View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Detecting I/O Access Patterns of HPC Workloads at Runtime

Jean Luca Bez, Francieli Zanon Boito, Ramon Nou, Alberto Miranda, Toni Cortes, Philippe Navaux

▶ To cite this version:

Jean Luca Bez, Francieli Zanon Boito, Ramon Nou, Alberto Miranda, Toni Cortes, et al.. Detecting I/O Access Patterns of HPC Workloads at Runtime. SBAC-PAD 2019 - International Symposium on Computer Architecture and High Performance Computing, Oct 2019, Campo Grande, Brazil. hal-02276191

HAL Id: hal-02276191

https://hal.inria.fr/hal-02276191

Submitted on 2 Sep 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Detecting I/O Access Patterns of HPC Workloads at Runtime

Jean Luca Bez[1], Francieli Zanon Boito[2], Ramon Nou[3], Alberto Miranda[3],
Toni Cortes[3,4], Philippe O. A. Navaux[1]

[1]Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS) — Porto Alegre, Brazil
[2]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
[3]Barcelona Supercomputing Center (BSC) — Barcelona, Spain
[4]Universitat Politècnica de Catalunya — Barcelona, Spain

{jean.bez, navaux}@inf.ufrgs.br, francieli.zanon-boito@inria.fr, {ramon.nou, alberto.miranda}@bsc.es, toni@ac.upc.edu

***Abstract*—In this paper, we seek to guide optimization and tuning strategies by identifying the application's I/O access pattern. We evaluate three machine learning techniques to automatically detect the I/O access pattern of HPC applications at runtime: decision trees, random forests, and neural networks. We focus on the detection using metrics from file-level accesses as seen by the clients, I/O nodes, and parallel file system servers. We evaluated these detection strategies in a case study in which the accurate detection of the current access pattern is fundamental to adjust a parameter of an I/O scheduling algorithm. We demonstrate that such approaches correctly classify the access pattern, regarding file layout and spatiality of accesses – into the most common ones used by the community and by I/O benchmarking tools to test new I/O optimization – with up to 99% precision. Furthermore, when applied to our study case, it guides a tuning mechanism to achieve 99% of the performance of an Oracle solution.**

***Index Terms*—high-performance computing, parallel I/O, access pattern detection, I/O forwarding, classification**

I. Introduction

In the context of HPC applications, I/O is often considered a bottleneck. That is due to a complex interplay of factors and their impact on performance. The organization and configuration of shared storage infrastructure, network topology to access the shared parallel file system (PFS) servers, and the application's *access pattern* play a role in determining the performance as seen by the applications. The last factor is the focus of many optimization techniques as it represents the way applications issue their I/O requests, i.e., type of operation, file layout, sequentiality, and size of the requests.

Optimization techniques that seek to improve I/O performance by modifying the application I/O access patterns often cover aggregations, request reordering, aggregation for collective operations, and I/O request scheduling [1]–[4]. These techniques can be applied at different layers of the I/O stack. In general, **optimization techniques typically provide improvements for particular system configurations and access patterns**, but not for all of them. Furthermore, they often rely on the right selection of parameters, as demonstrated for request scheduling at different levels [5], [6]. Therefore, achieving the best results proposed by an optimization technique often relies on correctly applying them to the proper workload, and configuring it accordingly. The main issue that arises in practice is that the workload keeps changing as new applications start and finish their I/O phases. Therefore, it becomes of paramount importance for systems that seek to auto-tune its parameters to correctly detect the access patterns, at runtime, to make decisions.

Nonetheless, runtime detection techniques should pose minimum overhead and should be able to perform its detection as fast as possible to allow tuning mechanisms and optimizations techniques to act on the information. That would enable the system to benefit from good choices quickly and promptly adapt once the observed I/O behavior changes. In this scenario, we could consider applying machine learning techniques. Although training phases could be expensive, once the model has learned its parameters, inferences on previously unseen data, at runtime, are fast. That would allow such techniques to be applied at runtime, with minimum overhead, given that metrics regarding I/O accesses are available.

Therefore, in this paper, **we demonstrate how machine learning techniques can aid in automatically detecting the I/O access pattern of HPC applications at runtime**. We investigate decision trees, random forests, and neural networks to classify metrics collected at runtime into common access patterns that are often used by the HPC I/O community to evaluate new I/O optimizations. Furthermore, to demonstrate its applicability, **we evaluated these detection strategies in a case study in which the accurate detection of the access pattern is paramount to tune an I/O scheduler's parameter at the I/O forwarding layer**.

The rest of this paper is organized as follows. Section II discusses related work on access pattern detection. Since experimental results are not restricted to a single section but presented throughout the paper, the experimental methodology is presented before any contributions, in Section III. The three approaches we use to detect the access pattern are detailed in Section IV, alongside their evaluation regarding precision. Section V introduces our case study and discusses our results when applying those techniques. Finally, Section VI concludes the paper and points future work.

II. Related Work

Detecting access patterns is an essential topic as it allows adapting the I/O system to the workload. For that, both postmortem and runtime approaches are popular. After the execution, information is often obtained from traces and applied to future executions of the same applications [5], [7], targeting repeated patterns with similar characteristics.

In this work, we prefer a runtime technique to avoid imposing the profiling effort and also to benefit from similarities between different applications. Furthermore, runtime detection allows the applications to start profiting from tuning optimizations more quickly, differently from postmortem analysis.

At runtime, techniques can typically only use information from operations already performed. To predict future accesses, the technique proposed by Dorier et al. [8] named Omnisc'IO, intercepts I/O operations and builds a grammar. The approach employed by Tang et al. [9] periodically analyzes previous accesses and applies a rule library to predict future accesses (for prefetching). They collect metrics regarding spatiality of read requests from the MPI-IO library.

Other techniques benefit from information obtained from I/O libraries. Ge et al. [10] collect data from MPI-IO covering the type of operation, data size, spatiality, and whether or not operations are collective and synchronous. Liu et al. [11] collect the number of processes, the number of aggregators, and binding between nodes and processes. Lu et al. [12] use the offsets accessed by each process during collective operations. The processes' access spatiality is used in the approach proposed by Song et al. [13].

These are client-side techniques. Consequently, they would not work at the forwarding layer or at the server-side, where less information is available, and the observed pattern is the interaction of multiple concurrent patterns. Conversely, in this work, we focus our efforts on detecting the access pattern at the I/O forwarding layer as it is a perfect place to apply optimization since this layer is transparent to applications.

III. I/O Workload Metrics

Modifying the I/O forwarding layer or the file system configuration in production machines to evaluate new mechanisms or optimization techniques is often not allowed. Such actions could disrupt services or even harm an application's performance at scale. Furthermore, to detect the I/O access patterns of HPC workload, we require an initial dataset of metrics. Such datasets with all the necessary metrics to do so are not often made available from production machines sites.

Therefore, we deployed an I/O stack with forwarding infrastructure in clusters from the Grid'5000 [14] testbed. That gives us the flexibility to evaluate our proposal and demonstrate its feasibility to be later applied to large-scale machines.

We collected metrics on each I/O node every second throughout the execution of multiple benchmarks and configurations (details in Section III-A). Since the patterns have a fixed duration, the number of observations is *not* the same for each benchmark, as it depends on the execution time. These metrics comprise a data set of over one million observations.

A. Experimental Methodology

We used two clusters from the Nancy site: four PFS servers in Grimoire; 32 clients and multiple (1, 2, 4, and 8) forwarding nodes in separated Grisou nodes. Nodes from both clusters have similar characteristics. Each one has two 8-core Intel Xeon E5-2630 v3, 128 GB of RAM, and a 558 GB hard disks. A 10 Gbps Ethernet network interconnect the nodes and the two clusters. For our evaluation, both clusters were exclusively reserved during the experiments to minimize interference.

PVFS version 2.8.2 was used with default 64 KB stripe size and striping through all four servers. Data servers perform writes directly to their disks, bypassing caches, to ensure the scale of tests would be enough to see access pattern impact on performance. Clients are equally distributed among the I/O nodes, that communicate directly with the file system through the IOFSL dispatcher. The IOFSL daemon was executed with all its default parameters.

To cover the most common I/O access pattern of HPC applications, we used the MPI-IO Test benchmark tool [15], to issue requests using the MPI-IO library. We varied the number of processes (128, 256, or 512), operation (read or write), file layout (shared-file or file-per-process), spatiality (contiguous or 1D-strided), and request sizes (32 or 256 KB – smaller than the stripe size or larger enough so that all servers are accessed). For each experiment, a total of 4 GB of data was accessed. We also deployed a different number of intermediate I/O nodes (1, 2, 4, or 8). In total, 144 different scenarios (we do not test the file-per-process 1D-strided combinations as this access pattern is not usual) were considered. The full set was executed in random order to minimize unforeseen impacts.

The number of patterns representing different benchmark parameters is detailed in Table I. The complete data set is public and available in the companion repository at `jeanbez.gitlab.io/sbac-pad/2019`.

TABLE I
Representativity of the access patterns in the dataset.

	Observations			
Read	29,929	*vs.*	100,406	**Write**
Shared-file	72,802	*vs.*	57,533	**File-per-process**
Contiguous	94,997	*vs.*	35,338	**1D-strided**

IV. Access Pattern Detection

In this section, we detail three approaches we consider to detect the I/O access pattern of HPC applications at runtime. We explore different machine learning techniques to identify the access pattern based on metrics collected at runtime by the system. **We classify the access patterns by file layout and spatiality into three classes.** File layout relates to the number of files, i.e., if all processes use a single shared file or if each process writes and read to its file. Spatiality expresses if requests issued by the application are contiguous or follow a strided pattern.

The three distinct classes cover I/O patterns that are common among scientific applications and that are used by several

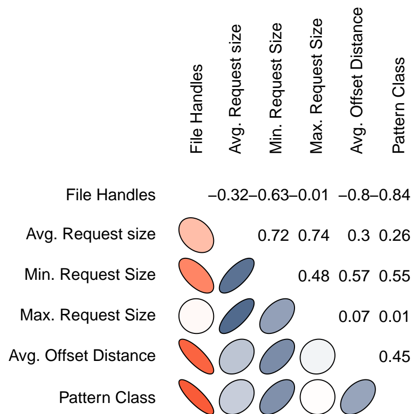


Fig. 1. Spearman's nonparametric correlation coefficient for the metrics selected to classify the access pattern into the three pre-defined classes. Positive correlations are displayed in blue and negative correlations in red. The color intensity and the size of the ellipse are proportional to the coefficients.

benchmarks [15], [16] to test I/O optimizations. Furthermore, they group situations that in our experience, present similar behavior. The three classes are:

- file-per-process with contiguous accesses (FPP);
- shared-file with 1D-strided accesses (SS);
- shared-file with contiguous accesses (SC).

The access pattern detection mechanism receives as input information collected at each I/O node. This information, regarding the past observation period, consists of the number of file handles, the request size (maximum, minimum, average), and the average offset distance between consecutive requests to the same file handle. These parameters were selected by calculating the Spearman's nonparametric correlation [17] to identify the ones most related to the access pattern class, as that correlation determines the strength and direction of the monotonic relationship between two variables.

Fig. 1 shows the coefficients for the selected metrics. It is possible to see a strong negative relationship between the number of file handles and the pattern class. This metric should allow us to detect the file layout. Additionally, the minimum and average request size, and the average offset distance exhibit a direct correlation to the pattern we want to classify. Intuitively these last metrics should allow us to detect if requests are contiguous or 1D-strided.

To build our access pattern detection mechanism, and to eliminate potential noise from start-up and tear-down phases, we extracted the observations from the center of each test. We also made sure to take the same number of observations for each access pattern and configuration, to avoid bias toward one of the classes. The final data set contains 40 observations, from each of the $1,008$ experiments (144 scenarios $\times$ 7 window sizes), yielding a total of ≈ 40 thousand observations.

In this paper, we explore three mechanisms to detect the access pattern. The **results are not restricted to a single section but presented throughout the paper after discussing each mechanism.** We explore decision trees in Section IV-A, and random forests in Section IV-B. We investigate neural networks in Section IV-C. For all the approaches, we have split the 40.240 observations into two: 70% for training (28.168) and 30% for testing (12.072 observations).

Furthermore, it is paramount for any detection mechanism to be able to generalize when faced with previously unseen metrics. That is one of the main reasons we compare distinct learning approaches to this problem. To evaluate their behavior in such scenario we collected additional $54,210$ metrics with requests sizes of 64KB and 128KB covering read and write operations, different number of I/O nodes, processes, file layout, and spatiality.

A. Decision Trees Approach

Decision trees are often an efficient approach to classification problems. Therefore they might prove suitable for detecting the I/O access pattern at runtime. To build our tree, we applied the C5.0 algorithm [18] which is a data mining tool for discovering patterns that delineate categories, assembling them into classifiers, and using them to make predictions. The classifiers are expressed as decision trees or sets of if-then rules, that are generally easy to understand and implement.

Fig. 2 depicts the generated tree with its decisions. It uses three attributes and five predictors: the number of file handles, the average offset distance, and the average received size.

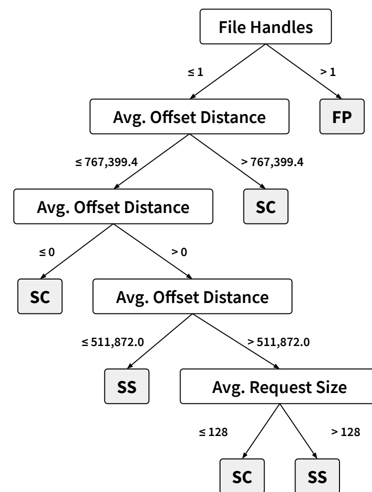
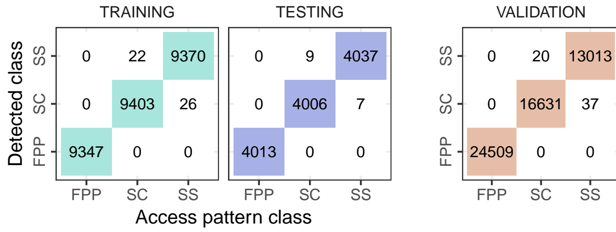


Fig. 2. Decision Tree to classify access patterns into the tree classes: file per process (FP); shared file, contiguous (SC); and shared file, 1D strided (SS).

Fig. 3 illustrates the confusion matrices for the training and testing data sets. The overall accuracy was of 0.9976. Table II details the sensitivity and specificity considering a *one-vs-all* scenario. As we have three classes, these results are calculated by comparing each level to the remaining levels. Sensitivity measures the proportion of actual positives that are correctly identified as such. On the other hand, specificity measures the proportion of actual negatives that are correctly identified.

It is paramount for any detection mechanism to generalize when faced with previously unseen metrics. Therefore, we also evaluated the decision tree behavior with the validation dataset composed of $54,210$ metrics. As depicted by Fig. 3(b), only



(a) $40,240$ patterns for training and testing (b) $54,210$ patterns

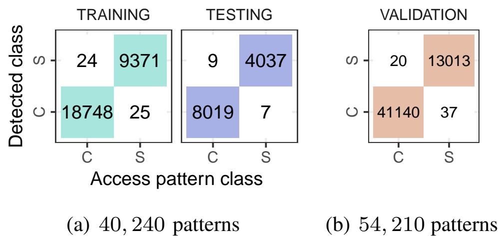
Fig. 3. Confusion matrices for the training, testing, and validation datasets. The x-axis shows the real class, and the y-axis shows what was detected by the DT. The classes are: file per process (FPP); shared file, contiguous (SC); and shared file, 1D strided (SS).

TABLE II
C5.0 ALGORITHM STATISTICS FOR EACH ACCESS PATTERN.

	FPP	**SC**	**SS**
Sensitivity	1.0000	0.9972	0.9977
Specificity	1.0000	0.9988	0.9986

57 observations were incorrectly classified, which represents approximately 0.1% of the total. The accuracy was 0.9989 with Kappa of 0.9984. Kappa [19] can handle both multi-class and imbalanced class problems. It represents how much better the classifier is over another that randomly makes guesses according to the frequency of each class. Values are in the interval $[0, 1]$ where values closer to zero indicate that the classifier is not performing well.

One could argue that identifying file layout, i.e., file-per-process or shared-file could be simplistic or even yield better results if such parameter was not considered when building the three. To evaluate such a hypothesis, we have removed that parameter and repeated our analysis. Differently from our previous approach, the decision tree uses only four predictor variables, and a tree of size equals to five. Furthermore, only two attributes were used to build the tree: the average offset distance, and the average received request size.



(a) $40,240$ patterns (b) $54,210$ patterns

Fig. 4. Confusion matrices for the training, testing, and validation datasets. The x-axis shows the real class, and the y-axis shows what was detected by the DT. The classes are: contiguous (C) and 1D strided (S) accesses.

Fig. 4 depicts the confusion matrix with the training and testing data sets. During training, our model correctly classified $28,119$ of the $28,168$ inputs, an accuracy of 0.9983. When compared to the decision tree using the three classes, the accuracy is very similar. Therefore, we could select the simpler model using it only to identify the spatiality of accesses. However, in practice, as a decision tree is implemented in a series of if-else statements, we do not expect any changes in performance using the simple method.

B. Random Forests Approach

Random forests are an ensemble method which makes predictions by averaging over the predictions of several independent decision trees [20]. Such approach often demonstrates improvements in classification accuracy due to the ensemble of trees and the voting for the most popular class.

To train the random forest we employed a re-sampling using cross-validation (25 fold). Kappa was used to select the optimal model using the largest value considering different values for the *mtry* parameter, which represents number of variables available for splitting at each tree node. Cutler et al. [21] reported that the classification rates and performance metrics of their model were stable with different values for *mtry*. Conversely, Strobl et al. [22] noticed a strong influence on predictor variable importance. Thus, we explored different values for the *mtry* parameter. Table III summarizes the results.

TABLE III
RANDOM FORESTS TO DETECT THE PATTERN CLASS.

	$mtry = 2$	$mtry = 3$	$mtry = 5$
Accuracy	0.99801	0.99818	0.99733
Kappa	0.99701	0.99728	0.99600

Using the best option, i.e. $mtry = 3$, with a forest of 500 trees, the accuracy for training was of 0.9983 and of 0.9986 for testing. Once more, the file-per-process (FP) class is the one with perfect detection, as it only depends on the number of file handles detected at the forwarding layer. For the other two classes, each one has only misclassified 47 and 17 metrics during training and testing.

Due to the perfect detection of the file-per-process class, we also investigated if there are performance improvements by simplifying the model to detect only the spatiality of the requests. We employ the same methodology, data set, and configuration of the random forest as before. Table IV details the results. It is possible to notice that accuracy is also not impacted by simplifying the model. Nonetheless, when applied in practice, an additional verification would be required to detect the file layout, i.e., if each process is issuing an operation to its own file or a single shared-file is used. We do not depict the confusion matrices for the random forest approach as they only presented minor differences to the decision tree. Nonetheless, the complete evaluation is available in the companion repository.

C. Neural Network Approach

Our classifier was built using Keras [23], a high-level Neural Network API, using TensorFlow [24] as back-end. The data set was split into two: 70% for training and 30% for testing.

TABLE IV
RANDOM FORESTS TO DETECT THE SPATIALITY OF THE ACCESSESS.

	$mtry = 2$	$mtry = 3$	$mtry = 5$
Accuracy	0.99827	0.99823	0.99765
Kappa	0.99614	0.99606	0.99478

Before feeding our metrics to the NN, we applied Yeo-Johnson [25], scale, and center data transformations so that the data is better suited for the network, and to speed up training. Yeo-Johnson is a power-transform similar to Box-Cox [26], but it supports features with zero or negative value. The scale transformation computes the standard deviation for a feature and divides each value by that deviation. Finally, the center transform calculates the mean and subtracts it from each value.

Our model consists of three layers, as illustrated by Fig. 5: an input layer with the five features, a hidden layer with the same number of neurons, and an output layer with three units, one for each class. The first two layers use a Rectified Linear Unit (ReLU) [27] activation function with a normal kernel initialization function. The output layer uses *softmax* to squash the outputs of each unit in the range $[0, 1]$ and to ensure that the total sum of the outputs is equal to one.

We used the *RMSProp* optimizer with learning rate of 0.001 and a momentum of 0.9. The loss function was the categorical cross-entropy, where the output has an n-dimensional vector that is all-zeros except for a 1 at the index corresponding to the class of the sample. In our case, we have a 3-dimensional vector for each sample. We trained our model on $22,534$ samples and tested it on $5,634$ samples with a batch size of 32 and 50 epochs. The training accuracy was 99.76% and the testing accuracy was 99.73%.

Fig. 6 shows the confusion matrix of the generated NN with the training and testing data sets. During training, our model correctly classified $28,099$ of the $28,168$ inputs, an accuracy of 99.76%. We also checked the performance of our model with our testing data set (30% of the original data). During testing, our model incorrectly classified only 20 samples out of the $12,072$. It is also important to notice all three classes are correctly identified with a reasonable probability.

We applied the neural network with the validation dataset.

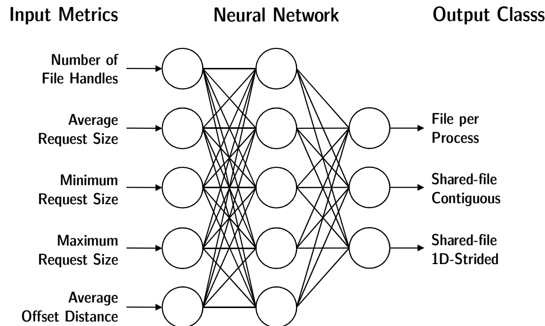
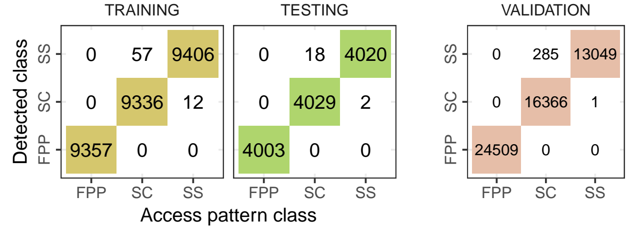


Fig. 5. Neural Network architecture employed to classify the metrics into the three classes, regarding the file layout and the spatiality of access.



(a) $40,240$ patterns for training and testing (b) $54,210$ patterns

Fig. 6. Confusion matrices for training, testing, and validation. The x-axis shows the real class, and the y-axis the class detected by the NN: file per process (FPP); shared file, contiguous (SC); and shared file, 1D strided (SS).

As depicted by Fig. 6(b), 286 observations were incorrectly classified. This represents approximately 0.53% of the total.

D. Discussion

Our results have shown that all detection approaches covered in this work can correctly detect the access pattern. The simplest approach, represented by the decision tree, presented 0.99 accuracy for the training, testing, and validation datasets. The random forest approach also yielded similar results. Therefore the most straightforward method is favored as it could be implemented as a series of if-else statements, which would speed up detection during runtime. Despite yielded equal accuracy, the neural network represents a "black-box" model if compared to the alternatives, and it is slower in practice than a set of if-else statements.

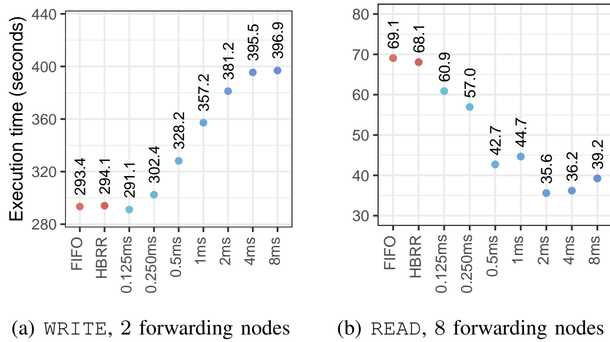
TABLE V
RUNTIME TO TRAIN AND MAKE PREDICTIONS.

	Train (s)	**Predict (ms)**
Decision Tree	0.369	1.371
Random Forest	364.505	1.363
Neural Network	54.177	9.825

We also evaluated the time taken by each approach to train and to make predictions. The training phase is often done offline and can take longer to complete. Moreover, it will only be required to update the model, which should not happen as frequently as the predictions in runtime. Table V summarizes the median of 10 repetitions to train each model and the median prediction time of all the metrics available in the dataset used to train and validate. All the approaches take in the order of milliseconds to predict once the model is trained. However, the neural network takes $\approx 7.2\times$ more than any of the other two methods.

V. CASE STUDY: TUNING AN I/O FORWARDING SCHEDULER PARAMETER

TWINS [6] is an I/O scheduling algorithm designed for the I/O forwarding layer of large-scale clusters and supercomputers. It seeks to coordinate the I/O nodes' accesses to the shared PFS servers to mitigate contention. The scheduler act in the context of each I/O node. It keeps multiple request



(a) `WRITE`, 2 forwarding nodes (b) `READ`, 8 forwarding nodes

Fig. 7. The impact of the window size on performance: makespan (the smaller, the better) from experiments where 128 processes access a 4 GB shared file in 32 KB requests. Baseline algorithms are presented in red and TWINS results are presented in blue. The best window size for each scenario depends on the system configuration and on the application's access pattern.

queues, one for each data server. During a configurable *time window*, requests are taken (in arrival order) from only one of the queues, to access only one of the servers. In this phase, each I/O node is focusing its accesses to different data servers. When the time window ends, the scheduler moves to the next queue following a round-robin scheme.

TWINS can increase performance in up to 48% over other schedulers available for the I/O forwarding layer (FIFO and HBRR provided by IOFSL). These high-performance improvements are obtained transparently, without requiring any modifications in the applications or runtime systems (except the forwarding software). Nevertheless, for some patterns, performance can be decreased by using TWINS, if not correctly configured. Both situations depend on adequately selecting the time window duration parameter, which depends on the I/O access pattern of the applications. Fig. 7 illustrates the impact of the window selection in two scenarios. The values on the x-axis represent different TWINS window sizes. Further details are available in previous work [6], [28].

To demonstrate the applicability of the access pattern detection techniques described in Section III, we evaluate those strategies in a case study in which the accurate detection of the access pattern is fundamental, i.e., selecting the TWINS window size. For that, metrics are collected at the I/O forwarding layer and are used by the detection mechanism to identify the access pattern and tune the scheduler's configuration.

A. Applying the I/O Access Pattern Detection

When applied to a real tuning mechanism, mispredictions could have higher or lower impact, depending on the class and on how optimizations applied to that given class behave in such non-optimal scenarios. Therefore, we complete our investigation by implementing the **decision tree**, **random forest**, and **neural network** methods to the case study presented in Section V, where **we seek to tune the window size of the TWINS scheduler**. We applied the detection mechanism to each observation in the entire data set of over one million entries and used this detection to determine the best window at each instant. In this section, we assume that if a pattern is correctly detected, the best window is always selected. Since we have a comprehensive set of experiments with performance metrics of multiple access patterns using distinct TWINS window sizes, we can evaluate it in an offline fashion.

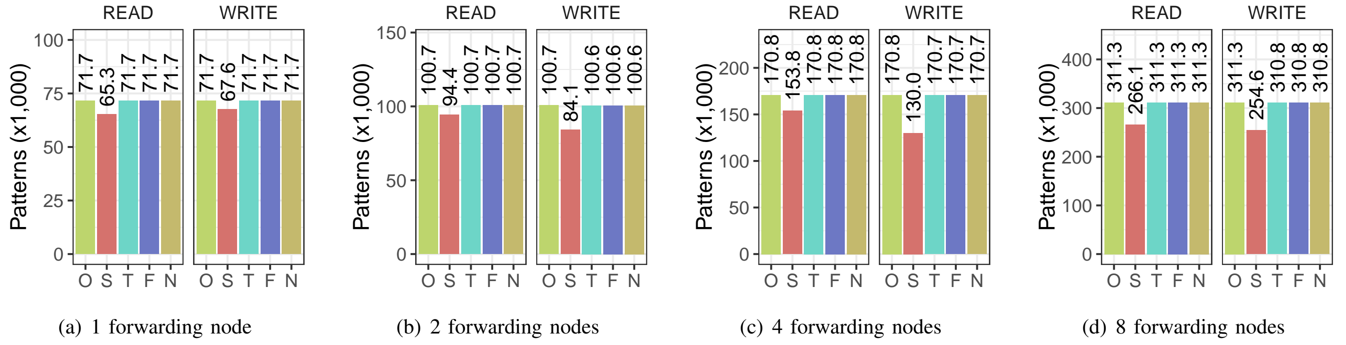
By comparing the performance results with a baseline, we count the number of decisions that resulted in performance improvements or decreases, as presented in Fig. 8 and Fig. 9. The baseline to compare performance was using a 1ms windows, which is a conservative value that decreases (and increases) performance for the least number of scenarios. We compare the results of the three mechanisms with an oracle and a static solution. Because we have performance results with all the seven TWINS window sizes considered in our experimental campaign, we were able to build the oracle by selecting the window size that yielded the highest bandwidth in each scenario. For the static solution, a 125µs window is always used. This value was chosen as it increases performance for the highest number of scenarios [28].

Results, where performance was increased, are presented in Fig. 8 and are grouped by the number of I/O forwarding nodes and operation. The y-axis of each plot is on a different scale, and they all represent the number of patterns $\times 10^3$. All approaches to detect the access pattern at runtime were able to perform better than the static solution, for all scenarios. Table VI summarizes the differences with higher precision.

For read operations, the decision tree was able to improve performance for all the situations where the oracle did when 1, 2, and 4 I/O forwarding nodes are used, respectively. However, for 8 I/O nodes, it improved performance for 99.9% of the cases. That represents 9.7%, 6.5%, 11.0%, and 16.9% more than the static solution. The random forest approach increased performance by the same amount as the decision tree. The neural network behaved similarly, with minor differences when 4 and 8 I/O nodes were used. Considering write operations, all detection approaches are also comparable to the oracle, with 1 and 2 I/O nodes. With 4 and 8, the decision tree achieves 99.9% and 99.8% of the oracle, respectively. That represent 6.0%, 19.7%, 31.2%, and 22.0% more than using a fixed 125µs window size. In summary, using such detection approaches, we can increase performance on average by 17% over using a statically defined window size.

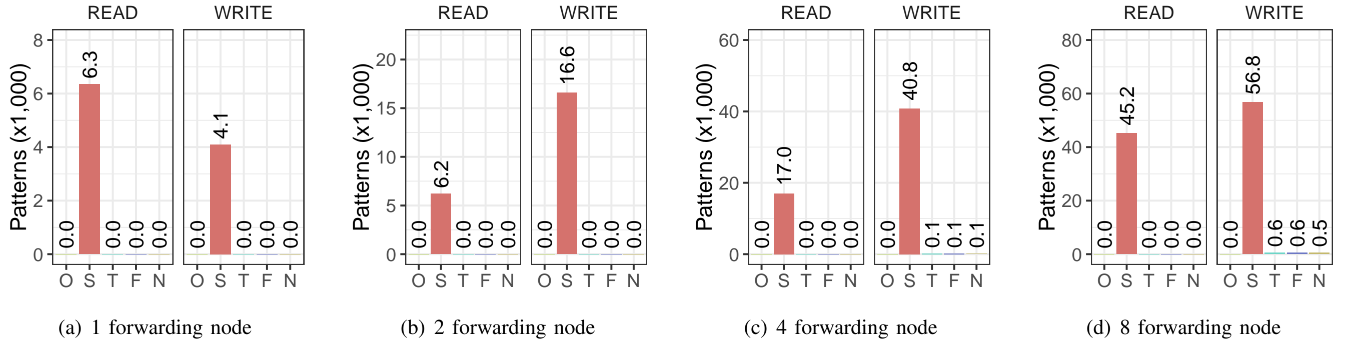
TABLE VI
NUMBER OF PATTERNS WHERE PERFORMANCE WAS INCREASED.

		1 ION	2 IONs	4 IONs	8 IONs
`READ`	**Oracle**	$71,672$	$100,653$	$170,833$	$311,322$
	Static	$65,326$	$94,438$	$153,824$	$266,126$
	Tree	$71,672$	$100,653$	$170,833$	$311,316$
	Forest	$71,672$	$100,653$	$170,833$	$311,316$
	Network	$71,672$	$100,653$	$170,830$	$311,310$
`WRITE`	**Oracle**	$71,677$	$100,653$	$170,834$	$311,314$
	Static	$67,575$	$84,063$	$130,038$	$254,552$
	Tree	$71,676$	$100,636$	$170,687$	$310,757$
	Forest	$71,676$	$100,637$	$170,691$	$310,756$
	Network	$71,676$	$100,644$	$170,727$	$310,782$



(a) 1 forwarding node (b) 2 forwarding nodes (c) 4 forwarding nodes (d) 8 forwarding nodes

Fig. 8. The number of patterns where performance was **increased** considering different policies to tune the I/O scheduler parameter. Results are grouped by the number of I/O nodes. The y-axis is not the same in all the plots. *O = Oracle, S = Static, T = Decision tree, F = Random forest, and N = Neural network.*



(a) 1 forwarding node (b) 2 forwarding node (c) 4 forwarding node (d) 8 forwarding node

Fig. 9. The number of patterns where performance was **decreased** considering different policies to tune the I/O scheduler parameter. Results are grouped by the number of I/O nodes. The y-axis is not the same in all the plots. *O = Oracle, S = Static, T = Decision tree, F = Random forest, and N = Neural network.*

Fig. 9 depicts all the scenarios with performance decrease. Results are also grouped by the number of I/O forwarding nodes and operation. The y-axis of each plot is on a different scale, and they all represent the number of patterns multiplied by 10^3. The three detection approaches were able to outperform the static choice, for all scenarios, guiding the scheduler to avoid using windows sizes that would be harmful to the I/O performance. Table VII compares the results with higher precision to detail the differences.

TABLE VII
NUMBER OF PATTERNS WHERE PERFORMANCE WAS DECREASED.

		1 ION	2 IONs	4 IONs	8 IONs
READ	**Oracle**	0	0	0	0
	Static	6,346	6,215	17,009	45,196
	Tree	0	0	0	6
	Forest	0	0	0	6
	Network	0	0	3	12
WRITE	**Oracle**	0	0	0	0
	Static	4,102	16,590	40,796	56,762
	Tree	1	17	147	557
	Forest	1	16	143	558
	Network	1	9	107	532

For read operations, the decision tree was able to avoid performance decrease for all the situations where the oracle did when 1, 2, and 4 I/O forwarding nodes are used, respectively. Minor impacts were seen using 8 I/O nodes. The random forest and the neural network behaved similarly. Only the later presented some tiny differences when using 4 and 8 I/O nodes. For write operations, incorrect detection of the access pattern at runtime causes loss of performance for some patterns. Table VII summarizes these differences.

Nonetheless, all detection approaches were able to avoid most of those scenarios. With 4 and 8, the decision tree decreased performance for 0.36% and 0.98% of the patterns where the static solution did. Applying the random forest instead, similar behavior is observed 0.35 and 0.98%. Finally, with the neural network to make the detection, similar behavior is also observed with slight differences 0.26 and 0.93%.

VI. CONCLUSION

Different optimization techniques have been proposed to improve performance of I/O operations at many levels of the I/O stack. These techniques typically achieve their goals in situations they were designed to improve performance, but not for all possible scenarios. Furthermore, they often require fine-tuning of parameters to yield the best effects.

In this paper, we demonstrated the applicability of machine learning techniques to automatically detect the I/O access pattern of HPC applications at runtime. We investigated decision trees, random forests, and neural networks to classify runtime

metrics into common access patterns. Furthermore, to illustrate its applicability, we evaluated these strategies by tuning the window size of the TWINS scheduler at the forwarding layer.

Our results have shown that all detection approaches covered in this work can correctly detect the access pattern. The simplest approached, represented by the decision tree, presented 0.99 accuracy for the training, testing, and validation datasets. Therefore the most straightforward method is favored as it can be trained more quickly and be easily implemented as a series of if-else statements.

When applied to our study case – guiding the TWINS scheduler at the I/O forwarding layer to adapt its window size for the detected access pattern – the detection mechanisms achieved 99% of the performance of an Oracle solution. We also demonstrated improvements of approximately 17% on average over a statically defined window. Finally, by correctly identifying the access pattern at runtime, we were able to avoid performance drops.

The proposed approaches to detect the access pattern are not specific to tuning the TWINS window size parameter, as they can be applied in the context of other optimization techniques that also required runtime knowledge of the current I/O access pattern. Finally, all data, source-codes, and analysis conducted in this paper are available in the companion repository at `jeanbez.gitlab.io/sbac-pad/2019`.

Future work will focus on extending our detection approach to consider a scenario where mixed patterns are detected on I/O forwarding nodes shared by more than one application. Additionally, we plan on demonstrating the applicability of our technique in the context of other tuning mechanisms.

ACKNOWLEDGMENT

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. It has also received support from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Brazil; the LICIA International Laboratory; NCSA-Inria-ANL-BSC-JSC-Riken Joint-Laboratory on Extreme Scale Computing (JLESC); the Spanish Ministry of Science and Innovation under the TIN2015–65316 grant; and the Generalitat de Catalunya under contract 2014–SGR–1051. This research received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 800144. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (www.grid5000.fr).

This project is partially funded by the European Union.

REFERENCES

[1] Z. Wang, X. Shi, H. Jin, S. Wu, and Y. Chen, "Iteration based collective I/O strategy for Parallel I/O systems," in *CCGRID '14 Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2014, pp. 287–294.

[2] F. Tessier, V. Vishwanath, and E. Jeannot, "TAPIOCA: An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 70–80.

[3] G. Congiu, S. Narasimhamurthy, T. SuB, and A. Brinkmann, "Improving Collective I/O Performance Using Non-volatile Memory Devices," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, sep 2016, pp. 120–129.

[4] S. Kumar, R. Ross, M. E. Papkafa, J. Chen, V. Pascucci, A. Saha *et al.*, "Characterization and modeling of PIDX parallel I/O for performance optimization," in *SC '13 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM Press, 2013, pp. 1–12.

[5] F. Z. Boito, R. V. Kassick, P. O. A. Navaux, and Y. Denneulin, "Automatic I/O scheduling algorithm selection for parallel file systems," *Concurrency and Computation: Practice and Experience*, 2015.

[6] J. L. Bez, F. Z. Boito, L. M. Schnorr, P. O. A. Navaux, and J.-F. Méhaut, "TWINS: Server Access Coordination in the I/O Forwarding Layer," in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, March 2017, pp. 116–123.

[7] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces," in *FAST'14 Proceedings of USENIX conference on File and Storage Technologies*, 2014, pp. 213–228.

[8] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'IO: a grammar-based approach to spatial and temporal I/O patterns prediction," in *Proceedings of the Int. Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 623–634.

[9] H. Tang, X. Zou, J. Jenkins, D. A. Boyuka II, S. Ranshous, D. Kimpe, S. Klasky, and N. F. Samatova, "Improving Read Performance with Online Access Pattern Analysis and Prefetching," in *Euro-Par 2014*. Springer, 2014, pp. 246–257.

[10] R. Ge, X. Feng, and X. H. Sun, "SERA-IO: Integrating energy consciousness into parallel I/O middleware," in *CCGRID '12 Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2012, pp. 204–211.

[11] J. Liu, Y. Chen, and Y. Zhuang, "Hierarchical I/O scheduling for collective I/O," in *Proceedings of the 13th International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2013, pp. 211–218.

[12] Y. Lu, Y. Chen, R. Latham, and Y. Zhuang, "Revealing applications' access pattern in collective I/O for cache management," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. ACM Press, 2014, pp. 181–190.

[13] H. Song, Y. Yin, Y. Chen, and X.-H. Sun, "A cost-intelligent application-specific data layout scheme for parallel file systems," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. ACM, 2011, pp. 37–48.

[14] R. Bolze *et al.*, "Grid5000: A large scale and highly reconfigurable experimental grid testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[15] Los Alamos National Laboratory, "MPI-IO Test Bechmark," http://freshmeat.sourceforge.net/projects/mpiiotest, 2008.

[16] HPC IO Benchmark Repository, "IOR," github.com/hpc/ior, 2019.

[17] C. Spearman, "The Proof and Measurement of Association between Two Things," *The American Journ. of Psychology*, vol. 15, pp. 72–101, 1904.

[18] M. Kuhn and K. Johnson, *Applied Predictive Modeling*, ser. SpringerLink : Bücher. Springer New York, 2013.

[19] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[20] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001.

[21] D. R. Cutler, T. C. Edwards Jr., K. H. Beard, A. Cutler, K. T. Hess, J. Gibson, and J. J. Lawler, "Random forests for classification in ecology," *Ecology*, vol. 88, no. 11, pp. 2783–2792, 2007.

[22] C. Strobl, A.-L. Boulesteix, T. Kneib, T. Augustin, and A. Zeileis, "Conditional variable importance for random forests," *BMC Bioinformatics*, vol. 9, no. 1, p. 307, Jul 2008.

[23] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[24] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://tensorflow.org

[25] I.-K. Yeo and R. A. Johnson, "A New Family of Power Transformations to Improve Normality or Symmetry," *Biometrika*, vol. 87, no. 4, pp. 954–959, 2000.

[26] G. E. P. Box and D. R. Cox, "An analysis of transformations," *Journal of the Royal Statistical Society*, pp. 211–252, 1964.

[27] R. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. Douglas, and H. Sebastian Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *Nature*, vol. 405, pp. 947–51, 07 2000.

[28] J. L. Bez, "Evaluating I/O Scheduling Techniques at the Forwarding Layer and Coordinating Data Server Accesses," Master's thesis, PPGC - Federal University of Rio Grande do Sul, 2016.