Adaptive Request Scheduling for the I/O Forwarding Layer

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

Adaptive Request Scheduling for the I/O Forwarding Layer

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
{jean.bez, navaux}@inf.ufrgs.br
[2]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
francieli.zanon-boito@inria.fr
[3]Barcelona Supercomputing Center (BSC) — Barcelona, Spain
{ramon.nou, alberto.miranda, toni.cortes}@bsc.es
[4]Universitat Politècnica de Catalunya — Barcelona, Spain

***Abstract*—In this paper, we present an approach to adapt the HPC I/O forwarding layer to the application access patterns. I/O forwarding is a technique used in most supercomputers today to alleviate contention in the access to the shared storage infrastructure. Because of its location, between processing nodes and parallel file system servers, it has been used to propose optimization techniques such as request reordering, aggregation, and scheduling. Such techniques can usually provide good results only for some of the situations, or depend on the right choice of parameter values. Our case study for this work is the TWINS request scheduling algorithm, which aims at coordinating the access of intermediate I/O nodes to the data servers. Our approach uses a neural network to classify application access patterns, and a reinforcement learning technique to empower the scheduler to learn the best parameter values to each access pattern during its execution, without the need of a previous training phase. Our evaluation of the access pattern detection neural network shows average precision of 98% during write experiments, and minimum precision of 98% during reads. The latter is an important result as most performance improvements by TWINS were observed for read experiments. Furthermore, we demonstrate that our contextual bandit strategy is able to learn the best value for the window size, achieving approximately 75% of precision — 98% of the performance provided by the best window size — in the first hundreds of steps.**

***Index Terms*—I/O scheduling, I/O forwarding, Reinforcement Learning, Access Pattern Detection, Neural Networks, HPC**

I. Introduction

Scientific applications impose strong performance requirements to the High-Performance Computing (HPC) field to build knowledge and understanding of complex phenomena. These performance requirements justify the appearance of ever-increasing large-scale platforms, comprised of tens of thousands of nodes. Such massive platforms commonly depend on a shared storage infrastructure which is built over a dedicated set of nodes, powered by a Parallel File System (PFS) such as Lustre [1] or Panasas [2]. In these machines, if all the compute nodes were to access the same shared file system servers concurrently, the contention would compromise overall performance and cause interference [3], [4].

To alleviate this problem, the I/O forwarding technique [5] aims at reducing the number of clients concurrently accessing the file system servers. It defines a set of special *I/O nodes* that are in charge of receiving I/O requests from the processing nodes and forward them to the PFS. This technique is successfully used in most current supercomputers, such as Tianhe-2 [6], Titan [7] and Sequoia [8] (#4, #7, and #8, respectively, from the TOP500[1]). In this configuration, illustrated in Fig. 1, the number of I/O nodes is typically larger than the number of file system servers, and smaller than the number of processing nodes. Besides alleviating contention, the concept of I/O forwarding has the advantage of creating an additional layer between applications and file system, a layer that can be used to apply specific I/O optimizations such as request reordering, aggregation, and scheduling [4], [9].

These optimization techniques typically provide improvements for specific system configurations and application access patterns, but not for all of them. Moreover, they often depend on the right choice of parameters. This was demonstrated to be the case for request scheduling at different levels [10], [11]. In such situations, finding the most suitable configuration is elusive as it is often hard to predict the future I/O workload of the system. For these reasons, in this paper we propose a novel approach to allow a scheduling algorithm, at the I/O forwarding layer, to adapt itself based on the access pattern observed at this layer.

Our case study for this work is the TWINS scheduling algorithm [11], designed to decrease I/O contention by

[1]TOP 10 — June 2018 — https://www.top500.org/lists/2018/06/

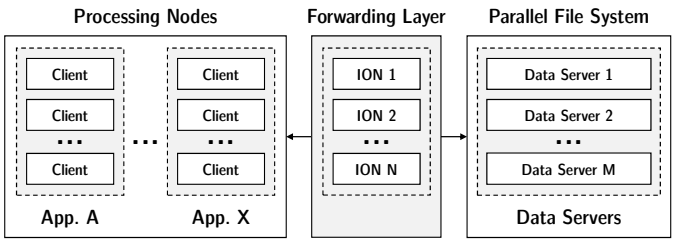


Fig. 1: The I/O forwarding layer

coordinating accesses from intermediate nodes to the PFS data servers. TWINS focuses on a single server during each time window of configurable duration. Achieving the best performance depends on adequately selecting the time window duration to fit the current situation. Therefore, in this paper, we propose a mechanism to adapt an I/O scheduler to different situations and demonstrate it applied to TWINS. In our proposal, a centralized council receives information from the I/O forwarding nodes and detects the current access pattern using a previously trained neural network. A reinforcement learning technique, contextual bandits [12], is used so that our system can learn what are the best choices during its lifetime.

By making the system capable of learning, we eliminate the need for a previous training step. That is important because designing and executing a set of tests that adequately represent the whole set of applications that will run in the supercomputer, and their diverse combinations of I/O characteristics, is both difficult and time-consuming. Moreover, the I/O forwarding software will be present during the whole lifetime of the machine. Hence any knowledge it obtains will be used to provide good results in the long term. Furthermore, the continuous learning process gives it the ability to adapt to changes in the system.

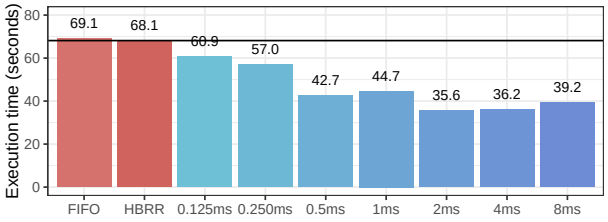
The rest of this paper is organized as follows. Section II describes TWINS, providing information required to understand this work. It also discusses some results that evidence the importance of choosing the right parameter value depending on the situation. Section III presents examples of applications executing in HPC machines and discusses the viability of learning and adapting in this context. Section IV presents our reinforcement learning approach to the problem of finding the best parameter value to each situation. Our approach to detect application access patterns with neural networks is detailed in Section V. The results obtained from this research are presented in Section VI. Finally, Section VII discusses related work and Section VIII details our conclusions and perspectives of future work.

II. TWINS: Server Access Coordination

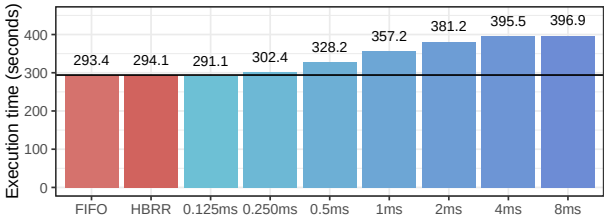
The main idea behind TWINS is to coordinate intermediate I/O nodes' accesses to the file system so that, at any given moment:

I. an I/O node directs its accesses to only one of the parallel file system data servers;
II. the different I/O nodes direct their accesses to different servers, so concurrency at each server is minimized.

TWINS achieves that by keeping multiple request queues, one per data server. During a configurable time window, requests are taken from only one of the queues (to access only one of the servers) according to a FIFO policy. When the time window ends, the scheduler moves to the next queue following a round robin scheme. Each I/O node applies a translation rule to server identifiers to ensure they are accessed at different times by different I/O nodes. Further details about this I/O scheduler can be found in [11].



(a) 1D-strided read through 8 I/O nodes



(b) Contiguous write through 2 I/O nodes

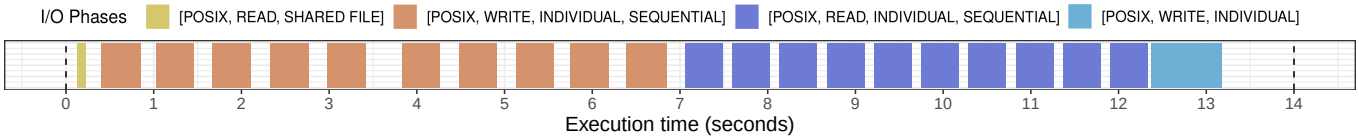
Fig. 2: The impact of the window size on performance: makespan (the smaller, the better) from experiments where 128 processes access a 4 GB shared file in 32 KB requests. Baseline algorithms are presented in red, with the horizontal line marking the best of them, and TWINS results are presented in blue.

We integrated TWINS into the IOFSL forwarding framework [3] using the AGIOS scheduling library [10]. Using AGIOS instead of implementing TWINS directly into IOFSL makes our solution more generic, as it can be used by other I/O forwarding frameworks or I/O services which use this library.

Previous performance evaluation compared TWINS to other schedulers provided by IOFSL (FIFO and HBRR — Handle-Based Round-Robin) and showed performance improvements for shared-file read workloads of up to 28% over the baseline (up to 50% compared to not using the I/O forwarding layer). For a multi-application scenario, TWINS decreases the interference factor. More details can be found in our previous work [11]. Despite the good results, TWINS' design means that if $server\ i$ is the current server being accessed but there are no requests to it, the scheduler will wait until either requests to $server\ i$ arrive or the time window ends, even if there are queued requests to other servers. For this reason, choosing an appropriate time window duration is critical to provide sustained good performance results.

Fig. 2 illustrates the benefits of correctly picking a window based on the access pattern, but also the overhead a wrong choice could introduce. In the situation presented in Fig. 2a, TWINS provides performance improvements between 10% and 48%, depending on the selected window duration. On the other hand, in the situation from Fig. 2b, the performance *decrease* of approximately 35% obtained with windows of 8 ms is avoided by using a smaller window. Therefore, to achieve the best performance with TWINS, we need to tune its time window duration to the current situation.

(a) OLAM application running in the Santos Dumont supercomputer



(b) Application `2201660091` (job `15335183665324813784`) running in the Intrepid supercomputer

Fig. 3: Two real-world executions in HPC systems

III. Motivation from Real-World HPC

As discussed in the previous section, TWINS improves performance over existing algorithms, but achieving its best results depends on selecting the best time window duration for the current situation. This means that TWINS needs to be capable of adapting to a changing workload. In this section, we discuss the viability of applying such a solution for a real-world HPC workload.

Fig. 3 illustrates the executions of two applications as reported by Darshan [13] traces. The x-axis represents the execution time, different colors represent different access patterns, and the boxes identify I/O phases. The duration of each phase is defined by the interval between the first and the last I/O operations from a sequence of operations with the same access pattern.

The first one, in Fig. 3a, is the Ocean-Land-Atmosphere Model (OLAM) [14], an important application executed in the Santos Dumont[2] supercomputer, at the National Laboratory for Scientific Computation, in Brazil. We selected a job from the year of 2017 that used 240 processes and ran for 2433 seconds, of which 221 seconds were spent on I/O operations. OLAM processes read time-dependent input files and write per-process logs of important events during most of the execution (the purple portions), and periodically write to a shared-file with MPI-IO (the yellow parts between purple phases).

The second application, in Fig. 3b, is anonymously identified as `2201660091`, job `15335183665324813784` from the Argonne Leadership Computing Facility I/O Data Repository [15]. This execution was randomly chosen from those who spent at least 30% of their time on I/O. We can see the application performs sequential writes to individual files in roughly the first half of the execution, and then it moves on to perform sequential reads from individual files.

A behavior present in both examples is that although application behavior changes during the execution, there are a few access patterns that are repeated multiple times and over an extensive period. The literature points resource-heavy applications tend to present a consistent I/O behavior, with patterns being repeated in future executions [16].

Therefore, one way of achieving our goal of adapting the TWINS window size parameter, without requiring previous information about the applications, would be to observe the current access pattern over some time. This information could then be combined with a selection strategy, such as decision trees [10], to decide on the appropriate value for the parameter. However, that decision tree would require prior training in order to cover most of the possible future executions. Considering I/O performance is sensitive to a large number of parameters, creating such a training set to represent all applications and executing it would be a time-consuming task. Therefore the desired approach should be able to learn the time window to use in each situation during the execution of applications, without disturbing them. The next section describes our proposal for such an approach.

IV. Adaptive Request Scheduling

As discussed in the previous section, we require a solution where the scheduler learns what is the best time window duration for different situations while they are being observed, but without a prior training process due its high cost and difficulty. Therefore we approach this as a Reinforcement Learning (RL) problem [12]. Among RL techniques, we could see it as a *k-armed bandit problem* [17], where at each step an agent takes one of the k possible actions (in our case, each action is a different duration for the time window) and receives a corresponding reward (performance). The expected reward from an action is called its *value* and denoted q_*. Since the value of an action a is *not* known beforehand, at the time step t we only have an estimate of it $Q_t(a)$, based on rewards obtained in the past whenever that action was taken. Using these estimates, the algorithm selects actions with the highest values (*exploitation*), but also sometimes chooses other actions in order to have better estimates of their values (*exploration*).

[2]https://www.top500.org/system/178568

The problem of seeing it as a simple k-armed bandit is that the policy to be learned (what is the best time window) changes when the access pattern changes, and then the learning process would need to be "restarted". That would mean having to try all possible actions a few times, to find good estimates of their values and converge on the best one. Considering that assessing the current situation has some cost, each step of this algorithm needs to be long enough so that the overhead does not surpass the obtained performance improvements. Looking at the example from Fig. 3b, we can see that using the described approach, changing the time window every second or so, and choosing among seven possible values for the time window would not lead to good results, as behaviors are stable only for a few seconds, and there is not enough time to both learn a policy and profit from it.

Therefore we model our problem as a *contextual bandit* (or *associative search* task) [12]: we actually have multiple concurrent "instances" of the k-armed bandit problem, one for each different situation (i.e., access pattern). At each step, only one of these instances will be active. This choice carries the assumption that taking an action does not have an impact on the next observed situation (access pattern). However, the selection of a time window value will impact the performance of the current I/O phase and can therefore slightly anticipate or delay the next I/O phase. Nonetheless, we consider this effect to be negligible, as the application primarily dictates the access pattern. Moreover, this possible effect is alleviated by the fact that we characterize the access pattern during an interval by looking at the majority of its accesses.

We implemented each of the concurrent armed bandit instances as an ϵ-*greedy* algorithm, that at step t takes the action a of the highest estimated value $Q_t(a)$, with probability $(1-\epsilon)$, or with probability ϵ takes a randomly selected action. Value estimates use *incrementally computed sample averages*, i.e., after obtaining this step's reward R, the estimate for a is updated as ($N(a)$ is the number of times a has been taken) :

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{N(a)}[R - Q_t(a)]$$

It is important to notice that since the proposed mechanism will run at the intermediate I/O nodes of an HPC architecture, its lifetime will not be constrained to the jobs execution time. Consequently, since what is learned is never lost, after observing an access pattern enough times it will be capable of consistently providing performance improvements for that situation by selecting the best time window duration.

A. Architecture of the proposed mechanism

The global coordination nature of TWINS, described in Section II, means that although request scheduling happens independently in the context of each I/O forwarding node, decisions about the window size should not. Hence we included a centralized agent called "council". Regarding possible the scalability issues of this decision, it is important to notice that only the I/O nodes will interact with the council, and not all the compute nodes. For instance, the Mira supercomputer (#21 in Top 500) has $49,152$ compute nodes and 384 I/O nodes (this gives a $1/128$ ratio). In this scenario, only the 384 nodes would be required to exchange messages with the council.

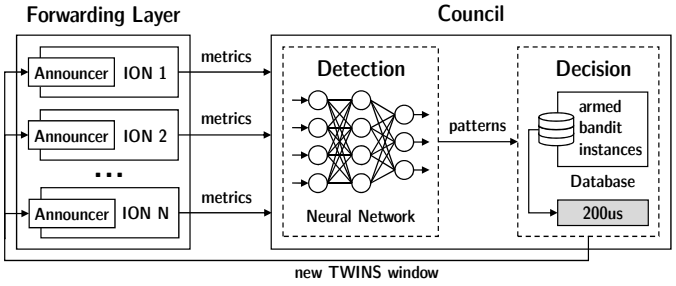


Fig. 4: The proposed architecture

In our approach, the council receives information from all I/O nodes, which are metrics (e.g., average request size, request arrival rate, etc.), and uses them to identify one of the armed bandit instances. It does so by using parameters we identified in a previous work [11] as impacting the window size choice: the number of file system servers being accessed, the number of I/O nodes being used, the number of clients and processes performing I/O operations, the type of operation, approach regarding files (shared file or file-per-process), and spatiality. The last two are not available as ready-to-use metrics; they need to be inferred from other metrics. Therefore, we feed some of the received data into a neural network to detect file approach and spatiality, as discussed in the next section. Additionally, if the received metrics do not carry enough information for detection, the decision for that I/O node is to do nothing (i.e., to keep using the current time window duration).

A single armed bandit instance will be identified *per I/O node*. It is possible that the council receives metrics that represent a mixed pattern, in a given I/O node, composed of read and write operations. In these cases, it will only consider the ones that represent the majority of accesses for the detection phase. Once the pattern is detected, and the correct armed bandit instance is selected and executed, an action is returned. This is done with the metrics received from each forwarding node. Therefore, this means that each I/O node will select a time window based on the most representative pattern it has observed. Consequently, it is possible that not all I/O nodes are in consensus, i.e., each one could have determined a different window size. In such scenarios, the council will act to ensure that the most common window size is chosen. In the end, the chosen value for the parameter is then sent to all I/O nodes. Investigating different policies to reach a consensus, or the co-existence of different values for the targeted parameter will be subject of future work.

V. Access Pattern Detection

In this work, we classify the access patterns by the operation, the number of files, spatiality, and request size. From these, the information regarding the number of files and spatiality is not readily available. Instead, they are derived

from the metrics reported by the I/O nodes. In this section, we describe our approach of using a Neural Network (NN) to classify the access patterns observed by each I/O node – regarding spatiality and number of files – into three distinct classes. These cover patterns that are common among scientific applications, and group situations that in our experience present similar behavior. Furthermore, this classification has the additional advantage of decreasing the number of armed bandit instances, hence facilitating the learning process. The three classes are:

- file-per-process with contiguous accesses;
- shared-file with 1D-strided accesses;
- shared-file with contiguous accesses.

The neural network receives as input information sent by each I/O node's announcer to the council. 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 [18] over our training dataset to identify the ones most related to the access pattern class. Figure 5 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. 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.

A. *Design of the Neural Network*

To build our access pattern detection neural network we used a dataset that consists of the input metrics obtained every second during the execution of several benchmarks, as detailed in Section VI-A. To eliminate potential noise from start-up

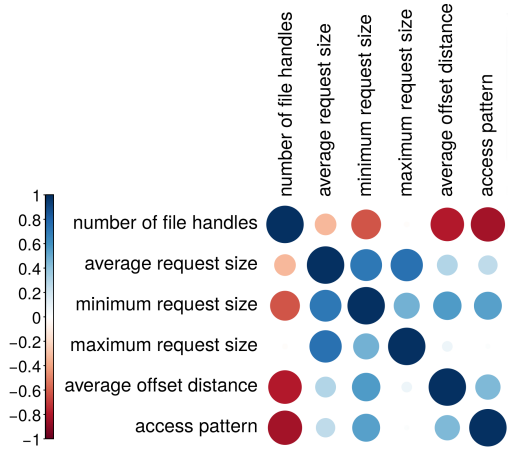


Fig. 5: Spearman's nonparametric correlation coefficient for the metrics selected to classify the access pattern into the three pre-defined classes. Positive correlations are displayed in blue and negative correlations in red. The color intensity and the size of the circle are proportional to the coefficients.

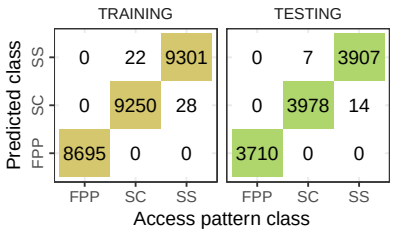


Fig. 6: Confusion matrices for the training and testing datasets. The x-axis shows the real class, and the y-axis shows what was detected by the NN. The classes are: file per process (FPP); shared file, contiguous (SC); and shared file, 1D strided (SS).

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 dataset contains 39 observations from each of the 1004 experiments, yielding a total of approximately 39 thousand observations.

Our classifier was built using Keras [19], a high-level Neural Network API, using TensorFlow [20] as backend. The dataset was split into two: 70% for training and 30% for testing. Before feeding our metrics to the NN, we applied Yeo-Johnson [21], 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 [22], 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 for each feature and subtracts it from each value.

Our model consists of three layers: an input layer containing the five features, a hidden layer with the same number of neurons, and an output layer with three units, one to represent each class. The first two layers use a Rectified Linear Unit (ReLU) [23] activation function with a normal kernel initialization function. The output layer uses the *softmax* function which squashes the outputs of each unit in the range $[0, 1]$ and ensures 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 $21,836$ samples and validated it on $5,460$ samples with a batch size of 32 and 50 epochs. The training accuracy was 99.82% with loss equal to 0.007634 and the validation accuracy was 99.76% with loss equal to 0.01165.

Fig. 6 shows the confusion matrix of the generated NN with the training and testing datasets. During training, our model correctly classified $27,246$ of the $27,296$ inputs, an accuracy of 99.81%. We also checked the performance of our model with our testing dataset (30% of the original data). During

testing, our model incorrectly classified only 21 samples out of the $11,616$ in the dataset. It is also important to notice all three classes are correctly identified with a reasonable probability.

In Section IV we presented a reinforcement learning approach aiming at *not* requiring a previous training phase. Hence the fact our proposal includes the access pattern detection NN described in this section, which needs to be trained, may seem counter-intuitive. However, it is important to notice that only detecting file approach and spatiality, in a single stream of requests, is a smaller problem and requires a much smaller dataset for training than what would be required to cover all possible I/O workloads in the whole system.

VI. Results and Discussion

In this Section, we present the results of our proposal. Section VI-A details our experimental methodology. In Section VI-B we conduct an offline evaluation of the council's ability to learn the best window size. We evaluate the accuracy of the access pattern detection mechanism in VI-C, and the overhead of our proposal in Section VI-D. Finally, in Section VI-E we discuss the implications of these results and limitations of our approach.

A. Experimental methodology

All experiments in this paper were carried out in clusters from the Grid'5000 [24] at Nancy: four PVFS2 servers in the Grimoire machine, 32 clients and multiple IOFSL nodes in separated Grisou nodes. Each of these nodes has two 8-core Intel Xeon E5-2630 v3 and 128 GB of RAM. A 558 GB hard disk was used for storage at the servers. Nodes are interconnected through a 10 Gbps Ethernet network, and there is a 10 Gbps link between the clusters. Both clusters were completely reserved during the experiments to minimize network interference.

PVFS version 2.8.2 was used with its default parameters, including 64 KB stripe size and striping through all four servers. Data servers were configured to perform I/O operations directly to their storage devices, bypassing buffer caches. That was done to avoid a situation where the scale of tests would hide the 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 and state machines. The maximum number of requests that can be aggregated from the dispatch queue (batch size) was 16. Minimum and maximum numbers of threads are four and 16, respectively.

The MPI-IO Test benchmarking tool [25] was executed to generate requests through the MPI-IO library. Using this tool, we cover the most usual access patterns in HPC by varying parameters: number of intermediate I/O nodes (1, 2, 4, or 8), number of processes (128, 256, or 512), shared-file or file-per-process, read or write, contiguous or 1D-strided access, and different request sizes (32 or 256 KB – smaller than the stripe size or larger enough so that all servers are accessed). In each experiment, 4 GB of data are accessed.

These 144 different situations (we do not consider the file-per-process 1D-strided combinations as this access pattern is not usual) were executed with the seven values considered in this analysis for the time window duration for a total of $1,008$ experiments. Metrics were collected from all I/O nodes every second during the executions. These metrics compose a dataset of over one million observations. A small part of this dataset ($\approx 39,000$ observations) was used in SectionV to train and evaluate the neural network. Nevertheless, the entire dataset is used in the next sections.

B. Offline evaluation of the learning aspect

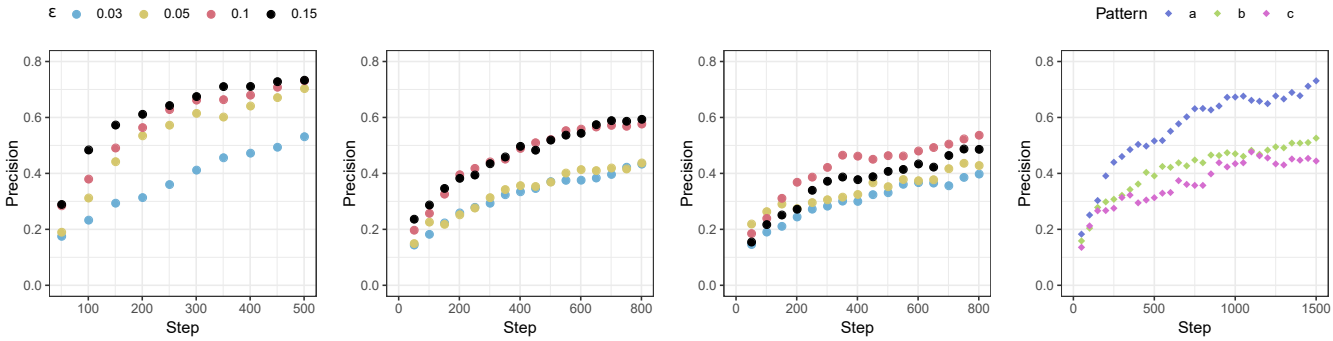
To evaluate the council's ability to learn the best parameter value without prior knowledge, we conducted a simulation of the ϵ-greedy approach described in Section IV, assuming perfect access pattern detection. The algorithm has seven possible actions (window size from 0.125 to 8 ms) and starts with value estimates of zero. After deciding on an action, to determine its reward (performance), it samples the dataset for real measurements obtained with that window size. We repeat each simulation 100 times to account for the variability in the sampling process and the performance measurements.

For each simulation, we summarize the precision of consecutive steps into bins (by averaging them) and then group the 100 simulations of each bin by also taking their average. The reason for grouping into bins is to allow the calculation of precision. Their size (50 steps) was chosen to facilitate the visualization of the general tendencies. To compute the precision, we compare the number of times the council correctly selected the best window size in each bin. Due to our previous extensive experimentation, we have the performance of each pattern and configuration when using each one of the tested window sizes. Therefore, we can compare the council's decision with the window that yielded the best I/O performance.

Fig. 7 shows the evolution of the precision metric (how often the correct window size is selected) throughout the simulations with different values for ϵ. The first three graphs represent simulations of a single armed bandit instance, and Fig. 7d simulates the same three instances concurrently. The simulation patterns are named:

- **Pattern *a*:** 128 processes read a shared file through 8 I/O nodes in 32 KB 1D-strided requests, in Fig. 7a. Performance for this pattern was shown in Fig. 2a.
- **Pattern *b*:** 128 processes write a shared file through 2 I/O nodes in 32 KB contiguous requests, in Fig. 7b. Performance was presented in Fig. 2b.
- **Pattern *c*:** 512 processes read a shared file through 8 I/O nodes in 32 KB contiguous requests, in Fig. 7c. For this pattern, all the tested windows yield similar performance.

We can see that as the algorithm reaches better estimates for performance with different window sizes, it improves by selecting the best value for the parameter more often (increasing precision). The smaller the ϵ (the probability of exploration), the slower the convergence is, as the algorithm may fall into local maxima and then take longer to find the



(a) Pattern *a* (b) Pattern *b* (c) Pattern *c* (d) Concurrent patterns, $\epsilon = 0.15$

Fig. 7: Precision (right selection rate) during simulations of the learning approach. The x-axis is different in each plot.

global maximum. On the other hand, in the long-term, an ϵ of 0.15, for instance, will choose the best action only at 85% of the times. Therefore a good option could be to start with a high value for ϵ and decrease it over time.

Normalizing performance obtained during the simulations by the average of all metrics with the best window size, when $\epsilon = 0.15$ the simulation of "pattern *a*" improves performance from 0.87 in the first bin to 0.98 in the last one, of "pattern *b*" from 0.89 to 0.95, and of "pattern *c*" from 0.95 to 0.98. The higher the performance difference between the window size options, the faster the algorithm's convergence. In many simulations of *pattern* c, for instance, due to the variability in the measurements, the algorithm converges to considering another window size duration as the best one. Nonetheless, in this case, selecting the best parameter value is not as important, as it only has a small impact on performance.

Finally, we can see in Fig. 7d that having concurrent armed bandit instances do not change their convergence, as they all evolve as before. The only difference is that they take longer to converge, as each of them is executing for only one-third of the simulation.

C. Evaluation of the access pattern detection

To evaluate the access pattern detection approach described in Section V, in this section we assume that if a pattern is correctly detected, the best window size is always selected.

We applied the neural network to each observation in the entire dataset of over one million entries, which contains the $\approx 39,000$ used in Section V, and used this detection to determine the best window size for that observation. Then we separated the observations by experiment (i.e., $1,008$ experiments) to calculate precision. Table I summarizes results separated by I/O operation. Although precision is low in some of the write experiments, the mean (97.99%), and median (100%) demonstrate that the model makes a correct detection in most of the cases. Furthermore, the minimum precision of 98.4% for read operations is an important result as most performance improvements by TWINS are observed for read access patterns [11].

TABLE I: Pattern detection results grouped by I/O operation.

Operation	Precision (%)			
	Min.	Mean	Median	Max.
Read	98.0	99.9	100.0	100.0
Write	53.8	97.9	100.0	100.0

Table II presents the same data but now grouped by access pattern (comparable to the ones in Fig. 6). File per process is the easiest pattern to detect and gives perfect scores. Lower precision was observed in some shared-file experiments, but they are not the rule, as indicated by high mean and median precision for these experiments.

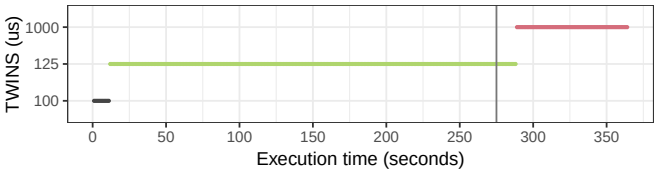
TABLE II: Pattern detection results grouped by access pattern.

Access Pattern	Precision (%)			
	Min.	Mean	Median	Max.
File per process	100.0	100.0	100.0	100.0
Shared file contigous	54.7	98.0	100.0	100.0
Shared file 1D strided	53.8	98.9	100.0	100.0

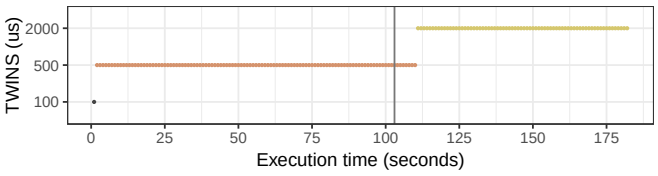
Fig. 8 presents three real (not simulated) benchmark executions using the council to select the best window size every one second. It is important to notice that these represent stable executions and patterns as they were generated by the benchmark. For this test, the council was previously informed of the best window size to each situation, and it does not use the learning portion. We can see the proposed neural network can detect the current access pattern (and hence select the best window size) during most of the executions. After the end of the write portions, the same window is kept for a few steps until the I/O nodes start to report metrics that allow for the detection of the read access pattern. Future work will extend our evaluation to real-world HPC applications with varying access patterns during the execution.

D. Overhead of the mechanism

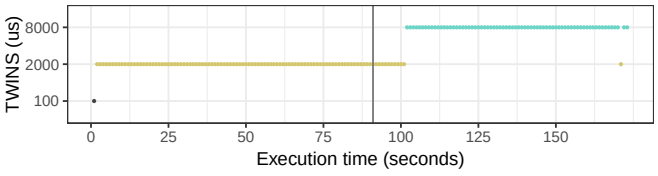
To analyze the overhead of our proposal, we selected the write portions of the experiments showed in Fig. 8 because



(a) 1 I/O node, file per process, contiguous, 256KB requests



(b) 8 I/O nodes, shared file, 1D-strided, 256KB requests



(c) 8 I/O nodes, shared file, contiguous, 256KB requests

Fig. 8: Selected window size when 128 processes write and then read 4 GB. Vertical lines indicate the first read requests.

they were executed with the correct window size for most of the time. We compare their execution time with the result previously obtained with that window size without the council. Therefore we can measure the overhead imposed by the approach itself, and not by using the wrong parameter value for part of the execution (although part of the reported decrease in performance is still due to the wrong window being used for some time). When 128 processes write 4 GB:

- to **independent files** with **contiguous** 256 KB requests through 1 I/O node (Fig. 8a), it takes 268.7 seconds using windows of $0.125ms$ and $279.7s$ with the council, an overhead of 4.0%;
- to a **shared file** with **1D-strided** 256 KB requests through 8 I/O nodes (Fig. 8b), it takes 107.8 seconds using windows of $5ms$ and $115.8s$ with the council, an overhead of 7.4%;
- to a **shared file** with **contiguous** 256 KB requests through 8 I/O nodes (Fig. 8c), it takes 106.5 seconds using windows of $2ms$ and $106.6s$ with the council, an overhead of less than 0.1%.

We conclude that our proposal does not impose a heavy overhead on applications. Moreover, we have shown in the previous sections that it is able to learn the best window for TWINS without prior knowledge and to detect the access pattern observed by the I/O nodes with high precision.

E. Discussion

All experiments in this paper consider the metrics being collected and a new window size being selected every one second. This choice was made because we have measured the time between the announcer starting to send metrics and it receiving the council's decision to be of tens of milliseconds (median of 27 ms), and it is important to give it enough time after changing the parameter to observe an impact on performance before reporting new metrics. On the other hand, it is desirable to make it as often as possible so shorter application I/O phases can still benefit from the adaptation. Finally, the time to reach a decision is not expected to directly affect the overhead because it happens asynchronously at the council node, not forcing the I/O nodes to wait for a decision. Once the later has a new value for the parameter that differs from the current one, then the new window size takes effect.

The continuous numerical parameter window size was represented as a set of different values to enable our armed bandit approach to be applied to this problem. We believe this is an appropriate strategy because when optimization techniques are proposed, they are typically experimented with a set of values for their parameters, as developers have an idea of what would be reasonable (for instance, for TWINS it is clear windows of several milliseconds or a few seconds would cause requests to starve). It is possible the true optimal values lie between classes, but reaching performance of the best among the observed classes is already an improvement over having no adaptation at all. Moreover, in Section VI-B we showed that having options with similar values slows down learning. Finally, the same approach described in this paper can be used with different sets of values for the parameter.

In this paper, the bandwidth observed by I/O nodes was used directly as the reward value to the armed bandit algorithm. The caveat of using bandwidth as the reward is that a low demand from the I/O subsystem (below its capacity) can result in a low bandwidth that is not related to the success of the optimization technique. Hence it brings noise to the learning process. This strategy is not a problem in our experiments because measurements to each situation are taken from the same experiments, i.e., with the same "intensity" of access. To mitigate this problem in practice, we could either consider additional metrics about the load to separate armed bandit instances (and this information is readily available at the I/O nodes) or apply some normalization to the bandwidth to account for this. Investigating these approaches is the subject of future work.

VII. Related Work

The I/O forwarding layer has been the focus of considerable research effort to improve its performance and transparently benefit the applications. Vishwanath et al. [4] improved I/O performance of an IBM Blue Gene/P supercomputer in up to 38% by improving this layer. Their modifications allowed for asynchronous operations in the I/O nodes and included a simple FIFO request scheduler to coordinate accesses from the multiple threads. The same authors later optimized data movement between layers through a topology-aware approach [26]. For the same architecture, Isaila et al. [27] proposed a two-level prefetching scheme.

Ohta et al. [9] implemented two request schedulers for the IOFSL framework: a FIFO and the quantum-based HBRR. The latter aims at reordering and aggregating requests to improve the access pattern. TWINS, on the other hand, was the first request scheduler for the forwarding layer that seeks to coordinate accesses to the data servers to avoid contention. Our previous work [11] showed it to improve performance and alleviate interference compared to FIFO and HBRR.

A. On the access pattern detection

Detecting access patterns is an important topic as it allows for 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 [10], [28], targeting patterns that are repeated 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.

At runtime, techniques can typically only use information from operations already performed. To predict future accesses, the technique proposed by Dorier et al. [29], Omnisc'IO, intercepts I/O operations and builds a grammar; the proposal by Tang et al. [30] periodically applies a rules library to recent accesses. These are client-side techniques, hence they would not work at server-side, where less information is available and the observed pattern is the interaction of multiple concurrent patterns. Runtime information at server-side is typically simple, like file system server load [31], while we needed an access pattern classification.

B. On the adaptation of the I/O subsystem

A number of parameters affect I/O performance, but tuning the system to each application requires a large number of experiments, being a difficult and time-consuming task. Research has been conducted aiming at facilitating the configuration of the I/O stack while keeping simplicity for users [32]–[34].

Building models to represent the impact of parameters is an usual strategy, as done by McLay et al. [33] to optimize MPI-IO collective writes to Lustre. Nonetheless, that requires prior investigation and is unavoidably specific at some extent to the system where this investigation was conducted, while we wanted a truly generic approach. The same disadvantage is present in proposals like the one by Boito et al. [10], where decision trees are used to select a scheduling algorithm.

Behzad et al. [32] proposed an auto-tuning system that intercepts HDF5 calls at runtime and applies a genetic algorithm to tune a set of parameters like Lustre stripe size and count, and MPI-IO collective nodes and buffer size. Their approach decreases the number of experiments to be executed to tune the parameters, but it does not eliminate the profiling phase.

Focusing on local storage, at block level, Nou et al. [35] used pattern matching to maintain a set of known patterns and their performance with different disk schedulers, using this information to adapt. This work is close to ours in the sense that it does not require prior knowledge. Nonetheless, the server-side file-level patterns we observe are more diverse, with concurrency, variability caused by the network, and more parameters. That means such a knowledge base would grow to a point where overhead, memory footprint, and slow convergence would make the approach unfeasible.

CAPES, the tuning system proposed by Li et al. [36], takes periodic measurements of a machine and train (online) a deep neural network that uses Q-learning to change parameters. They used it to select congestion window size and I/O rate limit, improving write performance by up to 45%. We also used neural networks, but for detecting access patterns, while they used it to represent the behavior of the whole system, with metrics from all levels. As the observed situations get more diverse, their training can take over 24 hours, while our system can learn and start to benefit a new access pattern after minutes of its start. Compared to their approach, we believe ours is the most adequate for a well-contained case where we know in advance what are the parameters that affect results, and actions can be represented by a small set of options.

VIII. Conclusion

Several optimization techniques have been proposed to improve performance at different levels of the I/O stack. These techniques typically achieve good results for the situations they were designed to improve, but not for all possible situations in a real HPC setting. Moreover, they often require fine-tuning of parameters to yield best results. In this paper, we focused on the I/O forwarding layer and proposed an approach to make it adapt to different access patterns. Our case study was TWINS, a request scheduler that provides performance improvements over other algorithms, but it is strongly dependant on selecting the right values for the window size parameter.

Our approach has a centralized council that periodically receives access pattern and performance metrics observed by the I/O nodes. From these metrics, it uses a neural network to classify the observed access pattern regarding the files approach and spatiality. A contextual bandit reinforcement learning strategy is used to learn what is the best window size to each situation during the execution. This mechanism has the advantage of not requiring previous training, which is a complex and time-consuming task.

Our results have shown the neural network is able to correctly detect the access pattern with an average precision of 98%. For read experiments, where TWINS was reported to provide the best performance improvements, the minimum observed precision was of 95%. Moreover, we have shown our learning strategy is capable of reaching a precision of $\approx 75\%$ (and achieve 98% of the best option's performance) in the first hundreds of observations of a given access pattern. Finally, the overhead imposed by our proposal is inferior to 8%.

The proposed approach is not specific to tuning the TWINS window size parameter, and it can be applied to other optimization techniques. Future work will focus on exploring the situations where diverging selections are made for different I/O nodes and the different policies to reach a consensus.

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. 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 (see https://www.grid5000.fr). The authors acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources of the SDumont supercomputer, which have contributed to the research results reported within this paper. URL: http://sdumont.lncc.br".

REFERENCES

[1] SUN, "High-performance storage architecture and scalable cluster file system," Sun Microsystems, Inc, Tech. Rep., 2007. [Online]. Available: http://www.csee.ogi.edu/~zak/cs506-pslc/lustrefilesystem.pdf

[2] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *Proceedings...*, ser. FAST'08, USENIX Conf. on File and Storage Technologies. USENIX Association, 2008, pp. 2:1–2:17.

[3] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O Forwarding Framework for High-Performance Computing Systems," in *Proceedings...*, IEEE International Conference on Cluster Computing and Workshops. IEEE, Aug. 2009, pp. 1–10.

[4] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. Ross, and K. Yoshii, "Accelerating I/O forwarding in IBM blue gene/p systems," in *Proceedings...*, ser. SC'10, 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, Nov. 2010, pp. 1–10.

[5] G. Almási, R. Bellofatto, J. Brunheroto, C. Caşcaval, J. G. Castanos, L. Ceze, P. Crumley, C. C. Erway, J. Gagliano, D. Lieber, X. Martorell, J. E. Moreira, and A. Sanomiya, "An overview of the Blue Gene/L system software organization," in *Proceedings...*, Euro-Par 2003 Conference, Lecture Notes in Computer Science. Springer-Verlag, 2003, pp. 543–555.

[6] W. Xu, Y. Lu, Q. Li, E. Zhou, Z. Song, Y. Dong, W. Zhang, D. Wei, X. Zhang, H. Chen, J. Xing, and Y. Yuan, "Hybrid hierarchy storage system in MilkyWay-2 supercomputer," *Frontiers of Computer Science*, vol. 8, no. 3, pp. 367–377, 2014.

[7] C. Zimmer, S. Gupta, and V. G. V. Larrea, "Finally, A Way to Measure Frontend I/O Performance," in *Proceedings...* Cray User Group Meeting, 2016.

[8] Prabhat and Q. Koziol, *High Performance Parallel I/O*, 1st ed. Chapman & Hall/CRC, 2014.

[9] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa, "Optimization Techniques at the I/O Forwarding Layer," in *Proceedings...* International Conference on Cluster Computing, 2010, pp. 312–321.

[10] 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. [Online]. Available: http://dx.doi.org/10.1002/cpe.3606

[11] 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.

[12] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," http://incompleteideas.net/book/the-book-2nd.html, 2017, accessed: August 2018.

[13] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access Through Continuous Characterization," *Trans. Storage*, vol. 7, no. 3, pp. 8:1–8:26, Oct. 2011.

[14] R. L. Walko and R. Avissar, "The Ocean–Land–Atmosphere Model (OLAM). Part I: Shallow-Water Tests," *Monthly Weather Review*, vol. 136, no. 11, pp. 4033–4044, 2008.

[15] P. H. Carns, "ALCF I/O Data Repository," http://ftp.mcs.anl.gov/pub/darshan/data/, 2013.

[16] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Server-side log data analytics for i/o workload characterization and coordination on large shared storage systems," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 819–829.

[17] D. A. Berry and B. Fristedt, "Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability)," *London: Chapman and Hall*, vol. 5, pp. 71–87, 1985.

[18] C. Spearman, "The Proof and Measurement of Association between Two Things," *The American Journal of Psychology*, vol. 15, no. 1, pp. 72–101, 1904. [Online]. Available: http://www.jstor.org/stable/1412159

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

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

[21] 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. [Online]. Available: http://www.jstor.org/stable/2673623

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

[23] 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.

[24] 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.

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

[26] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for I/O acceleration on blue gene/p supercomputing systems," in *Proceedings...*, ser. SC '11, 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011, pp. 19:1–19:11.

[27] F. Isaila, J. Garcia Blas, J. Carretero, R. Latham, and R. Ross, "Design and evaluation of multiple-level data staging for blue gene systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 6, pp. 946–959, 2011.

[28] 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 12th USENIX conference on File and Storage Technologies*. USENIX Association, 2014, pp. 213–228.

[29] 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 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 623–634.

[30] 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 – Parallel Processing*. Springer, 2014, pp. 246–257.

[31] B. Dong, X. Li, Q. Wu, L. Xiao, and L. Ruan, "A dynamic and adaptive load balancing strategy for parallel file system with large-scale I/O servers," *Journal of Parallel and Distributed Computing*, vol. 72, no. 10, pp. 1254–1268, 2012.

[32] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, R. Aydt, Q. Koziol, M. Snir *et al.*, "Taming parallel I/O complexity with auto-tuning," in *Proceedings...*, ser. SC '13. ACM, 2013, pp. 1–12.

[33] R. McLay, D. James, S. Liu, J. Cazes, and W. Barth, "A user-friendly approach for tuning parallel file operations," in *Proceedings...*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 229–236.

[34] F. Isaila, J. Carretero, and R. Ross, "CLARISSE: A middleware for data-staging coordination and control on large-scale HPC platforms," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 346–355.

[35] R. Nou, J. Giralt, and T. Cortes, "Automatic I/O scheduler selection through online workload analysis," in *9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing*. IEEE, 2012, pp. 431–438.

[36] Y. Li, O. Bel, K. Chang, E. L. Miller, and D. D. E. Long, "Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning," in *Supercomputing '17*, Nov. 2017.