 9–17, 2014.

[3] 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 *2014 43rd International Conference on Parallel Processing Workshops*, pages 9–17. IEEE, 2014.

[4] Sergey Blagodurov and Alexandra Fedorova. Towards the contention aware scheduling in hpc cluster environment. In *Journal of Physics: Conference Series*, volume 385, page 012010. IOP Publishing, 2012.

[5] Sergey Blagodurov and Alexandra Fedorova. Optimizing shared resource contention in hpc clusters. In *Super Computing*, pages 1–25, 2013.

[6] Julian Borrill, Leonid Oliker, John Shalf, Hongzhang Shan, and Andrew Uselton. Hpc global file system performance analysis using a scientific-application derived benchmark. *Parallel Computing*, 35(6):358–373, 2009.

[7] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. Predicting job completion times using system logs in supercomputing clusters. In *Proceedings of the 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 1–8, 2013.

[8] Renato LF. Cunha, Eduardo R. Rodrigues, Leonardo P. Tizzei, and Marco AS. Netto. Job placement advisor based on turnaround predictions for hpc hybrid clouds. *Future Generation Computer Systems*, 67:35–46, 2017.

[9] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 610–621. IEEE, 2014.

[10] Robert Dietze, Michael Hofmann, and Gudula Rünger. Analysis and modeling of resource contention effects based on benchmark applications. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 330–337. IEEE, 2018.

[11] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. Calciom: Mitigating i/o interference in hpc systems through cross-application coordination. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 155–164. IEEE, 2014.

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

[13] Ana Gainaru, Valentin Le Fèvre, and Guillaume Pallez. I/o scheduling strategy for periodic applications. 2019.

[14] Peter Harrington, Wucherl Yoo, Alexander Sim, and Kesheng Wu. Diagnosing parallel i/o bottlenecks in hpc applications. In *International Conference for High Performance Computing Networking Storage and Analysis (SCI7) ACM Student Research Competition (SRC)*, 2017.

[15] 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 *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 69–80, New York, NY, USA, 2016. ACM.

[16] Harsh Khetawat, Christopher Zimmer, Frank Mueller, Scott Atchley, Sudharshan S Vazhkudai, and Misbah Mubarak. Evaluating burst buffer placement in hpc systems. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11. IEEE, 2019.

[17] Shonali Krishnaswamy, Seng Wai Loke, and Arkady Zaslavsky. Estimating computation times of data-intensive applications. *IEEE Distributed Systems Online*, 5(4), 2004.

[18] Glenn K Lockwood, Shane Snyder, Teng Wang, Suren Byna, Philip Carns, and Nicholas J Wright. A year in the life of a parallel file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 74. IEEE Press, 2018.

[19] Jay Lofstead, F. Zheng, Q. Liu, Scott Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. Managing variability in the io performance of petascale storage systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, Nov 2010.

[20] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. Managing variability in the io performance of petascale storage systems. In *Proceedings of the 22nd ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 1–12, 2010.

[21] Jakob Lüttgau, Shane Snyder, Philip Carns, Justin M Wozniak, Julian Kunkel, and Thomas Ludwig. Toward understanding i/o behavior in hpc workflows. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 64–75. IEEE, 2018.

[22] Maxime Martinasso and Jean-François Méhaut. A contention-aware performance model for hpc-based networks: A case study of the infiniband network. In *European Conference on Parallel Processing*, pages 91–102. Springer, 2011.

[23] Andréa Matsunaga and José AB Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 495–504, 2010.

[24] 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)*, pages 255–258, Sept 2016.

[25] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th Neural Information Processing Systems Conference*, NIPS, pages 3111–3119, 2013.

[26] MC Miller. Design & implementation of macsio. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2015.

[27] Ali Mohammed, Ahmed Eleliemy, Florina M Ciorba, Franziska Kasielke, and Ioana Banicescu. An approach for realistically simulating the performance of scientific applications on high performance computing systems. *Future Generation Computer Systems*, 2019.

[28] Misbah Mubarak, Philip Carns, Jonathan Jenkins, Jianping Kelvin Li, Nikhil Jain, Shane Snyder, Robert Ross, Christopher D Carothers, Abhinav Bhatele, and

Kwan-Liu Ma. Quantifying i/o and communication traffic interference on dragonfly networks equipped with burst buffers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 204–215. IEEE, 2017.

[29] Misbah Mubarak, Christopher D Carothers, Robert B Ross, and Philip Carns. Enabling parallel simulation of large-scale hpc network systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):87–100, 2016.

[30] Sarah Neuwirth, Feiyi Wang, Sarp Oral, and Ulrich Bruening. Automatic and transparent resource contention mitigation for improving large-scale parallel file system performance. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 604–613. IEEE, 2017.

[31] Sarah Neuwirth, Feiyi Wang, Sarp Oral, Sudharshan Vazhkudai, James Rogers, and Ulrich Bruening. Using balanced data placement to address i/o contention in production environments. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 9–17. IEEE, 2016.

[32] 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.

[33] Sarp Oral, David A. Dillow, Douglas Fuller, Jason Hill, Dustin Leverman, Sudharshan S. Vazhkudai, Feiyi Wang, Youngjae Kim, James Rogers, James Simmons, and Ross Miller. Olcf's 1 tb/s, next-generation lustre file system. In *Cray User Meeting*, CUG, 2013.

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[35] Eduardo R. Rodrigues, Renato LF. Cunha, Marco AS. Netto, and Michael Spriggs. Helping hpc users specify job memory requirements via machine learning. In *Proceedings of the Third International Workshop on HPC User Support Tools*, pages 6–13, 2016.

[36] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 42. IEEE Press, 2008.

[37] Hongzhang Shan and John Shalf. Using ior to analyze the i/o performance for hpc platforms. 2007.

[38] Mehdi Sheikhalishahi, Ignacio Llorente, and Lucio Grandinetti. Energy aware consolidation policies. 09 2011.

[39] Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times using historical information. In *Workshop Job Scheduling Strategies for Parallel Processing*, JSSPP, pages 122–142, 1998.

[40] Warren Smith, Valerie Taylor, and Ian Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In *Workshop on Job Scheduling Strategies for Parallel Processing*, JSSPP, pages 202–219, 1999.

[41] Hanul Sung, Jiwoo Bang, Alexander Sim, Kesheng Wu, and Hyeonsang Eom. Understanding parallel i/o performance trends under various hpc configurations. In *Proceedings of the ACM Workshop on Systems and Network Telemetry and Analytics*, pages 29–36, 2019.

[42] 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), 2007.

[43] Teng Wang, Suren Byna, Glenn K Lockwood, Shane Snyder, Philip Carns, Sunggon Kim, and Nicholas J Wright. A zoom-in analysis of i/o logs to detect root causes of i/o performance bottlenecks. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 102–111, 2019.

[44] 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 *Proceedings of the 47th International Conference on Parallel Processing*, page 46. ACM, 2018.

[45] Yiqi Xu, Dulcardo Arteaga, Ming Zhao, Yonggang Liu, Renato Figueiredo, and Seetharami Seelam. vpfs: Bandwidth virtualization of parallel storage systems. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2012.

[46] Ming Zhao and Yiqi Xu. vpfs+: Managing i/o performance for diverse hpc applications. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 51–64. IEEE, 2019.

Appendix

PERMISSIONS

Chapter 3 is based on my publications in the IEEE 2018 International Conference on Parallel Processing (ICPP). Chapter 4 is based on my publication in the 2020 IEEE International Parallel and Distributed Processing Symposium. Based on the IEEE's copyright policy on re-use of work in a thesis available at https://www.ieee.org/content/dam/ieee-org/ieee/web/org/pubs/permissions_faq.pdf, I, the original author, note that "The IEEE does not require individuals working on a thesis to obtain a formal reuse license".