AIIO: Using Artificial Intelligence for Job-Level and Automatic I/O Performance Bottleneck Diagnosis

Bin Dong
dbin@lbl.gov
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

Jean Luca Bez
jlbez@lbl.gov
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

Suren Byna
byna.1@osu.edu
The Ohio State University
Columbus, OH, USA
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

ABSTRACT

Manually diagnosing the I/O performance bottleneck for a single application (hereinafter referred to as the "job level") is a tedious and error-prone procedure requiring domain scientists to have deep knowledge of complex storage systems. However, existing automatic methods for I/O performance bottleneck diagnosis have one major issue: the granularity of the analysis is at the platform or group level and the diagnosis results cannot be applied to the individual application. To address this issue, we designed and developed a method named "Artificial Intelligence for I/O" (AIIO), which uses AI and its interpretation technology to diagnose I/O performance bottlenecks at the job level automatically. By considering the sparsity of I/O log files, employing multiple AI models for performance prediction, merging diagnosis results across multiple models, and generalizing its performance prediction and diagnosis functions, AIIO can accurately and robustly identify the bottleneck of an even unseen application. Experimental results show that real and unseen applications can use the diagnosis results from AIIO to improve their I/O performance by at most 146×.

CCS CONCEPTS

• **Information systems** → *Information storage systems*; • **Computing methodologies** → **Causal reasoning and diagnostics**.

KEYWORDS

I/O Bottleneck, Job-Level, Diagnosis, Artificial Intelligence, Machine Learning, AI Interpretation, Prediction, Darshan

ACM Reference Format:
Bin Dong, Jean Luca Bez, and Suren Byna. 2023. AIIO: Using Artificial Intelligence for Job-Level and Automatic I/O Performance Bottleneck Diagnosis. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23), June 16–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3588195.3592986

1 INTRODUCTION

When executing data-intensive applications, high performance computing (HPC) faces constant challenges because data access from slow storage devices still takes longer than the computing process [45]. Identifying the causes of slow I/O performance is the very first and most important step to reducing the I/O cost. This paper explores novel methods using Artificial Intelligence (AI) and its interpretation technologies to diagnose I/O performance bottlenecks (also called I/O bottlenecks) for HPC applications.

The I/O bottleneck diagnosis can be performed manually, but this is a tedious and error-prone procedure [10, 12, 36]. Automatic methods for I/O bottleneck diagnosis can be realized by utilizing large-scale I/O trace logs that contain abundant performance factors (generally known as I/O counters, e.g., data access sizes and histograms) [11, 12]. Existing I/O bottleneck diagnosis can happen at the platform, group, or job level. The platform-level methods [5, 48–51] identify the cumulative distribution functions (CDF) of various performance factors and therefore infer "good" and "bad" I/O patterns. The group-level methods [3, 14–16, 25, 26] use clustering methods, e.g., HDBSCAN [26] and KNN [3], to group I/O trace logs and then perform diagnosis for each group. Both platform-level and group-level I/O approaches have the same limitation: the statistical consensus of a group or the whole system can differ from that of its group members [21].

The job-level[1] and automatic bottleneck diagnosis can avoid this limitation by providing a diagnosis for an individual job. But, to realize the method of automatic and job-level I/O bottleneck diagnosis, we need to address the following issues:

- How to develop an accurate performance prediction function (also called a performance function) for a job. This function maps I/O counters to the performance of a job. Each job has unique I/O counters and varying performance, making finding an accurate prediction function difficult.
- How to develop an accurate and robust diagnosis function for a job. Based on the performance function, a diagnosis function should calculate the impact of each performance factor on the performance. The diagnosis function is impacted by its sampling methods [33, 38] and the variance of the performance functions.
- How to integrate the sparsity of the I/O log into the performance and diagnosis functions. I/O counters in one job's log file may not exist in another, resulting in sparse input when developing performance and diagnosis functions. Taking data sparsity into account could improve the robustness of the diagnosis. Robustness refers to whether a diagnosis result

HPDC '23, June 16–23, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0155-9/23/06.
https://doi.org/10.1145/3588195.3592986

[1]A job is defined as a single run of an application. For a SLURM or PBS script file with more than one "srun" or "mpirun", each is viewed as a distinct job.

only contains the I/O counters that an application uses. Take an application that only reads data as an example; if a diagnosis function finds only read-related counters as potential bottlenecks, this function is robust. If a diagnosis function finds write-related counters to be potential bottlenecks, this function is non-robust.

- How to generalize the performance and diagnosis functions for unseen job logs from applications. Existing I/O diagnosis methods [3, 14–16, 25, 26] usually require either rebuilding their models by combining the new data with existing ones or may have a high error rate in classifying the new data into existing groups, both of which limit their applications to processing an unseen I/O log. In processing an unseen I/O log, generalizing performance and diagnosis functions in a job-level method can avoid the high classification error and the time-consuming rebuilding step.

To address these issues of realizing job-level I/O bottleneck diagnosis, this work explores how to use AI prediction-based performance functions combined with new AI interpretation technologies to calculate the impact of various factors on I/O performance. To the best of our knowledge, this is the first work to incorporate AI to automatically diagnose I/O bottlenecks at the job level. Our work identified the following key insights and contributions:

- We designed and developed an approach called "Artificial Intelligence for IO" (AIIO[2]) to diagnose I/O performance bottlenecks at the job level. AIIO includes an AI prediction-based performance function and an AI interpretation-based diagnosis function;
- We used multiple AI prediction models as I/O performance functions for an individual job. We explored five AI models, including XGBoost[13], MLP [37], TabNet [1], LightBGM [27], and CatBoost [19], to reduce the error in predicting the I/O performance of a single job. When compared to a single model, our multiple model-based method can reduce prediction error by up to 3.11×;
- For each AI prediction model tested, we used the AI interpretation method SHAP [33] as the diagnosis function. We also devised two methods to combine the results of multiple models to improve the accuracy of the diagnosis. When compared to a single model, our combining method can reduce the error of the diagnosis function by at most 2.19×;
- We incorporated the sparsity of I/O counters into the performance and diagnosis functions of AIIO to provide a robust diagnosis. We also incorporated early stop training into AIIO to improve its ability to handle unseen job logs;
- We evaluated our method with both synthetic workloads (by simulating different I/O access patterns) and real application workloads from diverse domains. Experimental results demonstrate that AIIO can accurately capture these I/O access patterns, and more importantly, AIIO can help improve application I/O performance by up to 146×.

As almost all scientific contributions have some limitations, we describe ours and the scope for improvement in the following items:

[2]The code repository of AIIO is available at: https://github.com/hpc-io/aiio, which also includes the Darshan log files used in the evaluation section and the link to the web service of AIIO.

- Portability of the proposed approach: Our AI models are based on analyzing a large number of I/O logs on a single system. While the AI model-building approach is portable and able to be used on another system's I/O logs, the models of a system themselves are not portable to another system.
- High-level I/O counters: This work only considers POSIX-IO counters in I/O logs. Our experimental results show that the performance bottleneck identified at the POSIX-IO level can still help applications with HDF5 and MPI-IO interfaces. One may use I/O counters from MPI-IO [22] and HDF5 [8] in AI models; however, we did not attempt that in this effort.
- Automatically fixing I/O issues: We have manually removed diagnosed I/O performance bottlenecks. Further effort is needed to automatically map these bottlenecks to performance tuning techniques.

Further details on the work are presented in the following sections. Section 2 covers the background. Section 3 introduces AIIO and its major components. We report the evaluation results in Section 4. Section 5 concludes the paper and discusses future work.

2 BACKGROUND

2.1 I/O Logs of Applications on HPC Systems

A popular method to capture I/O behaviors is to record the data access operations from applications. These recording results can be organized as a list of I/O counters of operations for each job. Given the diversity of applications, these I/O counters provide a common space to which different applications can be mapped. Various tools (e.g., Darshan [11, 12], STAT [2], mpiP [46], and TAU [42]) and I/O log databases (Charisma [35], HPCT-IO[41], and IOT [39]) have been created. Darshan is a *de facto* standard at an extreme scale. It is used by the National Energy Research Scientific Computing Center (NERSC), Argonne Leadership Computing Facility (ALCF), and other supercomputing centers to collect I/O profiling logs of applications. In this work, we used the Darshan I/O logs from NERSC to demonstrate the efficiency of our methods in diagnosing I/O performance bottlenecks. Details of the Darshan I/O logs used in this work will be presented in Section 3.1.

2.2 Related Work

The I/O bottleneck diagnosis can happen at different levels, including platform, group, and job. We briefly discuss each approach and highlight its benefits and limitations.

Platform-level I/O bottleneck diagnosis: A platform-level I/O bottleneck diagnosis takes the whole storage (or all available applications' logs) as a study target and finds the I/O access patterns for the whole system. Recently, cumulative distribution functions (CDF) for data access size and other parameters were studied on the Cori and Summit supercomputers [5]. Machine learning models can also be used to find metadata load and load skew, which can impact the writing performance of a job [50, 51]. The number of processes strongly correlates with job bandwidth at the platform level [48, 49]. The performance variability of the whole system has also been studied [25, 30]. These platform-level findings provide essential guidelines for system administrators to optimize the whole system, but these methods ignore the diversity of individual applications.

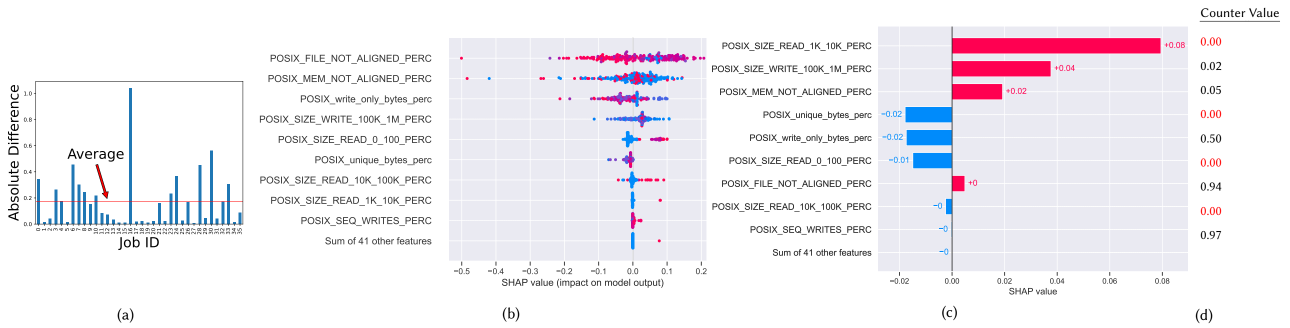


Figure 1: Comparison of prediction and feature importance for a cluster named *Gamma* and its member with Gauge and its open-sourced data [16]. The cluster is created with HDBSCAN. (a) Prediction error for the whole cluster (denoted as "average") and its members; (b) Counter importance for the whole cluster; (c) Counter importance for the 204th member in the cluster; (d) Counter values for the 204th member in the cluster.

Group-level I/O bottleneck diagnosis: The group-level I/O bottleneck diagnosis has arisen recently. The naive way is to manually group jobs in the same domain for analysis, such as in the work by Paul et al. [36], which focuses on studying I/O access patterns for machine learning applications. A more generic way is to use clustering algorithms, such as HDBSCAN [16, 25, 26], KNN [3], and agglomerative hierarchical clustering [14, 15] to group jobs for analysis. Based on Gauge and its dataset [16, 25, 26], we demonstrate four major issues with group-level bottleneck analysis below:

- The I/O performance prediction model selected by Gauge is based on the statistical average of a group. As shown by Fig. 1 (a), the prediction error of the whole group is significantly different from that of an individual job. Selecting a single prediction model for the whole group could cause large errors for an individual job.
- The impact of I/O counters (i.e., the likelihood of bottlenecks for performance) for the whole group is different from that for a single job. As shown by Fig. 1 (b) and Fig. 1 (c), the most impacted factor for the group is `POSIX_FILE_NOT_ALLIGENED_PERC` but the most impacted factor for the job is `POSIX_SIZE_READ_1K_10K_PERF`.
- The I/O counters with zero values (shown in Fig. 1 (d)) are assigned impact values by the Gauge method, which is referred to as non-robust in this work. These zero I/O counters can lead to false diagnoses. Fig. 1 (d) shows that the `POSIX_unique_bytes_perc` is assigned a -0.02 impact value, but its actual value is zero, which means there is no unique read or write in the application.
- The I/O bottleneck analysis by Gauge only focuses on its training data. It does not provide a method to work on an unseen I/O log from an application. Processing an unseen I/O log may require either classifying an unseen job into an existing cluster or rerunning clustering algorithms by combining the new one with existing ones. The former may face high error rates, and the latter may be time- and resource-consuming.

Job-level I/O bottleneck diagnosis: As illustrated by Fig. 2, manually [10, 12] identifying I/O performance bottlenecks via interactive exploration [6] works for a single job. But, it needs the involvement of a human at the diagnosis step. Also, the diagnosis may need to be repeated $N (N \geq 1)$ times for multiple counters. For M applications, the interactive exploration step may be repeated by $M (M \geq 1)$ times. The human involvement, N repetitions for a single application and M repetitions for multiple applications in interactive exploration, can be avoided using the AI-based method developed in this work. Diagnosis can be fully automated. Bez et at. [4] and ongoing work DigIO [52] have the initial target of providing a semi-automatic way for I/O bottleneck analysis for an individual application. But, those are not fully automatic because they may depend on static rules that must be defined manually. To the best of our knowledge, generic methods to perform fully automatic and job-level I/O performance bottleneck diagnosis are still missing. Hence, this work fills this gap by introducing AI and its AI interpretation technologies for I/O performance bottleneck diagnosis in a fully automatic way.

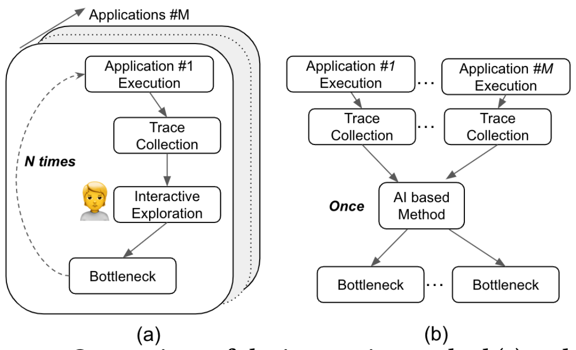


Figure 2: Comparison of the interactive method (a) and the AI-based method (b) for I/O bottleneck analysis.

3 ARTIFICIAL INTELLIGENCE FOR I/O (AIIO)

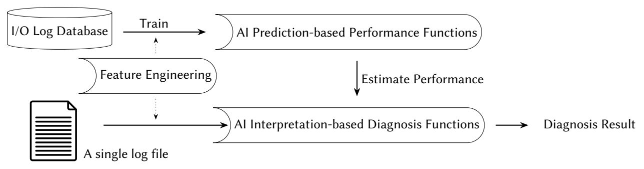


Figure 3: A high-level overview of our AIIO approach to applying Artificial Intelligence (AI) and its interpretation technologies to diagnose I/O performance bottlenecks for a job.

To automatically diagnose the I/O performance bottleneck at the job level, we show our AI for IO (or shorted for AIIO) method, as presented in Fig 3. AIIO contains multiple AI prediction-based performance functions trained from a historical I/O log database. These performance functions map the values of I/O counters to the performance of a job. Another major component of AIIO is the AI interpretation-based diagnosis functions. Based on the prediction capability of the performance functions, the diagnosis functions provide the diagnosis result for a single log file. The diagnosis result can be merged with multiple AI-based performance functions. Following diagnosis results, users can change their code to get better I/O performance if their codes have I/O issues. In the following subsections, we introduce the I/O log database and then each major component of the AIIO. At the end of this section, we introduce the AIIO Web Service, which puts AIIO into real-world practice.

Table 1: Total 825 GB Darshan logs with 6,647,219 jobs are used in this study.

Year	Size	# of Jobs
2019	182GB	3,013,293
2020	157GB	1,554,827
2021	387GB	2,854,583
2022	102GB	963,035
SUM	**825GB**	**6,647,219**

3.1 I/O Log Database and Feature Engineering

Jobs running at major HPC centers by default produce log files containing their I/O behaviors, such as how much data to write or read. After accumulating over time, these I/O logs have become a large I/O log database. In this study, we use one of the most recent and largest I/O log databases as the foundation to diagnose application I/O performance bottlenecks. Specifically, we use the Darshan I/O log database from the Cori supercomputer at NERSC. This Darshan I/O log database comprises logs collected from 2019 through the first four months of 2022 (40 months). Details of the logs for each year are presented in Table 1.

A Darshan log file collects the I/O operations of an application issued using different APIs, such as MPI-IO, HDF5, and STDIO. We choose to use POSIX I/O counters because it is a popular I/O method that supports MPI-IO and HDF5 too. A Darshan log has 90 I/O counters[3] for the POSIX-IO interface. Among them, 25 are time-related counters and 65 are not. These time-related counters capture the cost of the I/O operation itself, the interference of other applications, and the system noise. We use time-related counters to generate the tag (i.e., performance) for the job. The tag is used to build a performance prediction function, as presented in the following paragraph and Section 3.2. Then, we drop these time-related counters because they are the "effects" but not the "causes" of I/O performance. We drop non-Lustre file-based data accesses, such as the GPFS file system, which usually supports users' `HOME` directories or archiving systems with small I/O workloads on the Cori supercomputer. The Lustre file system is a high-performance production storage system used on Cori. We also drop nearly empty I/O counters (such as `POSIX_DUPS` and `POSIX_RENAME_SOURCES`) across all six million jobs because they don't contain any useful information. In summary, we use 45 Darshan I/O counters in our work, which are listed in Table 4 of the Appendix.

[3]Full counter list can be found at https://www.mcs.anl.gov/research/projects/darshan/docs/darshan-util.html

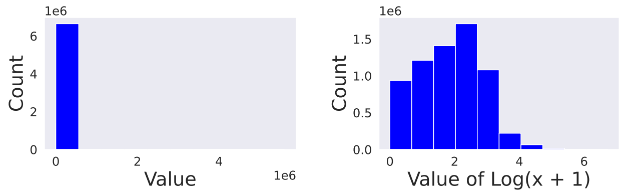


Figure 4: Performance of all jobs before (left) and after (right) the $log(x+1)$ data transformation.

Finding the ground truth (tag) for each job's performance: Darshan has time-related counters that record the time of most I/O operations. Darshan uses these time-related counters to estimate the performance of each job with high accuracy [16, 24, 26]. Our work uses the estimated performance as the tag during model training and accuracy measurement. The idea behind the estimated performance for a job is:

$$performance = \frac{total\ bytes\ of\ transferred\ data}{time\ of\ the\ slowest\ process} \tag{1}$$

The *total bytes of transferred data* are the aggregated data sizes of all processes involved in I/O and the *time of the slowest process* is the time required by the last process to complete its I/O. Unless specifically noted, our work sticks to the performance estimated by Darshan with the megabyte per second (MiB/s) unit.

Feature engineering: The feature engineering method presented below is applied to process a large-scale I/O log database and a single I/O job log from users. As stated in Section 2, each application has unique data access behaviors. The I/O counters in one job may not exist in another. For example, an application that only reads data will not have counters for applications that write data. Our feature engineering method handles all counters listed in Table 4 for all job logs, but fills the missing value for I/O counters from an application with 0. Since the goal of this work is to find the I/O performance bottleneck (i.e., I/O counters with a negative impact on the performance) and provide hints for users to tune the I/O performance of their applications, we tried to keep I/O counters as original as possible without merging them, as was done in previous work [16, 25, 26]. On all I/O counters, we performed a $log(X+1)$ transform:

$$x_{new} = log_{10}(x_{orignal} + 1) \tag{2}$$

$x_{orignal}$ is the value of the original counters, and x_{new} is the transformed one. Two-fold benefits come with this transformation: 1) The log_{10}-based transformation scales wide-scope values into small ranges. The $log_{10}(x+1)$, as illustrated in Fig. 4, reduces the performance value from the range (1, 6309573) to (0.3, 6.8). All features are narrowed into similar ranges, which improves the accuracy of the performance prediction function [21]. 2) The $log_{10}(x+1)$ transforms the original 0 values of I/O counters to 0, representing

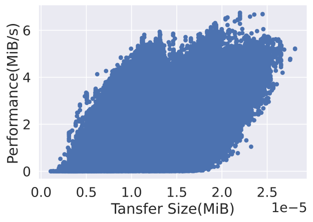


Figure 5: A scatter plot of the performance and total data transfer size of all Darshan logs.

the missing values (sparsity) in the original Darshan log. Considering the sparsity of a Darshan I/O log improves the accuracy and robustness of performance diagnosis at the job level.

Sparsity in our Darshan log database: We use the following formula to measure the average sparsity of jobs in our database: $sparsity = \frac{1}{R}\sum_{r}^{R}\frac{count\ of\ zeros\ of\ job\ r}{total\ counters\ of\ job\ r}$, where the *total counters of job r* are 45. The average *sparsity* of all jobs in our Darshan log database is 0.2379, which indicates that, on average, each job has around 10 I/O counters with zero values. As a result, consideration of the sparsity of a Darshan log is required.

Implications for performance and diagnosis functions: Visually inspecting the performance bottleneck is difficult. Taking data transfer size (Fig. 5) as an example, the relationship between performance and data transfer size is neither linear nor nonlinear. This is only in 2D space. Given the 45 dimensions (i.e., features) in our Darshan logs, more sophisticated performance and diagnosis functions need to be developed.

3.2 AI Prediction-Based Performance Function

The performance function captures the relationship between the I/O counters and the I/O performance of a job. By changing the inputs, i.e., the counters of I/O, the performance function also changes its output, i.e., predicted performance. This can be used to replace the simulation of expensive runs during the manual performance bottleneck diagnosis. As discussed in Section 3.3, we used the diagnosis function to measure the impact of these changes. This subsection focuses on how to find the performance function. The error between real performance and predicted performance should be as small as possible for the performance function.

Modeling the performance of any I/O system is a difficult task because of the complexity of these systems. Traditional analytical modeling methods [17, 28, 40] have been developed. However, these analytical models are limited because they only consider a few factors, such as access size and the count of I/O servers in a file system. Instead, we explored AI-based performance functions to capture the relationship between the value of Darshan I/O counters and the performance of a job.

To reduce the error of the performance function for a single job, AIIO uses multiple AI models as the performance function. Based on the latest research [7] on comparing different AI models for a regression problem, we selected three models with the highest accuracy for this work. These three models include LightGBM, CatBoost, and XGBoost. All the models are based on a tree structure. AI models have evolved quickly in recent years [44]. Therefore, we also chose MLP and TabNet. MLP represents a neural network-based method. TabNet uses the latest idea of transformer networks in modeling data. In summary, we investigated five AI models as performance functions and expanded our methods to include more AI models. Different models capture different features or patterns from a job's I/O counters, reducing errors. The explanations for how these AI models work and why they have different performances on regression is detailed in the references [7, 44]. The five AI models we investigated are briefly described in the following text:

- **MLP**: We developed a MLP (multilayer perceptron) model [21] as the performance function. The MLP model uses a fully-connected neural network. Its architecture is presented in Table 5 in the Appendix. We used the "Relu" function as the activation function and added the Batch Normalization (BN) and Dropout layers to the network. These are typical optimization methods used to improve the model's accuracy.
- **XGBoost**: XGBoost [13] is based on gradient boosting, a well-proven method for building machine learning models. In XGBoost, gradient boosting makes predictions from multiple decision trees. XGBoost has high accuracy in the regression of tabular data. Moreover, XGBoost supports sparse data input, as required by our Darshan I/O logs. XGBoost has a well-optimized cache access pattern and scale well on large-scale training data.
- **LightGBM**: LightGBM [27] is also a machine learning model that uses the gradient boosting decision tree. Unlike XGBoost, which uses a level-wise growing tree, LightGBM uses a leaf-wise growing tree. LightGBM has a one-sided sampling method and exclusive feature bundling methods to speed up its training. LightGBM has high accuracy and a fast convergence speed when modeling tabular data like our Darshan logs.
- **CatBoost**: CatBoost [19] uses a symmetric (balanced) decision tree for weak learners in gradient boosting. The balanced tree structure in CatBoost has efficient execution on modern computers. CatBoost also uses ordered boosting, a permutation-driven alternative to the classic algorithm for solving the prediction shift problem. With ordered boosting, CatBoost uses different datasets to train and calculate gradients and avoids prediction shifts to produce more general models.
- **TabNet**: TabNet [1] is a deep neural network model for tabular datasets like our Darshan I/O database. Specifically, TabNet uses sequential attention to choose different features for prediction. TabNet also has batch normalization, allowing it to process the original dataset.

How to choose the performance metrics in model training. We chose the root-mean-square deviation (RMSE) to measure the loss of an AI model during training:

$$RMSE = \sqrt{(\frac{1}{n})\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}. \tag{3}$$

The y_i is the transferred performance with Eq. 2 and $\hat{y}_i$ is the predicted performance.

How to train the model and avoid overfitting. During the training of each model, we shuffled the training data and split it into two parts: one half for training and the other for evaluations. We also used early stopping rounds (= 10) across all models to avoid overfitting, improving our models' generalization to handle

Table 2: RMSE of the prediction function and diagnosis function. The RMSE for the prediction function is calculated with Eq. 3. The RMSE for the diagnosis function is calculated using Eq. 5. The Closest Method and Average Method are discussed in Section 3.3.

	Prediction Func.	Diagnosis Func.
CatBoost	0.2686	0.2637
LightGBM	0.2632	0.2599
XGBoost	0.5634	0.2604
MLP	0.5416	0.4611
TabNet	0.3078	0.3077
Closest Method	**0.1860**	**0.2130**
Average Method	**0.2405**	**0.2471**

unseen data. Since each model has many parameters to tune, we kept all their default values. The convergence plot for the loss of these models is presented in Fig. 16 in the Appendix.

How to use multiple AI models for an individual job. Our goal with this work is to provide a job-level diagnosis for I/O performance bottlenecks. We assumed that the Darshan log file, on which these performance models work, contains unseen data. As a result, the predicted error for some of these models could be higher than expected. To reduce error, we fed each model the same Darshan log file to perform independent predictions. Then, as presented in Section 3.3, we ran the diagnosis models on each of them independently. The diagnosis results from these multiple models were merged to reduce error. The AI-based performance functions are trained with all Darshan logs, which contain information about all these job logs. In the following section, we present how to focus on the diagnosis of a single job with locality in mind.

How to handle sparsity in AI models. AI models naturally accept zero-based sparse input (e.g., SciPy sparse matrices), but some software (e.g., pytorch-tabnet [4] for TabNet) built for these AI models only accepts dense input. For these I/O models, we trained the models with dense input and later used the diagnosis function (explained in the following section) to handle the sparsity.

Experimental results of different models on performance. We evaluated multiple AI models as the performance function on half of our I/O log database. The RMSE results are presented in Table 2. We provide two methods, namely Closest Method and Average Method, to merge the results of different models. Merging results from different models helps reduce the error in predicting the performance of a single job. The RMSE for the Closest Method is calculated by replacing the $\hat{y}_i$ of a single job with the closest predicted performance among all models. The RMSE for the Average Method is calculated by replacing the $\hat{y}_i$ of a single job with weighted predicted performance among all models. The idea for the Closest Method and the Average Method is presented in Eq. 6 and Eq. 7, respectively. For the performance function, the RMSE is reduced at most by 3.11×. In other words, multiple models reduce the RMSE of the performance prediction for a single job. In the following section, we present our methods to use these performance prediction functions to diagnose the I/O performance bottleneck.

[4]https://pypi.org/project/pytorch-tabnet/

3.3 AI Interpretation-Based Diagnosis Function

The performance function presented in the previous section connects the I/O counters with the I/O performance of an application. By utilizing AI interpretation technologies, the diagnosis function presented in this section calculates the contribution (C) of these I/O counters to the performance. The C is either a negative or positive value. The negative C means that the corresponding counter decreases the I/O performance; the positive C means the corresponding I/O counter increases the performance. The I/O counters with negative C are bottlenecks of I/O performance. The larger the absolute value of C of an I/O counter, the larger the impact the I/O counter has on the I/O performance.

How to find the diagnosis functions. Various methods can be used for the diagnosis. Traditional methods, such as linear regression and the partial dependence plot (PDP) [20], may have atypical results for tabular data like our Darshan log file. Recent advances in AI have brought more accurate interpretation methods, such as SHAP [33] and LIME [38], which are used in this work. The SHAP provides a job-level diagnosis by perturbing the data around the input log and then fitting a regression model with the perturbation results. The SHAP is based on game theory to allocate the performance contribution for all I/O counters. SHAP unifies other methods, such as LIME, PDP, and DeepLIFT [43]. Although our AIIO supports different interpretation methods, we mostly focused on discussing SHAP. The SHAP value, i.e., the contribution C in our work, for an IO counter j can be expressed by

$$C_j = \sum_{s \in I-j} \frac{|S|!(|I| - |S| - 1)!}{|I|!}(\hat{y}_{S+j} - \hat{y}_{S-j}). \tag{4}$$

The I is the set of I/O counters, S is a subset of I without I/O counter j, $\hat{y}_{S+j}$ is the predicted performance with I/O counter subset S with I/O counter j and $\hat{y}_{S-j}$ is the predicted performance without I/O counter subset with I/O counter j. Our work uses the SHAP package[5] to compute SHAP values for each I/O counter. The SHAP uses local accuracy [32] to measure how well the SHAP works. Based on the local accuracy and RMSE for regression, we defined RMSE for the diagnosis functions to measure their precision:

$$RMSE\ for\ SHAP = \sqrt{(\frac{1}{n})\sum_{i=1}^{n}(E_i + \sum_{j=1}^{|I|} C_j - y_i)^2}, \tag{5}$$

where E_i is the expected performance of the job i from SHAP. The sum of $\sum_{j=1}^{|S|} C_j$ and E_i should provide the prediction for the y_i that is the real performance of a job. By calculating the RMSE for SHAP, we know how accurately SHAP understands the impact of I/O counters on performance.

The locality awareness in SHAP allows AIIO to use global performance functions for job-level diagnosis. In Section 3.2, we reported our performance functions, which are trained with all I/O logs. Therefore, we call them globally-trained performance functions. As shown here, SHAP uses locality-aware sampling methods [33, 38] to create a set of synthetic samples that are "neighbors" to a job. The neighborhood is measured in this context using cosine distance or Euclidean distance. The impact of each I/O counter can be calculated based on the set of synthetic samples.

[5]https://github.com/slundberg/shap

Sparse Darshan log input is required for diagnosis functions. The SHAP package takes a performance function as input and runs perturbations on an input Darshan log. The sparse data in the Darshan log needs specific consideration here. We used the SHAP Kernel Explainer from the SHAP package to calculate the SHAP values to be model agnostic. The SHAP Kernel Explainer accepts sparse data as input as well as a background filter against which to sample. Hence, we initialized the background filter of the SHAP Kernel Explainer as zero, then provided I/O counters (sparse list) from a Darshan log as the input of the SHAP Kernel Explainer. The SHAP Kernel Explainer calculates the SHAP value for the I/O counters with values that are not zero. For the I/O counters with zero values, SHAP Kernel Explainer skips the sampling data around them and assigns zero contribution to them.

Merge diagnosis results from multiple performance functions. As stated in Section 3.2, we used multiple AI models as the performance function to connect the I/O counters with their performance. Using multiple models reduces the RMSE of the prediction of a single job. Here, we used these performance models to independently build different diagnosis functions. For each AI-based performance model, we built SHAP and LIME diagnosis models separately. Now the question becomes, "How do we utilize all of these different interpretation results from different diagnosis functions?" We consider the following two methods:

- **Closest Method: Pick the diagnosis results from the most accurate model.** This method picks the diagnosis function that is built from the performance function with a minimum difference from the real performance (from Eq. 1). This can be presented as:

$$m^* = \underset{m}{\operatorname{argmin}}(\hat{y}_m^x - y_m^x), \tag{6}$$

where the $\hat{y}_m^x$ is the predicted value from model m and y_m^x is the estimated value by Darshan for x. The m^* is the model selected with a minimum difference between the predicted and estimated values. Hence, we choose the diagnosis function built from m^*.

- **Average Method: Average diagnosis results from all models with sophisticated weights.** This method merges the results across diagnosis functions. Specifically, we obtained diagnosis results (for the same Darshan log) independently from different diagnosis functions, then we merged their diagnostic results. We did not merge results for diagnosis functions built from different interpretation technologies, such as SHAP and LIME, because their results are on different scales and are difficult to merge. For the diagnosis functions built with the same interpretation technology, we used Eq. 7 to merge results:

$$C^j = \sum_m^M w_m C_m^j \tag{7}$$

where C_m^j is the impact of I/O counter j. The effect on performance can be either positive or negative. The negative impact means the current I/O counters degrade the whole performance, and the positive impact means the current I/O counters improve the performance. Specifically, we used Eq. 8 to calculate the ratio (r_m) of a model among all models and then normalized it to get the final weight w_m. The calculation of r_m is based on the difference between the predicted performance $\hat{y}_m$ and the estimated performance y_m from Darshan. The more accurately that the model predicts the job's performance, the larger weight this performance model becomes.

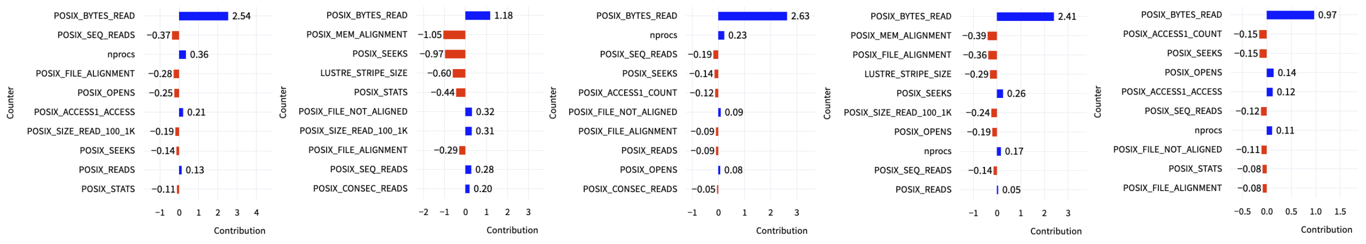
$$w_m = \frac{r_m}{\sum_{m=1}^M r_m} \quad \text{where} \quad r_m = \frac{\sum_{m=1}^M (\hat{y}_m - y_m)}{\hat{y}_m - y_m} \tag{8}$$

Comparison of the two methods mentioned above. Using the RMSE defined in Eq. 5 for SHAP, we calculated the RMSE for individual models and used the results from our two methods to merge them, as reported in Table 2. Both of our methods outperform individual models. The Closest Method and the Average Method have very similar results. Based on our experimental studies, we intended to use the Average Method. As previously stated, SHAP samples data points surrounding the input data with locality awareness. These data points are used to build a linear regression function and obtain the impact of each factor. Different AI-based performance functions have different accuracy for both the input and sampled data points. One possibility is that a model has a large error on the input data but high accuracy on the sampled data. This possibility can be inversely proportional to the difference between its performance and real performance: the larger the difference, the lower the possibility of an accurate prediction. Hence, merging the diagnosis results with the weight, based on performance difference, can take advantage of different models with different ways to model the data, thus increasing the possibility of finding I/O bottlenecks. An example is shown in Fig. 6 and Fig. 8a (in the following section). Fig. 6 contains the results of five models and Fig. 8a contains the merged results from the average method. The Closest Method will select the CatBoost diagnosis results with the `POSIX_SEQ_READS` as the negative contribution. The most negative impact factor for the Average Method is `POSIX_SEEKS`. As demonstrated in the following section, `POSIX_SEEKS` is proven to be one negative factor for the performance. This diagnosis result, on the other hand, is for a sequential reading where the `POSIX_SEQ_READS` should have less impact on the performance. `POSIX_SEEKS` is ranked as the fifth negative factor by CatBoost. By considering the contributions of other models, `POSIX_SEEKS` becomes the first negative factor.

Robustness in our diagnosis methods. As presented above, we integrated the sparsity of the Darshan I/O log into both performance and diagnosis functions. As a result, the sampling method in SHAP and LIME voids the sampling and impact calculation for I/O counters with zeros. Our experimental results in Fig. 6 and all the following results show that our method only assigns impact values to I/O counters with non-zero values. By contrast to the results from Gauge in Fig. 1, which assigns impact to I/O counters with zeros, our method for job-level and automatic I/O bottleneck diagnosis is robust.

3.4 AIIO in Action

So far, we have presented the details of AIIO, which diagnoses the I/O bottleneck of a job by analyzing its I/O log file. Here, we describe the different ways to put our approach into practice. The first is to extend the tool like darshan-util, which generates the report for Darshan I/O logs. The second is to build a web service for users. The first method faces the difficulty of model management, as darshan-util may be installed by different users at different places. In this case, we decided to create an AIIO web service that allows



(a) CatBoost 395MiB/s (b) TabNet 370MiB/s (c) LightGBM 199MiB/s (d) MLP 125MiB/s (e) XGBoost 1591MiB/s

Figure 6: Diagnosis results of five models in AIIO. The real performance of the job is 412MiB/s. The configurations of the workload and weighted diagnosis results of these five models are presented in Fig. 8a in the following section.

users to obtain diagnosis results while also allowing us to manage its AI models easily. In the AIIO web service, all AI models are pre-trained and saved in files. Since AI models evolve quickly, our AIIO web service may accept new models from users. The web server loads these pre-trained models to provide diagnosis results, as shown in Fig. 17 in the Appendix.

4 EVALUATION

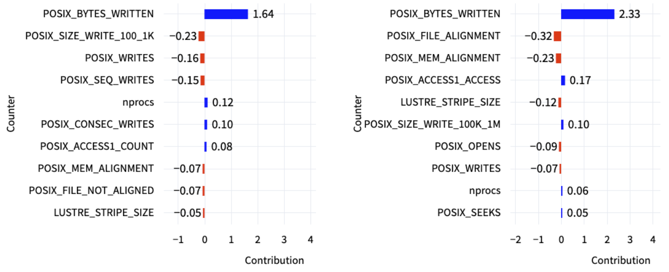
Our experiments aim to evaluate if AIIO can accurately identify the I/O bottlenecks of HPC applications. We manually tuned the application and measured its performance to see if the diagnosis result was accurate. Our experiments were performed on the Cori supercomputer[6] at NERSC. Cori is a Cray XC40 system with at least 2,388 Intel Xeon Haswell processor nodes and a Luster file system. Our tests only used the Luster file system with its default settings, e.g., 1 OST and 1 MB stripe size.

Our analysis mostly focuses on the most negative impact factors from AIIO, and we tuned the applications to address these factors. In reality, this is an iterative process with multiple rounds. We combine different experiments to show how this iterative process works. We also used the performance estimated by Darshan as the metric. We tested AIIO with both synthetic I/O workloads and real applications. Details of these experiments and their results are presented in the following parts of this section. Our AIIO Web Service produced all the plot results.

[6]Cori: https://docs.nersc.gov/systems/cori/

Table 3: Summary of the IOR Configuration

	IOR Configuration
Fig. 7 (a)	`ior -w -t 1k -b 1m -Y`
Fig. 7 (b)	`ior -w -k 1m -b 1m -Y`
Fig. 8 (a)	`ior -r -t 1k -b 1m`
Fig. 8 (b)	`ior -r -t 1k -b 1m` with a seek for the first read
Fig. 9	`ior -w -t 1k -b 1k -s 1024 -Y`
Fig. 10	`ior -w -t 1k -b 1k -s 1024 -Y`
Fig. 11	`ior -w -t 1k -b 1m -z -Y`
Fig. 12	`ior -a POSIX -r -t 1k -b 1m -z`



(a) Performance: 1.55 MiB/s Transfer size: 1K **(b) Performance: 162.01 MiB/s Transfer size: 1M**

Figure 7: Results of IOR that writes sequentially.

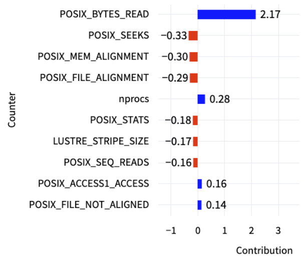
4.1 Synthetic I/O workloads from IOR

We executed the latest IOR (version 3.3.0)[7] to simulate six low-performing I/O access patterns identified by previous research [12, 17, 34, 47]. These access patterns included sequential reading and writing with small request sizes, stride reading and writing, and reading and writing with random offsets. Then, we saw whether AIIO captures these I/O access patterns and provides hints to improve their performance. All tests in this subsection were run with 256 MPI processes using the POSIX API. The configurations of IOR are summarized in Table. 3.

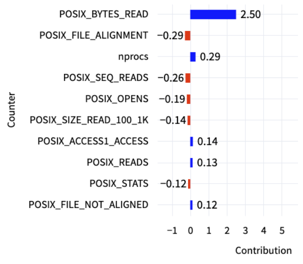
*4.1.1 **Pattern 1: Sequential Writing with Small I/O Requests**.*

This test used the IOR with small write sizes. Our test used IOR's `-Y` option to perform `fsync` after each POSIX write, which can avoid the impact of the cache. The experimental results and configurations are presented in Fig. 7. First, we run the IOR with a write size of one kilobyte (i.e., `-t 1K`), which results in a performance of 1.55 MiB/s. According to Fig. 7a, AIIO reports that a large number of small writes (i.e., `POSIX_SIZE_WRITE_100_1K` and `POSIX_WRITES`) have the greatest negative impact on the performance. Then, we increase the size of each write in IOR to one megabyte (i.e., `-t 1M`). The writing performance increases to 162.01MiB/s, which is 104× faster. According to Fig. 7b, this large size of each write (i.e., `POSIX_SIZE_WRITE_100k_1M`) has a positive impact here. Clearly, AIIO can capture this small I/O request pattern in the sequential writing workload from IOR.

[7]https://github.com/hpc/ior



(a) Performance: 412.70 MiB/s



(b) Performance: 644.67 MiB/s

Figure 8: Results for IOR that reads sequentially. The (a) uses the original IOR code that calls seek for read every time, but the (b) only calls seek once for the first read.

4.1.2 *Pattern 2: Sequential Reading with Small Requests.*

This experiment tests the sequential reading of IOR with small I/O request sizes. The experimental configurations and results are reported in Fig. 8. From Fig. 8a, we can see that the IOR only has 412 MiB/s of performance because there are lots of `POSIX_SEEKS`. After studying the IOR benchmark, we found that IOR enforces seek for each read operation, even though it is a sequential read. Hence, we changed IOR's code by only seeking once each time for the first read. With this change, we re-ran the same code and found its performance had increased to 644.47 MiB/s (in Fig. 8b). Also, based on the new diagnosis results of AIIO on this run, the `POSIX_SEEKS` is not a negative impact factor. Hence, AIIO can find the cause of this pattern of sequential reading with small requests.

4.1.3 *Pattern 3: Noncontiguous Writing with a Fixed Stride.*

By setting the transfer size to be equal to the block size in IOR and the segment number (-s) to 1024, we simulate noncontiguous writing with a fixed stride size. Experimental results are presented in Fig 9. This configuration attains 1.46 MiB/s. With the exception of the `POSIX_SIZE_WRITE_100_1K`, the AIIO reports two counters (i.e, `POSIX_STRIDE1_COUNT` and `POSIX_STRIDE1_COUNT`) as the most negative impact factors for the performance. As stated in Section 4.1.1, the `POSIX_SIZE_WRITE_100_1K` is one major negative factor for performance. Now the stride pattern becomes the second negative factor. Our AIIO accurately captures this pattern. To improve the performance, we converted the stride pattern into sequential writing (like one for 8a) and had a large write size (like one for 8b). The latter can have a performance of 162 MiB/s, as stated in Section 4.1.1.

4.1.4 *Pattern 4: Noncontiguous Reading with a Fixed Stride.*

Using the same idea as the previous experiment, we tested the pattern of noncontiguous reading with a fixed stride size. Experimental results are presented in Fig. 10. The diagnosis results from the AIIO say that `POSIX_SEEKS` and `POSIX_FILE_ALIGMENT` are the two factors with the largest negative impact[8]. For noncontiguous reading, many `POSIX_SEEKS` are required to jump to the next position to read. Hence, `POSIX_SEEKS` dominates the performance. Since our request size is 1K and `POSIX_FILE_ALIGMENT` is 1M, this makes `POSIX_FILE_ALIGMENT` the top negative factor. To optimize this pattern's performance (65.33MiB/s), we easily convert the noncontiguous reading into a contiguous one, as shown in Fig. 8a, which has a performance of 412.70MiB/s. The contiguous access can be further diagnosed by AIIO to improve its performance to 644.67MiB/S, as shown in Fig. 8a.

[8]We ignore the `POSIX_MEM_ALIGMENT` since we focus on the I/O operation.

4.1.5 *Pattern 5: Writing with a Random Offset.*

Experimental results of writing at random offset are presented in Fig. 11. Except for small writes and their amounts (i.e., `POSIX_SIZE_WRITE_100_1K` and `POSIX_WRITES`), the two I/O counters (i.e., `POSXI_FILE_NOT_ALIGNED` and I/O counter `POSIX_STRIDE1_COUNT`) are the top negative factors. These two factors are caused by the I/O writing pattern with random offset. Clearly, AIIO can capture this pattern. The performance of this pattern is 1.43MiB/s. The performance of this pattern can be improved by converting it to a contiguous pattern (shown in Fig. 7a) with 1.55MiB/s and further by using a large write size (shown in Fig. 7b) with 162.01MiB/s.

4.1.6 *Pattern 6: Reading with Random Offset.*

The experimental results of IOR, which reads with a random offset, are reported in Fig. 12. The `POSIX_STRIDE3_STRIDE` and `POSIX_STRIDE4_STRIDE` are all selected as the negative factors, produced by this random reading pattern. Other I/O counters, e.g. `POSIX_SIZE_READ_100_1K` and `POSIX_SEQ_READS`, are identified bottlenecks from the previous tests. The performance for this pattern is 94.52MiB/s. The performance can be improved by converting this random reading into a contiguous reading (as shown in Fig. 8a) to 412.70MiB/s. Then, the performance can be improved again to 644.67MiB/s by using a large read size (as shown in Fig. 8b)

In summary, we tested the AIIO with six different I/O patterns usually viewed as patterns with bad performance. Experimental results manifest that AIIO accurately identifies these patterns and also helps to improve their performance. As shown by the experiments with noncontiguous access (§4.1.3, §4.1.4, §4.1.5, §4.1.5), contiguous but small access size (Fig. 7a and Fig. 8a), and contiguous and large access size (Fig. 7b and Fig. 8b)", AIIO can identify the cause of the I/O performance bottleneck as the optimization moves on.

4.2 Real Applications

4.2.1 *E2E.*

E2E [31] contains the I/O kernels of the Chimera Supernova code and Pixie3D. This test used `write_3d_nc4.c` of the E2E to evaluate AIIO. The function accepts six parameters, where (np_x, np_y, np_z) defines the points per block and (nd_x, nd_y, nd_z) defines the number of blocks. The ($np_x \times nd_x$, $np_y \times nd_y$, $np_z \times nd_z$) is the size of the data. Following the recommendations of Bez et al. [6], we disabled filling the default value before the run. Furthermore, we set (np_x, np_y, np_z) = (32, 32, 16) and (nd_x, nd_y, nd_z) = (32, 32, 32), which write 3D data of size (1024, 1024, 512). Our test used 64 processes, where each process writes a cubic (subset) based on its rank in the MPI communicator.

Experimental results of E2E are presented in Fig. 13. The initial run of E2E has a performance of 3.28 MiB/s. The results of the diagnosis show that a large number of small writes, which can be revealed by `POSIX_SIZE_WRITE_100_1K` and `POSIX_WRITES`, have a significant negative impact on its performance. After we studied the code and parameters of `write_3d_nc4.c`, we found it only writes a subset of the whole 3D space based on this configuration. Because of this, even though it already uses the collective I/O inside, these

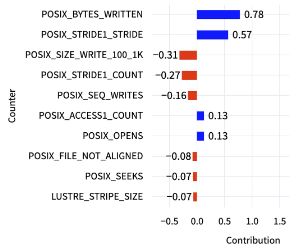


Figure 9: Results for IOR which writes with a stride size. Performance: 1.46 MiB/s

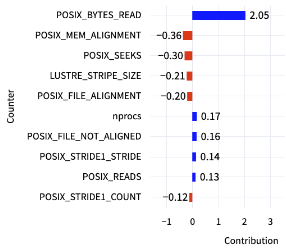


Figure 10: Results for the IOR which reads with a stride size. Performance: 65.33 MiB/s

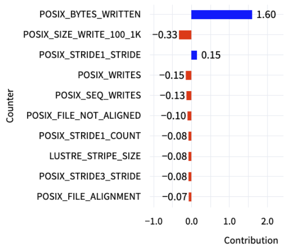


Figure 11: Results for the IOR which writes with random offset. Performance: 1.43 MiB/s

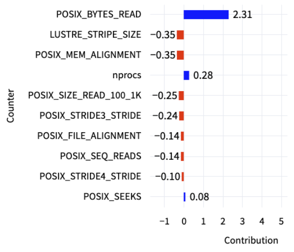
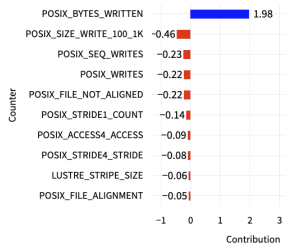
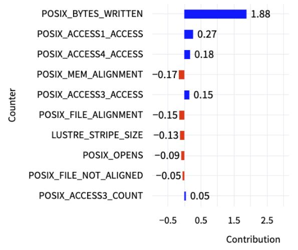


Figure 12: Results of IOR which reads with random offset. Performance: 94.52 MiB/s

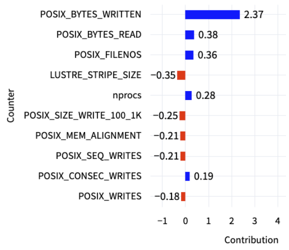


(a) Performance: 3.28 MiB/s

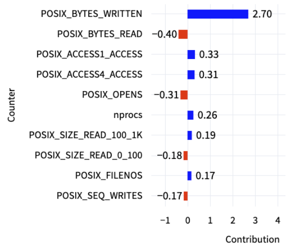


(b) Performance: 482.22 MiB/s

Figure 13: Performance and AIIO diagnosis for E2E.



(a) Performance: 713.65 MiB/s



(b) Performance: 1303.27 MiB/s

Figure 14: Performance and AIIO diagnosis for OpenPMD.

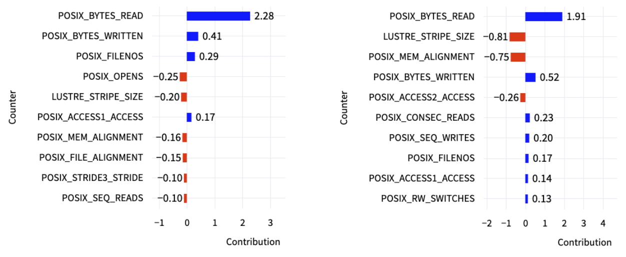
small writes can not be merged because they are not contiguous at their physical offsets. As a result, the E2E has very low I/O performance. To evaluate if our diagnosis was correct, we set the 3D data size to (1024, 64, 32), matching the exact size of the writes of all processes. With this setting, the collective I/O merges small writes into large ones. As shown by Fig. 13b, the performance of E2E with the changed 3D data size increases to 482.22 MiB/s (146× faster than the previous one). These results prove that our AIIO can find the bottleneck in the E2E. Note that here we only focus on evaluating our diagnostic results and changing the data size to prove our idea. In reality, the application stakeholder may insist on the previous size. So, in reality, the I/O strategy of E2E can be implemented in a way that the 3D data can be stored in multiple files. Each file contains a logically contiguous subset, e.g., (1024, 64, 32) in this case. Another possible optimization method is subfiling [9], which splits a large file into smaller files. Incorporating these changes is outside the scope of this paper.

4.2.2 ***OpenPMD***.

OpenPMD [23] defines methods to store and exchange particle- and mesh-based data from scientific simulations and experiments. OpenPMD supports ADIOS BP, HDF5, netCDF, and other file formats. In this experiment, we used the I/O kernel of OpenPMD from h5bench [29] as the representative application for accessing particle and mesh data. We ran OpenPMD with 1024 processes with the following configuration: `dim=3, balanced=true, ratio=1, steps=1, minBlock=64 32 32`, and `grid=1024 8 8`. Experimental results are shown in Fig. 14a. The AIIO diagnosis in Fig. 14a reveals that the stripe size and small writes are the major bottlenecks (i.e., `POSIX_SIZE_WRITE_100_1K`), resulting in a performance of 713.65MiB/s. As a result of the diagnosis, we increased the stripe size to 4M and also disabled the independent option of the OpenPMD (i.e., `OPENPMD_HDF5_INDEPENDENT`) to turn off its independent I/O calls (i.e., small writes). We used the collective I/O to merge these small writes and create large ones. As shown by Fig. 14b, we obtained higher I/O performance at 1303.27 MiB/s (1.82× faster than the original code). The results of the diagnosis in Fig. 14b also show that the stripe size and the small writes (i.e., `POSIX_SIZE_WRITE_100_1K`) are not negative factors. This proves that AIIO can identify I/O bottlenecks in OpenPMD and improve its performance.

4.2.3 ***DASSA***.

DASSA [18] is an open framework with various analysis functions for distributed acoustic sensing (DAS) data. It supports large-scale DAS data analysis on HPC via data partition methods. In this experiment, we used the earthquake search function with the cross-correlation (*xcorr*) method as the study object for our AIIO method. The earthquake search task partitions the data equally among nodes and independently calculates the *xcorr*. Each partition contains *m* 1-minute files and *n* template files. These template files contain identified seismic waves. We ran DASSA with a single node (which uses multiple threads inside for computing). The code accesses 21 1-minute files and *one* template. Since DASSA balances the workload very well, large-scale runs should have the same pattern as this small setting. Fig. 15a reports the AIIO diagnosis of the original DASSA run with a performance of 695.91 MiB/s. `POSIX_OPENS` has the most negative impact on performance. `POSIX_OPENS` means that each process opens too many files here (at least 21 1-minute files). Based on this diagnosis, we merged these 21 files and then changed the DASSA code to work on a single file. These changes delivered 1482.06 MiB/s (2.1× faster than the one



(a) Performance: 695.91 MiB/s **(b) Performance: 1482.06 MiB/s**

Figure 15: Performance and AIIO results of DASSA.

without merging 21 files). The new diagnosis results are shown in Fig. 15b, where `POSIX_OPENS` has no negative impact. The results show that our AIIO helps DASSA improve its performance. In reality, this merging operation may take extra time, but the DAS data analysis is meant to be repeated many times. Thus, merging small DAS files into large ones is practically useful.

In summary, we have demonstrated through experimentation how AIIO can be used to diagnose I/O bottlenecks in three real applications. The results show that AIIO identifies the I/O bottlenecks in each of these applications. By addressing these bottlenecks, we improved the I/O performance of these applications by $1.8\times$,. $2.1\times$, **and** $146\times$, **respectively.**

5 CONCLUSIONS

Identifying the cause of I/O performance bottlenecks for HPC applications can significantly reduce I/O costs and eventually shorten the runtime of the whole application. A user-centric and automatic method for I/O performance bottleneck diagnosis is possible because of the accumulated I/O logs from large-scale supercomputers and recent advances in AI techniques.

However, the existing methods for I/O bottleneck diagnosis only work at the group or platform level. The results from group- and platform-level methods are based on statistical consensus and can be significantly different from the results of an individual job. To realize a job-level and automatic I/O performance bottleneck diagnosis method, this work resolves a few issues, including 1) how to improve the performance prediction accuracy for a single job; 2) how to merge the diagnosis results of multiple models; 3) how to consider the sparsity of the I/O log, and 4) how to generalize our method for an unseen job log. With these solutions, we designed a method, namely AIIO, for job-level and automatic I/O performance bottleneck diagnosis.

AIIO is evaluated extensively with synthetic workloads of different I/O patterns and three real scientific applications. AIIO can help improve the performance of applications by up to 146×. This work can be extended to work on I/O logs from multiple platforms as well as directly supporting HDF5 and MPI-IO interfaces. Automating the map from diagnosis results to code tuning can reduce resource consumption in I/O performance tuning. Our work views it as a regression problem. Another direction to further improve our work is by viewing the I/O diagnosis as a classification problem. With the classification problem, a dataset with accurately tagged bottlenecks can help train the classification models. The recall and precision for diagnosis can be calculated with the availability of the classification models and the tagged dataset.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This effort was supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. This research used the resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] Sercan Ömer Arik and Tomas Pfister. 2019. TabNet: Attentive Interpretable Tabular Learning. *CoRR* abs/1908.07442 (2019). arXiv:1908.07442 http://arxiv.org/abs/1908.07442
[2] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. 2007. Stack Trace Analysis for Large Scale Debugging. In *IPDPS*. 1–10. https://doi.org/10.1109/IPDPS.2007.370254
[3] Jiwoo Bang, Chungyong Kim, Kesheng Wu, Alex Sim, Suren Byna, Hanul Sung, and Hyeonsang Eom. 2021. An In-Depth I/O Pattern Analysis in HPC Systems. In *HiPC*. 400–405. https://doi.org/10.1109/HiPC53243.2021.00056
[4] Jean Luca Bez, Hammad Ather, and Suren Byna. 2022. Drishti: Guiding End-Users in the I/O Optimization Journey. In *2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW)*. 1–6. https://doi.org/10.1109/PDSW56643.2022.00006
[5] Jean Luca Bez, Ahmad Maroof Karimi, Arnab K. Paul, Bing Xie, Suren Byna, Philip Carns, Sarp Oral, Feiyi Wang, and Jesse Hanley. 2022. Access Patterns and Performance Behaviors of Multi-Layer Supercomputer I/O Subsystems under Production Load. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (Minneapolis, MN, USA) *(HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 43–55. https://doi.org/10.1145/3502181.3531461
[6] Jean Luca Bez, Houjun Tang, Bing Xie, David Williams-Young, Rob Latham, Rob Ross, Sarp Oral, and Suren Byna. 2021. I/O Bottleneck Detection and Tuning: Connecting the Dots using Interactive Log Analysis. In *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*. 15–22. https://doi.org/10.1109/PDSW54622.2021.00008
[7] Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, and Gjergji Kasneci. 2021. Deep Neural Networks and Tabular Data: A Survey. *CoRR* abs/2110.01889 (2021). arXiv:2110.01889 https://arxiv.org/abs/2110.01889
[8] Suren Byna, M Scot Breitenfeld, Bin Dong, Quincey Koziol, Elena Pourmal, Dana Robinson, Jerome Soumagne, Houjun Tang, Venkatram Vishwanath, and Richard Warren. 2020. Exahdf5: delivering efficient parallel i/o on exascale computing systems. *Journal of Computer Science and Technology* 35, 1 (2020), 145–160.
[9] Suren Byna, Mohamad Chaarawi, Quincey Koziol, John Mainzer, and Frank Willmore. 2017. Tuning HDF5 subfiling performance on parallel file systems. (5 2017). https://www.osti.gov/biblio/1398484
[10] Surendra Byna, Jerry Chou, Oliver Rubel, Prabhat, Homa Karimabadi, William S. Daughter, Vadim Roytershteyn, E. Wes Bethel, Mark Howison, Ke-Jou Hsu, Kuan-Wu Lin, Arie Shoshani, Andrew Uselton, and Kesheng Wu. 2012. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1109/SC.2012.92
[11] P Carns, K Harms, R Latham, and R Ross. 2012. *Performance analysis of Darshan 2.2. 3 on the Cray XE6 platform.* Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).
[12] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 2009. 24/7 Characterization of petascale I/O workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops (CLUSTER)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–10. https://doi.org/10.1109/CLUSTR.2009.5289150
[13] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. *CoRR* abs/1603.02754 (2016). arXiv:1603.02754 http://arxiv.org/abs/1603.02754
[14] Emily Costa, Tirthak Patel, Benjamin Schwaller, James Brandt, and Devesh Tiwari. 2021. Lessons From Examining Repetitive Job Behavior and I/O Performance Variability on a Production HPC System Emily Costa Northeastern University, USA Tirthak Patel Northeastern University, USA Benjamin Schwaller. *"OSTI"* (8

2021). https://www.osti.gov/biblio/1884199

[15] Emily Costa, Tirthak Patel, Benjamin Schwaller, Jim M. Brandt, and Devesh Tiwari. 2021. 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* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 33, 15 pages. https://doi.org/10.1145/3458817.3476186

[16] Eliakin Del Rosario, Mikaela Currier, Mihailo Isakov, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B Ross, Kevin Harms, Shane Snyder, and Michel A Kinsy. 2020. 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)*. IEEE, 15–21.

[17] Bin Dong, Xiuqiao Li, Limin Xiao, and Li Ruan. 2012. A New File-Specific Stripe Size Selection Method for Highly Concurrent Data Access. In *2012 ACM/IEEE 13th International Conference on Grid Computing*. 22–30. https://doi.org/10.1109/Grid.2012.11

[18] Bin Dong, Verónica Rodríguez Tribaldos, Xin Xing, Suren Byna, Jonathan Ajo-Franklin, and Kesheng Wu. 2020. DASSA: Parallel DAS Data Storage and Analysis for Subsurface Event Detection. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 254–263. https://doi.org/10.1109/IPDPS47924.2020.00035

[19] Anna Veronika Dorogush, Andrey Gulin, Gleb Gusev, Nikita Kazeev, Liudmila Ostroumova Prokhorenkova, and Aleksandr Vorobev. 2017. Fighting biases with dynamic boosting. *CoRR* abs/1706.09516 (2017). arXiv:1706.09516 http://arxiv.org/abs/1706.09516

[20] Jerome H. Friedman. 2001. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics* 29, 5 (2001), 1189 – 1232. https://doi.org/10.1214/aos/1013203451

[21] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2001. *The Elements of Statistical Learning*. Springer New York Inc., New York, NY, USA.

[22] Dean Hildebrand, Arifa Nisar, and Roger Haskin. 2009. pNFS, POSIX, and MPI-IO: a tale of three semantics. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. 32–36.

[23] Axel Huebl, Rémi Lehe, Jean-Luc Vay, David P. Grote, Ivo F. Sbalzarini, Stephan Kuschel, and Michael Bussmann. 2017. Open Science with openPMD. https://doi.org/10.5281/zenodo.822396

[24] M. Isakov, M. Currier, E. Rosario, S. Madireddy, P. Balaprakash, P. Carns, R. B. Ross, G. K. Lockwood, and M. A. Kinsy. 2022. A Taxonomy of Error Sources in HPC I/O Machine Learning Models. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*. IEEE Computer Society, Los Alamitos, CA, USA, 205–218. https://doi.ieeecomputersociety.org/

[25] Mihailo Isakov, Eliakin del Rosario, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B. Ross, and Michel A. Kinsy. 2020. Toward Generalizable Models of I/O Throughput. In *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*. 41–49. https://doi.org/10.1109/ROSS51935.2020.00010

[26] Mihailo Isakov, Eliakin del Rosario, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B. Ross, and Michel A. Kinsy. 2020. HPC I/O Throughput Bottleneck Analysis with Explainable Local Models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13. https://doi.org/10.1109/SC41405.2020.00037

[27] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[28] Edward K Lee and Randy H Katz. 1993. An analytic performance model of disk arrays. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. 98–109.

[29] Tonglin Li, Suren Byna, Quincey Koziol, Houjun Tang, Jean Luca Bez, and Qiao Kang. 2021. h5bench: HDF5 I/O Kernel Suite for Exercising HPC I/O Patterns. In *Proceedings of Cray User Group Meeting, CUG 2021*.

[30] Glenn K. Lockwood, Shane Snyder, Teng Wang, Suren Byna, Philip Carns, and Nicholas J. Wright. 2019. A Year in the Life of a Parallel File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 74, 13 pages. https://doi.org/10.1109/SC.2018.00077

[31] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. 2011. Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing* (San Jose, California, USA) *(HPDC '11)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/1996130.1996139

[32] Scott M. Lundberg, Gabriel G. Erion, Hugh Chen, Alex J. DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2019. Explainable AI for Trees: From Local Explanations to Global Understanding. *CoRR* abs/1905.04610 (2019). arXiv:1905.04610 http://arxiv.org/abs/1905.04610

[33] Scott M Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 4765–4774. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

[34] T.M. Madhyastha and D.A. Reed. 2002. Learning to classify parallel input/output access patterns. *TPDS* 13, 8 (2002), 802–813. https://doi.org/10.1109/TPDS.2002.1028437

[35] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Sclatter Ellis, and M.L. Best. 1996. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems* 7, 10 (1996), 1075–1089. https://doi.org/10.1109/71.539739

[36] Arnab K. Paul, Ahmad Maroof Karimi, and Feiyi Wang. 2021. Characterizing Machine Learning I/O Workloads on Leadership Scale HPC Systems. In *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 1–8. https://doi.org/10.1109/MASCOTS53633.2021.9614303

[37] Allan Pinkus. 1999. Approximation theory of the MLP model in neural networks. *Acta numerica* 8 (1999), 143–195.

[38] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. 1135–1144.

[39] Philip C. Roth. 2007. Characterizing the I/O Behavior of Scientific Applications on the Cray XT. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07* (Reno, Nevada) *(PDSW '07)*. Association for Computing Machinery, New York, NY, USA, 50–55. https://doi.org/10.1145/1374596.1374609

[40] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. 1998. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal* 7, 1 (1998), 48–66.

[41] Seetharami Seelam, I-Hsin Chung, Ding-Yong Hong, Hui-Fang Wen, and Hao Yu. 2008. Early experiences in application level I/O tracing on blue gene systems. In *IPDPS*. 1–8. https://doi.org/10.1109/IPDPS.2008.4536550

[42] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 20, 2 (may 2006), 287–311. https://doi.org/10.1177/1094342006064482

[43] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning Important Features Through Propagating Activation Differences. *CoRR* abs/1704.02685 (2017). arXiv:1704.02685 http://arxiv.org/abs/1704.02685

[44] Ravid Shwartz-Ziv and Amitai Armon. 2021. Tabular Data: Deep Learning is Not All You Need. *CoRR* abs/2106.03253 (2021). arXiv:2106.03253 https://arxiv.org/abs/2106.03253

[45] Mukund Subramaniyan, Anders Skoogh, Jon Bokrantz, Muhammad Azam Sheikh, Matthias Thürer, and Qing Chang. 2021. Artificial intelligence for throughput bottleneck analysis – State-of-the-art and future directions. *Journal of Manufacturing Systems* 60 (2021), 734–751. https://doi.org/10.1016/j.jmsy.2021.07.021

[46] Jeffrey S. Vetter and Michael O. McCracken. 2001. Statistical Scalability Analysis of Communication Operations in Distributed Applications. *SIGPLAN Not.* 36, 7 (jun 2001), 123–132. https://doi.org/10.1145/568014.379590

[47] Feng Wang, Qin Xin, Bo Hong, Scott A Brandt, Ethan L Miller, and Darrell Long. 2004. File system workload analysis for large scale scientific computing applications. (2004).

[48] Teng Wang, Suren Byna, Glenn K. Lockwood, Shane Snyder, Philip Carns, Sunggon Kim, and Nicholas J. Wright. 2019. A Zoom-in Analysis of I/O Logs to Detect Root Causes of I/O Performance Bottlenecks. In *CCGRID*. 102–111. https://doi.org/10.1109/CCGRID.2019.00021

[49] Teng Wang, Shane Snyder, Glenn Lockwood, Philip Carns, Nicholas Wright, and Suren Byna. 2018. IOMiner: Large-Scale Analytics Framework for Gaining Knowledge from I/O Logs. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 466–476. https://doi.org/10.1109/CLUSTER.2018.00062

[50] Bing Xie, Zilong Tan, Philip Carns, Jeff Chase, Kevin Harms, Jay Lofstead, Sarp Oral, Sudharshan S. Vazhkudai, and Feiyi Wang. 2019. Applying Machine Learning to Understand Write Performance of Large-scale Parallel Filesystems. In *PDSW*. 30–39. https://doi.org/10.1109/PDSW49588.2019.00008

[51] Bing Xie, Zilong Tan, Philip Carns, Jeff Chase, Kevin Harms, Jay Lofstead, Sarp Oral, Sudharshan S. Vazhkudai, and Feiyi Wang. 2021. Interpreting Write Performance of Supercomputer I/O Systems with Regression Models. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 557–566. https://doi.org/10.1109/IPDPS49936.2021.00064

[52] Izzet Yildirim, Hariharan Devarajan, Anthony Kougkas, Xian-He Sun, and Kathryn Mohror. 2022. A Multifaceted Approach to Automated I/O Bottleneck Detection for HPC Workloads. https://sc22.supercomputing.org/proceedings/tech_poster/tech_poster_pages/rpost186.html

A APPENDIX

Table 4 details the I/O counters collected by Darshan that were used in the context of this research. Table 5 sums up MLP used to predict I/O performance. Figure 16 presents the loss plot of training XGBoost. Figure 17 presents the architecture of our AIIO web service to put AIIO into practice.

Table 4: Darshan I/O counters used in this research.

Counter	Description
`nprocs`	count of MPI ranks
`LUSTRE_STRIPE_SIZE`	stripe size
`LUSTRE_STRIPE_WIDTH`	count of OSTs
`POSIX_OPENS`	count of POSIX opens
`POSIX_FILENOS`	count of POSIX fileno operations
`POSIX_MEM_ALIGNMENT`	memory alignment size
`POSIX_FILE_ALIGNMENT`	file alignment size
`POSIX_MEM_NOT_ALIGNED`	count of accesses not memory aligned
`POSIX_FILE_NOT_ALIGNED`	count of accesses not file aligned
`POSIX_READS`	count of writes
`POSIX_WRITES`	count of writes
`POSIX_SEEKS`	count of seeks
`POSIX_STATS`	count of stat/lstat/fstats
`POSIX_BYTES_READ`	total bytes read
`POSIX_BYTES_WRITTEN`	total bytes written
`POSIX_CONSEC_READS`	count of consecutive reads
`POSIX_CONSEC_WRITES`	count of consecutive writes
`POSIX_SEQ_READS`	count of sequential reads
`POSIX_SEQ_WRITES`	count of sequential writes
`POSIX_RW_SWITCHES`	count of switch between read and write
`POSIX_SIZE_READ_*`	read sizes: `0_100,100_1K,1K_10K,10K_100K,100K_1M`
`POSIX_SIZE_WRITE_*`	write sizes: `0_100,100_1K,1K_10K,10K_100K,100K_1M`
`POSIX_STRIDE[1,2,3,4]_STRIDE`	the four most frequently appearing strides
`POSIX_STRIDE[1,2,3,4]_COUNT`	count of each of the most frequent strides
`POSIX_ACCESS[1,2,3,4]_ACCESS`	the four most frequently appearing access sizes
`POSIX_ACCESS[1,2,3,4]_COUNT`	count of each of the most frequent access sizes

Table 5: Model summary for MLP.

Layer (Type)	Output Shape	Param #
`dense_35 (Dense)`	`(None, 90)`	4140
`dense_36 (Dense)`	`(None, 89)`	8099
`batch_norm_25 (BN)`	`(None, 89)`	356
`dropout_25 (Dropout)`	`(None, 89)`	0
`dense_37 (Dense)`	`(None, 69)`	6210
`batch_norm_26 (BN)`	`(None, 69)`	276
`dropout_26 (Dropout)`	`(None, 69)`	0
`dense_38 (Dense)`	`(None, 49)`	3430
`batch_norm_27 (BN)`	`(None, 49)`	196
`dropout_27 (Dropout)`	`(None, 49)`	0
`dense_39 (Dense)`	`(None, 29)`	1450
`batch_norm_28 (BN)`	`(None, 29)`	116
`dropout_28 (Dropout)`	`(None, 29)`	0
`dense_40 (Dense)`	`(None, 9)`	270
`batch_norm_29 (BN)`	`(None, 9)`	36
`dropout_29 (Dropout)`	`(None, 9)`	0
`dense_41 (Dense)`	`(None, 1)`	10

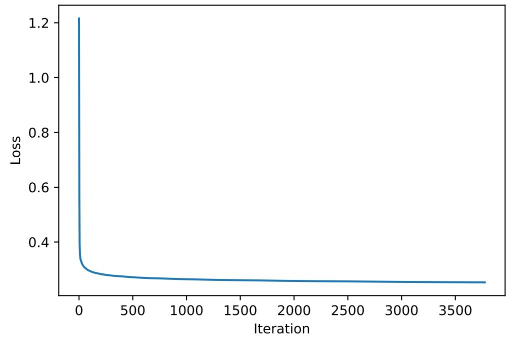
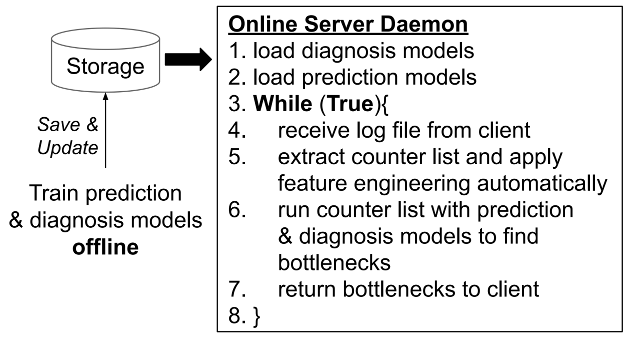


Figure 16: Plot for the loss for XGBoost during its training. The y-axis is the loss, which is defined as RMSE. The x-axis is the number of iterations. We observed similar plots for other models used in this paper.



```
1. load diagnosis models
2. load prediction models
3. While (True){
4.     receive log file from client
5.     extract counter list and apply
       feature engineering automatically
6.     run counter list with prediction
       & diagnosis models to find
       bottlenecks
7.     return bottlenecks to client
8. }
```

Figure 17: Architecture of web service for our AIIO, which loads pre-trained models to identify I/O bottleneck for users.