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[4], and Philippe O. A. Navaux[1]

[1] Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS), Brazil
jean.bez@inf.ufrgs.br
[2] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
francieli.zanon-boito@inria.fr
[3] Barcelona Supercomputing Center, Spain {ramon.nou, alberto.miranda}@bsc.es
[4] Universitat Politècnica de Catalunya, Spain toni@ac.upc.edu

Abstract. In this paper, we present an approach to adapt the HPC I/O forwarding layer to the applications' access patterns. I/O optimization techniques can typically provide good results only for their target access patterns, or depend on the right choice of parameters. Our case study is the TWINS scheduling algorithm, where performance improvements depend on the right choice for the time window parameter. Our approach uses a neural network to classify access patterns, and a reinforcement learning technique to empower the scheduler to learn the best parameter values during its execution, without a previous training phase. Our evaluation of the neural network shows average precision of 98% for writes, and minimum precision of 98% for reads. Furthermore, we demonstrate that our contextual bandit strategy is able to learn the best window size, achieving approximately 88% of precision, and the performance provided by the best window, in hundreds of steps.

Keywords: I/O scheduling, I/O forwarding, Reinforcement Learning, Access Pattern Detection, Neural Networks

1 Introduction

Scientific applications impose performance requirements to the High-Performance Computing (HPC) field. These justify the appearance of ever-increasing large-scale platforms. Such massive platforms rely on a shared storage infrastructure which is built over a dedicated set of nodes, powered by a Parallel File System (PFS). In these machines, if all the compute nodes were to access the same file system servers concurrently, contention would compromise performance [1,18].

To alleviate this problem, the I/O forwarding technique [2] aims at reducing the number of clients concurrently accessing the file system servers. It defines a set of *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, Titan, and Sequoia (4th, 9th, and 10th, from the TOP500[5]). Besides alleviating contention, I/O forwarding creates

[5] TOP 10 — November 2018 — https://www.top500.org/lists/2018/06/

an additional layer between applications and file system, that can be used for optimizations such as request reordering, aggregation, and scheduling [15,18]. These 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 [5,6], where finding the most suitable configuration is elusive as it is hard to predict the future I/O workload of the system.

For these reasons, we propose a novel approach to allow the I/O forwarding layer to adapt itself based on the observed access pattern. Our case study is the TWINS scheduling algorithm [5], designed to decrease I/O contention by coordinating accesses from the I/O nodes to the servers. TWINS focuses on a single server during each window of configurable duration. Achieving the best performance depends on adequately selecting the window duration.

In our proposal, a centralized council receives information from the I/O nodes and detects the current access pattern using a neural network. A reinforcement learning technique, contextual bandits [16], is used so that the system can learn the best choices during its lifetime. Consequently, any knowledge it obtains will be used to provide good results in the long term. Furthermore, 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 represents 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. Finally, the continuous learning process enables it to adapt to system changes.

The rest of this paper is organized as follows. Section 2 describes TWINS, our case study. Section 3 discuss the viability of learning and adapting in HPC machines. Section 4 presents our reinforcement learning approach to find the best parameter for each situation. Our approach to detect access patterns using neural networks is detailed in Section 5. The results are presented in Section 6. Finally, Section 7 discusses related work and Section 8 concludes this paper.

2 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: (1) an I/O node directs its accesses to only one of the PFS data servers; (2) the different I/O nodes direct their accesses to different servers, so concurrency at each server is minimized.

TWINS achieves that by keeping one request queue per data server. During a time window, requests are taken from only one of the queues (to only one 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 access then at a different order.

We integrated TWINS into the IOFSL forwarding framework [1] using the AGIOS scheduling library [6]. Previous performance evaluation compared TWINS to the schedulers provided by IOFSL (FIFO and HBRR) and showed improvements for shared-file read workloads of up to 28% (up to 50% compared to not

using the I/O forwarding layer). For a multi-application scenario, TWINS also decreases interference. More details can be found in our previous work [5]. 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, to achieve the best performance with TWINS, it is critical to tune its time window duration, taking into account the system configuration and the applications' access patterns.

3 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 correct 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. 1 shows an execution of Ocean-Land-Atmosphere Model (OLAM) [19], executed in the Santos Dumont supercomputer, at the National Laboratory for Scientific Computation (LNCC), in Brazil. The x-axis represents the execution time, different colors represent the 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. We selected a job 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 (purple), and periodically write to a shared-file with MPI-IO (yellow).

It is possible to see that there are a few access patterns that are repeated multiple times over an extensive period. Resource-heavy applications tend to present a consistent I/O behavior, with patterns being repeated in future executions [12]. Therefore, one way 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 to decide on the appropriate value. Considering I/O performance is sensitive to a large number of parameters, creating a training set to represent all applications and executing it would be a time-

Fig. 1: OLAM execution at the Santos Dumont Supercomputer (LNCC)

consuming task. Therefore the desired approach should be able to learn the time window to use in each situation during the execution of applications.

4 Adaptive Request Scheduling

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 to its high cost and difficulty. Therefore we approach this as a Reinforcement Learning (RL) problem [16]. We could see it as a *k-armed bandit problem* [4], 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*).

We model our problem as a *contextual bandit* (or *associative search* task) [16]: we have multiple concurrent "instances" of the k-armed bandit problem, one for each different 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. 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): $Q_{t+1}(a) = Q_t(a) + \frac{1}{N(a)}[R - Q_t(a)]$.

Since the proposed mechanism will run at the intermediate I/O nodes its lifetime will **not** be constrained to the jobs' execution time. Consequently, after observing an access pattern enough times it will be capable of consistently providing performance improvements by selecting the best time window duration.

4.1 Architecture of the proposed mechanism

The global coordination nature of TWINS, described in Section 2, 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*. This should not pose scalability issues as

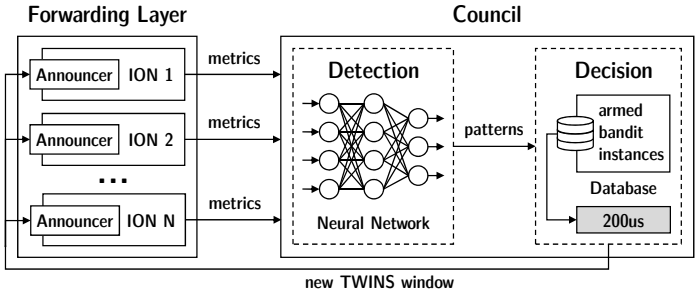


Fig. 2: The proposed architecture

only the I/O nodes will interact with the council. To give an idea, the Mira supercomputer (21st on Top500) has 49,152 compute nodes and 384 I/O nodes.

In our approach, the council receives metrics from all I/O nodes and uses them to identify one of the armed bandit instances. It uses parameters we previously identified [5] as impacting the window choice: the number of PFS servers, I/O nodes, clients, and processes the type of operation; approach regarding files (shared file or file-per-process); and spatiality. The last two are 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.

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. In these cases, it will only consider the ones that represent the majority of accesses. Once the pattern is detected, and the correct armed bandit instance is selected and executed, an action is returned. Different time windows could be selected for different I/O nodes, and then the council will choose the best for the majority, and send this value to all I/O nodes. Investigating different consensus policies, and the co-existence of divergent values will be subject of future work.

5 Access Pattern Detection

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. Additionally, these patterns are accepted by the HPC I/O community to represent common observed behaviors and used by *de facto* standard benchmarks such as IOR and MPI-IO Test. Furthermore, this classification has the additional advantage of decreasing the number of armed bandit instances, hence facilitating the learning process. The three classes are: (1) *file-per-process with contiguous accesses*; (2) *shared-file with 1D-strided accesses*; and (3) *shared-file with contiguous accesses*.

The NN 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 (*max.*, *min.*, *avg.*), and the average offset distance between consecutive requests to the same file handle. These parameters were selected by calculating the Spearman's nonparametric correlation to identify the ones most related to the access pattern class.

5.1 Design of the Neural Network

To build the NN we used a dataset of metrics obtained every second during the execution of several benchmarks, as detailed in Section 6. To eliminate potential noise from start-up and tear-down phases we extracted the same number of observations from the center of each test. This avoids bias toward one of the classes. The final data set contains ≈ 39,000 observations from 1,008 scenarios.

Our classifier was built using Keras[6] with TensorFlow[7] as backend, using 70% of the data set for training and 30% for testing. We applied Yeo-Johnson, scale, and center data transformations to improve results and to speed up training.

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 for each class. The first two layers use a Rectified Linear Unit (ReLU) activation function with a normal kernel initialization function. The output layer uses *softmax*. 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*. We trained our model on 21,836 samples and validated it on 5,460 with a batch size of 32 and 50 epochs. The training accuracy was 99.82% and the validation accuracy was 99.76%.

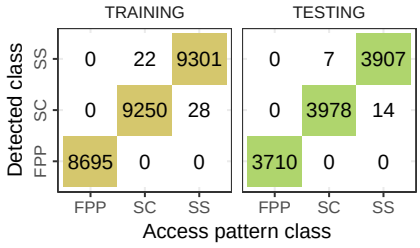


Fig. 3: Confusion matrices: *file per process* (FPP); *shared file, contiguous* (SC); *shared file, 1D strided* (SS).

Fig. 3 shows the confusion matrices. During training, our model correctly classified 27,246 of the 27,296 inputs (99.81%). With the testing data set, it incorrectly classified only 21 samples out of the 11,616. In Section 4 we aimed at *not* requiring a previous training phase, and here use a previously trained model for access pattern detection. However, it is important to notice that detecting file approach and spatiality, in a single stream of requests, requires less training than to cover all possible I/O workloads in the whole system.

6 Results and Discussion

All experiments were carried out in Grid'5000[8] clusters at Nancy: four PVFS2 servers in Grimoire, 32 clients and multiple IOFSL nodes in separated Grisou

[6] `https://keras.io/` [7] `https://www.tensorflow.org/` [8] `https://www.grid5000.fr/`

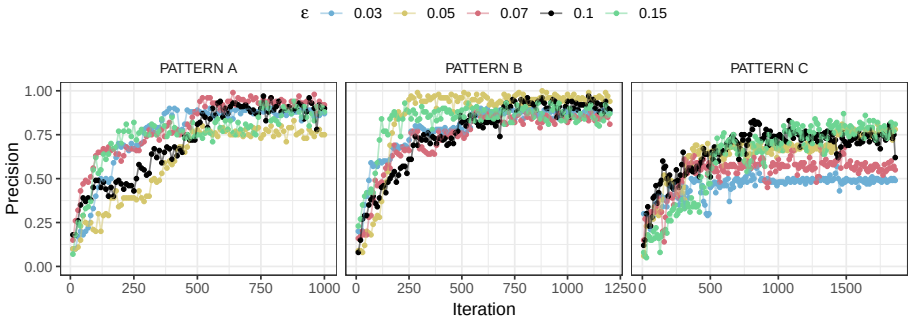


Fig. 4: Precision during simulations. The x-axis is different in each plot.

nodes. Each node has two 8-core Intel Xeon E5-2630 v3 and 128 GB of RAM. A 558 GB hard disk was used at the servers. Nodes have a 10 Gbps Ethernet interconnection, and there is a 10 Gbps link between the clusters.

PVFS version 2.8.2 was used with 64 KB stripe size. 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.

The MPI-IO Test benchmarking tool was executed using the MPI-IO library. We cover the most usual access patterns in HPC by varying parameters: number of 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, 32 or 256 KB requests (smaller than the stripe size or larger enough so that all servers are accessed). In each experiment a total of 4 GB of data are accessed.

These 144 different situations (we excluded the unusual 1D-strided file-per-process) were executed with seven different values for time window, for a total of $1,008$ experiments. Metrics were collected from all I/O nodes every second, composing a dataset of over one million observations. A small part of this dataset ($\approx 39,000$ observations) was used in Section 5 to train and evaluate the NN.

6.1 Offline evaluation of the reinforcement learning

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 4, 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 sampling variability. We group consecutive steps into bins to allow the calculation of precision, comparing the council decisions to what we know from the experiments results. We then summarize the 100 simulations of each bin by taking their average. The bin size of 10 was chosen to facilitate the visualization of the tendencies.

Fig. 4 shows the evolution of precision (how often the correct value is selected) for simulations with different ϵ, showing single armed bandit instances, and Fig. 5 simulates three instances concurrently. The patterns are: **(A)** 128 processes read a shared file through 8 I/O nodes in 32 KB 1D-strided requests; **(B)** 128 processes write a shared file through 2 I/O nodes in 32 KB contiguous requests; and **(C)** 512 processes read a shared file through 8 I/O nodes in 32 KB contiguous requests.

As the algorithm reaches better estimates for performance with different window sizes, it selects the best value for the parameter more often (increasing precision). The smaller the ϵ (probability of exploration), the slower the convergence is, as the algorithm may fall into local maxima. 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 we could start with a high ϵ and decrease it over time.

Considering performance normalized by the average of all observations for the best window, when $\epsilon = 0.15$ performance of **A** is improved from 0.87, in the first bin, to 1.00, in the last one; from 0.79 to 0.99 for **B**; from 0.95 to 0.97 for **C**. The higher the impact of window in performance, the faster the convergence.

Finally, in Fig. 5, having concurrent armed bandit instances do not change their convergence, as they all evolve as before. The difference is that they take longer to converge, as each one is executing for only one-third of the simulation.

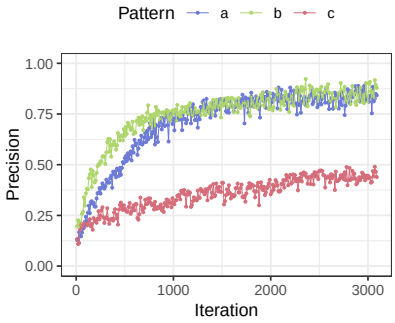


Fig. 5: Precision during simulations of concurrent patterns, $\epsilon = 0.10$

6.2 Evaluation of the access pattern detection

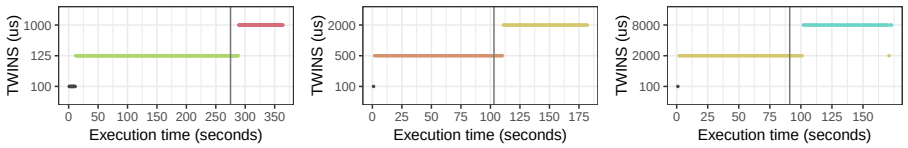
To evaluate the access pattern detection approach described in Section 5, in this section, we assume that if a pattern is correctly detected, the best window size is always selected. We applied the NN to each of the over one million observations, and used each detection to determine the best window. Then we separated the observations by the 1,008 experiments to calculate precision. Table 1 summarizes results by I/O operation. Despite the low precision in some write experiments, the mean (97.99%), and median (100%) show a correct detection in most of the cases. Furthermore, the minimum precision of 98.4% for reads is an important result as most performance improvements by TWINS are observed for reads [5].

Table 1: Precision (%) by operation.

	Min.	Mean	Median	Max.
READ	98.0	99.9	100.0	100.0
WRITE	53.8	97.9	100.0	100.0
Total	53.8	98.9	100.0	100.0

Table 2: Precision (%) by access pattern.

	Min.	Mean	Median	Max.
FPP	100.0	100.0	100.0	100.0
SFC	54.7	98.0	100.0	100.0
SFS	53.8	98.9	100.0	100.0



(a) (b) (c)

Fig. 7: Selected window when 128 processes write and then read 4 GB. Vertical lines indicate the first read request. (7a) 1 I/O node, file per process, contiguous, 256KB requests; (7b) 8 I/O nodes, shared file, 1D-strided, 256KB requests; and (7c) 8 I/O nodes, shared file, contiguous, 256KB requests.

Table 2 presents the same data grouped by access pattern (comparable to the ones in Fig. 3). The *file per process* (FPP) is the easiest pattern to detect and gives perfect scores. Lower precision was observed in some *shared-file* scenarios with *contiguous* (SFC) and *1D-strided* (SFS) requests, but they are not the rule, as indicated by high mean and median precision for these experiments.

6.3 Overhead of the mechanism

To determine the overhead of our proposal, we repeated all 144 tests using the neural network and bandit mechanisms, but ignoring their decisions at the I/O nodes. Hence we can measure the overhead without the impact of the window on performance. The minimum observed overhead was of 0.83%, the median, 1.77%, and the maximum of 32.29%. The latter is an outlier when a single I/O node is used. We conclude that our proposal imposes a low overhead (median $< 2\%$). Moreover, we have shown that it can learn the best window for TWINS without prior knowledge and detect the access pattern with high precision.

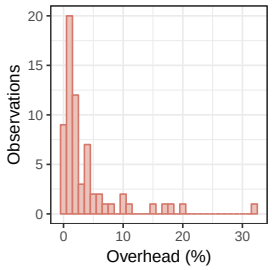


Fig. 6: The overhead imposed by the council.

6.4 Online evaluation of the council and discussion

Fig. 7 presents three real (not simulated) benchmark executions using the council to select the best window size every one second. The council was previously informed of the best value for each situation. We can see the proposed NN can detect the current pattern. The same window is kept for a few steps after the write portion, until the reported metrics allow to detect the read pattern.

All experiments in this paper consider council decisions every second. We measured the total time took by the announcer to send metrics and receive the decision to be of tens of milliseconds (median of 27ms). It is important to give enough time after changing the parameter to observe an impact on performance. On the other hand, it is desirable to make it as often as possible so shorter I/O

phases can still benefit. Finally, the time to reach a decision is not expected to directly affect the overhead as it happens asynchronously at the council node.

The bandwidth observed by I/O nodes was used as the reward to the bandit. The caveat is that a low demand can result in a low bandwidth that is not related to the success of the optimization. Hence, it brings noise to the learning process. This strategy is not a problem in our experiments because measurements have the same "intensity" of access. To mitigate this problem in practice, we could either consider load metrics or apply some bandwidth normalization. Investigating these approaches is the subject of future work.

7 Related Work

Research has focused on improving the I/O forwarding layer. Vishwanath et al. [18] improved I/O performance of an IBM Blue Gene/P supercomputer by up to 38% by improving this layer, whereas Isaila et al. [9] proposed a two-level prefetching scheme. Ohta et al. [15] implemented a FIFO and the quantum-based HBRR request schedulers for the IOFSL framework. The latter aims at reordering and aggregating requests. TWINS, on the other hand, was the first to aim at coordinating accesses to the data servers to avoid contention.

Detecting access patterns is important as it allows to adapt the I/O system to them. 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 [6,11], targeting patterns that are repeated with similar characteristics. We prefer a runtime technique to avoid the profiling effort and to benefit from similarities between different applications. At runtime, techniques can only use information from past operations. To predict future accesses, Dorier et al. [7] built a grammar, whereas Tang et al. [17] periodically applied a rules library to recent accesses. These client-side techniques would not work at server-side, where less information is available and the observed pattern is the interaction of multiple concurrent patterns.

A number of parameters affect I/O performance, and tuning the system requires a large number of experiments. Research has been conducted to facilitate the configuration of the I/O stack [3,8,13]. Building models to represent the impact of parameters is an usual strategy, as done by McLay et al. [13] to optimize MPI-IO collective writes to Lustre. Nonetheless, that is unavoidably specific at some extent to the tested system, while we wanted a truly generic approach. The same disadvantage is present in decision tree proposals like the one by Boito et al. [6]. Focusing on block-level local storage, Nou et al.[14] used pattern matching to record known patterns and their performance with different disk schedulers. However, our server-side patterns are diverse, with concurrency and variability caused by the network, i.e., the knowledge base would grow to a point where overhead, memory footprint, and slow convergence would make it unfeasible.

CAPES, the tuning system proposed by Li et al. [10], 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 the congestion window size and I/O

rate limit, improving write performance by up to 45%. Their training can take over 24 hours, while our system can learn and start to benefit from a new access pattern after minutes of its start. We believe our approach is the most adequate for a well-contained case where we know in advance what are the parameters that affect results, and actions that can be represented by a small set of options.

8 Conclusion

Many optimization techniques aim at improving performance at different levels of the I/O stack. These techniques achieve good results for their target situations, but not for all. Moreover, they often require fine-tuning of parameters. 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 improvements over other algorithms, but it is strongly dependant on selecting the right values for the window size parameter.

Our approach has a council that periodically receives access pattern and performance metrics observed by the I/O nodes. It uses a neural network to classify the pattern regarding the files approach and spatiality. A contextual bandit is then used to learn the best window to each pattern during execution. This mechanism does not require previous training, which is a complex and time-consuming task. Our results have shown that the neural network is able to correctly detect the access pattern with an average precision of 98%. For read experiments, the minimum observed precision was of 98%. Moreover, our learning strategy is capable of reaching a precision of $\approx 88\%$ (and achieve the best option's performance) in the first hundreds of observations of a given access pattern. Finally, the median overhead imposed by our proposal is inferior to 2%. All data, source-codes, and analysis conducted in this paper are available in the companion repository at: `https://jeanbez.gitlab.io/europar-2019`.

The proposed approach is not specific to tuning the TWINS window size parameter, and it can be applied to other scenarios. Future work will focus on extending our approach to other tunable parameters on the HPC I/O stack and exploring diverging decisions, and different policies to reach a consensus.

References

1. Ali, N., Carns, P., Iskra, K., Kimpe, D., Lang, S., Latham, R., et al.: Scalable I/O Forwarding Framework for High-Performance Computing Systems. In: Proceedings..., pp. 1–10. IEEE International Conference on Cluster Computing and Workshops, IEEE (2009). DOI 10.1109/CLUSTR.2009.5289188
2. Almási, G., Bellofatto, R., Brunheroto, J., Caşcaval, C., Castanos, J.G., Ceze, L., et al.: An overview of the Blue Gene/L system software organization. In: Proceedings..., pp. 543–555. Euro-Par 2003 Conference, Lecture Notes in Computer Science, Springer-Verlag (2003). DOI 10.1007/978-3-540-45209-6_79
3. Behzad, B., Luu, H.V.T., Huchette, J., Byna, S., Aydt, R., Koziol, Q., et al.: Taming parallel I/O complexity with auto-tuning. In: SC '13: Proceedings of the

International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, pp. 1–12. ACM (2013). DOI 10.1145/2503210.2503278
4. Berry, D.A., Fristedt, B.: Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability). London: Chapman and Hall **5**, 71–87 (1985)
5. Bez, J.L., Boito, F.Z., Schnorr, L.M., Navaux, P.O.A., Méhaut, J.F.: TWINS: Server Access Coordination in the I/O Forwarding Layer. In: 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pp. 116–123 (2017). DOI 10.1109/PDP.2017.61
6. Boito, F.Z., Kassick, R.V., Navaux, P.O.A., Denneulin, Y.: Automatic I/O scheduling algorithm selection for parallel file systems. Concurrency and Computation: Practice and Experience (2015)
7. Dorier, M., Ibrahim, S., Antoniu, G., Ross, R.: 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, pp. 623–634. IEEE Press (2014)
8. Isaila, F., Carretero, J., Ross, R.: CLARISSE: A middleware for data-staging coordination and control on large-scale HPC platforms. In: 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing), pp. 346–355 (2016)
9. Isaila, F., Garcia Blas, J., Carretero, J., Latham, R., Ross, R.: Design and evaluation of multiple-level data staging for blue gene systems. Parallel and Distributed Systems, IEEE Transactions on **22**(6), 946–959 (2011)
10. Li, Y., Bel, O., Chang, K., Miller, E.L., Long, D.D.E.: Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning. In: Supercomputing '17 (2017)
11. Liu, Y., Gunasekaran, R., Ma, X., Vazhkudai, S.S.: Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces. In: FAST'14 Proceedings of USENIX conference on File and Storage Technologies, pp. 213–228 (2014)
12. Liu, Y., Gunasekaran, R., Ma, X., Vazhkudai, S.S.: 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, pp. 819–829. IEEE (2016)
13. McLay, R., James, D., Liu, S., Cazes, J., Barth, W.: A user-friendly approach for tuning parallel file operations. In: Proceedings..., SC '14, pp. 229–236. IEEE Press, Piscataway, NJ, USA (2014)
14. Nou, R., Giralt, J., Cortes, T.: Automatic I/O scheduler selection through online workload analysis. In: 9th International Conference on Autonomic and Trusted Computing, pp. 431–438. IEEE (2012)
15. Ohta, K., Kimpe, D., Cope, J., Iskra, K., Ross, R., Ishikawa, Y.: Optimization Techniques at the I/O Forwarding Layer. In: Proceedings..., pp. 312–321. International Conference on Cluster Computing (2010). DOI 10.1109/CLUSTER.2010.36
16. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. `http://incompleteideas.net/book/the-book-2nd.html` (2017). Accessed: August 2018
17. Tang, H., Zou, X., Jenkins, J., Boyuka II, D.A., Ranshous, S., Kimpe, D., Klasky, S., Samatova, N.F.: Improving Read Performance with Online Access Pattern Analysis and Prefetching. In: Euro-Par 2014, pp. 246–257. Springer (2014)
18. Vishwanath, V., Hereld, M., Iskra, K., Kimpe, D., Morozov, V., Papka, M.E., et al.: Accelerating I/O forwarding in IBM Blue Gene/P systems. In: Proceedings..., SC'10, pp. 1–10. IEEE Computer Society (2010)
19. Walko, R.L., Avissar, R.: The Ocean–Land–Atmosphere Model (OLAM). Part I: Shallow-Water Tests. Monthly Weather Review **136**(11), 4033–4044 (2008)