ML-based Modeling to Predict I/O Performance on Different Storage Sub-systems

Yiheng Xu*, Pranav Sivaraman[†], Hariharan Devarajan[§], Kathryn Mohror[§], Abhinav Bhatele[†]

*Palantir Technologies, Inc., Denver, USA
[†]Department of Computer Science, University of Maryland, College Park, USA
[§]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, USA
E-mail: [†]bhatele@cs.umd.edu

***Abstract*—Parallel applications can spend a significant amount of time performing I/O on large-scale supercomputers. Fast near-compute storage accelerators called burst buffers can reduce the time a processor spends performing I/O and mitigate I/O bottlenecks. However, determining if a given application could be accelerated using burst buffers is not straightforward even for storage experts. The relationship between an application's I/O characteristics (such as I/O volume, processes involved, etc.) and the best storage sub-system for it can be complicated. As a result, adapting parallel applications to use burst buffers efficiently is a trial-and-error process. In this work, we present a Python-based tool called PrismIO that enables programmatic analysis of I/O traces. Using PrismIO, we identify performance bottlenecks when using burst buffers and parallel file systems, and explain why certain I/O patterns perform poorly. Further, we use machine learning to model the relationship between I/O characteristics and file system selections. We use IOR, an I/O benchmark to gather performance data for training the machine learning model. Our model can predict the better performing storage system for unseen IOR scenarios with an accuracy of 94.47% and for four real applications with an accuracy of 95.86%.**

***Index Terms*—I/O performance, trace analysis tool, benchmarking, machine learning, modeling**

I. Introduction

I/O-intensive high performance computing (HPC) applications can suffer from I/O performance bottlenecks due to slower advances in storage hardware technology as compared to compute hardware [1]. Such I/O bottlenecks can have a significant impact on the overall application performance [2]. Hence, newer I/O sub-systems are being designed to improve applications' I/O performance. Burst buffers are one such example that have gained popularity in recent years.

Burst buffers (BBs) are fast intermediate storage components positioned between compute nodes and hard disk storage [3]. Instead of performing I/O on the parallel file system (PFS), applications can perform I/O via burst buffers and read from/write to the PFS at the end of the job. Studies show that burst buffers can accelerate I/O for a variety of HPC applications [4]–[8]. For instance, Bhimji et al. [7] demonstrate that several types of applications including scientific simulations, data analysis, etc. can achieve better performance on BBs. BBs are also suitable for applications with frequent checkpointing and restart [4].

Although burst buffers have the potential to improve I/O performance, determining if an application could be accelerated by putting files on BBs is a complex task. First, the underlying relationship between an application's I/O characteristics and its performance on I/O sub-systems can be considerably complicated. In this paper, we use *I/O characteristics* to refer to things that decide the I/O behavior of an application, such as I/O volume, I/O library used, etc. For instance, the amount of data transferred per read or write, which we refer to as *transfer size* in the rest of the paper, can impact the I/O sub-system choices. For example, we did several IOR [9] runs keeping all configuration parameters the same except transfer size when writing files on GPFS (the regular parallel file system on Lassen) and BBs on Lassen. We find that for a smaller transfer size, IOR performs better when using BBs, while for a larger transfer size, it performs better on GPFS (Figure 1). Varying other configurations of IOR can also lead to significant performance differences. We did a suite of IOR runs with various configurations, and the fastest configuration runs 10x faster on GPFS than on BB.

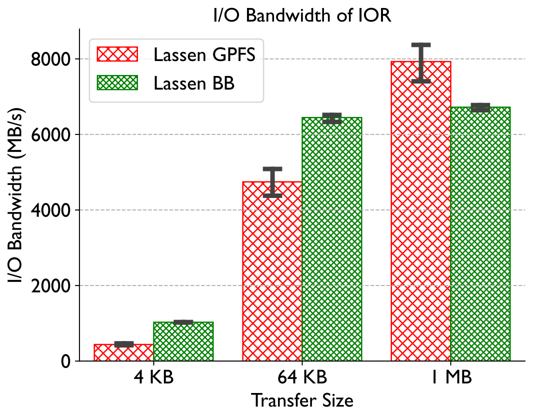


Fig. 1. Comparison of I/O bandwidth achieved for different transfer sizes by IOR when using Lassen GPFS versus burst buffers (BB). Depending on the transfer size, BBs do not always achieve better I/O performance than GPFS.

Second, even though we can manually figure out whether to direct the I/O of an application to BBs by doing experiments, it is a trial-and-error process. To the best of our knowledge, there is no prior work that describes an efficient workflow for

I/O sub-system selection for arbitrary applications. Moreover, even a single application can read from or write to multiple files with different I/O characteristics. I/O to some files can have characteristics that perform better on GPFS, whereas to other files can have characteristics that perform better on BBs. Figuring out the placement of each file manually for large-scale HPC applications is unrealistic.

Lastly, doing detailed performance analyses is challenging as well. Understanding why certain I/O characteristics result in better or poor performance of I/O sub-systems through detailed analyses is important. They also help us validate our decisions of choosing BBs or not. However, doing so is not trivial. Users have to make considerable efforts to write their own codes for the analysis, as existing tools are not efficient enough for detailed analysis. They also lack interfaces that enable users to customize their analysis.

We address the challenges above by designing a methodology to conduct detailed analysis and model the selection of storage sub-systems using machine learning (ML) for HPC applications. We first present our pandas-based I/O performance analysis tool, PrismIO, and several case studies that demonstrate its usefulness. PrismIO provides a Python data structure and API that enable users to customize their analysis programmatically. It also provides an API to extract the I/O characteristics of an application. Next, we model the relationship between I/O characteristics of HPC applications and performant I/O sub-system selections using machine learning. We run IOR with various I/O configuration parameters on several I/O sub-system installations and collect performance data. Using this data, we train ML models that select the best I/O sub-system for files read/written in a program based on its I/O characteristics. Finally, we present a workflow that efficiently extracts I/O characteristics of applications, executes the ML model, and outputs a file placement plan.

The main contributions of this work are:

- We build a Python-based tool called PrismIO that enables detailed analysis of I/O traces. We identify and reason about different patterns/bottlenecks of I/O sub-systems.
- We conduct detailed experiments to study the relationship between I/O characteristics and the best storage sub-system choices. The resulting dataset can be used to model this relationship.
- We train a machine learning model that approximates this relationship on Lassen. The model predicts whether to place files on BBs or GPFS based on I/O patterns with 94.47% accuracy.
- To make it easy to train ML models, we add API functions in PrismIO that extract I/O characteristics and integrate the modeling workflow into PrismIO. Using this workflow, we predict the best storage sub-system choices for four real applications with 95.86% accuracy.

II. Background & related work

In this section, we provide relevant literature needed to understand our work. We introduce BB, its architecture, and how it benefits I/O performance based on previous research. We also provide previous works that study how to better use BB and the limitations they found. Besides, we introduce performance analysis tools for I/O and their limitations.

A. *Burst buffers*

Burst Buffers (BBs) are fast intermediate storage layers between compute nodes and storage nodes [3]. The traditional HPC I/O architecture consists of on-node memory, I/O nodes for handling I/O requests, and a disk-based parallel file system [10]. But this architecture is not enough to fill the gap between computational performance and I/O performance [7]. Therefore, BB is a natural solution to be introduced between these layers. BBs are implemented on several state-of-the-art HPC clusters such as Lassen, Summit, and Cori.

Burst buffers can be implemented in various ways. In terms of hardware, BBs are mostly made with SSDs. For example, Lassen at LLNL uses 1.6 TB Samsung PM1725a NVMe SSDs [11]. In terms of architecture, there are two major types: node-local and shared BBs [12]. Lassen and Summit adapt node-local BBs. In this case, a BB is directly connected to a compute node. All BBs are independent, which means they don't share the same namespace. For the shared BBs (as used in Cori), BB nodes are separated from compute nodes and can be accessed by multiple compute nodes through interconnected networks [13]. In this case, compute nodes perform I/O on the global BB file system with a shared namespace. In this paper, we focus on node-local BBs.

Research has demonstrated the benefit BB brings to I/O performance [2]. For example, Bhimji et al. analyze the performance of a variety of HPC applications on Cori burst buffers, including scientific simulations, visualization, image reconstruction, and data analysis [7]. They demonstrate that burst buffers can accelerate the I/O of these applications and outperform regular Parallel File Systems (PFS). Pottier et al. [14] also demonstrate that node-local burst buffer significantly decreases the runtime and increases the bandwidth of an application called SWarp. From the research mentioned above [2], [7], [14], it is evident that BB has the potential to improve the I/O performance of certain HPC applications.

Although these papers demonstrate BB is promising in improving I/O performance, most of them raise the same concern: The performance is sensitive to application I/O patterns [7], [8], [14]. Pottier et al. demonstrated different workflows may not get the same benefit from BBs [14]. Some of them have decreased performance compared with the performance on regular file systems. For instance, they indicated that the shared burst buffer can perform worse than the regular PFS and is sensitive to the amount of data transferred per read/write operation. However, their work is only based on a single application called SWarp so it might not be general enough to cover common I/O patterns of HPC application. Existing research on this issue has not given a general solution to the problem due to insufficient experiments and modeling to cover different I/O patterns. Besides, the metric they use is the total runtime, which may not be a good I/O metric as it can be significantly affected by work other than I/O like computation. To summarize, the underlying relationship between an application's I/O characteristics and the storage sub-system choice is still unclear.

B. I/O performance analysis tools

There exist I/O performance analysis tools that can assist in studying such relationships. Two popular I/O performance analysis tools are Darshan [15], [16] and Recorder [17]. Both of them trace application runs and capture basic performance information such as time and I/O volume of an operation. They provide visualization scripts upon their traces. With them users can derive aggregated I/O metrics such as I/O bandwidth, file sizes, etc. These tools can be helpful to start with the I/O sub-systems selection research question, however, they are not efficient enough for detailed analysis due to the lack of interfaces that enable users to customize their analysis. Users have to make significant efforts to write their own codes to achieve deeper discoveries. Besides these two tracing tools, there is an automated analysis project called Tokio [18], [19], short for total knowledge of I/O. The project collects the I/O activities of all applications running on NERSC Cori for a time period and then provides a clear view of the system behavior. It does a great job in holistic I/O performance analyses such as detecting I/O contention and unhealthy storage devices at the system level. However, there is no application-level analysis so it does not map I/O characteristics to I/O sub-system selections. Two other tools, Hatchet [20] and Pipit [21], provide programmatic APIs similar to PrismIO for analyzing performance profiles and traces respectively.

III. Data collection

In order to understand the relationship between I/O characteristics and performance on different I/O sub-systems, we decided to collect a significant amount of benchmarking data. This data would also be useful for training the machine learning models. A cornerstone of our work is to carefully design experiments to build a large and unbiased dataset. In this section, we introduce our experimental setup and data collection in detail. We go through machines, benchmark configurations, and how we process the data.

A. HPC platform used for the experiments

We conduct our experiments on Lassen. Lassen is a GPU-based cluster at LLNL with IBM Power9 CPUs and NVIDIA V100 GPUs. It uses IBM's Spectrum Scale parallel file system, formerly known as GPFS. GPFS is one of the most common implementations of parallel file systems (PFS). It is used by many of the world's fastest supercomputers [22]. In terms of burst buffers, Lassen uses node-local BBs implemented with Samsung NVMe PCIe SSDs.

B. The IOR benchmark and its configuration

We use the IOR benchmark [9] for benchmarking the I/O sub-systems on Lassen and collecting I/O performance traces. We chose IOR because it is easily configurable to emulate various HPC workflows. Also, it has been shown to be an effective replacement for full-application I/O benchmarks [23], [24]. IOR has more than 40 command line configuration parameters. In our experiments, we select representative I/O characteristics that are common in real applications and configure the corresponding configuration parameters in IOR. We emulate the I/O characteristics of real applications by covering all possible combinations of the selected configuration parameters. For instance, we include collective MPI-IO because it is commonly used in checkpoint/restart applications. We further separate I/O into six types: RAR, RAW, WAR, WAW, RO, and WO, to better map IOR runs to different types of I/O in production applications (RAR stands for read after read, RAW for read after write, RO for read only, etc.). For instance, RAR and RO are common in ML applications between epochs. We again cover all feasible combinations of these read/write types and other I/O characteristics (configuration parameters). In addition, all runs are done using GPFS and node-local burst buffers on Lassen separately.

We do the following to take performance variability into account. We repeat each run five times at different times and days to minimize the impact of temporal system variability. Since our whole experiment contains around 35,000 runs, we avoid adding abnormal burdens to the system by distributing our IOR experiments over time. We run our jobs sequentially so that at one time there is only one IOR run submitted by us running on the system. We collect all I/O metrics from IOR standard output, including total I/O time, read/write time, bandwidth, etc, and take the mean over the five repetitions. Each IOR run produces a sample for the training dataset.

TABLE I
Description of data used for training the ML models.

Input I/O feature	Description
I/O interface	Categorical feature Can be POSIX, MPI-IO, HDF5
collective	Enable collective MPI-IO
fsync	Perform fsync after POSIX file close
preallocate	Preallocate the entire file before writing
transfer size	The amount of data of a single read/write
unique dir	Each process does I/O in its own directory
read/write per open to close	The number of reads/writes from open to close of a file
use file view	Use an MPI datatype for setting the file view to use individual file pointer
fsync per write	perform fsync upon POSIX write close
I/O type	Categorical feature can be RAR, RAW, WAR, WAW, etc.
random access	whether the file is randomly accessed

C. Data processing

The data used for analysis and model training is derived from IOR runs with different configurations on Lassen GPFS and BB, where each IOR run is a sample in the dataset. The independent variables are all the I/O features (configuration parameters) we included in the experiments. Descriptions of these input features are presented in Table I. We first process them to make them suitable for common ML models. For example, most ML models cannot directly handle string-formatted categorical columns. Also, if a column is nominal instead of ordinal, simply converting it into integers ranging from 0 to the number of categories may result in erroneous inference. One popular solution is using one-hot vector encoding. It works by creating as many columns as the number of categories for a categorical column in the data set. Each column represents a category. We set a column to 1 if the original value is the category represented by that column and 0 for all other columns. We encode all of our categorical features such as file system used, I/O interface, etc., into one-hot encoded vectors.

As our goal is to predict the best I/O sub-system to place a file, the dependent variable should be the I/O sub-system. Since we run every I/O feature combination on both GPFS and BBs, the ground truth (oracle) label is the I/O sub-system (GPFS or BB) that gives the best I/O bandwidth for each run. The exhaustive I/O feature combinations plus the true label make our final dataset. The final dataset has 19 feature columns and 2967 rows.

IV. PrismIO: An I/O performance analysis tool

We have created PrismIO, a Python library on top of pandas [25], [26], as a solution to better facilitate detailed I/O trace analysis. It has two primary benefits. First, it enables users to perform analysis on a pandas DataFrame instead of an unfamiliar trace format. Based on the DataFrame structure, PrismIO provides an analysis API that assists users in efficiently doing detailed analyses. Second, PrismIO provides functions that can automatically extract the I/O characteristics of applications from their I/O traces. This is the foundation for applying our ML modeling workflow to new applications because I/O characteristics are the input to the model.

A. Data organization in PrismIO

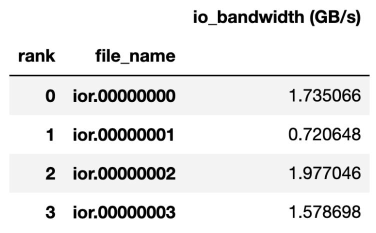
PrismIO is primarily designed for analyzing I/O traces, and we currently support traces gathered using Recorder [17], which is an I/O tracing tool. Recorder instruments function calls from common I/O libraries such as fwrite, MPI_file_write, etc. We also plan to support Darshan traces in the future via their Python API. We implement PrismIO as a Python library that builds on pandas to organize trace data into a DataFrame, and build analysis functions on top of it.

The primary class in PrismIO that holds the trace data and provides user-facing functions is called IOFrame. It reorganizes the input I/O trace into a pandas DataFrame. Each row in the DataFrame is a record of a function call. Columns store numerical and categorical information of the function calls, including start time, end time, MPI rank that calls the function, etc. It also explicitly lists information that is non-trivial to retrieve from complex trace structures such as read/write access offsets of a file.

B. API functions for analyzing I/O performance

PrismIO provides several analytical functions that report aggregated information from different aspects, such as I/O bandwidth, I/O time, etc. It also provides a feature extraction API that extracts important I/O characteristics that end users may be interested in. Feature extraction will be used in our prediction workflow that we will discuss in Section V. In addition, PrismIO provides preliminary visualization capabilities for inspecting I/O traces visually. Below, we discuss some of the important analysis functions provided in the PrismIO API.

io_bandwidth Users often inspect the achieved I/O bandwidth for different file read/write operations by different processes to understand the I/O performance of their code. The `io_bandwidth` function provides the I/O bandwidth for individual files per MPI process. This is implemented by a groupby operation on the DataFrame by file name and MPI rank, and then calculating the bandwidth by dividing the total I/O volume by I/O time in each group (per file, per MPI rank). This function also provides options for quick filtering. Sometimes, the user wants to focus on certain MPI ranks in a parallel execution. We provide a "`rank`" argument that takes a list of ranks and filters by ranks that are listed before the groupby and aggregation. We also provide a general filter option where users can specify a filter function. Figure 2 shows the few lines of code to use this function and the resulting DataFrame output.



		io_bandwidth (GB/s)
rank	**file_name**	
0	**ior.00000000**	1.735066
1	**ior.00000001**	0.720648
2	**ior.00000002**	1.977046
3	**ior.00000003**	1.578698

```
ioframe = IOFrame.from_recorder("/path/to/recorder/
    trace")
ioframe.io_bandwidth(rank=[0,1,2,3], filter=lambda x
    : x["I/O_type"] == "read")
```

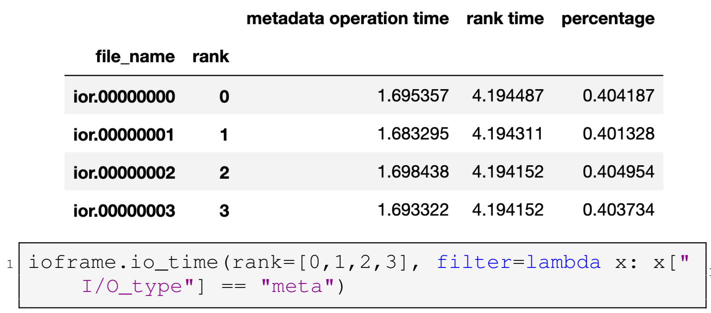
Fig. 2. I/O bandwidth achieved by MPI ranks 0–3 for each file they read (in a sample IOR trace.) `io_bandwidth` returns a DataFrame with a hierarchical index. We utilize the filter option to only report read bandwidths of ranks 0-3 for different files. The function automatically selects the most readable unit for bandwidth, in this case GB/s.

io_time `io_time` returns the time spent in I/O operations per file and per MPI process. Similar to `io_bandwidth`, users can filter by different things. Figure 3 demonstrates an example of using it to print the time spent in metadata operations (open, close, sync, etc.) on each file by ranks 0, 1, 2, and 3.

shared_files Another common analysis in parallel I/O is to study which files are shared across MPI processes. `shared_files` reports for each file, how many processes are sharing it and what their ranks (IDs) are. Figure 4 shows part of the output of `shared_files` for an I/O trace.

C. Feature extraction API

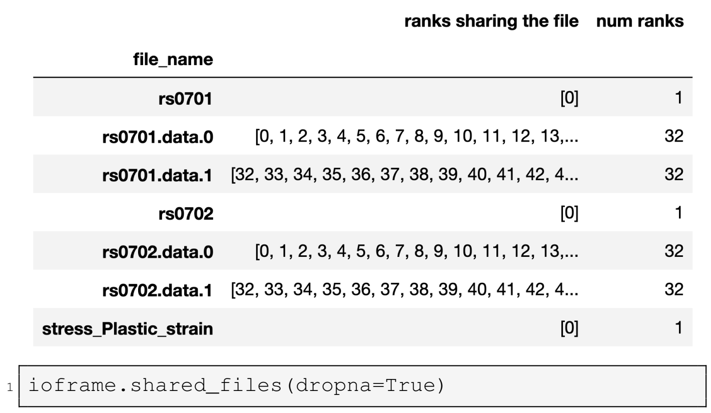
Feature extraction is a class of API functions in PrismIO that extract the I/O characteristics of an application from



file_name	rank	metadata operation time	rank time	percentage
ior.00000000	0	1.695357	4.194487	0.404187
ior.00000001	1	1.683295	4.194311	0.401328
ior.00000002	2	1.698438	4.194152	0.404954
ior.00000003	3	1.693322	4.194152	0.403734

```
ioframe.io_time(rank=[0,1,2,3], filter=lambda x: x["
    I/O_type"] == "meta")
```

Fig. 3. Time spent in metadata operations by ranks 0–3 for each file they read (in a sample IOR trace.) In addition to absolute time, `io_time` also reports the percentage of total time spent performing I/O.



file_name	ranks sharing the file	num ranks
rs0701	[0]	1
rs0701.data.0	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,...	32
rs0701.data.1	[32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 4...	32
rs0702	[0]	1
rs0702.data.0	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,...	32
rs0702.data.1	[32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 4...	32
stress_Plastic_strain	[0]	1

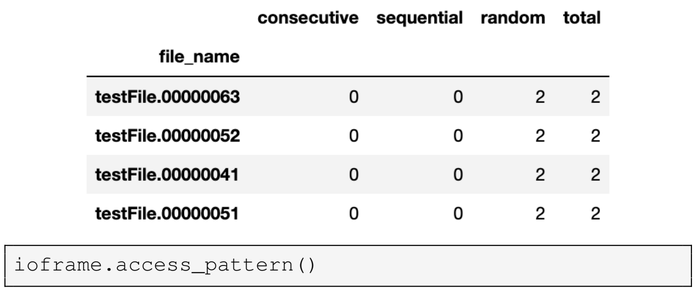
```
ioframe.shared_files(dropna=True)
```

Fig. 4. A screenshot of a part of the DataFrame output by `shared_files` for a sample trace. `shared_files` demonstrates how files are shared across processes. Users can easily observe that some files are shared while others are not in a run using 32 MPI processes.

its I/O trace. As mentioned earlier, the primary purpose of these functions is to prepare inputs to the machine learning model. In the machine learning context, the inputs to a model are called features. As our model predicts whether to place files on BBs based on I/O characteristics of applications, I/O characteristics are the input features to our model. In the rest of the paper, when we discuss inputs to the model, we use I/O features to refer to I/O characteristics. Such features include I/O type (RAR, RAW, WAR, etc.), access pattern (random or sequential), file per process (FPP), I/O interface (POSIX, MPI-IO, etc.), etc. For an arbitrary application, users either need to read the source code or analyze the trace to obtain these features, which may require considerable time. Using PrismIO, users can get these features via simple function calls as shown below.

access_pattern `access_pattern` counts the number of different access patterns per file including consecutive, sequential, and random access. Figure 5 demonstrates the `access_pattern` output of an example trace. This can be a useful feature in determining I/O performance.

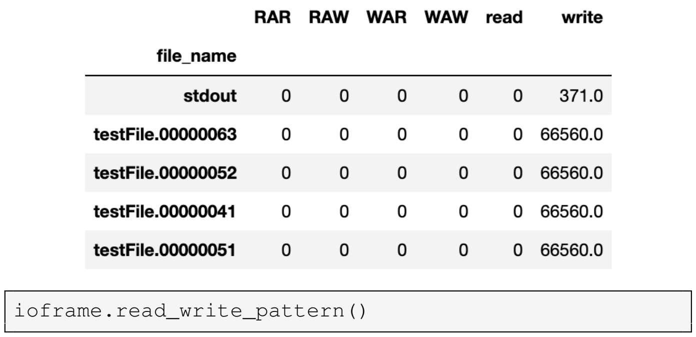
readwrite_pattern `readwrite_pattern` identifies all the read/write patterns in an I/O trace such as RAR, RAW, etc., and calculates the amount of data transferred using those patterns. Figure 6 demonstrates the output of `readwrite_pattern` of a sample trace.



file_name	consecutive	sequential	random	total
testFile.00000063	0	0	2	2
testFile.00000052	0	0	2	2
testFile.00000041	0	0	2	2
testFile.00000051	0	0	2	2

```
ioframe.access_pattern()
```

Fig. 5. A screenshot of a part of the DataFrame when using `access_pattern` for a sample trace. It counts the number of accesses of different types. In this case, there are two random accesses on files and no other type of access. From the counts, users can figure out what kind of accesses are the most frequent in a run.



file_name	RAR	RAW	WAR	WAW	read	write
stdout	0	0	0	0	0	371.0
testFile.00000063	0	0	0	0	0	66560.0
testFile.00000052	0	0	0	0	0	66560.0
testFile.00000041	0	0	0	0	0	66560.0
testFile.00000051	0	0	0	0	0	66560.0

```
ioframe.read_write_pattern()
```

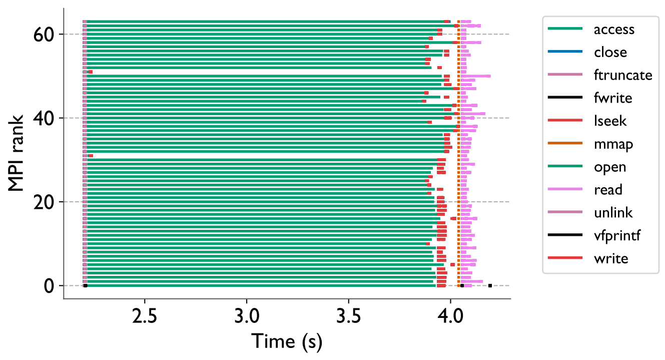
Fig. 6. A screenshot of a part of the DataFrame when using `readwrite_pattern` for a sample trace. It calculates the amount of data transferred using different read/write patterns. This example run only does write so the pattern of all files is write only (WO).

D. Visualization API

timeline Often times, end users want to inspect I/O traces visually to understand and explore the trace. While this is not recommended for extremely large traces due to scalability issues, we provide basic visualization support in PrismIO through the `timeline` function. It plots the timeline of function calls for each MPI process. Each horizontal block represents the time interval of a function execution, from start to end. The `timeline` function also provides filter options to select certain types of functions. Figure 7 demonstrates an example timeline plot of I/O functions for a sample trace.

V. ML MODEL FOR SELECTING BEST I/O SUB-SYSTEM

We use machine learning models to predict and recommend which file(s) should be placed on which I/O sub-system for best performance, which we refer to as "file placement" in the rest of the paper. We train an ML model that selects the best I/O sub-system for application file placements. Moreover, to efficiently apply the model to an application whose I/O characteristics are not known, we make the selection process



```
ioframe.timeline(filter=lambda x: x["function_type"]
    == "I/O")
```

Fig. 7. Timeline of I/O functions in a sample trace. X-axis shows time, and y-axis shows the MPI rank. Users can easily make observations such as functions that are abnormally slow, load imbalance, temporal orders of function calls, etc. For this example, users can easily identify that `access` runs slow on most processes, but fast on two specific processes. Users can efficiently get important insights and get directions for further analysis.

part of a workflow with assistance from PrismIO's feature extraction API. In this section, we discuss the design of our ML methodology and overall workflow.

A. Models for predicting file placement

The key to answering whether to place files written by an application on burst buffers is to understand the relationship between the I/O characteristics of an application and its performance on different I/O sub-systems. As mentioned in Section IV, we use the term I/O features to refer to I/O characteristics that we use as inputs to our model. Since some features are continuous such as transfer size and numbers of reads/writes, we need a model to interpolate the relationship. And since the parameter space is large, heuristic-based methods may not work well and are difficult to extend when adding new I/O features. Therefore, we decided to use machine learning (ML) techniques to model such relationships. An application can have multiple files with very different I/O patterns. These files from the same application can usually be placed on different I/O sub-systems. Therefore, our model focuses on predicting for individual files instead of the whole application. Ideally, it predicts the best I/O sub-system for each file and thus results in the best file placement plan overall for the application.

Since Lassen uses node-local BBs, the decision to be made is either using them or not. So we pose the ML prediction as a binary classification problem. We evaluate several different classifiers and compare their test accuracies. For our evaluation We do a 90-10 split of our dataset. We train the model with 90% of the data, and then use the rest 10%, which is unseen by the model, to obtain test accuracies by comparing predictions with the true labels. We train and evaluate each classifier 10 times with random data splits. and report the average test accuracy of the classifiers.

B. Workflow for identifying best performing I/O sub-system

We make the selection process efficient and easy-to-use for the end user by developing a workflow. This is because applying the ML model to unseen large-scale HPC applications with a large number of different files is non-trivial. When end users do not know the I/O features of an application, they cannot use the model easily. To solve this problem, we combine the feature extraction API in PrismIO with the model and make them into a workflow. Figure 8 highlights the workflow design and demonstrates how different parts and their inputs/outputs fit together in the workflow. Given an application, the user should only need to run it once on any machine and trace it with Recorder. Then the workflow takes the Recorder trace as input, extracts I/O features for each file, and feeds them into the pre-trained model to predict the best file placement per file i.e. which file should be placed on which I/O sub-system.

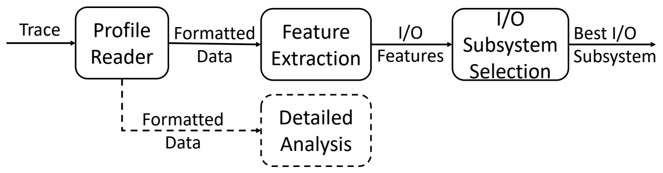
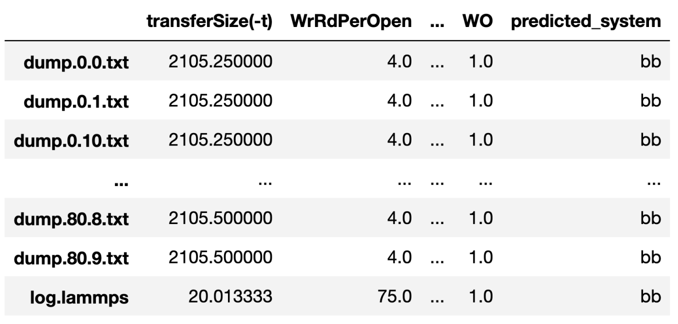


Fig. 8. Overview of our modeling workflow. Users provide the raw trace of their application run and PrismIO restructures it into a programmable data structure (IOFrame). Then PrismIO extracts I/O features from the IOFrame object. These features are provided as input to the model that decides the best I/O sub-system to use for this application.

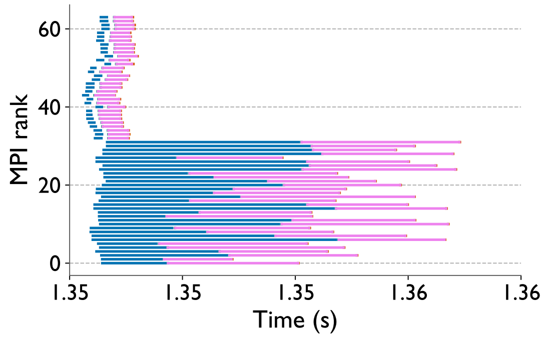
We have implemented this workflow into a function called `predict` in PrismIO. Users simply need to call it with the trace path. It groups the data by file names and calls the feature extraction API on each of them. It then loads the pre-trained model and predicts whether to use BBs for each file. The I/O features and predictions are organized as a DataFrame indexed by file name. Figure 9 demonstrates a part of the workflow output of a sample trace.



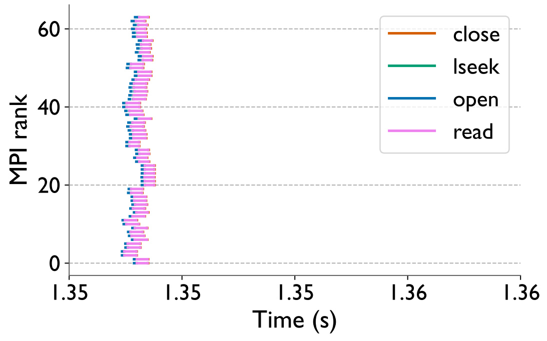
	transferSize(-t)	WrRdPerOpen	...	WO	predicted_system
dump.0.0.txt	2105.250000	4.0	...	1.0	bb
dump.0.1.txt	2105.250000	4.0	...	1.0	bb
dump.0.10.txt	2105.250000	4.0	...	1.0	bb
...	...	...	...	...	...
dump.80.8.txt	2105.500000	4.0	...	1.0	bb
dump.80.9.txt	2105.500000	4.0	...	1.0	bb
log.lammps	20.013333	75.0	...	1.0	bb

```
result = predict("/path/to/recorder/trace")
```

Fig. 9. A screenshot of a part of the DataFrame of the prediction workflow output for a sample trace. Each row corresponds to a file. We can easily understand what are the I/O features of this file, and given those features, which I/O sub-system should be used for this file. In this example, we know that all I/O to these files are write operations, and there are four writes with an average size of 2105.25 bytes. Based on this, all of these writes are predicted to perform better when placed on BB.



(a) Lassen GPFS



(b) Lassen BB

Fig. 10. Timeline of I/O function calls in IOR traces from GPFS and BB using the configuration from the extreme point "Good BB Bad GPFS". We run them on two nodes and 32 processes per node.

VI. I/O ANALYSIS CASE STUDIES

In this section, we present some case studies to demonstrate the process of I/O performance analysis using PrismIO.

A. *Extreme runs analysis*

We first demonstrate how PrismIO can be used for performance analysis of specific runs where the I/O performance is unusual. We look at all IOR runs in the dataset introduced in Section III, and identify some "outlier" or "extrema" runs with extremely poor or good performance when using the PFS versus BB. We call such runs as extreme runs. Figure 11 plots the achieved I/O bandwidth of IOR runs (using 64 processes on 2 nodes) on GPFS or BB on Lassen. Each point on the plot represents a single run with a certain I/O feature combination. The x-axis represents the I/O bandwidth on GPFS and the y-axis shows the I/O bandwidth on BB. The farther the point from the diagonal, the IOR run with that configuration performs more differently on PFS and BB.

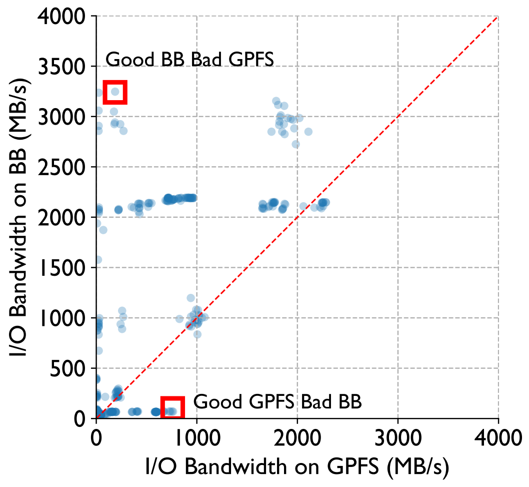
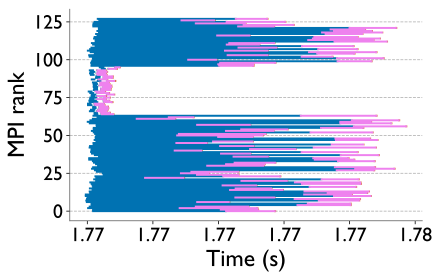


Fig. 11. I/O bandwidth of IOR benchmarking runs on PFS vs BB. Each point on the plot represents a run with a certain I/O feature combination. The x-axis is the I/O bandwidth on the PFS and y-axis is the I/O bandwidth on BBs. The plotted value is the average over five repetitions of that I/O configuration as mentioned in Section III. The red boxes highlight the most extreme runs we picked for deeper analysis.

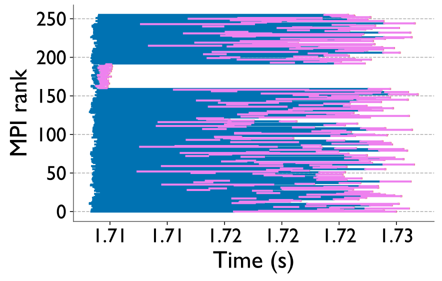
We pick the highlighted runs (red boxes) in Figure 11 to conduct detailed analyses. The upper left one performs much better on BBs than on GPFS. The lower middle one performs much better on GPFS than on BB. We refer to them as "Good BB Bad GPFS" and "Good GPFS Bad BB" in later text. Since they perform very differently on the two I/O sub-systems, they may indicate bottlenecks in the individual systems. We trace these extreme runs with Recorder, and then do detailed analyses with PrismIO.

Lassen GPFS bottleneck analysis: We first present the analysis for the "Good BB Bad GPFS" configuration on Lassen. The I/O bandwidth per process for that run on GPFS is 155.01 MB/s, whereas on BB it is 3236.24 MB/s. Such a significant gap indicates there must be performance issues on GPFS for this run. We first use PrismIO's `io_time` function, and find that the run on GPFS spends 37.8% of the I/O time in metadata operations. We then use the `timeline` function introduced in Section IV to visualize the I/O function traces (Figure 10). We immediately observe a significant load imbalance of `open` and `read` over MPI processes when using GPFS: the first 32 processes on one node run much slower than the second 32 processes on the other node. This does not seem to be the case when the same configuration is run using BBs. We checked the source code of IOR and logically all processes do exactly the same thing, so it is likely an issue with the I/O sub-system. To further drill down, we check if this slowdown is happening because a particular node is problematic or if it is a broader system issue.

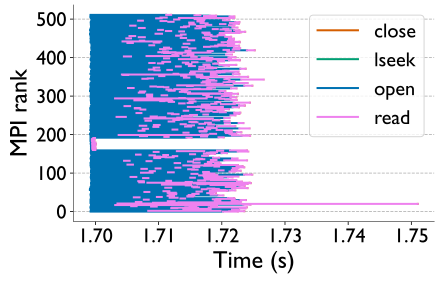
We run IOR with the same configuration but on more nodes. Figure 12 shows the timeline of IOR runs on 4, 8, and 16 nodes. We notice that there is a single node that performs much better than the other nodes, so we checked if it is the same physical node in these three runs. Unfortunately, we did not observe a unique node. Therefore, we conclude that it is an issue with GPFS in balancing metadata operations such as `open`. We checked with Lassen system administrators and confirmed that the most likely reason is that GPFS does not have metadata or data servers. When writing a large number of files to the same directory, one of the nodes becomes the



(a) IOR on Lassen GPFS (4 nodes)

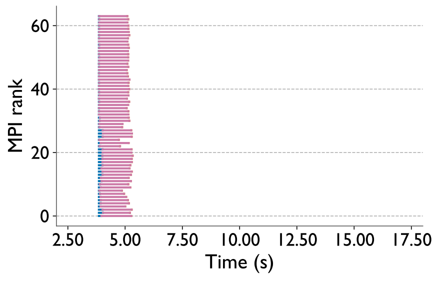


(b) IOR on Lassen GPFS (8 nodes)

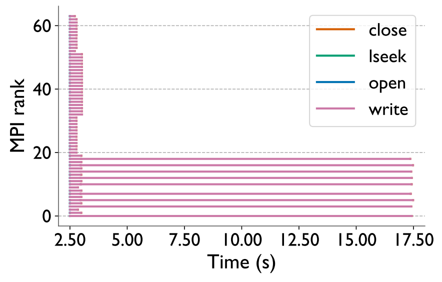


(c) IOR on Lassen GPFS (16 nodes)

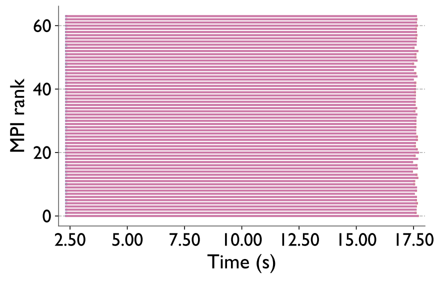
Fig. 12. I/O timeline of IOR runs with "Good BB Bad GPFS" configuration using Lassen GPFS on 4, 8, and 16 nodes. Each node has 32 MPI processes. Irrespective of the number of nodes used, there is one specific node that performs much better than other nodes.



(a) Lassen GPFS



(b) Lassen BB



(c) Lassen BB with direct I/O

Fig. 13. Timeline of I/O function calls in IOR traces from GPFS and BB using the configuration from the extreme point "Good GPFS Bad BB". We run them on two nodes and 32 processes per node.

metadata server for that directory and thus the MPI processes on it are much faster in metadata operations than processes on other nodes.

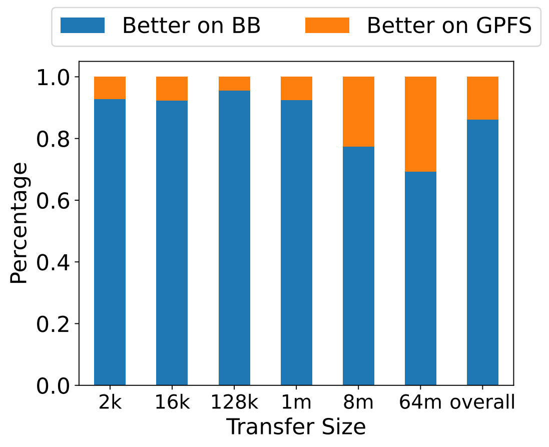
Lassen BB bottleneck analysis: Similar to the analysis above, we also dive deeper to analyze the bottlenecks of Lassen BBs for the "Good GPFS Bad BB" configuration. The configuration is also run on 2 nodes with 32 processes per node. Figure 13 (a) and (b) show the timeline for I/O function calls of each process in that configuration on GPFS and BB, respectively. This time we observe a significant load imbalance over processes in BB. Some processes perform much worse than others, and such processes are spread across both nodes. However, faster processes in the BB run actually perform better than in the GPFS run, and have have unexpectedly high bandwidths, which indicates that they are most likely taking advantage of the SSD hardware cache. As the SSD hardware cache is part of the BB architecture, in most real-world use cases, applications use it by default. To confirm this hypothesis, we run these IOR runs with the same configurations but with direct I/O enabled, in which case all I/O traffic would skip any kind of cache and directly go to storage.

Figure 13 (c) demonstrates the I/O timeline of the same run on BB but with direct I/O enabled. All processes now take the same long time on BBs. This confirms that the load imbalance over ranks on BBs is due to the SSD hardware cache. GPFS is a shared file system so its hardware cache is likely to be occupied by millions of other workloads, while the hardware cache of node-local BBs is not filled until the total amount of I/O exceeds the cache size. When the hardware cache is filled, processes performing I/O that cannot use the cache run much slower than processes that are able to use the cache. Therefore, the empirical conclusion that large I/O may not perform well on BB is not always true. A single large I/O can perform better on BB than on GPFS if the BB hardware cache is not full. Users can do large I/O on BB, as long as the total amount of I/O at a time does not overwhelm the hardware cache.

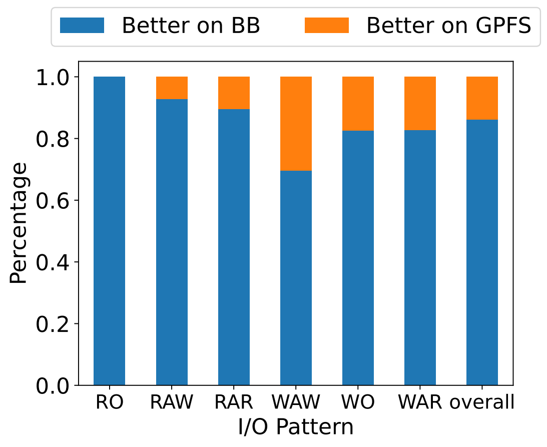
The above detailed case studies demonstrate that PrismIO can help in identifying detailed I/O performance issues in parallel executions. The above analyses used PrismIO's Python API and the `timeline` visualization.

B. *Overall trends*

From the previous case studies, we also identified that configurations with certain values of some I/O features are better suited to using BB. To summarize such trends, we look at two specific I/O features: transfer size and I/O pattern. We look at all values of each of these features that we experimented with and for each fixed value, we plot the fraction of configurations that perform better when using GPFS vs BB. Figure 14 shows the percentage of configurations that perform better on Lassen BB as the values of the two features change. We can see that as transfer size grows, the fraction of runs that perform better on BB decreases. Similarly, the fraction of runs that perform better on BB is higher when the I/O pattern is reading than writing. We can conclude that although overall BB results in better performance than GPFS, it is preferable for smaller transfer sizes and read operations (over write operations).



(a) transfer size



(b) I/O pattern

Fig. 14. Plots showing the percentage of runs that perform better on Lassen BB versus GPFS for different transfer sizes and I/O patterns.

VII. Machine learning modeling results

In this section, we present a comprehensive evaluation of our ML models. First, we report test accuracy for all classifiers we have tried on the IOR benchmarking data. Second, we evaluate our prediction workflow with four real applications and report the accuracy of the models on real applications.

A. *Comparing different ML models*

We use the percentage of correct predictions to evaluate our classifiers. In other words, we compare the predictions with the ground truth and calculate the percentage of predictions that match the ground truth. We refer to this metric as the accuracy. We tried nine different classifiers (Figure 15), and among them, the Decision Tree classifier gives the highest test accuracy of 94.47%. Therefore, in the rest of the experiments, we use the Decision Tree classifier.

The Decision Tree classifier can also output importances for all input features. Figure 16 plots the importance of all features. Since our model predicts whether a combination of I/O features performs better on GPFS or BB, higher feature importance for model prediction implies that the feature can be a key factor that distinguishes GPFS and BB. It may also imply underlying bottlenecks of a certain system. On the other hand, features with very low importance may be insignificant for prediction and accuracy. We observe that transfer size and the number of reads/writes per open are the most important features, after which the importances drop considerably.

To reduce the number of features we need to use in practice, we eliminate the least important features recursively until the accuracy starts to drop significantly. After selecting the set of important features, we retrain the Decision Tree classifier with the smaller feature set.

B. *Model evaluation using production applications*

In the previous sub-section, we showed a test accuracy of 94.47% for unseen IOR runs. However, IOR is a benchmark and we know the I/O features of IOR apriori from

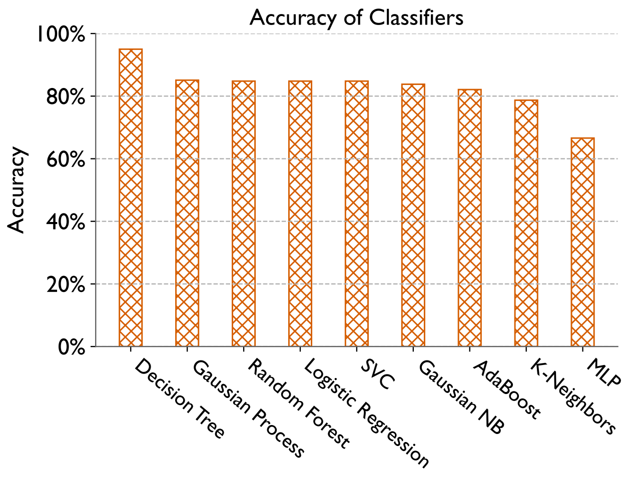


Fig. 15. Accuracy of all tested classifiers. We model whether to put files on BBs based on the I/O features of an application. We train classifiers with IOR data produced in our experiments. The dataset contains 2967 samples and we do a 90-10 split for training and testing. We do 10 random splits and evaluate the model and take the average accuracy to minimize the possible impact from certain data splits. We pick the classifier with the highest accuracy to use in our model.

the configurations of IOR runs. To further evaluate our prediction workflow on real applications, we test it with four unseen applications: LAMMPS, CM1, Paradis, and HACC-IO. LAMMPS is a molecular dynamics application, CM1 is an extreme climate simulator, Paradis is a dislocation dynamics application, and HACC-IO is the I/O proxy application of HACC, which is a cosmology application.

First, we run these applications on both Lassen GPFS and BBs, and trace them with Recorder. Although we only need a trace on one of the I/O sub-systems to make the prediction, we need both to obtain the ground truth values. We run them with configurations from example use cases provided in the documentation or repositories of these applications. Second, for each run, we apply our prediction workflow on either

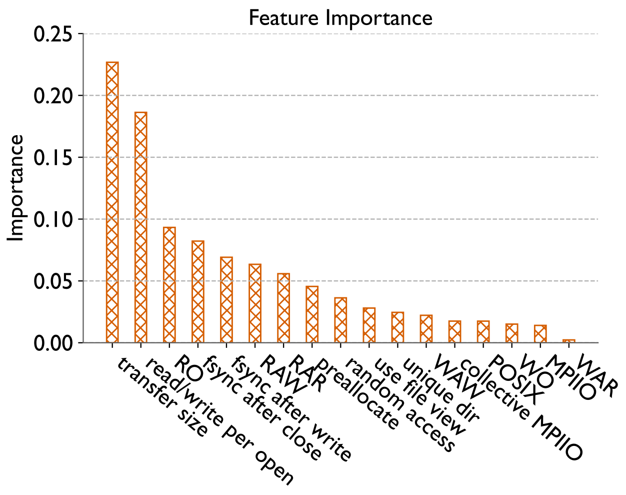


Fig. 16. Feature importances when using a Decision Tree classifier. A higher value signifies that the feature is more important. For example, the feature importance of transfer size is 0.23, which suggests that it is important in deciding whether to use burst buffers.

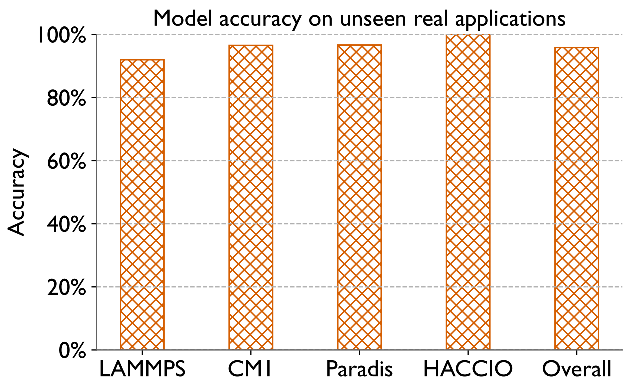


Fig. 17. Accuracy of our prediction workflow for four real applications. We trace these four applications and feed their trace into our workflow to obtain predictions for the better-performing I/O sub-system. For prediction, users only need one trace on any system. But since we need to use ground truth to compute the accuracy, we run them on both systems and compute the bandwidth of each file from their trace. Overall we achieve a good accuracy of 95.86%.

the GPFS trace or BB trace. As discussed in Section V, the model predicts which I/O sub-system to use for each file. Third, we get the ground truth of whether to use BBs for each file by comparing its I/O bandwidth on GPFS and BBs. We use the PrismIO `io_bandwidth` function to obtain the I/O bandwidths for reading/writing each file. Finally, we compare the predicted I/O sub-system with the ground truth to calculate the accuracy metric.

Since our model predicts whether to use BBs for a file based on its I/O features, each individual file produced by the runs should be a sample. However, although these applications write a large number of files, many of them have exactly the same I/O features. For example, one LAMMPS run in our experiment has 386 different files, but most of them are dump files. LAMMPS does multiple dumps and it writes files in the same way for each dump. Such files will have exactly the same I/O features and the same prediction, meaning they cannot be used as different samples. Therefore, to create more samples, we run the applications by varying their input configurations such as problem size, the number of processes, interfaces, etc. This results in multiple configurations per application with different I/O features and thus more samples. We group files with the same I/O features and treat them as a single sample, and use average bandwidth to figure out the ground truth for the group. In total, we have 121 samples in our test data set.

Figure 17 demonstrates the test accuracy for production applications. We achieve 95.86% accuracy on the whole test data set. For the individual applications, we achieved 92%, 96.55%, 96.67%, and 100% accuracy for LAMMPS, CM1, Paradis, and HACC-IO respectively. This shows that training a classifier model using the highly configurable IOR benchmark leads to a significantly high test accuracies for even production applications which were not present in the training data, and may have more complex I/O patterns.

VIII. Conclusion and Future Work

In this paper, we presented PrismIO, a Python-based library that enables programmatic analysis of I/O traces. It provides an API for users to analyze I/O performance efficiently. Using PrismIO, we showed two case studies that demonstrate the use of PrismIO for detailed I/O analyses, and for explaining the causes of performance issues on Lassen GPFS and burst buffers. The case studies demonstrate the potential of PrismIO in making data analytics of I/O traces quick and convenient for the end user.

We also presented an ML-based workflow that predicts whether to put files on burst buffers or not based on the I/O characteristics of an application. We used IOR to create training data and showed that the model achieved 94.47% accuracy on unseen IOR data. We also developed feature extraction functions in the PrismIO API to automate the process of creating training data from new applications. We demonstrated that the ML model achieves 95.86% overall accuracy on four real applications.

In the future, we plan to apply the same methodology to build prediction workflows for other platforms that have burst buffers. Moreover, the current feature extraction APIs are not fast enough, therefore, we plan to improve their performance by designing faster algorithms and creating APIs that extract multiple features at the same time. We believe that this will make our prediction workflow more efficient for large application runs.

Acknowledgment

This work was performed in part under the auspices of the U.S. Department of Energy (DOE) by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-858971). This material is based upon work supported in part by the National Science Foundation under Grant No. 2047120. This research used resources of the Oak Ridge

Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. DOE under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the i/o of hpc applications under congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 1013–1022.

[2] N. T. Hjelm, "libhio: Optimizing io on cray xc systems with datawarp," 2017.

[3] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 807–818.

[4] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka, "Design and modeling of a non-blocking checkpointing system," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–10.

[5] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "Veloc: Towards high performance adaptive asynchronous checkpointing at large scale," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 911–920.

[6] K. Sato, K. Mohror, A. Moody, T. Gamblin, B. R. d. Supinski, N. Maruyama, and S. Matsuoka, "A user-level infiniband-based file system and checkpoint strategy for burst buffers," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 21–30.

[7] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, S. Byna, S. Farrell, D. Gursoy, C. Daley, V. Beckner, B. Van Straalen, D. Trebotich, C. Tull, G. H. Weber, N. J. Wright, K. Antypas, and n. Prabhat, "Accelerating science with the nersc burst buffer early user program," 1 2016. [Online]. Available: https://www.osti.gov/biblio/1393591

[8] A. Ovsyannikov, M. Romanus, B. Van Straalen, G. H. Weber, and D. Trebotich, "Scientific workflows at datawarp-speed: Accelerated data-intensive science using nersc's burst buffer," in *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2016, pp. 1–6.

[9] "Ior." [Online]. Available: https://ior.readthedocs.io/en/latest/index.html

[10] Y.-F. Guo, Q. Li, G.-M. Liu, Y.-S. Cao, and L. Zhang, "A distributed shared parallel io system for hpc," in *Fifth International Conference on Information Technology: New Generations (itng 2008)*, 2008, pp. 229–234.

[11] Using lc's sierra systems. [Online]. Available: hpc.llnl.gov/documentation/tutorials/using-lc-s-sierra-systems

[12] L. Cao, B. W. Settlemyer, and J. Bent, "To share or not to share: Comparing burst buffer architectures," in *Proceedings of the 25th High Performance Computing Symposium*, ser. HPC '17. San Diego, CA, USA: Society for Computer Simulation International, 2017.

[13] B. R. Landsteiner, D. Henseler, D. Petesch, and N. J. Wright, "Architecture and design of cray datawarp," 2016.

[14] L. Pottier, R. F. da Silva, H. Casanova, and E. Deelman, "Modeling the performance of scientific workflow executions on hpc platforms with burst buffers," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 92–103.

[15] Pydarshan documentation — pydarshan 3.3.1.0 documentation. [Online]. Available: https://www.mcs.anl.gov/research/projects/darshan/docs/pydarshan/index.html

[16] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Trans. Storage*, vol. 7, no. 3, oct 2011. [Online]. Available: https://doi.org/10.1145/2027066.2027068

[17] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient parallel i/o tracing and analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 1–8.

[18] G. K. Lockwood, S. Snyder, S. Byna, P. Carns, and N. J. Wright, "Understanding data motion in the modern hpc data center," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, 2019, pp. 74–83.

[19] T. Wang, S. Byna, G. K. Lockwood, S. Snyder, P. Carns, S. Kim, and N. 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)*, 2019, pp. 102–111.

[20] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: Pruning the overgrowth in parallel profiles," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Nov. 2019, lLNL-CONF-772402. [Online]. Available: http://doi.acm.org/10.1145/3295500.3356219

[21] A. Bhatele, R. Dhakal, A. Movsesyan, A. K. Ranjan, and O. Cankur, "Pipit: Scripting the analysis of parallel execution traces," 2023.

[22] D. Quintero, J. Bolinches, J. Chaudhary, W. Davis, S. Duersch, C. H. Fachim, A. Socoliuc, and O. Weiser, "Ibm spectrum scale (formerly gpfs)," 2020.

[23] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12.

[24] J. M. Kunkel, J. Bent, J. Lofstead, and G. S. Markomanolis, "Establishing the io-500 benchmark," 2017.

[25] W. McKinney, *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, 2017.

[26] ——, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 51 – 56.