A Taxonomy of Error Sources in HPC I/O Machine Learning Models

Mihailo Isakov*, Mikaela Currier*, Eliakin del Rosario*, Sandeep Madireddy†,
Prasanna Balaprakash†, Philip Carns†, Robert B. Ross†, Glenn K. Lockwood‡, Michel A. Kinsy*
* *Secure, Trusted and Assured Microelectronics (STAM) Center*
Ira A. Fulton Schools of Engineering, Arizona State University, Tempe, AZ
misakov1@asu.edu, mikaela.currier@gmail.com, eliakin.drosario@tamu.edu, mkinsy@asu.edu
† *Argonne National Laboratory*, Lemont, IL
smadireddy@anl.gov, {pbalapra, carns, rross}@mcs.anl.gov
‡ *Lawrence Berkeley National Laboratory*, Berkeley, CA
glock@lbl.gov

***Abstract*—I/O efficiency is crucial to productivity in scientific computing, but the growing complexity of HPC systems and applications complicates efforts to understand and optimize I/O behavior at scale. Data-driven machine learning-based I/O throughput models offer a solution: they can be used to identify bottlenecks, automate I/O tuning, or optimize job scheduling with minimal human intervention. Unfortunately, current state-of-the-art I/O models are not robust enough for production use and underperform after being deployed.**

We analyze four years of application, scheduler, and storage system logs on two leadership-class HPC platforms to understand why I/O models underperform in practice. We propose a taxonomy consisting of five categories of I/O modeling errors: poor application and system modeling, inadequate dataset coverage, I/O contention, and I/O noise. We develop litmus tests to quantify each category, allowing researchers to narrow down failure modes, enhance I/O throughput models, and improve future generations of HPC logging and analysis tools.

***Index Terms*—High performance computing, I/O, storage, machine learning**

I. Introduction

As scientific applications push to leverage ever more capable computational platforms, there is a critical need to identify and address bottlenecks of all types. applications, the I/O subsystem is often a major source of performance bottlenecks, and it is common for applications to attain only a small fraction of the peak I/O rates [1]. These performance problems can severely limit the scalability of applications and are difficult to detect, diagnose, and fix. Data-driven machine learning-based models of I/O throughput can help practitioners understand application bottlenecks (e.g., [2]–[7]), and have the potential to automate I/O tuning and other tasks. However, current machine learning-based I/O models are not robust enough for production use [6]. A thorough investigation of *why* these models underperform when deployed on high performance computing (HPC) systems will provide key insights and guidance on how to address their shortcomings. The goal of our study is to help machine learning (ML)-driven I/O modeling techniques make the transition from theory to practice.

There are several reasons why machine learning-based I/O models underperform when deployed: poor modeling choices [2], [7], concept drift in the data [5], and weak generalization [6], among others. I/O models are often opaque, and there is no established methodology for diagnosing the root cause of model errors. In this work, we present a taxonomy of ML-based I/O modeling errors, as shown in Figure 1. Through this taxonomy, we show that I/O throughput prediction errors can be separated and quantified into five error classes: inadequate (1) application and (2) system models, (3) novel application or system behaviors, (4) I/O contention and (5) inherent noise. For each class, we present data-driven litmus tests that estimate the portion of modeling error caused by that class. The taxonomy enables independent study of each source of error and prescribes appropriate ML techniques to tackle the underlying sources of error.

Our contributions in this work are as follows:

1) We introduce a taxonomy of ML-based I/O throughput modeling errors which consists of five classes of errors.
2) We show that the choice of ML model algorithm, scaling the model size, and tuning hyperparameters cannot reduce all potential errors. We present two litmus tests that quantify error due to poor application and system modeling.
3) We present a litmus test that estimates what portion of error is caused by rare jobs with previously unseen behavior, and apply uncertainty quantification methods to classify those jobs as out-of-distribution jobs.
4) We present a method for quantifying the impact of I/O contention and noise on I/O throughput, which (1) defines a fundamental limit in how accurate ML models can become, and (2) gives HPC system users and administrators a practical estimate of the I/O throughput variance they should expect. We show that underlying system noise is the dominant source of errors, and not poor modeling or lack of application or system data.
5) We present a framework for how the proposed taxonomy is practically applied to new systems and evaluate it on two leadership-class supercomputers: Argonne Leadership Computing Facility (ALCF) Theta and National Energy Research Scientific Computing Center (NERSC) Cori.

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

II. Related work

In recent years, automating HPC I/O system analysis through ML has received significant attention, with two prominent directions: (1) workload clustering to better understand groups of HPC jobs and automate handling of whole groups, and (2) I/O subsystem modeling and make predictions of HPC job I/O time, I/O throughput, optimal scheduling, etc. Clustering HPC job logs has been explored in [2], [8], [9] with the goal of better understanding workload distribution, scaling I/O expert effort more efficiently, and revealing hidden trends and I/O workload patterns. ML-based modeling has been used for predicting I/O time [4], I/O throughput [2], [7], optimal filesystem configuration [10], [11], as well as for building black boxes of I/O subsystems in order to apply ML model interpretation techniques [2]. While there have been some attempts at creating analytical models of I/O subsystems [12], most attempts are data-driven, and rely on HPC system logs to create models of I/O [1], [2], [4], [7], [13]. Although the challenges of developing accurate machine learning models are well known, the nature of the domain requires special consideration: I/O subsystems have to service multiple competing jobs, their configuration evolves over time, they have periods of increased variability, they experience occasional hardware faults, etc. [14]–[16]. Diagnosing this I/O variability, where the performance of a job depends on external factors to the job itself has been extensively studied [3], [4], [14], [16], [17]. Finally, the deployment of I/O models has been shown to require special consideration as these models often significantly underperform on new applications [5], [6]. While different sources of model error have been studied individually, no prior work characterizes the relative impact of different sources of error on model accuracy.

III. Modeling HPC applications and systems

The behavior of an HPC system is governed by both complex rules and inherent noise. By formalizing the system as a mathematical function (or, more generally, a stochastic process) with its inputs and outputs, the process may be decomposed into smaller components more amenable to analysis.

The I/O throughput of a system running specific sets of applications may be treated as a data-generating process from which I/O throughput measurements are drawn. While building a perfect model of an HPC system may not be possible, it is useful to understand the inputs to the 'true' process and the process's functional properties. The theoretical model of the process must include all causes that might affect a real HPC system, such as: how well a job uses the system, hardware and software configurations over the life of the system, resource contention between concurrent jobs, inherent application-specific and system noise, application-specific noise sensitivity, etc. Although many of these causes are not directly observable since they work at short time scales or below the instruction set architecture where gaining low-level insight is not possible, the effects are cumulative and the system is affected by them. ML models of the system must take these causes into account or suffer modeling errors.

To model system behavior, we adopt the system modeling formulation from [4], expressing the relationship between an HPC job j and its I/O throughput on the system $\phi(j)$ as:

$$\phi(j) = f(j,\, \zeta,\, \omega) \tag{1}$$

Here, j represents HPC job behavior (e.g., I/O volume and access patterns, distribution of POSIX operations, etc.), ζ represents system state (e.g., file system health, system configuration, node availability, etc.) and system behavior (e.g., the behavior of other applications co-located with the modeled application during its run, contention from resource sharing, etc.) *at a given time*. ω represents the randomness acting on the system. The system ζ can be further decomposed as:

$$\zeta = \zeta_g(t) + \zeta_l(t,\, j) \tag{2}$$

The component $\zeta_g(t)$ represents the *global system impact* on all jobs running on the system (e.g., a service degradation that equally impacts all jobs) and is only a function of time t. The component $\zeta_l(t,\, j)$ represents the *local system impact* on the I/O throughput of job j caused by resource contention and interactions with other jobs running on the system. Contrary to the $\zeta_g(t)$ component, $\zeta_l(t,\, j)$ is job-specific and depends on the behavior of the current set of applications running on the system and their location relative to j, the sensitivity of j to resource contention and noise, etc. Without loss of generality, the I/O throughput from Equation 1 can be expressed as:

$$\begin{aligned}\phi(j) &= f(j,\, \zeta_g(t),\, \zeta_l(t,j),\, \omega) \qquad (3)\\ &= f_a(j) + f_g(j,\, \zeta_g(t)) + f_l(j,\, \zeta_l(t,\, j)) + f_n(j,\, \zeta,\, \omega)\end{aligned}$$

Here, $f_a(j)$ represents the I/O throughput of a job j on an idealized system where the j is alone on the system, the system does not change over time, and there is no resource contention. $f_g(j,\, \zeta_g(t))$ represents how the evolving configuration of the system (hardware provisioning, software updates, etc.) affects a job's I/O throughput. The $f_l(j,\, \zeta_l(t,\, j))$ component represents the per-job impact of resource contention and j's I/O noise sensitivity. Finally, $f_n(j,\, \zeta,\, \omega)$ represents the impact of inherent system noise (e.g., dropped packets) on the job.

A. Modeling assumptions

The task of modeling a system's I/O throughput involves predicting the behavior of the system when tasked with executing a job from some application on some data. Modeling I/O throughput requires modeling both the HPC system and the jobs running on it. Machine learning models used in this work attempt to learn the true function ϕ by mapping observable features of the job j and the system ζ to measured I/O throughputs $\phi(j)$. A model $m(j_o,\, \zeta_o)$ is tasked with predicting throughput $\phi(j)$, where $j_o \subseteq j$ and $\zeta_o \subseteq \zeta$ are the observable job and system features.

When designing ML models, the choice of model architecture and model inputs is based on implicit assumptions about the process that generates the data. When incorrect assumptions are made about the domain, the model will suffer from errors that cannot be fixed within that modeling framework, e.g., through hyperparameter tuning or further data collection. We investigate four common assumptions about the HPC domain, shown by the branches in Figure 1.

All data is in-distribution: a common assumption that ML practitioners make is that all model errors are the product of insufficiently trained models, inadequate model architectures, or missing discriminative *features*. However, some jobs in the dataset may be *Out of Distribution (OoD)*, that is, they may be collected at a different time or environment, or through a different process. The model may underperform on OoD jobs due to the lack of similar jobs in the training set and not due to lack of insight (features) into the job. The cause of the problem is *epistemic uncertainty (EU)* - the model suffers from *reducible uncertainty*, i.e., lack of knowledge, since a broader training set would make the OoD jobs in-distribution (ID). In the HPC domain, epistemic uncertainty is present in cases of rarely ran or novel jobs or uncommon system states. Without considering the possibility that a portion of the error is a product of epistemic uncertainty, practitioners may put effort into tuning models instead of collecting more underrepresented jobs. Referring to Equation 1, this assumption may be expressed as: deployment time j_d and ζ_d are drawn from a different distribution from training time j_t and ζ_t.

Noise is absent: all systems have some inherent noise that cannot be modeled and will impact predictions. *Aleatory uncertainty* (AU) refers to *irreducible uncertainty* which stems from inherent noise or lack of insight into jobs on the system. Modeling errors due to aleatory uncertainty are different from epistemic uncertainty because collecting more jobs may not reduce AU, and these errors may be fundamentally unfixable. Understanding and characterizing a system's inherent I/O noise is necessary to quantify ML model uncertainty, and because the amount of noise in the data has a strong effect on the optimal choice of ML model. HPC I/O domain experts note that certain systems do have significantly higher or lower I/O noise [18], [19], but I/O modeling works rarely attempt to quantify ML model uncertainty [20]. The assumption that noise is not present in the dataset can be expressed as follows: The practitioner assumes that the data-generating process ϕ has the form of $\phi(j) = f(j, \zeta)$ instead of $\phi(j) = f(j, \zeta, \omega)$, i.e., that the inherent noise impact is zero: $f_n(j, \zeta, \omega) = 0$.

Sampling is independent: running a job on a system can be viewed as sampling the combination of application behavior and system state and measuring I/O throughput. Most I/O modeling works implicitly assume that multiple samples taken at the same time are independent of each other. The system is modeled as equally affecting all jobs running on it, that is, the placement of different jobs on nodes, the interactions between neighboring jobs, network contention, etc. *do not affect the job*. This assumption can then be expressed as: the process has the form of $\phi(j) = f(j, \zeta_g(t))$, not $\phi(j) = f(j, \zeta, \omega)$ i.e., that the resource contention impact is zero: $f_l(j, \zeta_l(t, j)) = 0$.

Process is stationary: a common assumption ML practitioners make is that the data-generating process is stationary, and that the same job ran at different times achieves the same I/O throughput. As hardware fails, as new nodes are provisioned, and shared libraries get updates, the system evolves over time. The stationarity assumption is therefore incorrect, and ignoring it by e.g., not exposing the ML model to *when* a job is ran may cause hard-to-diagnose errors. This assumption implies sampling independence and absence of noise, and can be expressed using the system modeling formulation as: $\phi(j) = f(j)$ and $f_g(j, \zeta_g(t)) = 0$.

IV. Classifying I/O throughput prediction errors

No matter the problem to which machine learning is applied, a systematic characterization of the sources of errors is crucial to improve model accuracy. While there is no substitute for 'looking at the data' to understand the root cause of the problem, this approach does not scale for large datasets. We seek a systematic way to understand the barriers to greater accuracy and improve ML models applied to systems data.

While the work presented here can be generalized past just I/O to e.g., compute or network modeling, we study I/O because I/O bottlenecks are more difficult to diagnose than compute bottlenecks, and because I/O has a coarser temporal granularity allowing software to observe I/O subsystems without the need for e.g., hardware performance counters or binary instrumentation. The key questions we ask in this work are: What are the impediments to the successful application of learning algorithms in understanding I/O? Should ML practitioners focus on acquiring more data on HPC applications or the HPC system? How much of the error stems from poor ML model architectures? How much of the error can be attributed to the dynamic nature of the system and the interactions between concurrent jobs? How much of the performance variation is caused by the system? What fraction of jobs exhibit truly novel I/O behavior compared to jobs observed thus far? At what point are the applications *too novel*, so much so that users should no longer trust the predictions of the I/O model? We now describe five error classes and dive deeper into error attribution in Sections VI, VII, IX and VIII.

The lack of application and system observability, the interaction between running jobs, the inherent system noise, and the novel or rare applications prevent ML models from fully capturing system behavior, causing errors. We define the I/O throughput prediction error of a model m in a job j as:

$$e(j) = \phi(j, \zeta, \omega) - m(j_o, \zeta_o) \tag{4}$$

Following the $\phi(j)$ terms from Eq. 3 and including the out-of-distribution error, the error can be broken down as follows:

$$e(j) = e_{app} + e_{system} + e_{ood} + e_{contention} + e_{noise} \tag{5}$$

Here, the application modeling error e_{app} is caused by a poor model fit of application behavior ($f_a(j)$ component), the global system error e_{system} is caused by poor predictions of global system impact ($f_g(j, \zeta_g(t))$ component), the out-of-distribution error e_{ood} is caused by weak model generalization on novel applications or system states, the contention error $e_{contention}$ is caused by poor predictions of job interactions ($f_l(j, \zeta_l(t, j))$ component), and the noise error e_{noise} is caused by the inability of any model to predict inherent noise ($f_n(j, \zeta, \omega)$ component). These five classes of errors are shown as leaf nodes at the bottom of Figure 1. While attributing cumulative job error to each class may be difficult on a per-job basis, we will show that estimating each component across a whole dataset is possible.

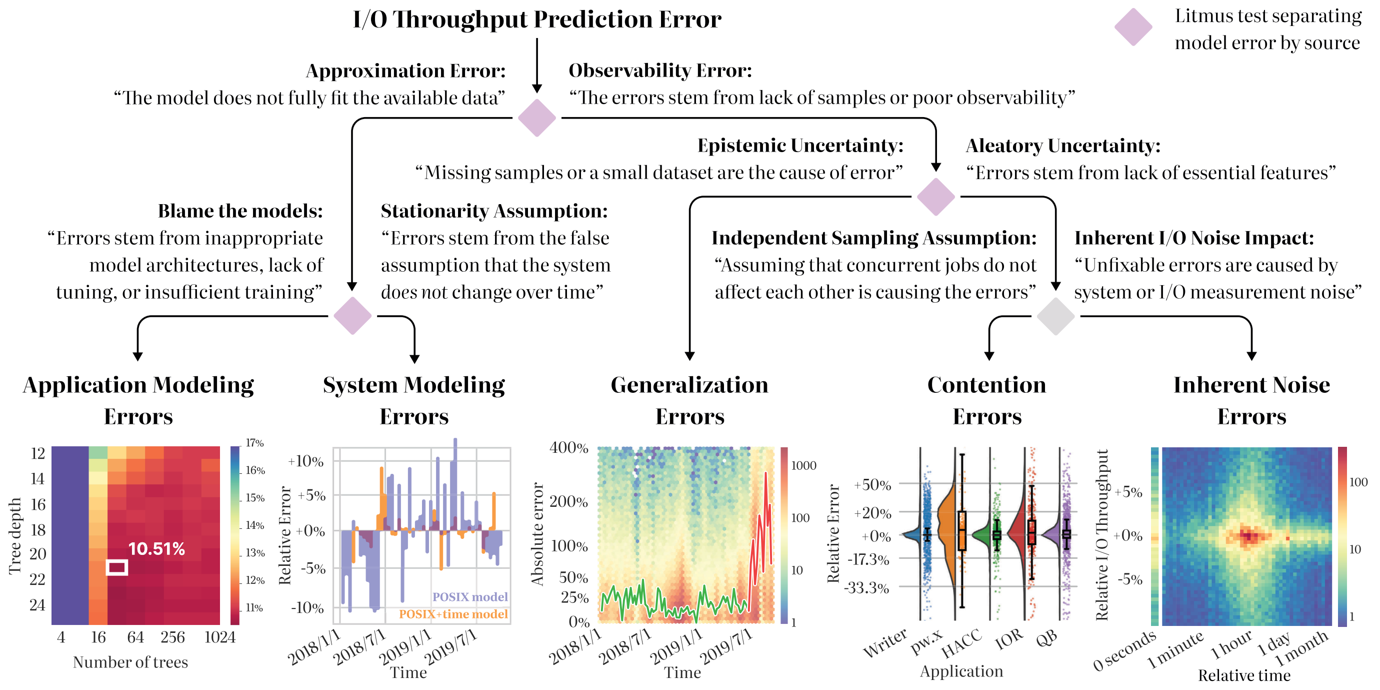

I/O Throughput Prediction Error
Litmus test separating model error by source
Approximation Error:
"The model does not fully fit the available data"
Observability Error:
"The errors stem from lack of samples or poor observability"
Epistemic Uncertainty:
"Missing samples or a small dataset are the cause of error"
Aleatory Uncertainty:
"Errors stem from lack of essential features"
Blame the models:
"Errors stem from inappropriate model architectures, lack of tuning, or insufficient training"
Stationarity Assumption:
"Errors stem from the false assumption that the system does not change over time"
Independent Sampling Assumption:
"Assuming that concurrent jobs do not affect each other is causing the errors"
Inherent I/O Noise Impact:
"Unfixable errors are caused by system or I/O measurement noise"
Application Modeling Errors
System Modeling Errors
Generalization Errors
Contention Errors
Inherent Noise Errors
Tree depth
Number of trees
10.51%
Relative Error
POSIX model
POSIX+time model
Time
Absolute error
Relative Error
Application
Writer
pw.x
HACC
IOR
QB
Relative I/O Throughput
Relative time
0 seconds
1 minute
1 hour
1 day
1 month

Fig. 1: Taxonomy of I/O throughput modeling errors, with examples of the effects of each error class shown in the bottom row (left to right): (1) median error of XGBoost models with varying numbers of estimators and estimator depth, (2) per-week averaged ML model error during the lifetime of a system, (3) median error before (green) and after training (red), with error distribution shown in the background, (4) I/O throughput prediction error for sets of identical (duplicate) jobs, for 5 different applications, (5) distribution of relative job start times and relative job I/O throughputs for pairs of duplicate jobs.

A. *I/O Model Error Taxonomy and Litmus Tests*

We adopt the term litmus test to mean a test that evaluates the presence, amount, or ratio of a certain quantity. In the following sections, we introduce a four litmus tests that split the error from Equation 5 into five separate classes. The error classes in Equation 5 must be estimated in the order shown in the bottom row of Figure 1 due to the specifics of individual litmus tests. For example, before the effect of aleatory and epistemic uncertainty can be separated, a good model must be found [21]. Similarly, before global and local system modeling errors can be separated, OoD jobs must be identified.

Application modeling errors: ML models can have varying expressivity and may not always have the correct structure or enough parameters to fit the available data. Models whose structure or training prevents them from learning the shape of the data-generating process are said to suffer from *approximation errors*. Approximation errors cannot be classified as epistemic or aleatory in nature because no new features or jobs are necessary to remove this error. To estimate AU and EU in the dataset, methods such as AutoDEUQ [21] first require that an appropriate model architecture is found and trained, placing approximation errors as the first branch of the taxonomy.

Approximation errors are further divided into *application* and *system modeling errors*. Application modeling errors are caused by poor predictions of application behavior which can be fixed through hyperparameter searches or better model architectures. The first column of Figure 1 illustrates the impact of application modeling errors with an example hyper-parameter search over two XGBoost parameters on the Theta dataset (introduced in the next section). The best configuration found by the grid search has 32 trees with a depth of 21, while the default XGBoost configuration uses 100 trees of depth 6.

System modeling errors: system behavior changes over time due to transient or long-term changes such as file system metadata issues, failing components, new provisions, etc. [22]. A model that is only aware of application behavior, but not of system state implicitly assumes that the process is stationary. It will be forced to learn the *average* system response to I/O patterns, and will suffer greater prediction errors during periods when system behavior is perturbed. *System modeling errors* occur due to poor (or complete lack of) modeling of the global system component $\zeta_g(t)$. To illustrate this class of errors, the second experiment in Figure 1 shows the per-week average error of two models trained to predict job I/O throughput. The blue model can be written as $m(j_o)$, i.e., it is only exposed to observable application behavior j_o. The orange model can be written as $m(j_o, t)$, i.e., it also knows the *job start time* t. During service degradations, the blue model has long periods of biased errors while the orange model does not, since it knows when the degradations happen.

Generalization errors: ML models should perform well on data drawn from the same distribution from which their training set was collected. When exposed to samples highly dissimilar from their training set, the same models tend to make mispredictions. These samples are called 'out-of-distribution' (OoD) because they come from new, shifted, distributions, or the training set does not have full coverage of the sample space. While models that generalize (perform well on OoD data) may exist, mispredictions on OoD samples are not always the fault of the model, and in those cases the only recourse is to (1) detect and exclude samples suspected as out-of-distribution, (2) seek an expanded training set covering those regions, or (3) apply domain-specific knowledge. In order not to pollute other classes of errors, samples that show high epistemic uncertainty must be detected and their error counted towards *generalization errors* before other errors are estimated. As an example, the third column of Figure 1 shows model error before (green) and after (red) deployment, with the error significantly rising when the model is evaluated on data collected outside the training time span.

Contention and resource sharing errors: a diverse and variable number of applications compete for compute, networking, and I/O bandwidth on HPC systems and interact with each other through these shared resources [17], [23]. Although the global system state will impact all jobs equally, the impact of resource sharing is specific to pairs of jobs that are interacting and is harder to observe and model. Prediction errors that occur due to lack of visibility into job interactions are called *contention errors* and are shown in the fourth column of Figure 1. Here, the I/O throughputs of a number of identical runs (same code and data) of different applications illustrate that some applications are more sensitive to contention than others, even when accounting for global system state.

Inherent noise errors: while hard to measure, contention and resource sharing errors can be potentially removed through greater insight into the system and workloads. What fundamentally cannot be removed are *inherent noise errors*: errors due to random behavior by the system (e.g., dropped packets, randomness introduced through scheduling, etc.). Inherent noise is problematic both because ML models are bound to make errors on samples affected by noise and because noisy samples may impede model training. The fifth column of Figure 1 shows the I/O throughput and start time differences between pairs of identical jobs. The leftmost column contains identical jobs that ran at exactly the same time, which often experience 5% or more difference in I/O throughput.

V. Datasets and experimental setup

This work is evaluated on two datasets, one collected from the Argonne Leadership Computing Facility (ALCF) Theta supercomputer in the period from the beginning of 2017 to end of 2020, and one collected from the National Energy Research Scientific Computing Center (NERSC) Cori supercomputer in the period from beginning of 2018 to the end of 2019. Theta collects Darshan [24] and Cobalt logs and the Theta dataset consists of about 100K jobs with an I/O volume larger than 1GiB, while Cori collects Darshan and Lustre Monitoring Tools (LMT) logs, and the Cori dataset consists of 1.1M jobs larger than 1GiB.

Darshan is an HPC I/O characterization tool that collects HPC job I/O access patterns on both POSIX and MPI-IO levels, and serves as our main insight into application behavior. It collects POSIX aggregate job-level data, e.g., the total number of bytes transferred, accesses made, read / write ratios, unique or shared files opened, distribution of accesses per access size, etc. MPI-IO is a library built on top of POSIX that offers higher-level primitives for performing I/O operations and can potentially offer the model greater insight into application semantics and behavior. Darshan collects MPI-IO information for jobs that use it, and all MPI-IO operations are also visible on the POSIX level. Darshan also collects the number of processes ran, which is typically equal to or greater than the number of cores allocated to a job, but Darshan does not currently measure a job's core count. This information is however available in Cobalt scheduler logs, which contain the number of nodes and cores assigned to a job, job start and end times, job placement, etc. Of the two systems observed in this work, only Theta stores Cobalt scheduler logs. LMT collects I/O subsystem information such as storage server load and file system utilization, and serves as our main insight into the I/O subsystem state as it changes over time. Every 5 seconds, LMT records the state of Lustre file system's object storage servers (OSS), object storage targets (OST), metadata servers (MDS) and metadata targets (MDT). Some of the features collected are OSS and MDS CPU and memory utilization, number of bytes transferred to and from the OSTs, file system fullness, number of metadata operations (e.g., `open`, `close`, `mkdir`, etc.) performed by the metadata targets, etc.

Because LMT logs are collected independently from jobs running on the system, during a data preprocessing phase each job's Darshan log is matched with all LMT measurements collected between the job's start and stop time. LMT separately logs each OSS, OST, MDS, and MDT I/O node state, but since a job is served by an arbitrary number of these I/O nodes, only the minimum, maximum, mean and standard deviation of collected features are exposed to the ML model. Overall, models have access to 48 Darshan POSIX, 48 Darshan MPI-IO, 37 LMT, and 5 Cobalt features. All logs are sanitized and pre-processed as according to [2]. During sanitization, jobs with missing features or illegitimate values (e.g., I/O throughput of zero) are removed. During preprocessing, bounded features (e.g., percentage features such as I/O R/W rate) are not modified, while unbounded features are either scaled first by taking a log_{10} and then applying min-max normalization, or alternatively these features are converted to artificial features (e.g., read and write access count features are converted to a bounded read / write ratio feature and an unbounded total access count feature). The final features exposed to the models are reported in Table I. The code and the dataset for this work are provided in the appendix.

The ML models in this work are trained using supervised learning on the task of predicting the I/O throughput of individual HPC jobs. The model error is defined as:

$$e(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}\left|log_{10}\left(\frac{y_i}{\hat{y}_i}\right)\right| \tag{6}$$

TABLE I: Condensed feature set and feature count

Darshan features (present on both Theta and Cori)	Count
log_{10} of the job I/O throughput	1
log_{10} of the total number of {processes, files, accesses, bytes}	4
log_{10} of the number of POSIX {`open`, `seek`, `stat`, `mmap`, `fsync`, `mode`} calls	6
log_{10} of {memory, file} alignment in bytes	2
% of all accesses that are {reads, writes}	2
% of all {reads, writes} that are {consecutive, sequential}	4
% of all accesses that switch between reading and writing	1
% of {read, write} accesses of size in ranges (0B, 100B], (100B, 1KiB], ..., (100MiB, 1GiB], (1GiB+)	20
% of non-aligned {file, memory} accesses	2
% of all bytes that are {read, written}	2
% of {shared, unique, read-only, read-write, write-only} files	5
% of bytes read/written from {shared, unique, read-only, read-write, write-only} files	5
Lustre features (Cori only)	**Count**
log_{10} of file system {byte, inode} fullness {min, mean, max, std}	8
log_{10} of metadata target {closes, getattrs, getxattrs, links, mkdirs, mknods, opens, renames, rmdirs, setattrs, statfss, unlinks} operation mean	12
% of data server {CPU, memory} {min, mean, max, std}	8
% of data target bytes {read, written} {min, mean, max, std}	8
% of metadata server CPU usage {min, mean, max, std}	1
Cobalt features (Theta only)	**Count**
log_{10} of {core, node} count	2
log_{10} of job runtime	1
% of job {start, end} time relative to total system time range	2

where y_i and $\hat{y}_i$ are the i-th job's measured and predicted I/O throughputs. Because $log(x) = -log(1/x)$, if a model overestimates or underestimates the I/O throughput by the same relative amount, the absolute error remains the same. We use percentages to write errors, where, e.g., a -25% error specifies that the model underestimated real I/O throughput by 25%. Some figures however show the absolute error when model bias is not important. While models try to minimize mean error, we report median values since some of the distributions have heavy tails that make mean estimates unreliable.

VI. Application modeling errors

When an ML practitioner is tasked with a classification or a regression problem, the first model they evaluate will likely under-perform on the task, due to e.g., inadequate data preprocessing, architecture, or hyperparameters. Therefore, the model will suffer from *approximation errors*, which can be removed by tuning the model hyperparameters or finding more appropriate domain-specific ML model architectures. Since the choice of model architecture and parameters typically has a dominant effect on model error, approximation errors must be resolved before more subtle classes of errors become a limiting factor in improving model performance.

Approximation errors can be split into errors caused by poor modeling of the available data (i.e., applications), and into errors caused by implicit assumptions about the domain (e.g., that I/O behavior of a system does not change over time). In this section we analyze application modeling errors, and in Section VII we analyze system modeling errors.

This section asks the following questions: do I/O models build faithful representations of application behavior? What are the limits of I/O application modeling? In practice, do I/O models faithfully learn application behavior? Can I/O application modeling benefit from extra hyperparameter fine-tuning or new application features?

A. *Estimating limits of application modeling*

Here we develop an application modeling error litmus test which separates the application modeling error e_{app} from the other four error classes in Equation 5. To do so, we seek a 'golden model' (GM) that predicts I/O throughput as accurately as possible given the observable application behavior. Application modeling error of a practical ML model is then estimated by comparing its error rate with that of a golden model.

To build this 'golden model', we rely on a property of synthetic datasets where the data-generating process can be freely and repeatedly sampled. When analyzing HPC logs, it is common to see records of the same application ran multiple times on the same data, or data of the same format. For example, system benchmarks such as IOR [25] may be run periodically to evaluate file system health and overall performance. We call these sets of repeated jobs *'duplicate jobs'*. Pairs of jobs are duplicates if they belong to the same application and all of their *observable* application features are identical, typically because the application was ran with the same configuration and input data. Because jobs from the same set of duplicates appear identical to an ML model, the model cannot distinguish between them. Given a training set that only contains sets of duplicate jobs, the highest possible accuracy can be achieved by mapping jobs from each individual set of duplicates to the set's mean I/O throughput. A model that does not learn to predict a set's mean value is said to suffer from application-modeling error.

By restricting the training set to only sets of duplicates, a golden model with a median absolute error e^g can be built for which $e^g_{app} = 0$. This golden model performs only memorization and does not generalize at all, but is nonetheless useful for comparison against real ML models. Any practical model with a median error e^p can then learn its application modeling error e^p_{app} on the restricted training set by comparing against the golden model e^g as $e^p_{app} = e^p - e^g$. Since duplicate sets can have as few as two jobs, I/O throughput estimates for duplicate sets are biased, and the golden model (GM) may appear to perform better on small sets than on large sets. By applying Bessel's correction [26], this effect is mitigated, and the litmus test is administered as:

Application modeling error litmus test:

1. Find sets of duplicate jobs. For each set:
    1.1. Calculate the set's mean I/O throughput;
    1.2. Apply Bessel's correction to mean;
    1.3. Use mean as golden model prediction;
    1.4. Calculate per-set mean GM absolute error;
2. Calculate median of real and GM mean set errors;
3. Calc. model's e_{app} as difference between the two;

Assuming that duplicate jobs are drawn from the same distribution of applications as the rest of the dataset, the golden model median absolute error represents the lower bound on median absolute error a model can achieve on the whole dataset. Note that different applications may have different distributions of duplicate I/O throughputs, as shown in the fourth column of Figure 1. For this litmus test to be accurate, a large sample of applications representative of the HPC system workload must be acquired. When applied to Theta, 19010 duplicates (23.5% of the dataset) over 3509 sets show a median absolute error of 10.01%. Cori has 504920 duplicates (54%) in 77390 sets with a median absolute error of 14.15%. If the litmus test is applied correctly, practical ML models may approach the golden model's error but cannot surpass it.

B. Minimizing application modeling error

The next question is whether ML models can practically reach the error lower bound estimate e^g. Several I/O modeling works have explored different types of ML models: linear regression [2], decision trees [27], gradient boosting machines [2], [27], [28], Gaussian processes [4], neural networks [5], etc. Here, we explore two types of models: XGBoost [29], an implementation of gradient boosting machines, and feedforward neural networks. These model types are chosen for their accuracy and previous success in I/O modeling.

Neither type of model achieves ideal performance 'out of the box'. XGBoost model performance can be improved through hyperparameter tuning, e.g., by exploring different (1) numbers of decision trees, (2) their depth, (3) the features each tree is exposed to, and (4) part of the dataset each tree is exposed to. Neural networks are more complex, since they require tuning hyperparameters (learning rate, weight decay, dropout, etc.), while also exploring different architectures (number of layers, their size, type, and connectivity). In the case of XGBoost, we exhaustively explore four hyperparameters listed above, for a total of 8046 XGBoost models. In the case of neural networks, exhaustive exploration is not feasible due to state space explosion, so we use AgEBO [30], a Network Architecture Search (NAS) method that trains populations of neural networks and updates each subsequent generation's hyperparameters and architectures through Bayesian logic.

The leftmost column of Figure 1 shows a heatmap of an XGBoost exhaustive search over two parameters on the Theta dataset, with the other two parameters (% of columns and rows revealed to the trees) selected from the best possible result found. The best performing model has an error of 10.51% - close to the predicted bound of 10.01%. The Cori search arrives at a similar configuration with an error of 14.92%.

In the case of neural networks, Figure 2 shows a scatter plot of test set errors of 10 generations of neural networks on the Cori system, with 30 networks per generation. The networks are evolved using a separate validation test to prevent leakage of the test set into the model parameters. Networks approach the estimated error limit, and the best result achieves a median absolute error of 14.3%. After extensive tuning both neural networks and XGBoost models asymptotically approach the estimated limit in model accuracy. Despite the 300 trained neural networks, NAS does little to improve models, since only 6 out of 300 different models improve on previous results (gold stars in Figure 2). This suggests that both types of ML models are impeded by the same barrier and that the architecture and the tuning of models are not the fundamental issue in achieving better accuracy, i.e., that the source of error lies elsewhere.

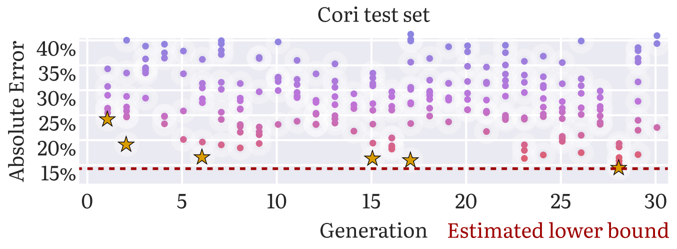


Fig. 2: Results of the Neural Architecture Search (NAS), with the estimated error lower bound highlighted in red.

C. Increasing visibility into applications

While hyperparameter and architecture searches approach but do not surpass the litmus test's estimated lower bound on error, this is not conclusive evidence that all application modeling error has been removed and that error stems from other sources. Possibly, there exist missing application features that might further reduce errors. We explore two such sets of features: MPI-IO logs and Cobalt scheduler logs.

Figure 3 shows the absolute error distribution of hyperparameter-tuned models trained on three Theta datasets: POSIX, POSIX + MPI-IO, and POSIX + Cobalt (Cori excluded because of the lack of Cobalt logs). None of the dataset enrichments help reduce error, corroborating the conclusion that poor application modeling is not a source of error for these models, and further insight into applications will not help. Note that this absence of evidence does not imply evidence of absence, i.e., it does not prove that there exist no features that may help improve predictions. However, this experiment does present a best-effort attempt at exposing novel features, and the model's predictions stay within predicted limits.

Adding Cobalt logs *does* reduce the error on the training set, and ablation studies show that the job start and end time features are the cause. Once timing features are present in the dataset, no two jobs are duplicates due to small timing variations. While previously the ML model was not able to overfit the dataset due to the existence of duplicates, this is no longer the case, and the ML model can differentiate and memorize each individual sample. In [2] authors remove timing features for a similar reason: ML models can learn Darshan's implementation of I/O throughput calculation and make good predictions without observing job behavior.

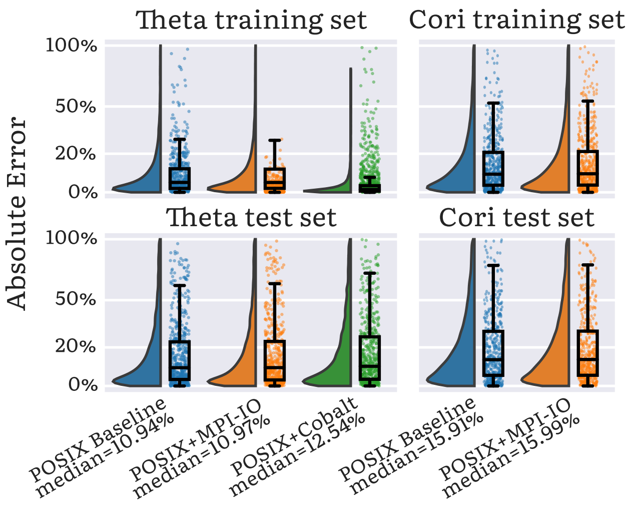


Fig. 3: Error distribution of models trained on POSIX, POSIX + MPI-IO, and POSIX + Cobalt feature sets.

VII. Global system modeling errors

The second part of the approximation error in the taxonomy is the global system modeling error. This error refers to I/O climate and I/O weather effects [22] that affect all jobs running on the system, and corresponds to the second component in Equation 3. While global and local system impact on job performance have complex and overlapping effects, factorizing system impact into impact applied to all jobs versus the impact that is dependent on pairs of concurrent jobs is useful for modeling purposes. The main difference between the two is that modeling local system impact requires modeling relationships between all pairs of concurrent jobs, while modeling global system impacts requires modeling only a single but pervasive influence. In other words, global system impact modeling is insensitive to the number of concurrent jobs running on the system, and can be seen as a form of lossy compression of system state and contention impact on jobs.

We now ask: How does I/O contention impact job I/O throughput prediction? What are the limits of global system modeling? Can I/O models approach this limit? What I/O subsystem features can help improve I/O throughput predictions?

A. *Estimating limits of global system modeling*

Global system impact $\zeta_g(t)$ on job j from Equation 3 can be formalized as some function $\zeta_g(t) = g(J(t))$ where J is the set of jobs running at time t. Since jobs have a start and end time, given a dataset with a dense enough sampling of J, $g(J(t))$ can be calculated for every point in time. During periods of time where e.g., the file system is suffering a service degradation, all jobs on the system will be impacted with varying severity. A model of the system does not need to understand how and why the degradation happened, it only needs to know degradation start and end times, and how different types of jobs were impacted. This time-based model is useless for predicting future performance, and its only utility is in evaluating how much of the degradation can be described as purely a function of time. A deployed model does not have insight into the future and will still need to observe the system.

To evaluate the global system impact, a golden model that exhibits no global modeling error is developed, against which other, 'real' ML models can be compared. Since the global system impact $\zeta_g(t)$ only depends on time t and may ignore the set of all jobs J, only application behavior j and the job start time feature are exposed to the golden model. Both real and golden models have optimized hyperparameters and should have $e_{app} = 0$, but only the golden model has $e_{system} = 0$ (assuming enough data to memorize $\zeta_g(t)$ is available). The litmus test compares these two models to determine $e^p_{system} = e^p - e^g$. Here, a golden model is an XGBoost model fine-tuned on a validation set and evaluated on a test set. Assuming that the golden model is exposed to enough jobs throughout the lifetime of the system, it will learn the impact of $\zeta_g(t)$ even without having access to the underlying system features causing that impact. This golden model is used in the following litmus test:

System modeling error litmus test:
1) Run grid search on real model, find lowest e^p;
2) Insert job start time feature t into the dataset;
3) Run grid search on golden model, find lowest e^g;
4) Calc. system modeling error $e^p_{system} = e^p - e^g$;

If the litmus test is applied correctly, the golden model only suffers from the last three classes of errors: poor generalization, local system impact, and inherent noise. Note that the litmus test is applied on the whole dataset, and not just duplicates, because the less numerous duplicate jobs do not cover the whole lifetime of the system well. In Figure 4 we evaluate a baseline model (blue) and a model enriched with the job start time (orange). Adding a start time feature has a large impact on error: on Cori, the error drops 40%, from 16.49% down to 10.02%, while on Theta the error drops by 30.8%. To obtain this higher accuracy on the POSIX+time dataset, a far larger model is needed, i.e., one that can remember the I/O weather throughout the lifetime of the system.

Note that the timestamp feature fed to the golden model serves no purpose at deployment time, since the ML model cannot learn the state of the system as it is happening. This golden model *is* useful to retrospectively analyze past states and validate that deployment-time models are not suffering from system modeling errors.

B. *Improving modeling through I/O visibility*

With an estimate of minimal error achievable assuming perfect application and global system modeling, we investigate whether I/O subsystem logs can help models approach this limit. Since Theta does not collect I/O subsystem logs, we

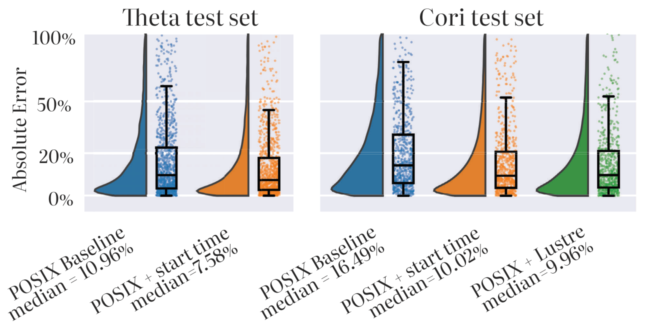


Fig. 4: Error distribution of models trained on (1) POSIX, (2) POSIX + the start time feature, and (3) Darshan and Lustre

analyze Cori, which collects both application and I/O logs. Figure 4 shows the XGBoost performance of three models: a baseline where $e_{app} = 0$ (blue), the litmus test's golden model where also $e_{system} = 0$ (orange), and a Lustre-enriched model (green). Cori's median absolute error is reduced by 40%, from 16.49% down to 9.96%. The Lustre-enriched results are surprisingly close to the litmus test's predictions, and suggest that predictions cannot be improved through further I/O insight since the litmus test's prediction is reached.

VIII. Generalization errors

The remaining three classes of error are caused by lack of data and not poor modeling, as the top branch of the taxonomy shows. While I/O contention and inherent noise errors are examples of aleatory uncertainty and are caused by lack of insight into specific jobs, generalization errors stem from epistemic uncertainty, i.e., the lack of other logged jobs around a specific job of interest. To motivate this section, in the third graph of Figure 1 we show error distribution of a model trained on data from January 2018 to July 2019. When evaluated on held-out data from the same period, the median absolute error is low (green line). Once the model is deployed and evaluated on the data collected after the training period (July 2019 and after), median error spikes up (red line).

A. Estimating generalization error

Estimating the amount of out-of-distribution error e_{ood} is important because any unaccounted OoD error will be classified as noise or contention. This will make systems that run a lot of novel jobs appear to be more noisy than they truly are. Because OoD and ID jobs likely have a similar amount of I/O and contention noise, false positives (ID jobs classified as OoD) are preferable over false negatives, since false negatives contribute to overestimating I/O noise. To estimate the impact of out-of-distribution jobs on error e_{ood}, we aim to quantify how much of the error is epistemic and how much is aleatory in nature, as shown in Figure 1 (upper right). The leading paradigm for uncertainty quantification works by training an ensemble of models and evaluating all of the models on the test set. If the models make the same error, the sample has high aleatory uncertainty, but if the models disagree, the sample has high epistemic uncertainty [31]. The intuition is that predictions on out-of-distribution samples will vary significantly on the basis of the model architecture, whereas predictions on ID but noisy samples will agree and exhibit the same bias. Since this method relies on ensemble to have great model diversity, several works have explored increasing diversity through different model hyperparameters [32], different architectures [33], or both [21]. We choose to use AutoDEUQ [21], a method that evolves an ensemble of neural network models and jointly optimizes both the architecture and hyperparameters of the models. While in theory any type of machine learning model can be used for the model population, neural networks are attractive due to their high hyperparameter count, diverse architectures found in practice, and high generalization capability. Additionally, AutoDEUQ's Neural Architecture Search (NAS) is compatible with the NAS search from section VI, reducing the computational load of applying the taxonomy. Note that in order for AutoDEUQ to correctly split error into e_{ood} vs. $e_{contention} + e_{noise}$, first all application and system modeling errors e_{app} and e_{system} must be removed. Therefore, the function of the NAS is two-fold in this litmus test: (1) eliminate application and system modeling errors, and (2) create a diverse model population. Figure 5 shows the distribution of epistemic (EU) and aleatory uncertainties (AU) of Theta and Cori test sets. For both systems, aleatoric uncertainty is significantly higher than epistemic uncertainty. Furthermore, *all* jobs seem to have AU larger than about 0.05, hinting at the inherent noise present in the system. The inverse cumulative distributions on the margins (red) show what percentage of total error is caused by AU / EU *below* that value. For example, for both systems 50% of all error is caused by jobs with EU below 0.04, while in case of AU, 50% of error is below AU=0.25. The low total EU is expected since the test set was drawn from the same distribution as the training set, and increases on the 2020 set (omitted due to space concerns).

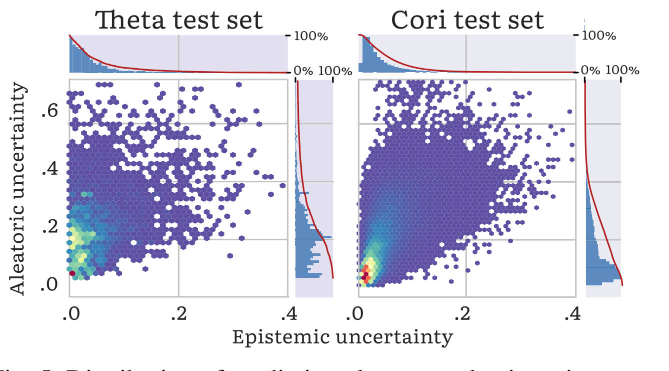


Fig. 5: Distribution of prediction aleatory and epistemic uncertainties for the two systems, with marginal distributions (blue) and inverse cumulative error (red) shown on the margins.

Epistemic uncertainty does not directly translate into the out-of-distribution error e_{ood} from Equation 5. When a sample is truly OoD, it may not be possible to separate aleatory and epistemic uncertainty, since a good estimate of AU requires dense sampling around the job of interest. Therefore, we choose to attribute all errors of a sample marked as out-of-distribution to e_{ood}. This error attribution requires classifying every test set sample as either in- or out-of-distribution, but since EU estimates are continuous values, an EU threshold which will separate OoD and ID samples is required. Although this threshold is specific to the dataset and may require tuning, the quick drop or 'shoulder' in the inverse cumulative error graph around EU=0.1 in Figure 5 makes the choice of an EU threshold robust. A litmus test that estimates the error due to out-of-order samples has the following steps:

Out-of-Distribution error litmus test:

1. Run network architecture search:
   1.1. Minimize e^p_{app} and e^p_{system} for each model;
   1.2. Collect best performing models;
2. Estimate epistemic uncertainty using AutoDEUQ;
3. Find a stable epistemic uncertainty threshold;
4. Classify jobs as either ID or OoD based on threshold;
5. Calculate e_{ood} as the sum of OoD job errors.

On Theta, for an EU threshold of 0.24, .7% of the samples are classified as OoD, but constitute 2.4% of the errors, while on Cori 2.1% of error gets removed for the same EU threshold. In other words, the selected jobs have $3\times$ larger average error than random samples. By visualizing the high-dimensional job features using the Gauge tool [8] and interactively exploring the types of jobs that do get removed, we confirm that OoD-classified jobs are typically rare or novel applications.

IX. I/O Contention and Inherent Noise Errors

With the ability to estimate the amount of application and system modeling error, as well as detect outlier jobs, leftover error is caused by system contention or inherent noise. Both of these error classes are caused by aleatory uncertainty, since the model lacks deeper insight into jobs or the system, as opposed the OoD case where the model lacks samples. While e.g., application error was explainable in terms of broad application behavior (e.g., this application is slow because it frequently writes to shared files, but the model fails to learn this effect), the impact of contention and noise on I/O throughput is caused by lower level, transient effects. Though it may be possible to observe and log such effects through microarchitectural hardware counters or network switch logs, such logging would require vast amounts of storage per job and may impact performance. Lack of practical logging tools makes the last two error categories typically *unobservable*. Furthermore, these two classes may only be separated in hindsight, and while I/O noise levels may be constant, the amount of I/O contention on the system is unpredictable for a job that is about to run.

The questions we ask in this section are: how can errors due to noise and contention be separated from errors due to poor modeling or epistemic uncertainty? Is there a fundamental limit to how accurate I/O models can become? What steps are necessary to quantify system I/O variability?

A. Establishing the bounds of I/O modeling

To separate contention and noise impacts from the first three classes of error, we develop a litmus test based on the test from Section VI. There, by observing sets of duplicates, the error of a golden model e^g was estimated, where $e^g_{app} = 0$. Comparing real models against this ideal model allows for calculating a real model's e_{app}. This litmus test works by 'holding constant' application behavior j within a set of duplicates, i.e., by preventing any input variance from reaching the model. The here introduced noise and contention litmus test seeks to hold constant not only application behavior, but also global system impact, and impact from poor generalization. We design a litmus test that works by enforcing a stronger requirement on duplicate sets, where pairs of jobs are duplicates only if they have the both same application behavior j and same start time t. The test assumes that identical jobs ran at the same time are exposed to the same global system impact $\zeta_g(t)$, but not necessarily the same local impact. The litmus test therefore estimates *the sum* of contention and noise error for a golden model, where only concurrent duplicates are observed and both application behavior j and global system behavior $\zeta_g(t)$ are held static for each duplicate set.

Contention and noise error litmus test:

1. Remove OoD jobs as per previous litmus test;
2. For each set of concurrent duplicate jobs ($\Delta t = 0$):
   2.1. Calculate the set's mean I/O throughput;
   2.2. Apply Bessel's correction to mean;
   2.3. Use mean as golden model prediction;
   2.4. Calculate per-set mean GM absolute error;
3. Calculate $e_{contention} + e_{noise}$ as median of golden model per-set errors.

In the fifth column of Figure 1 we show the distribution of I/O throughput differences $\Delta\phi$ and timing differences Δt between all pairs of Cori duplicate jobs, weighted so that large duplicate sets are not overrepresented. The vertical strip on the left contains Cori duplicate jobs that were ran simultaneously, largely because they were batched together. These jobs share j and ζ_g, but may differ in ζ_l and ω. Due to the denser sampling around 1 minute to 1 hour range, it is not immediately apparent how the I/O difference changes between duplicates ran at the same time and duplicates ran with a small delay. By grouping duplicates from different Δt ranges and independently scaling them, a better understanding of duplicate I/O throughput distributions across timescales can be made, as shown in Figure 6 (Theta shown, Cori omitted due to lack of space). For both systems, the distributions on the right contain jobs ran over large periods of time where global system impact ζ_g might have changed, explaining the asymmetric shape of some of them. The left-most distributions are similar, since variance only stems from contention ζ_l and noise ω. While some distributions (e.g., the 10^5 to 10^6 second) show complex multimodal behavior, all of the distributions seem to contain the initial zero-second ($0s$ to $1s$) distribution.

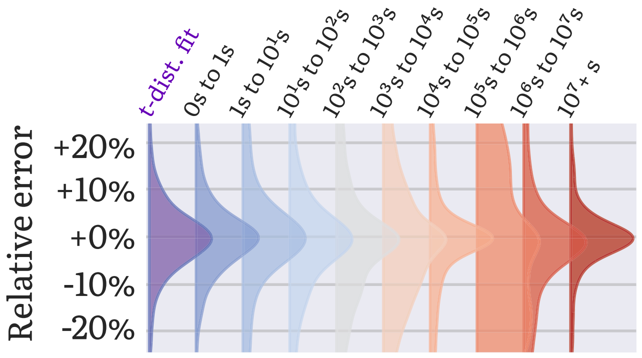
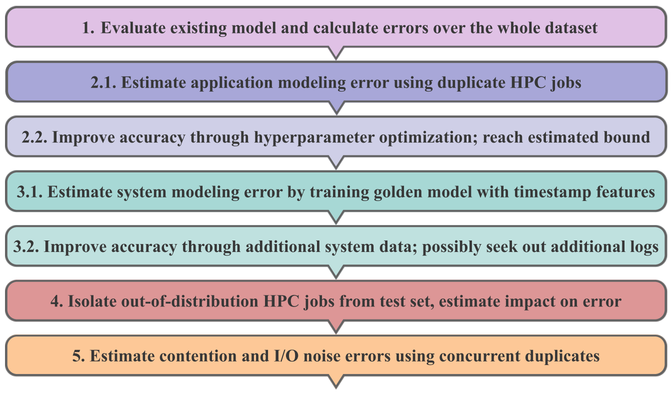


Fig. 6: Distribution of errors for different periods between duplicate runs.



1. Evaluate existing model and calculate errors over the whole dataset

2.1. Estimate application modeling error using duplicate HPC jobs

2.2. Improve accuracy through hyperparameter optimization; reach estimated bound

3.1. Estimate system modeling error by training golden model with timestamp features

3.2. Improve accuracy through additional system data; possibly seek out additional logs

4. Isolate out-of-distribution HPC jobs from test set, estimate impact on error

5. Estimate contention and I/O noise errors using concurrent duplicates

Fig. 7: Framework for applying the taxonomy.

By fitting a normal distribution to the $\Delta t = 0$ distribution (0s to 1s) in Figure 6, we can both (1) learn the lower limit on total modeling error and (2) learn the system's I/O noise level, i.e., how much I/O throughput variance should jobs running on the system expect. However, upon closer inspection, the $\Delta t = 0$ distribution *does not* follow a normal distribution. This is surprising, since if noise follows some (not necessarily normal) stationary distribution, and is independent over time, and its effects are cumulative, according to the central limit theorem the total noise impact is a normal distribution. The answer lies in how the concurrent ($\Delta t = 0$) duplicates are sampled. When observing duplicates, in general, duplicate sets have between 2 and hundreds of thousands of identical jobs in them. However, in duplicate sets with identical start times on Theta, 70% of the sets only have two identical jobs, and 96% have 6 jobs or less, with similar results on Cori. The issue stems from how small (sub-30 sample) duplicate set errors are calculated: when only a small number of jobs exist in the set, the mean I/O throughput of the set is biased by the sampling, i.e., the estimated mean is closer to the samples than the real mean is. This causes the set I/O throughput variance to decrease and therefore duplicate error estimate will be reduced as well. Student's t-distribution describes this effect: when the true mean of a distribution is known, error calculations follow a normal distribution. When the true mean is not known, the biased mean estimate makes the error follow the t-distribution. As the set size increases, the t-distribution approaches a normal distribution. However, naively taking the variance of the t-distribution will produce a biased sample variance σ^2, which can be de-biased by applying Bessel's correction as $\bar{\sigma}^2 = \frac{n}{n-1}\sigma^2$.

With de-biasing in place, we estimate the I/O noise variance of the two systems. Results show that a job running on Theta can expect an I/O throughput within $\pm 5.71\%$ of the predicted value *68% of the time*, or within $\pm 10.56\%$ 95% of the time. For Cori, these values are $\pm 7.21\%$ and $\pm 14.99\%$, respectively. This is a fundamental barrier not just to I/O model improvement, but to predictable system usage in general. Although some insight into contention can be gained through low-level logging tools, noise cannot be overcome. I/O practitioners can use this litmus test to evaluate the noise levels of their systems, and ML practitioners should reconsider how they evaluate models, since some systems may be simply harder to model.

X. Applying the Taxonomy

We now illustrate how the proposed taxonomy can be used in practice. In Figure 7, we show the steps a modeler can follow to evaluate the taxonomy on a new system. Step 1: The modeler splits the available data into training and test sets, and then trains and evaluates some baseline machine learning model on the task of predicting I/O throughput. This model does not have to be fine-tuned, as the taxonomy will reveal the main sources of error and approximately how much the quality of the model is at fault. Step 2.1: The modeler estimates application modeling errors by finding duplicate jobs and evaluating the mean predictor performance on every set of duplicates. Assuming that the distribution of duplicate HPC jobs is representative of the whole population of jobs, this step provides the modeler with a lower bound on the application modeling error. Step 2.2: By contrasting the baseline model error (Step 1) and the estimated application modeling error, the modeler can estimate the percentage of error that can be attributed to poor modeling. The modeler performs a hyperparameter or network architecture search and arrives at a good model close to the bound. Step 3.1: The modeler estimates system modeling errors by exposing the job start time feature to a golden model. This step requires that the modeler has developed a well-performing model in Step 2.2, i.e., one that achieves close to the estimated ideal performance. The test set error of the model serves as an estimate of the application + system modeling lower bound. Step 3.2: The modeler explores adding sources of system data to improve the performance of the baseline model up to the estimated limit of application and system modeling. Step 4: The modeler identifies out-of-distribution samples using AutoDEUQ, calculates OoD error that stems from these samples, and removes them from the dataset. Step 5: The modeler estimates the error that can be attributed to contention and noise, as well as I/O variance of the system. This estimate is made by observing the I/O throughput differences between sets of concurrent duplicates, i.e., duplicate jobs ran at around the same time.

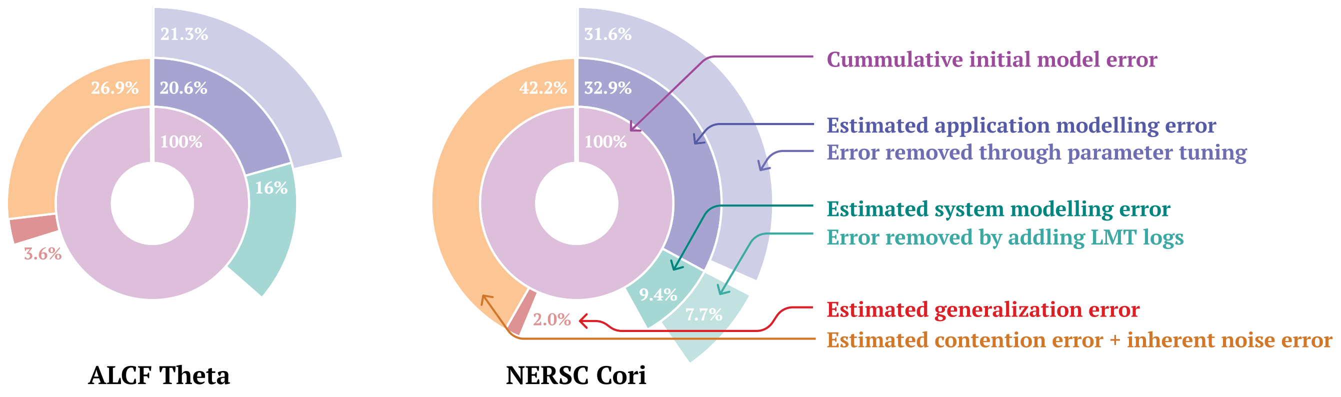


Fig. 8: Results from ALCF Theta and NERSC Cori systems.

In Figure 8 we show the average baseline model error (inner pink circle segment) of both ANL Theta and NERSC Cori systems, and how that error is broken down into different classes of error. We do not focus on the cumulative (total) error value of the two systems; instead, we focus on attributing the baseline model error into the five classes of errors in the taxonomy (middle circle segments of the pie chart), and on the percentage of error that can be removed through improved application and system modeling (outer segments of the pie chart). The inner blue section of the two pie charts represents the estimated application modeling error, as arrived at in Step 2.1. The outer blue section represents how much of the error can be fixed through hyperparameter exploration, as explored in Step 2.2. The inner green section represents the estimated system modeling error, derived in Step 3.1. Note that the total percentage of system modeling error is relatively small on both systems; i.e., I/O contention, filesystem health, hardware faults, etc., do not have a dominant impact on I/O throughput. The outer green circle segment represents the percentage of error that can be fixed by including system logs (LMT logs in our case), as described in Step 3.2. Only the Cori pie chart has this segment, as Theta does not collect LMT logs. On Cori, the inclusion of LMT logs helps remove most of the system modeling errors, reinforcing the conclusion that including other logs (i.e., topology, networking) may not help to significantly reduce errors. The inner red segment represents the percentage of error that can be attributed to out-of-distribution samples of the two systems, as calculated in Step 4. Finally, the yellow circle segment represents the percentage of error that can be attributed to aleatory uncertainty. For both Theta and Cori, this is a rather large amount, pointing to the fact that there exists a lot of innate noise in the behavior of these systems, and setting a relatively high lower bound on ideal model error.

The similarity between the modeling error estimates (Steps 2.1 and 3.1) and the actual updated model performance (Steps 2.2 and 3.2) is surprising and serves as evidence for the quality of the error estimates. However, the estimates of the five error classes *do not* add up to 100%. The first three error estimates are just that - estimates, derived from a subset of data (duplicate HPC jobs) that do not necessarily follow the same distribution as the rest of the dataset and may be biased. If we add the estimates, we see that on Theta 32.9% of the error is unexplained, and on Cori 13.5% of the error is unexplained. Cori's lower unexplained error may be due to the fact that we have collected some 1.1M jobs compared to 100K on Theta.

XI. Discussion and Future Work

Developing production-ready machine learning models that analyze HPC jobs and predict I/O throughput is difficult: the space of all application behaviors is large, HPC jobs are competing for resources, and the system changes over time. To efficiently improve these models, we present a taxonomy of HPC I/O modeling errors that enables independent study of different types of errors, helps quantify their impact, and identifies the most promising avenues for model improvement. Our taxonomy breaks errors into five categories: (1) application and (2) system modeling errors, (3) poor generalization, (4) resource contention, and (5) I/O noise. We present litmus tests that quantify what percentage of model error should be attributed to each class, and show that models improved by using the taxonomy are within a percentage point of an estimated best-case I/O throughput modeling accuracy. We show that a large portion of I/O throughput modeling error is irreducible and stems from I/O variability. We provide tests that quantify the I/O variability and establish an upper bound on how accurate I/O models can become. Our test shows that jobs ran on Theta and Cori can expect an I/O throughput standard deviation of 5.7% and 7.2%, respectively.

In future work, we plan to explore why error classes in Figure 8 do not add up to 100%. Our hypothesis that poor duplicate distribution is the source of this discrepancy, and that instead of duplicate jobs, a targeted set of repeated microbenchmarks may better inform the framework introduced in this work. By tuning and executing microbenchmarks representative of the system's application distribution, we hope to build a minimal set of workloads that evaluate system parameters such as I/O noise amount or application parameters such as I/O contention sensitivity. We also plan to explore how transferable this set of benchmarks is, and whether different HPC system workloads can be accurately represented by a set of weighted microbenchmarks.

References

[1] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A Multiplatform Study of I/O Behavior on Petascale Supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, Jun. 2015, pp. 33–44.

[2] M. Isakov, E. d. Rosario, S. Madireddy, P. Balaprakash, P. Carns, R. B. Ross, and M. A. Kinsy, "HPC I/O Throughput Bottleneck Analysis with Explainable Local Models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[3] K. W. Cameron, A. Anwar, Y. Cheng, L. Xu, B. Li, U. Ananth, J. Bernard, C. Jearls, T. Lux, Y. Hong, L. T. Watson, and A. R. Butt, "MOANA: Modeling and Analyzing I/O Variability in Parallel System Experimental Design," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1843–1856, 2019.

[4] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, and S. M. Wild, "Machine Learning Based Parallel I/O Predictive Modeling: A Case Study on Lustre File Systems," in *High Performance Computing*, 2018, pp. 184–204.

[5] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, G. K. Lockwood, R. Ross, S. Snyder, and S. M. Wild, "Adaptive Learning for Concept Drift in Application Performance Modeling," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019, 2019.

[6] M. Isakov, E. d. Rosario, S. Madireddy, P. Balaprakash, P. Carns, R. B. Ross, and M. A. Kinsy, "Toward Generalizable Models of I/O Throughput," in *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2020, pp. 41–49.

[7] S. Kim, A. Sim, K. Wu, S. Byna, Y. Son, and H. Eom, "Towards HPC I/O Performance Prediction through Large-Scale Log Analysis," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '20, 2020.

[8] E. del Rosario, M. Currier, M. Isakov, S. Madireddy, P. Balaprakash, P. Carns, R. B. Ross, K. Harms, S. Snyder, and M. A. Kinsy, "Gauge: An Interactive Data-Driven Visualization Tool for HPC Application I/O Performance Analysis," in *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*, 2020, pp. 15–21.

[9] E. Ates, O. Tuncer, A. Turk, V. J. Leung, J. Brandt, M. Egele, and A. K. Coskun, "Taxonomist: Application Detection Through Rich Monitoring Data," in *Euro-Par 2018: Parallel Processing*, 2018.

[10] S. Kim, A. Sim, K. Wu, S. Byna, T. Wang, Y. Son, and H. Eom, "DCA-IO: A Dynamic I/O Control Scheme for Parallel and Distributed File Systems," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 351–360.

[11] Y. Li, K. Chang, O. Bel, E. L. Miller, and D. D. E. Long, "CAPES: Unsupervised Storage Performance Tuning Using Neural Network-Based Deep Reinforcement Learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017.

[12] F. Isaila, P. Balaprakash, S. M. Wild, D. Kimpe, R. Latham, R. Ross, and P. Hovland, "Collective I/O Tuning Using Analytical and Machine Learning Models," in *Proceedings of the 2015 IEEE International Conference on Cluster Computing*, ser. CLUSTER '15, 2015.

[13] T. Wang, S. Snyder, G. K. Lockwood, P. Carns, N. J. Wright, and S. Byna, "IOMiner: Large-Scale Analytics Framework for Gaining Knowledge from I/O Logs," *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 466–476, 2018.

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

[15] G. K. Lockwood, S. Snyder, T. Wang, S. Byna, P. Carns, and N. 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*, ser. SC '18. IEEE Press, 2018.

[16] E. Costa, T. Patel, B. Schwaller, J. M. Brandt, and D. Tiwari, "Systematically Inferring I/O Performance Variability by Examining Repetitive Job Behavior," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[17] B. Li, S. Chunduri, K. Harms, Y. Fan, and Z. Lan, "The Effect of System Utilization on Application Performance Variability," in *Proceedings of the 9th International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '19, 2019.

[18] L. Wan, M. Wolf, F. Wang, J. Y. Choi, G. Ostrouchov, and S. Klasky, "Comprehensive Measurement and Analysis of the User-Perceived I/O Performance in a Production Leadership-Class Storage System," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 1022–1031.

[19] B. Xie, Y. Huang, J. S. Chase, J. Y. Choi, S. Klasky, J. Lofstead, and S. Oral, "Predicting Output Performance of a Petascale Supercomputer," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 181–192.

[20] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, and S. Wild, "Modeling I/O Performance Variability Using Conditional Variational Autoencoders," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 109–113.

[21] R. Egele, R. Maulik, K. Raghavan, P. Balaprakash, and B. Lusch, "AutoDEUQ: Automated Deep Ensemble with Uncertainty Quantification," *CoRR*, vol. abs/2110.13511, 2021.

[22] G. K. Lockwood, W. Yoo, S. Byna, N. J. Wright, S. Snyder, K. Harms, Z. Nault, and P. Carns, "UMAMI: A Recipe for Generating Meaningful Metrics through Holistic I/O Performance Analysis," in *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, ser. PDSW-DISCS '17, 2017.

[23] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan, "Watch Out for the Bully! Job Interference Study on Dragonfly Network," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 750–760.

[24] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular HPC I/O Characterization with Darshan," in *5th Workshop on Extreme-Scale Programming Tools (ESPT)*, 2016.

[25] H. Shan and J. Shalf, "Using IOR to analyze the I/O Performance for HPC Platforms," 6 2007. [Online]. Available: https://www.osti.gov/biblio/923356

[26] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[27] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Diagnosing Performance Variations in HPC Applications Using Machine Learning," in *High Performance Computing*, 2017, pp. 355–373.

[28] B. Xie, Z. Tan, P. Carns, J. Chase, K. Harms, J. Lofstead, S. Oral, S. S. Vazhkudai, and F. Wang, "Interpreting Write Performance of Supercomputer I/O Systems with Regression Models," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 557–566.

[29] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, 2016.

[30] R. Egele, P. Balaprakash, V. Vishwanath, I. Guyon, and Z. Liu, "AgEBO-Tabular: Joint Neural Architecture and Hyperparameter Search with Autotuned Data-Parallel Training for Tabular Data," *CoRR*, vol. abs/2010.16358, 2020.

[31] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6405–6416.

[32] F. Wenzel, J. Snoek, D. Tran, and R. Jenatton, "Hyperparameter Ensembles for Robustness and Uncertainty Quantification," *ArXiv*, vol. abs/2006.13570, 2020.

[33] S. Zaidi, A. Zela, T. Elsken, C. C. Holmes, F. Hutter, and Y. W. Teh, "Neural Ensemble Search for Uncertainty Estimation and Dataset Shift," 2020.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We run 10 experiments in the paper, most of which do data science on a set of preprocessed Darshan logs from the ALCF Theta supercomputer.

Figure 1a performs a grid search over XGBoost parameters and shows an error matrix. Figure 1b trains two XGBoost models, one with and one without a job start time feature, and plots their errors vs. time when the jobs were run. Figure 1c trains a model on data collected up to a date, and evaluates the model on data collected before (green) and after that date. Figure 1d finds sets of repeated jobs with identical features from 5 largest applications, and plots how much repeated jobs diverge in I/O throughput. Figure 1e shows the distribution of time and I/O throughput differences for sets of duplicate jobs. Figure 3 trains XGBoost models on different subsets of features (POSIX, POSIX+MPIIO, POSIX+Cobalt) and plots error distributions. Figure 4 evaluates how much the START_TIME feature helps improve the baseline POSIX predictions. Figure 5 plots the distribution of aleatory and epistemic uncertainties collected using the autodeuq.ipynb Jupyter Notebook found in the experiments/ directory. Due to the heavy computational requirements for the notebook, we store the aleatory and epistemic uncertainties in CSV files in the postprocessed_data/ directory. Figure 6 performs the same experiment as Figure 1e but groups samples in different time ranges and only plots 1D histograms fitted with KDE.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://doi.org/10.5281/zenodo.6632461`
Artifact name: Experiments needed to reproduce our SC22 submission titled "A Taxonomy of Error Sources in HPC I/O Machine Learning Models"

Reproduction of the artifact with container: The user needs to: 1. Download and unpack the zip file 2. Enter the directory 3. Build the docker container by running "docker build -t sc22 ." 4. Run the docker container by running "docker run -p 8888:8888 -v $PWD/figures/:/figures/ -it sc22"

The figures generated by the scripts will appear in the figures/ directory. The docker container will also start a Jupyter Notebook, which can be accessed at http://localhost:8888/notebooks/experiments/autodeuq-io.ipynb