Understanding and Predicting Cross-Application I/O Interference in HPC Storage Systems

Chris Egersdoerfer[1], Md. Hasanur Rashid[1], Dong Dai[1], Bo Fang[2], and Tallent Nathan[2]

[1]Department of Computer and Information Sciences, University of Delaware, {cegersdo, mrashid, dai}@udel.edu
[2]Pacific Northwest National Laboratory, {bo.fang, Nathan.Tallent}@pnnl.gov

***Abstract*—On High Performance Computing (HPC) systems, where multiple concurrent workloads may read and write vast amounts of data stored through a shared network on storage servers, competition for I/O resources between workloads is inevitable. Previous work has thoroughly recognized the impact of such competition-introduced resource contention, highlighting its potential to impact the performance of individual applications significantly. However, no prior work on such an issue has investigated the quantitative impact of inter-application I/O contention on individual applications, impeding a more efficient resource provision strategy. In this work, we first exemplify the dynamics of I/O interference towards I/O patterns and system status. We then propose a framework for collecting fine-grained I/O traces from applications and concurrent server-side metrics and train a machine learning model to accurately predict the existence of I/O interference and its quantitative impacts. Our results show that it is feasible to learn the complex factors and relationships which cause applications to underperform in the presence of I/O interference. Additionally, we show that a trained model can accurately predict the impact of I/O interference on HPC applications with F1 scores exceeding 90% for both synthetic benchmarks and real-world applications.**

I. Introduction

In high-performance computing (HPC) systems, data accesses are typically facilitated by globally shared Parallel File Systems [1]. Hence, multiple applications may conduct I/O operations simultaneously on the same storage or network resources, creating *I/O interference*: a type of I/O performance degradation that occurs due to the contention from concurrent I/O requests by applications to shared storage resources.

Given the importance of delivering high I/O performance, researchers have studied I/O interference extensively. These studies have primarily answered two questions: 1) what are the root causes of I/O interference and 2) how can I/O interference be avoided or minimized at runtime. For the former, researchers have found that various factors, such as frequent disk head seeks [2], network contention [3–5], file access patterns [6], or even application behaviors [7], can lead to I/O interference. For the latter, various methods have been proposed to alleviate I/O interference, including new I/O scheduling policies [8, 9], adaptive I/O middleware [10], using storage resources such as burst buffers to serve the I/O requests [11, 12] temporarily, and applying token bucket filters to limit the I/O rate to avoid congestion [13].

Despite this large body of existing work, there remains a limited quantitative understanding of I/O interference. The qualitative studies which show that multiple factors contribute to I/O interference can only help explain why certain slowdowns occurred, not predict when or how much an application may be impacted by the I/O interference, reducing their applicability to avoiding interference. For instance, previous work has shown that network contention could lead to I/O interference [3]. However, under the same level of network contention, some applications may be disproportionately affected compared to others. If the application issues intensive write requests, the effect may be significant; if the application only issues light write requests, the interference may be negligible. Hence, it is critical to quantitatively understand the impact of I/O interference, so that adaptive strategies can be leveraged to address the mitigation of its worst effects.

To fulfill this need, this study explores the feasibility of a machine learning-assisted framework to gain a quantitative understanding of I/O interference at runtime using collected system metrics. To each interested application, the framework aims to offer a key functionality: a prediction of the relative performance degradation that the application will experience in the near future given its current I/O request pattern and the system runtime.

However, there are many challenges to implementing such a framework. The first is deciding which system metrics should be leveraged to accurately indicate the presence of I/O interference. A further challenge is deciding what is an effective method for capturing the complex relationships between these metrics and a given application's I/O behaviour at runtime to predict the impact of interference. Finally, it is challenging to maintain real-time monitoring and modelling capabilities at the scale of HPC systems.

In this preliminary study, we provide a detailed analysis of the dynamics of I/O interference in the real-world and motivate the necessity of obtaining a quantitative understanding of I/O interference at runtime. We also introduce our machine learning system designed to address these challenges, and show the accuracy of our model.

The remainder of this work is structured as follows. In section II, we provide a detailed analysis of I/O interference. Section III outlines the design of our scalable monitoring infrastructure and machine learning model. Section IV evaluates the accuracy of our trained model on a set of benchmarks as well as real applications. In section V, we discuss related work. Finally, we conclude this work in section VI.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00174

IO500 Section	*ior-easy-read*	*ior-hard-read*	*mdt-hard-read*	*ior-easy-write*	*ior-hard-write*	*mdt-easy-write*	*mdt-hard-write*
ior-easy-read	29.304	10.722	10.895	1.004	1.285	1.002	1.003
ior-hard-read	5.747	15.156	5.789	3.593	1	3.394	0.998
mdt-hard-read	1.058	1.394	1.199	1.009	1.01	2.106	3.961
ior-easy-write	4.384	1.047	0.976	2.72	5.012	1.802	3.032
ior-hard-write	3.383	0.956	1.291	2.946	4.252	1.273	1.586
mdt-easy-write	1.441	1.018	1.022	1.044	1.032	1.465	1.539
mdt-hard-write	11.145	4.211	1.19	26.219	40.923	1.48	1.496

TABLE I: IO500 task slowdown when different types of interfering I/O patterns are present. Cells highlighted in red show the most impacted cases for each selected IO500 benchmark.

II. I/O Interference Cases and Analysis

In this section, we present a series of analyses investigating the observed variance in I/O performance of both I/O benchmarks and real-world HPC workloads in different scenarios. We demonstrate that interference widely exist across multiple scenarios and the strong need for interference prediction based on quantitative methods.

A. IO500 Benchmark Analysis

In the first set of experiments, we leveraged the IO500 benchmark suite [14], which consists of a set of configured benchmark tasks using IOR and MDTest [15]. These tasks cover a diverse set of data and metadata access patterns, and allow for simple control over which patterns are run at a given time. To observe how different I/O access patterns interfere with each other, we selected 7 representative tasks and compared the performance of running them alone vs. mixing them with another IO500 task as background noise.

The results are shown in Table I. Each row in the table corresponds to one of the 7 standalone IO500 tasks. Each column indicates a mixing case, where another IO500 benchmark task is running concurrently on other computing nodes to create unique cross-application I/O interference patterns. The value of each cell corresponds to the slowdown experienced by the standalone benchmark task (row label) when the corresponding I/O interference (column label) was used to create background noise. Also note, each value is based on the average of 3 consecutive runs and each node running interference tasks was configured to ensure 3 concurrent runs remain active for the entirety of the consecutive runs.

From these results we gather two key insights. First, the results show that the same I/O workload may experience significant variance in its observed performance degradation depending on the type of I/O patterns causing interference. For example, the first row of data shows that reads to a single file may be significantly affected by other contending read patterns but hardly affected by data writes or metadata access patterns. Additionally, the results show that applications undergoing multiple phases with different I/O behavior (in this case iterating through the IO500 tasks) may experience disproportionate performance degradation on a subset of its phases under the same type of I/O interference. For instance, an application that chronologically runs the 7 benchmarks one by one will experience slowdown ranging from 1.0x to 40.9x under the same *ior-hard-write* workload.

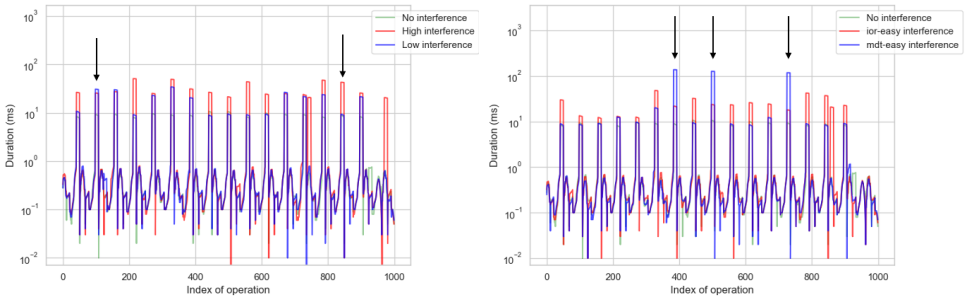
B. Real Application Results

In the second set of experiments, we leveraged a real-world application, Enzo [16], to conduct non-cosmological collapse tests and monitored the I/O performance. While running Enzo, we concurrently ran a set of IO500 benchmark instances from different computing nodes, inducing cross-application I/O interference in the system. To analyze the effect of I/O interference, we compared the performance of each I/O operation between 1) the standalone execution without any cross-application interference and 2) the concurrent execution with IO500 instances varying in type or level of intensity.

We report two major results in Figure 1. These plots consist of the same sequence of I/O requests extracted from the first 50 seconds of the baseline execution of Enzo. The x-axis shows the indices of I/O operations, and the y-axis shows the time spent by each I/O operation. The exact time of each I/O request is collected from Darshan DXT logs [17].

Figure 1(a) shows the I/O performance of Enzo with various amounts of concurrent *data write* workloads running in the background. In particular, we used a varying amount of IO500 *ior-easy-write* instances to simulate increasing *data write* contention on all object storage servers (OSTs). Here, we can observe that Enzo's I/O performance is indeed impacted by the external concurrent data access operations. More importantly, we can observe the impacts are not uniformly applied as some I/O requests experience nearly no impact while others experience significant slowdown under the same interference context. Further, most operations impacted by I/O performance are impacted more by more intense interference as shown by the rightmost arrow in Figure 1(a), but there are also cases where interference caused performance degradation but the amount of interference had no effect on the amount of performance degradation as shown by the leftmost arrow in Figure 1(a). This fact again suggests that a uniform treatment for mitigating the I/O interference may be inefficient, which unfortunately is the current practice.

Figure 1(b) further shows how different types of background contention may have varying effects on real applications. Here, we ran two types of applications to induce contention: the data-intensive IO500 *ior-easy-write* workload and a metadata-intensive IO500 *mdt-easy* workload to create contention on the metadata target (MDT). The result clearly shows that these two workloads impact the performance of I/O requests differently. For instance, the several arrows in Figure 1(b) point to cases where operations were affected far more negatively by the metadata-intensive workload while most other cases where



a) Enzo I/O performance with different levels of IO500 IOR-easy causing interference

b) Enzo I/O performance with different types of IO500 workloads causing interference

Fig. 1: I/O performance of Enzo with different levels and types of I/O interference from IO500. Enzo issues `read`, `write`, `open`, `close` and `stats` within the first 50 seconds of execution. All results are smoothed using a moving window.

performance degradation occurred show a larger negative impact from the data-intensive workload.

In light of these results, the goal of this study is to learn the combinations of features from both applications (ie. Enzo) as well as the contending I/O behaviors on the system which lead to the observed fine-grained performance impacts. To this end, we aim to accurately identify and predict the occurrence of I/O interference for an application and estimate the significance of its impacts on that application. With such a capability, users can develop more effective methods to mitigate such impacts in real-world systems.

III. Design and Implementation

Figure 2 shows the prototype design of our framework. As discussed earlier, to accurately understand I/O interference, we need to: 1) characterize the pattern of I/O requests from the application, 2) record the real-time utilization of shared storage resources such as MDTs or OSTs, and 3) learn the interactions among collected features and apply them at runtime. These tasks correspond to the three main components as shown in Fig 2: 1) Client-Side Monitor, 2) Server-Side Monitor, and the 3) Training Server.

A. Client-Side Monitor

The *Client-Side Monitor* runs on the computing nodes to collect real-time per-application I/O request information. We implemented this module by modifying the Darshan I/O profiling framework [18]. The collected I/O information is buffered in a shared memory (SHM) space and then aggregated by independent processes (part of MPI aggregator) running in the background across all computing nodes where the application is running. The aggregation is done periodically based on a user-defined time window size. The results are then sent to the training server for offline training and real-time prediction. The following list summarizes the metrics we collect in the client-side monitor:

- *# of I/O requests.* The individual and combined counts of different types of I/O requests conducted in the time window. Three types of I/O requests are collected: read, write, and metadata operations.
- *I/O sizes.* The individual and combined sum of bytes conducted by read and write requests in the time window.
- *Actual I/O time.* Total time doing I/O within the time window as well as calculated throughput and IOPS. This contextualizes the intensity and span of I/O requests within the time window.

B. Server-Side Monitor

The *Server-Side Monitor* runs on the storage servers to collect real-time, per-server status information. It runs as an independent process on each parallel file system (PFS) server and periodically pulls key statistical information to denote the usage of each storage server. Like the client-side monitor, the server-side monitor buffers the collected metrics and sends them to the training server periodically using the same time window size. All metrics in this section are recorded once every second and a sum, mean, and standard deviation over all seconds in a given time window are calculated. As summarized in Table II, the first set of metrics (*delivered read/write performance*) shows how the server performed in delivering read/write operations in the last time interval. The second set of metrics covers disk sector operations, which indicate the I/O request patterns the server is experiencing as different I/O patterns may cause the server to be sensitive to specific upcoming workloads. The third set of metrics focuses on the read/write queues. Specifically, we collect the total number of all I/O requests in the local queue. Additionally, we collect the total waiting time aggregated over all I/O requests in the queue to estimate if the current server is in a backlogged status.

C. Training Server

The *Training Server* component takes the periodically collected metrics from applications and storage servers and fills them into a set of *per-server vectors*. There will be one vector for each storage server and each vector consists of one time window worth of client-side metrics targeting the given server and server-side metrics collected from the server. The per-server vectors are then used as input to a kernel-based neural network model for training.

The kernel-based network design was chosen for this task in order to account for the fact that some applications may only

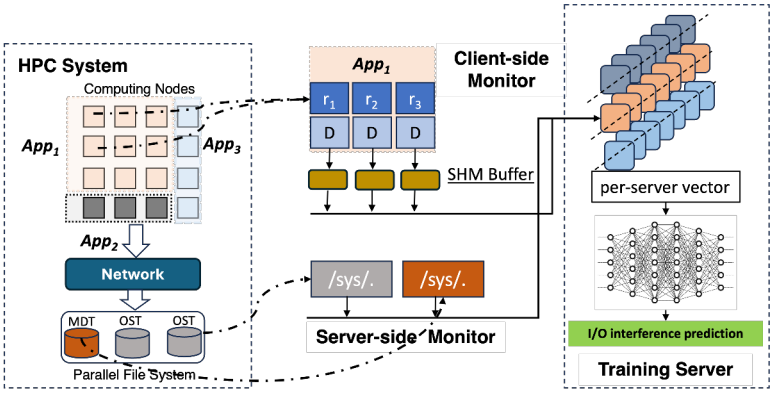


Fig. 2: The overall design and workflow of the proposed framework. It includes three runtime components: client-side monitor, server-side monitor, and the training servers.

TABLE II: List of server-side metrics.

I/O Speed	1) The number of completed I/O requests in the time window.
Device Metrics	1) The number of disk sectors read and written in the time window.
Read/Write Queue	1) The amount of I/O requests queued within the time window. 2) The number of read and write requests merged with others in the queue within the time window. 3) The sum of the amount of time in which I/O requests have been queued (irrespective of the amount of requests queued within the time window) 4) A sum of the amount of time which all requests in the queue have been in the queue for within the time window

utilize a subset of OSTs or target different ones in multiple runs. Specifically, the kernel-based model applies the same dense network to each of the server's vectors, and learns to generally interpret the data from any server. Once the kernel-based network has processed each of the per-server vectors, resulting in a single value for each server, all output values are concatenated and further fed through a simple MLP classification network for multi-bin classification. After training, the model is deployed in the same training server and receives time window metrics from both the server-side and client-side monitors in the same per-server vector format at runtime.

D. Training Data Collection

We collect high-quality labelled data by executing an application in the presence and absence of additional I/O workloads running on other computing nodes. For clarity, we refer to the application which serves as the baseline as the *target workload* and all other workloads as *interference workloads*. The relative difference for the same operations between these two cases determines the ground truth label as indicated by the following equation where degradation level corresponds to the relative impact of I/O interference on the given I/O requests:

$$Level_{degrade} = Avg_{i \in IORequests} \frac{iotime^{i}_{interference}}{iotime^{i}_{base}}$$

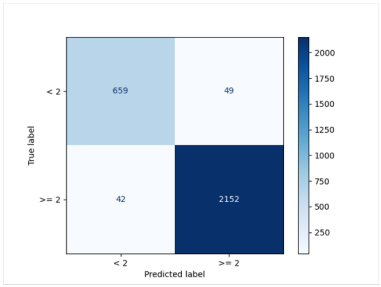
We created varying levels of background I/O requests (using IO500) to cover different types and levels of I/O interference and system statuses for training. The interference workloads always run on separate nodes from the original application to avoid complexity introduced by local resource contention. Also note, the labelling process is performed offline due to the time consuming nature of matching sets of operations between large trace logs. During the labelling process we randomly select time windows accounting for 20% of the total amount of windows and reserve these for a test set, resulting in a 80-20 split of training to testing data. The performance of the models trained using this approach is evaluated in section IV.

IV. Preliminary Evaluation

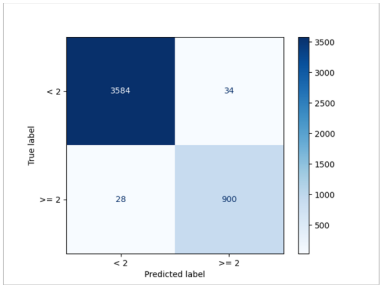
1) Cluster Configuration: The cluster used for evaluation was set up using 11 Linux machines and version 2.12.8 of the Lustre [1] parallel file system. Each machine is equipped with an Intel Xeon CPU (4-10 physical cores), DDR4 main memory ranging from 32GB to 140GB, a 7200 RPM SATA3 disk (1 TB), and a 1 GB/s network interface. Four of the 11 machines were configured as Lustre servers, with one serving dual roles as both the management and metadata server (MGS and MDS, repsectively), and the other three functioning as object storage servers (OSS). The remaining machines serve as Lustre clients. Each OSS is equipped with two object storage targets (OSTs) for data storage.

2) Training Data: We leveraged three training data sets:

- The first dataset was generated from the IO500 benchmark, which includes typical workloads such as ior-easy focusing on sequential data reads/writes and mdtest-easy focusing on metadata operations.
- The second dataset was generated from the DLIO benchmark [19]. DLIO simulates deep learning applications' I/O behaviour by executing their data loaders with different types of training data. We include two configurations of DLIO covering I/O behaviors of widely used models, including Unet3d[20] and BERT [21].
- In addition to the benchmarks, we also tested our framework using three HPC applications. 1) AMReX [22]



a) Confusion matrix results for IO500 benchmark workloads (on testing data only).



b) Confusion matrix results for DLIO benchmark workloads (on testing data only).

Fig. 3: The model evaluation results for models trained and tested on each respective benchmark dataset represented as confusion matrices. For each matrix, true positives correspond to the bottom right, false positive correspond to the top right, true negatives correspond to the top left, and false negative correspond to the bottom left.

is a framework for highly concurrent, block-structured adaptive mesh refinement. 2) Enzo [16] is a cosmological structure formation simulation framework. 3) OpenPMD [23] is a framework that assists scientists in adding context to their datasets, thereby streamlining development processes and simplifying workflow adjustments. AMReX and Enzo are I/O intensive and OpenPMD represents metadata-intensive applications.

A. Benchmark Results

The first evaluation measures the performance of our framework on benchmark workloads. Specifically, we trained the machine learning model on the data generated by each benchmark workload and evaluated the performance of the trained model on the corresponding test set sampled from the same benchmark executions. We used both IO500 and DLIO benchmarks to conduct these evaluations.

In this preliminary study, we start from training the binary classification model, predicting if the application would experience at least 2x slower I/O performance ($>= 2$) or not (< 2). Note that, we do not try to predict the exact slowdown ratio as the exact ratio (e.g., $2.5x$ v.s $2.7x$) is often less important than knowing the I/O slowdown is in certain category (e.g., $>= 2$ or $>= 5$) in quantitatively understanding I/O interference.

For IO500, we collected 11,638 total I/O request windows (samples) for training. Among them, 8,647 samples are in the I/O interference category ($>= 2$) and 2,991 samples were not affected by interference (< 2). The testing data consisted of 2,902 samples, of which 2,194 exhibit slowdown due to interference and 708 do not exhibit slowdown. The results of training and testing on this split are shown in Figure 3(a). We conducted similar experiments on the DLIO dataset, which contains 18,426 training samples, among them 3,702 are positive and 14,724 were negative. Its testing dataset has 4,546 samples with 928 positive and 3,618 negative. The results are shown in Figure 3(b).

From both figures, we can observe that the model trained on each benchmark training set exhibits a high degree of accuracy on its respective test set. This is evident by the small amount of both false positives (top-right portion of each figure) and false negatives (bottom-left portion of each figure).

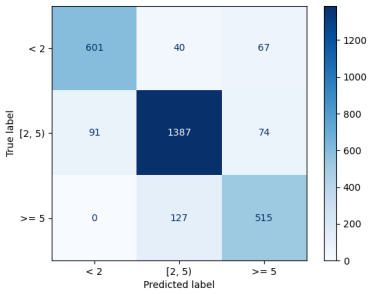
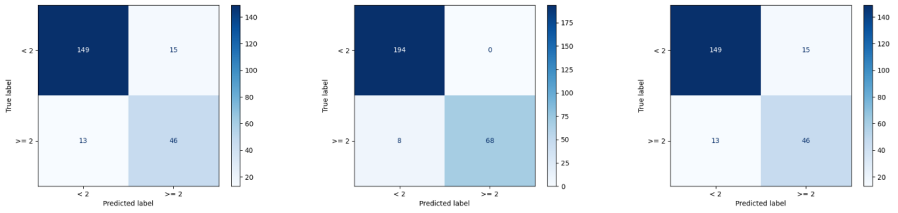


Fig. 4: Evaluation of model accuracy on IO500 data in a multi-class classification setting. Note the large amount of true positive in the top-left, central, and bottom-right bins.

B. Multi-class Prediction Results

In our design, the amount of classification bins is configurable. This allows for the key capability of the proposed model and data collection process to predict more than two levels of interference severity. To evaluate the performance of this, we minimally adjusted the output layer of our proposed model architecture to three output nodes. The ground truth classification labels in the training and testing data were also adjusted to reflect the 3-class classification. Specifically, the bin threshold values were set to 2 and 5, meaning the first bin contains time windows which experienced less than $2x$ slowdown, the middle bin contains time windows which experienced $2x - 5x$ slowdown, and the last bin contains time windows which experienced $5x$ or more slowdown. The configuration of bins are inspired by Lu et al. [24], which described them as mild, moderate, and severe slowdown. With these adjustments, the model was trained and tested on the training and testing sets of IO500. The results are shown in 4. The results show that in the vast majority of samples, the trained model predicts the correct ground-truth labels. Notably, the middle bin shows slightly better precision and recall due to its larger representation in the dataset.



a) Confusion matrix results for AMReX (On testing set only)

b) Confusion matrix results for Enzo (On testing set only)

c) Confusion matrix results for OpenPMD (On testing set only)

Fig. 5: Results for training and testing on AMReX, Enzo and OpenPMD (in order from left to right). Models trained and evaluated on AMReX and Enzo show good performance while the model trained and evaluated on OpenPMD is worse.

C. Real Application Results

To verify the proposed framework still works for real-world application, we evaluated the prediction accuracy of our model on several real applications using the same general data collection setup as for the previously mentioned benchmarks (IO500 and DLIO), where each application was run once without interference to generate a baseline and then repeated three times with increasing amounts of concurrent instances of IO500 launched on each of the other nodes in the cluster. As discussed earlier, we use three workloads: AMReX and Enzo for data-intensive workloads and OpenPMD for metadata intensive workloads. The results are shown in Figure 5.

As shown by Figure 5, the machine learning model can still quickly learn to perform well on the real applications' data. Notably the results for AMReX (left-most matrix in the figure) and especially Enzo (central matrix in the figure) represent the high-performance nature of the respective trained models as evidenced by their large amount of both true positives (bottom-right) and true negatives (top-left) when compared to their false positives (top-right) and false negatives (bottom-left). We do notice the lower performance in OpenPMD (right-most plot). We believe the small amount of data samples collected from this application is a major reason and we plan to further investigate it in future work.

V. Related Work

While approaching the exascale era, a large amount of studies have been conducted to understand cross-application I/O interference. We summarize these into two primary sets.

The first set of research focuses on understanding, defining, and characterizing I/O interference. For instance, Kuo et al. [6], Tseng et al. [25], Yildiz et al. [4], and Xu et al. [26] characterize the locations, including applications' I/O patterns, network, and storage servers, where interference can occur and their respective impacts . Alternatively, Zhang et. al. [2] show that frequent disk head seeks can hurt the performance of a system and Bhatele et al. [3] found the primary factor of performance variability in Cray machines to be interference of multiple jobs that share the same network links. However, these works only provide a qualitative understanding and thus cannot predict the impact of I/O interference.

The second set of works focuses on mitigating I/O inference via reactive or conservative strategies. For example, DFRA [7] reduce I/O interference by avoiding execution of historically interfering applications on the same forwarding nodes. Zhou et al. [27] presented an I/O-aware batch scheduler which coordinates I/O requests on the fly by considering the system state and I/O activities. Gainaru et al. [8] proposed a scheduler which coordinates I/O requests based on past historical behaviors to reduce congestion. AIOC2 [28] learns a model to control Lustre RPC parameters to reduce congestion at runtime. Lastly, IO-Sets [29], provides a fine-grained I/O scheduling strategy to avoid I/O interference based on heuristically defined groups. While all of these works generally provide interfaces for direct actions intended to limit resource contention, they do not differentiate between different types of interference patterns, marking notable difference in purpose from our proposed work. More precisely, our work aims to predict the quantitative performance impact of any observed I/O interference pattern on real-time application I/O behavior, which is not considered by existing I/O interference mitigation strategies. Thus, we consider our work to be complementary to these strategies and helpful motivate more effective I/O interference mitigation strategies.

VI. Conclusion and Future Plan

In this study, we propose a new framework to predict I/O interference in HPC system on the fly. We introduce scalable time window-based metrics monitors, kernel-based neural network, and benchmark-based training data generation strategies. Together, we show the trained model can accurately predict I/O interference for both synthetic benchmarks and real-world applications. We also show the framework can be easily adapted to different clusters. We believe such a framework can fill the gap and be widely adapted in current HPC systems. In the future, we plan to further investigate other possible network architectures, such as transformers [30]. We also plan to evaluate our framework in the leading infrastructure.

VII. Acknowledgement

We sincerely thank the anonymous reviewers for their valuable feedback. This work was supported in part by NSF grants CNS-2008265 and CCF-2412345. This research is also supported in part by the U.S. Department of Energy (DOE) through the Office of Advanced Scientific Computing Research's "Orchestration for Distributed & Data-Intensive Scientific Exploration" and the "Cloud, HPC, and Edge for Science and Security" LDRD at Pacific Northwest National Laboratory.

References

[1] P. Braam, "The lustre storage architecture," *arXiv preprint arXiv:1903.01955*, 2019.

[2] X. Zhang and S. Jiang, "Interferenceremoval: Removing interference of disk access for mpi programs through data replication," in *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010, pp. 223–232.

[3] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: performance degradation due to nearby jobs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[4] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the root causes of cross-application i/o interference in hpc storage systems," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 750–759.

[5] Y. Zhang, T. Groves, B. Cook, N. J. Wright, and A. K. Coskun, "Quantifying the impact of network congestion on application performance and network metrics," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 162–168.

[6] C.-S. Kuo, A. Shah, A. Nomura, S. Matsuoka, and F. Wolf, "How file access patterns influence interference among cluster applications," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2014, pp. 185–193.

[7] X. Ji, B. Yang, T. Zhang, X. Ma, X. Zhu, X. Wang, N. El-Sayed, J. Zhai, W. Liu, and W. Xue, "Automatic,{Application-Aware}{I/O} forwarding resource allocation," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 265–279.

[8] 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*. IEEE, 2015, pp. 1013–1022.

[9] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "Calciom: Mitigating i/o interference in hpc systems through cross-application coordination," in *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE, 2014, pp. 155–164.

[10] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020.

[11] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu, "Trio: Burst buffer based i/o orchestration," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 194–203.

[12] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X.-H. Sun, "Leveraging burst buffer coordination to prevent i/o interference," in *2016 IEEE 12th international conference on e-Science (e-Science)*. IEEE, 2016, pp. 371–380.

[13] Y. Qian, X. Li, S. Ihara, L. Zeng, J. Kaiser, T. Süß, and A. Brinkmann, "A configurable rule based classful token bucket filter network request scheduler for the lustre file system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[14] J. Kunkel, J. Bent, J. Lofstead, and G. S. Markomanolis, "Establishing the io-500 benchmark," *White Paper*, 2016.

[15] H. Shan and J. Shalf, "Using ior to analyze the i/o performance for hpc platforms," 2007.

[16] G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman *et al.*, "Enzo: An adaptive mesh refinement code for astrophysics," *The Astrophysical Journal Supplement Series*, vol. 211, no. 2, p. 19, 2014.

[17] C. Xu, S. Snyder, V. Venkatesan, P. Carns, O. Kulkarni, S. Byna, R. Sisneros, and K. Chadalavada, "Dxt: Darshan extended tracing," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2017.

[18] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular hpc i/o characterization with darshan," in *2016 5th workshop on extreme-scale programming tools (ESPT)*. IEEE, 2016, pp. 9–17.

[19] H. Devarajan, H. Zheng, A. Kougkas, X.-H. Sun, and V. Vishwanath, "Dlio: A data-centric benchmark for scientific deep learning applications," no. 81–91, 2021.

[20] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3d u-net: Learning dense volumetric segmentation from sparse annotation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2016*, S. Ourselin, L. Joskowicz, M. R. Sabuncu, G. Unal, and W. Wells, Eds. Cham: Springer International Publishing, 2016, pp. 424–432.

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[22] W. Zhang, A. Myers, K. Gott, A. Almgren, and J. Bell, "Amrex: Block-structured adaptive mesh refinement for multiphysics applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, pp. 508–526, 2021.

[23] A. Huebl, R. Lehe, J.-L. Vay, D. P. Grote, I. Sbalzarini, S. Kuschel, D. Sagan, C. Mayes, F. Perez, F. Koller *et al.*, "openpmd: A meta data standard for particle and mesh based data," *URL https://doi. org/10.5281/zenodo*, vol. 591699, 2015.

[24] R. Lu, E. Xu, Y. Zhang, F. Zhu, Z. Zhu, M. Wang, Z. Zhu, G. Xue, J. Shu, M. Li *et al.*, "Perseus: A {Fail-Slow} detection framework for cloud storage systems," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 49–64.

[25] S.-M. Tseng, B. Nicolae, F. Cappello, and A. Chandramowlishwaran, "Demystifying asynchronous i/o inter-

ference in hpc applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 391–412, 2021.

[26] H. Xu, S. Song, and Z. Mao, "Characterizing the performance of emerging deep learning, graph, and high performance computing workloads under interference," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2024, pp. 468–477.

[27] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan, "I/o-aware batch scheduling for petascale computing systems," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 254–263.

[28] W. Cheng, S. Deng, L. Zeng, Y. Wang, and A. Brinkmann, "Aioc2: A deep q-learning approach to autonomic i/o congestion control in lustre," *Parallel Computing*, vol. 108, p. 102855, 2021.

[29] F. Boito, G. Pallez, L. Teylo, and N. Vidal, "Io-sets: Simple and efficient approaches for i/o bandwidth management," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

Appendix: Artifact Description

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 Motivated the need for quantifying the impact of interference on HPC applications.
- C_2 Designed file system metrics that enable a machine learning model to identify and quantify interference.
- C_3 Developed a framework for runtime monitoring of metrics and inference using a machine learning (ML) model.
- C_4 Demonstrated the framework's efficacy in quantifying interference for both I/O benchmarks and real HPC applications.

B. Computational Artifact

The artifact for the evaluation is available in a single GitHub repository containing all necessary computational components and data to simulate different execution scenarios.

A_1 https://zenodo.org/doi/10.5281/zenodo.13759882

The following table shows how each contribution maps to the reported evaluation on the main paper.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figure 1
		Table I
	C_2	Table II
	C_3	Figure 2
	C_4	Figure 3-5

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

In support of C_1, the artifact includes the data collected for both Figure 1 and Table I as well as the computation components necessary for data collection. The artifact also includes the necessary code to recreate Figure 1 using the provided data. In support of C_2 and C_3, the artifact contains computational components necessary to collect the server-side and client-side metrics, as well as run the machine learning model. In support of C_4, the artifact contains all of the computational components and data necessary to conduct offline training of the binary prediction model (Figure 3 and Figure 5) and the multi-class prediction model (Figure 4). Additionally, the artifact contains the computational components necessary to visualize model test results, recreating Figures 3-5.

Expected Results

The experiments will demonstrate the necessity and effectiveness of deriving interference quantification. For contribution C_1, the results will illustrate how different I/O patterns are impacted by various types of I/O interference. For contributions C_2 and C_3, the results will include trained ML models based on designed metrics derived from collected PFS data.

For contribution C_4, the results will showcase the framework's ability to quantify interference across diverse evaluation scenarios: I/O benchmarks, multiple quantification categories, and real HPC applications.

Expected Reproduction Time (in Minutes)

Since the data used for evaluation in all cases is provided, the artifact evaluations primarily focus on verifying contributions C_1 and C_4. The evaluation consists of recreating the visualization of Figure 1 using the provided data and training/evaluation a model for each presented modelling scenario (6 total among Figures 3-5). The entire process is expected to take approximately 5 minutes on an Apple MacBook pro with an M2 Pro Max chip.

Artifact Setup (incl. Inputs)

Hardware: There is no specific hardware requirement to setup the cluster used to generate training/evaluation data for the proposed model. In our evaluation, the cluster was set up using 11 Linux machines. Each machine is equipped with an Intel Xeon CPU (4-10 physical cores), DDR4 main memory ranging from 32GB to 140GB, a 7200 RPM SATA3 disk (1 TB), and a 1 GB/s network interface.

Software: The artifact requires a Lustre PFS, and we implemented and evaluated our artifact using Lustre version 2.12.8. The Python version needs to be at least 3.10.

Datasets / Inputs: No dataset needs to be generated to evaluate the scenarios as the collected data and model training code are included in the artifact. However, for collecting data on a different platform, the artifact contains the scripts necessary run the data generation process.

Installation and Deployment: To deploy the artifact, ensure that Lustre is properly configured across all servers and clients. The I/O profiling tool, Darshan, must be installed on the executing clients. For evaluating different scenarios involving I/O benchmarks and HPC applications (e.g., IO-500, Enzo, etc.), the necessary benchmarks and applications must be installed and available in the environment.

Artifact Execution

If new data is intended to be collected and used, the complete workflow may consist of two tasks. The first task, T_1 includes choosing an application to collect data for and running it multiple times in a prescribed set of scenarios where various types and levels of interference are introduced. The

second task, T_2 includes training a new model based on the data generated by T_1.

Since the collected data from executing T_1 for various applications are provided in the artifact, execution of the artifact only requires the execution of T_2, where a new model is trained and evaluated for each application dataset provided.

Artifact Analysis (incl. Outputs)

To observe the outcomes of each evaluation scenario, the $generate\_eval\_results.py$ script automatically trains a new model and generates visualizations of the model's performance for each I/O benchmark or HPC application. These visualizations illustrate the model's accuracy in correctly quantifying interference for the specific evaluation scenario.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

To execute the scripts, the Python version needs to be at least 3.10 along with the following libraries present in the environment. Python, along with libraries can be set up using either the pip install command or the following conda command:

```
conda create -n test_env python=3.10
numpy pandas seaborn matplotlib
scikit-learn torch
```

To activate the conda environment:

```
conda activate test_env
```

To get the code from the GitHub repository to the local system, one needs to clone the GitHub repo in the system:

```
git clone
https://github.com/mrashid2/Quanterference.git
```

Artifact Execution

After cloning the GitHub repository, change directory to inside of the repository:

```
cd Quanterference
```

To execute the artifact, following python script needs to be executed:

```
python generate_eval_results.py
```

Running the $generate\_eval\_results.py$ script will load the data and train/evaluate a new model for each of the application datasets represented in the $NN\_Model/data$ directory by running the model training and evaluation script in $NN\_Model/scripts$. Afterward, it will load the data and run the scripts in the $enzo\_prelim$ directory.

Artifact Analysis (incl. Outputs)

Running the $generate\_eval\_results.py$ script will create the figures in the $eval\_results$ directory in *.png* format. All figures referenced in the paper will be found in this folder, with filenames indicating the type and corresponding figure number.