On server-side file access pattern matching

Francieli Zanon Boito, Ramon Nou, Laércio Lima Pilla, Jean Luca Bez, Jean-François Méhaut, Toni Cortes, Philippe Navaux

On server-side file access pattern matching

Francieli Zanon Boito[1], Ramon Nou[2], Laércio Lima Pilla[3], Jean Luca Bez[4],
Jean-François Méhaut[1], Toni Cortes[2,5], Philippe O.A. Navaux[4]

[1]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP*, LIG, 38000 Grenoble, France
francieli.zanon-boito@inria.fr
[2]Barcelona Supercomputing Center, Spain
[3]LRI, Univ. Paris-Sud – CNRS, Orsay, France
[4]Institute of Informatics — Federal University of Rio Grande do Sul, Porto Alegre, Brazil
[5]Universitat Politècnica de Catalunya, Spain

***Abstract*—In this paper, we propose a pattern matching approach for server-side access pattern detection for the HPC I/O stack. More specifically, our proposal concerns file-level accesses, such as the ones made to I/O libraries, I/O nodes, and the parallel file system servers. The goal of this detection is to allow the system to adapt to the current workload. Compared to existing detection techniques, ours differ by working at run-time and on the server side, where detailed application information is not available since HPC I/O systems are stateless, and without relying on previous traces. We build a time series to represent accesses spatiality, and use a pattern matching algorithm, in addition to an heuristic, to compare it to known patterns. We detail our proposal and evaluate it with two case studies — situations where detecting the current access pattern is important to select the best scheduling algorithm or to tune a fixed algorithm parameter. We show our approach has good detection capabilities, with precision of up to 93% and recall of up to 99%, and discuss all design choices.**

***Index Terms*—high-performance computing, parallel I/O, parallel file systems, access pattern detection, pattern matching**

I. Introduction

High-performance computing (HPC) applications — such as weather and seismic simulations — rely on parallel file systems (PFS) to access persistent, shared data. These file systems are deployed over a set of dedicated servers and shared by all concurrent applications running in the HPC architecture.

The **performance** observed when accessing a PFS **depends on the access pattern**, i.e. on the way this access is performed: to large contiguous portions of the files or to small sparse portions, for instance. That affects performance at different levels of the parallel I/O stack: (i) the access to hard disks in the storage servers, (ii) the number of connections between clients and servers, (iii) the efficacy of caching and prefetching techniques at all levels, etc. For this reason, over the decades many techniques were proposed to adapt the application access patterns to improve performance [1]–[4]. They include techniques to perform aggregations, reorder requests, align them to the PFS stripe size, select the best aggregators for collective operations, and perform request scheduling.

These **optimization techniques typically provide improvements for specific system configurations and access patterns**, but not for all of them. Moreover, they often depend on the right choice of parameters, as demonstrated for request scheduling at different levels [5], [6]. In this situation, **finding ways of adapting the optimizations is key to achieving good performance**. Although the system configuration (numbers of nodes and of PFS servers, network topology, etc.) tends to be stable, the I/O workload changes as applications start and end their I/O phases. Hence systems capable of auto-tuning require a way of detecting access patterns for making decisions.

Extensive work has been dedicated into client-side access pattern detection [7]–[12]. However, the access pattern **at server-side** is actually the result of the interaction of multiple concurrent applications sharing the file system. Particularly, this makes it harder to list and represent all possible patterns, and even to predict what decisions the system should make to each of them (for instance with a decision tree [5]).

For this scenario, a tuning technique capable of unsupervised learning is more adequate, as we argue in a previous work [13]. Such a technique will try to learn what is the best decision to each access pattern while they are observed. Hence, **the detection of the access pattern directly affects the system's ability to make good decisions**. Indeed, each access pattern has to be observed multiple times before the system can learn, so learning is slower if too many redundant patterns are represented. Furthermore, if different patterns are considered by the system to be the same, that will introduce noise to the learning algorithm and prevent it from converging.

In this paper, we advocate the use of pattern matching for server-side file access pattern detection in the HPC I/O stack. The main goal is to enable the system to learn the access pattern classification while observing accesses. That simplifies the adaptation of the I/O stack, as there is no need to build a training set of benchmarks to cover all possible access patterns. We propose a pattern matching technique and evaluate it for two case studies: scheduling algorithm selection at PFS data servers and parameter tuning at the I/O forwarding layer. We analyze all choices and parameters involved in our proposal and discuss its use in other contexts.

The rest of this paper is organized as follows. Section II discusses related work on access pattern detection. Section III explains our case studies. Our pattern matching technique is

This research received funding from the European Unions Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 800144.

*Institute of Engineering Univ. Grenoble Alpes

detailed in Section IV. Our research methodology is explained in Section V, and results are discussed in Section VI. Section VII concludes the paper and points future perspectives.

II. Related work

Access pattern detection is important for techniques that try to adapt to the workload. One popular strategy is postmortem analysis — metrics are collected from applications during their execution, and used for future executions. Liu et al. [14] use server-side traces containing the system throughput measured every two seconds. By gathering multiple traces from different executions of the same application, they are able to filter the interference of other concurrent applications and determine its I/O requirements. In the approach proposed by Yin et al. [15], the MPI-IO library was modified to generate traces detailing information about each request. They are later used to guide data replication. He et al. [16] use the IOSIG tool to capture a trace from an application, and use it to optimize data placement in future executions.

Using traces makes sense for such techniques that work at application level, whereas server-side access pattern detection is our focus. Server-side traces include requests from multiple applications, which makes them harder to reproduce. Moreover, we want to benefit from similarities between applications. Another important difference to the discussed techniques is that we want to detect access patterns during run-time.

Dorier et al. [7] propose Omnisc'IO, which intercepts requests and builds a grammar to predict future accesses. The approach by Tang et al. [8] periodically analyzes past accesses and applies a rules library to predict future accesses (for prefetching). They collect spatiality of read requests from the MPI-IO library. It is usual for run-time techniques to gather information from I/O libraries. Ge et al. [9] collect information from MPI-IO related to operation, data size, spatiality, if operations are collective, and if operations are synchronous. Liu et al. [10] collect the number of processes, the number of aggregators, and binding between nodes and processes. Lu et al. [11] use the offsets accessed by each process during collective operations. The processes' access spatiality is used in the approach proposed by Song et al. [12].

The discussed techniques for run-time detection work at the client side, where information is more easily obtained from I/O libraries, applications, etc. Dong et al. [17] use a time series model to estimate file system server load. The approach by Zhang et al. [18] applies a "reuse distance", defined as the time difference between consecutive requests from the same application to the same server. In contrast, our approach accounts for more aspects in its characterization.

III. Case studies

This section shows **two cases where access pattern detection is key to improve the I/O stack's performance**.

Previously [5], we studied **request scheduling for PFS data servers**. From five policies (TO, TO-agg, aIOLi, MLF, and SJF), we showed that the best choice depends on the system and workload. The scheduling algorithms that provided the best performance *improvements* for some applications (up to 200%) were the same ones that resulted in the worse *decreases* (up to 70%) for others. Therefore, this situation calls for a dynamic policy selection that adapts to the current workload.

We explore the aforementioned case study in this paper, reproducing a similar scenario from the previous experiments, but now using a different cluster and parallel file system (OrangeFS instead of dNFSp). In this new scenario, aIOLi and MLF did not provide performance improvements because they are similar to the algorithm already used by OrangeFS, and thus were not considered. However, we included as an option the scheduling algorithm proposed by Song et al. [19]. With different policies, we observed performance *improvements* of up to 67% and *decreases* of up to 79%. All policies appear as the best choice for at least one of the tested applications.

The second case study is regarding parameter tuning for TWINS. This request scheduler [6] works in the I/O nodes — from the forwarding layer between processing nodes and the PFS — aiming at decreasing the data servers access contention. Results obtained with TWINS depend on the right choice of the *time window* parameter, and this choice depends on application access patterns. For instance, we observed performance *improvements* of up to 48% and *decreases* of up to 35%, depending on the selected window duration.

We have recently proposed a reinforcement learning technique to learn the best values for the parameter and adapt it at run-time [13]. We use multiple k-armed bandit instances, one per access pattern. In that work, we explicitly characterize the access patterns using aspects we previously observed to be relevant to the TWINS situation. Here, we want to empower the system to perform this classification, facilitating the use of this learning technique for different situations.

IV. File access pattern matching

We view the accesses (requests from the clients) to a server as a time series, and **we define a pattern as the sequence of requests that arrived in a slice of time**. Each request is represented in the time series as a number that represents its spatiality, and patterns can be compared (to be matched) using a pattern matching algorithm such as the dynamic time warping (DTW [20], or the approximate, linear-time DTW algorithm used in this work, FastDTW [21]).

It is important to emphasize here that we talk about server-side access pattern, which may be different from the application access pattern due to striping over multiple servers and to concurrent applications. Thus, we *cannot* simply leverage application-side information to characterize these accesses.

A similar pattern matching strategy was used with success in a previous work [22] to detect patterns of *disk block accesses*, where each access is represented by its logical block number. However, in our scenario requests are not to blocks but at **file level** — they request a portion of data of a given size, from a given file, starting at a given offset.

A. Building the time series

We represent each request in the time series as the absolute difference between its starting offset and the final

offset of the previous request (the *offset distance*). As a result, contiguous requests will add zeroes to the time series, and the higher the values, the less contiguous the accesses are.

When consecutive requests are not to the same file, the offset distance between them is not defined. As we are working at file level, we have no information about the placement of portions of data in the underlying storage. In this situation, we use an *infinite* value to represent this large distance between the requests, as illustrated in Eq. 1. If the offset distance between consecutive requests to the same file happens to be larger than the defined infinite value, it is replaced by this value.

$$dist(r^i, r^{i+1}) \stackrel{\text{def}}{=} \begin{cases} inf, & \text{if } r^i_{\text{file}} \neq r^{i+1}_{\text{file}} \\ \min(|r^{i+1}_{\text{start}} - r^i_{\text{end}}|, inf), & \text{otherwise} \end{cases} \quad (1)$$

B. Comparing patterns

Given a period of T seconds, **every T seconds the current time series is ended and then compared to previously observed patterns using Algorithm 1**. The times series of two patterns are compared using the FastDTW algorithm (line 4). It returns a distance between them: the higher the distance, the more *different* they are. We then convert this distance to a score between 0 and 1, where the higher the score, the more *similar* patterns are. This is done by normalizing the distance by the highest distance ever observed and inverting it (line 6). This score is compared to a defined *threshold* to decide if the patterns match or not (line 7). If no match is found, the new pattern is added to the collection of known patterns (line 12).

Therefore the system will learn to correctly detect matches over its execution as its maximum distance is updated by new comparisons (line 5). That also means the first comparisons could result in false negatives (due to the small maximum distance). As systems such as a PFS or an I/O forwarding framework are expected to run for years under a multitude of workloads, the maximum distance is expected to stabilize as soon as the system has observed non-contiguous access patterns. The impact of the first incorrect detections can be mitigated by re-comparing all known patterns *if the maximum observed distance has changed in the last period*. This operation can be performed asynchronously to minimize overhead.

The fact that we normalize distances by the largest ever observed one is another reason why the *infinite* value is used in our solution. If there is no bound to the offset distance between consecutive requests, there will be no bound to the calculated distances, and the system will *not* converge to a stable knowledge base. In fact, non-contiguous accesses to a very large file would generate very high values in the time series, which would then produce large calculated distances when comparing the time series to others. This would push the maximum distance and thus skew all scores to be large, generating false positives in future comparisons.

C. Coverage of access pattern aspects

Spatiality is the access pattern aspect most commonly considered because it has a deep impact on performance [23].

Algorithm 1: File access pattern matching

Input: p access pattern, P list of n previous patterns
Output: $match$ boolean, pos position of a match

1 $match \leftarrow false$; $maxscore \leftarrow 0$
2 **for** $i \leftarrow 1$ **to** n **do**
3 **if** $diff(p, P[i]) < maxdiff$ **then**
4 $dist \leftarrow fastDTW(p_{ts}, P[i]_{ts})$
5 $maxdist \leftarrow \max(maxdist, dist)$
6 $score \leftarrow 1 - \frac{dist}{maxdist}$
7 **if** $score > thres$ *and* $score > maxscore$ **then**
8 $match \leftarrow true$
9 $pos \leftarrow i$
10 $maxscore \leftarrow score$
11 **if** *not* $match$ **then**
12 $P[n+1] \leftarrow p$
13 $pos \leftarrow n+1$
14 **return** $match, pos$

In addition to spatiality, the offset distance also accounts for request size, another commonly considered aspect. The number of files affects the time series because accesses to multiple files will generate a highly noncontinuous pattern, which is why the number of files affects *data* access performance. Finally, the requests arrival rate affects the length of the time series as more requests will arrive in a fixed period of time.

From the aspects shown to impact performance in our case studies and in the literature, the only one that is not represented by the time series is the proportion of read and write requests. Thus **we represent patterns as a time series, plus the number of accessed files, and the number of read and write requests it represents**. When comparing two patterns p and q, their difference is computed by Eq. 2 by taking into account their number of files (p_f), and reads and writes (p_r and p_w, respectively). The FastDTW algorithm is applied to their time series **only if** their difference is smaller than a *maxdiff* tolerated difference percentage (line 3 in Algorithm 1). This heuristic has three advantages: (i) it accounts for the operation, (ii) the number of calls to the DTW algorithm is decreased, as a new pattern will only be compared to others of similar size and read/write proportion, and (iii) we avoid comparing patterns of very different lengths, what would result in very large distances and might skew our scores.

$$diff(p,q) = \max\Big(\frac{|p_{\text{f}} - q_{\text{f}}|}{\min(p_{\text{f}}, q_{\text{f}})}, \frac{|p_{\text{r}} - q_{\text{r}}|}{\min(p_{\text{r}}, q_{\text{r}})}, \frac{|p_{\text{w}} - q_{\text{w}}|}{\min(p_{\text{w}}, q_{\text{w}})}\Big) \quad (2)$$

D. Compression of patterns

Finally, if we consider the case of a PFS server in a large-scale HPC machine and a period T of a few seconds, the patterns could become very long on periods of high intensity (millions of requests). This means that the time series comparisons take longer, and that the pattern matching mechanism has a larger memory footprint. To alleviate these problems,

patterns can be compressed by a fixed factor. Incoming requests are inserted in a "bucket" until reaching its size limit (the compression factor), and then the bucket is added to the time series as a single value (the average of its requests' offset distances). In this process, the information of the pattern length is *not* lost, as we still keep the number of requests for the patterns. Furthermore, as it will be discussed in Section VI, using compression improves results as it makes patterns less sensitive to variations in the requests arrival order.

V. Methodology

We generated two data sets of server-side traces (one per case study) using the Grid'5000[1] testbed. Although there are available traces from real systems, such as the ALCF I/O Data Repository[2], they contain application-side aggregated I/O statistics, whereas our evaluation requires **server-side fine-grained traces containing information about each request**. Such traces are not typically captured and shared as they can represent intractable volumes of data.

A. Methodology for the generation of the traces

We generated multiple benchmarks with the MPI-IO Test benchmarking tool[3] to cover representative access patterns. They are the combination of parameters such as number of files, spatiality, request size, operation (read or write), etc. The data sets were generated in two experimental campaigns, using different clusters and configurations. The OrangeFS parallel file system[4] and the IOFSL forwarding framework[5] were in turn enriched with the AGIOS I/O scheduling library [5]. The library was used for its tracing capabilities. Each test was repeated multiple times (6 for server and 10 for I/O node traces), accessing different files. The whole set was executed in random order to minimize unexpected impacts, including cache in the read experiments. Write experiments are not affected by caching because we configured OrangeFS (and PVFS) to always sync data to files. All clusters were completely reserved to minimize network interference.

The **traces from PFS servers (first case study)** were generated in the Rennes site of Grid'5000. OrangeFS version 2.8.7 was deployed over four nodes from the Parasilo cluster, each powered with two eight-core Intel Xeon E5-2630 v3, and 128 GB of RAM. The PFS data was written to 600 GB SATA Seagate ST600MM0006 HDDs (one per server). 64 nodes from the Paravance cluster, with similar configuration to the Parasilo nodes, were used as clients for the file system. In each group of 32 nodes from Parasilo, each node is connected to a switch with 2×10Gbps Ethernet links. The two switches are connected to another switch with 2×40Gbps links each. The latter switch is connected to the Parasilo nodes with 2×10Gbps Ethernet links to each node. 28 benchmarks were generated by varying these parameters (except file-per-process non-contiguous and contiguous with 32 KB requests): write or read; shared-file or file-per-process; contiguous or 1D-strided; 32 KB, 4 MB, or 16 MB requests. In each experiment we executed two concurrent instances of the same benchmark, for a total of $64+64$ or $64+32$ processes. Each process accesses 128 MB through the POSIX interface.

The **traces from I/O nodes (second case study)** were generated in the Nancy site, with four PVFS 2.8.2 servers in the Grimoire cluster, and 32 clients plus 1 to 4 IOFSL nodes over separated Grisou nodes. These nodes have a similar configuration than the Parasilo and Paravance ones (with a ST600MM0088 HDD in each server). Nodes from both clusters are connected to a shared switch through 4×10Gbps Ethernet connections. These tests include 12 benchmarks: shared-file contiguous or 1D-strided and file-per-process contiguous; read or write; 32 or 256 KB requests. In each one, 128 processes access a total of 4 GB though MPI-IO.

B. Methodology for the evaluation of our proposal

We implemented our approach as described in Section IV and use it within a code for **offline evaluation**. This code feeds a random sequence of trace files (each trace's requests in order) to the pattern matching mechanism, which outputs its decisions regarding pattern matching. We parse the output and calculate **precision and recall** from the number of false and true positives and negatives, which are available in an offline evaluation, because we know from what benchmark traces were obtained. Still, our evaluation estimates these metrics by assuming all matches between patterns from the same application are correct, and matches between patterns from different applications are incorrect. Those are rather conservative assumptions, and **make our results pessimistic regarding the quality of the results**.

Previous research [13] indicate the feasibility of taking decisions and adapting the system every few seconds. Hence for this paper we chose to work with 1-second long patterns, to represent a real usage while keeping a large number of patterns in our data sets. The data sets used in this research, the pattern matching mechanism source code and all scripts used to generate and evaluate results are available at: https://gitlab.inria.fr/frzanonb/apmatching.

VI. Evaluation of the proposed technique

The server data set contains 9216 patterns, and the I/O nodes data set 130335. Since the patterns have a fixed duration, the number of patterns is *not* the same for each benchmark, as it depends on the execution time. The number of patterns representing different benchmark parameters are presented in Table I. Server patterns contain a median of 605 requests, and I/O nodes patterns a median of 123. In both cases, read patterns are longer: median of 734 for reads and 551 for writes in the data servers, 302 for reads and 75 for writes in the I/O

TABLE I: Representativity of access patterns in the data sets

	Data server	I/O node
Read vs Write	1779 vs 7437	29929 vs 100406
Shared-file vs File-per-process	5063 vs 4153	72802 vs 57533
Contiguous vs 1D-strided	5154 vs 4062	94997 vs 35338

[1]https://grid5000.fr/ [2]https://mcs.anl.gov/research/projects/darshan/data/
[3]http://freshmeat.sourceforge.net/projects/mpiiotest [4]https://orangefs.org/
[5]https://mcs.anl.gov/research/projects/iofsl/

nodes. In traces from I/O nodes, patterns become smaller as we increase the number of intermediate nodes (median of 242 with one, 123 with two, and 67 with four).

A. All-to-all comparisons

We started by **comparing all patterns among themselves, calculating the score, and inspecting the distribution of scores. Comparing these distributions tells us if it is possible to identify a threshold that allows to match most of the pairs of patterns that should match (true positives) without matching too many of the pairs that should not (false positives)**. Table II shows the results for a single I/O node. We can see there is such a clear separation.

TABLE II: Scores between patterns from **I/O node traces**, obtained with a **single I/O node** and without compression.

	Min.	1st Qu.	Median	3rd Qu.	Max.
No match	0	0.9029	0.9290	0.9736	1
Match	0.8116	0.9995	1	1	1

However, results without compression were not good for all access patterns. Table III details the distribution of scores between shared-file *read* patterns. Without compression, a threshold of 0.99 allows to capture $\approx$ 100% of the right matches for shared-file 1D-strided and file-per-process patterns, while incorrectly matching only from 13 to 15% of the patterns that should not match. However, there are no good thresholds for shared-file contiguous patterns without compression. With a compression factor of 10, the 0.99 threshold captures between 99 and 100% of the correct matches between *write* patterns, shared-file 1D-strided and file-per-process read patterns, and 49% of shared-file contiguous *read* patterns, while also capturing 19% of the incorrect matches. Increasing the compression factor to 100, the mechanism matches between 19 and 20% of the wrong matches, but is able to capture over 99% of the right matches for *all* patterns.

TABLE III: Scores for **read** patterns from **traces for a single I/O node**. *SF = shared-file, C = contiguous, S = 1D-strided.*

		Min.	1st Qu.	Median	3rd Qu.	Max.
		No compression				
No match		0	0.4061	0.8543	0.9511	1
Match	**SF, C**	0.8116	0.9173	0.9390	0.9896	0.9985
	SF, S	0.9935	0.9970	0.9981	0.9987	0.9998
		Compression of 10				
No match		0	0.5846	0.9091	0.9810	1
Match	**SF, C**	0.9673	0.9856	0.9898	0.9979	0.9995
	SF, S	0.9877	0.9991	0.9994	0.9996	1
		Compression of 100				
No match		0	0.5895	0.8992	0.9856	1
Match	**SF, C**	0.9506	0.9955	0.9969	0.9987	1
	SF, S	0.9882	0.9990	0.9995	0.9997	1

The fact that writes benefited from increasing the compression factor only up to 10 means **there is merit in the representation of patterns as time series of requests**. At the same time, read patterns, which are longer, benefited from a higher compression factor, indicating **compression is important to adequately represent patterns. That happens because multiple observations of the same pattern will have requests arriving in different orders, and compression mitigates this variation**. Compression was important mainly for the contiguous application access pattern, because that is the most non-contiguous pattern when viewed from the server-side. At a given moment, each process is accessing its own contiguous portion of the file, and therefore requests from different processes have a large distance. Hence these patterns are more sensitive to variations in the order of requests.

These results point write and read patterns should not be treated the same, as achieving the best results depends on using different compression factors and maximum observed distances. In practice, that can be handled by building two time series concurrently (with different compression factors) and, at the end of the pattern, keeping only one of them depending on the most common type of requests (read or write).

Table IV presents the scores between *data server traces*. Without compression, we can see there is no clear threshold that allows matching read patterns without also making most of incorrect matches. This is similar to what happened with I/O node traces, where compression improved results.

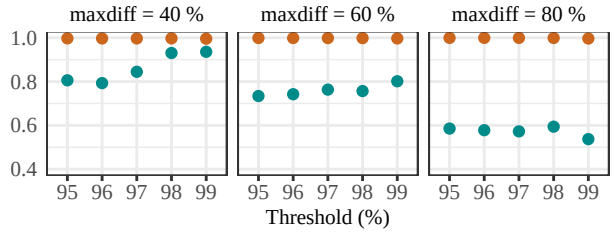
TABLE IV: Scores between patterns from **PFS server traces**

		Min.	1st Qu.	Median	3rd Qu.	Max.
		No compression				
No match		0	0.9673	0.9771	0.9976	1
Match	**Read**	0.0127	0.9647	0.9953	0.9990	1
	Write	0.8578	0.9835	1	1	1
		Compression of 10				
No match		0	0.8900	0.9683	0.9971	1
Match	**Read**	0.4859	0.9848	0.9987	0.9996	1
	Write	0.9391	0.9936	0.9999	1	1

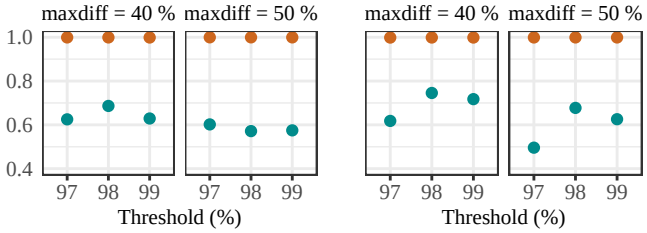
1) The impact of the infinite *value:* As discussed in Section IV-A, an "infinite" value is used to bound the offset distance, and for requests to different files (Eq. 1). The results presented so far were obtained using a limit of 10 GB. In Table V, the scores between read server patterns, we can see the 1D-strided **patterns are harder to match when the infinite value is low** (512 MB). When we increase it, we can match most of the 1D-strided patterns, with some loss in the quality of results for contiguous patterns. Increasing

TABLE V: Scores between **read** patterns from **server traces** with compression of 100. *SF = shared-file, FPP = file-per-process, C = contiguous, S = 1D-strided.*

		Min.	1st Qu.	Median	3rd Qu.	Max.
		infinite = 512 MB				
No match		0	0.6928	0.9608	0.9872	1
Match	**SF, C**	0.9392	0.9924	0.9962	0.9975	1
	SF, S	0.5000	0.9150	0.9742	0.9894	1
	FPP, C	0.9283	0.9986	0.9999	1	1
		infinite = 10 GB				
No match		0	0.9287	0.9558	0.9928	1
Match	**SF, C**	0.9649	0.9848	0.9975	0.9996	1
	SF, S	0.5095	0.9659	0.9969	0.9993	1
	FPP, C	0.9279	0.9987	0.9999	1	1
		infinite = 100 GB				
No match		0	0.9281	0.9641	0.9979	1
Match	**SF, C**	0.9667	0.9839	0.9997	1	1
	SF, S	0.5089	0.9659	0.9997	0.9999	1
	FPP, C	0.9280	0.9987	0.9999	1	1

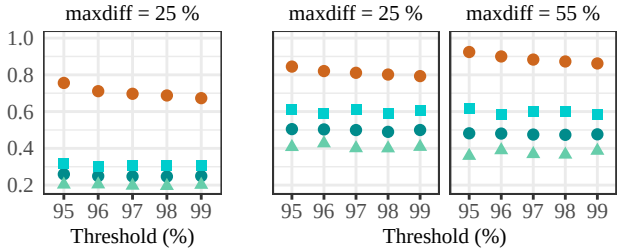


(a) Read experiments (compression of 100)

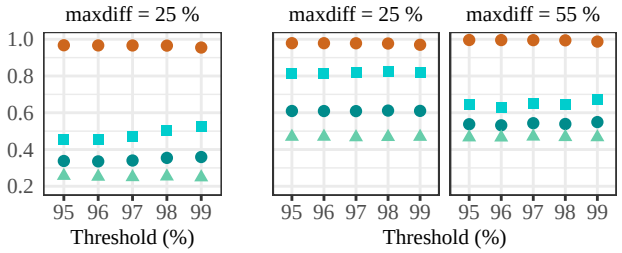


(b) Write (compression of 100) (c) Write (compression of 10)

Fig. 1: Precision (cyan, below) and recall (orange, above) for **2 I/O node traces**. *The y-axis do not start at zero.*



(a) Read (method 1) (b) Read (method 2)



(c) Write (method 1) (d) Write (method 2)

Fig. 2: Precision (cyan, dots, below) and recall (orange, dots, above) for **server traces**, with compression of 100. Squares show precision for 1D-strided, and triangles for contiguous. *The y-axis do not start at zero.*

it by a factor of ten (from 10 to 100 GB) does not affect results much, indicating that **as long as the infinite value is not too small, it does not have an important impact on results (the solution is robust)**. Here an infinite value that is *too small* is characterized by a reasonable offset distance for shared-file accesses. It is important to notice, however, **in practice shared-file and file-per-process patterns would not be compared because we only compare the time series when their numbers of accessed files are similar. Hence the actual results are expected to be better than what was observed in these all-to-all comparisons**.

B. Precision and recall from the offline analysis

In this section, we present the offline evaluation of our mechanism, as discussed in Section V-B. We set the maximum observed distance to be the one observed in the all-to-all comparisons in order to eliminate this factor from the evaluation. The offline evaluation was repeated ten times (to account for the random order of traces) and to each I/O node or server separately. We present the average precision and recall metrics.

Fig. 1 shows precision and recall from some of the experiments with 2 I/O node traces. For *all* I/O node traces, 40 was the best value for *maxdiff*, and in general increasing it harms precision, as comparisons between very different patterns are allowed. The best threshold was a high one (98 when using 2 I/O nodes and 99 with 1 or 4). Results for all write traces were better with compression of 10 than 100, reflecting what was observed in Section VI-A: we lose information when the whole pattern is reduced to a few values. For the same reason, results with *read* traces from 4 I/O nodes, which are shorter than with less intermediate nodes, were also better with compression of 10. Table VI summarizes these results.

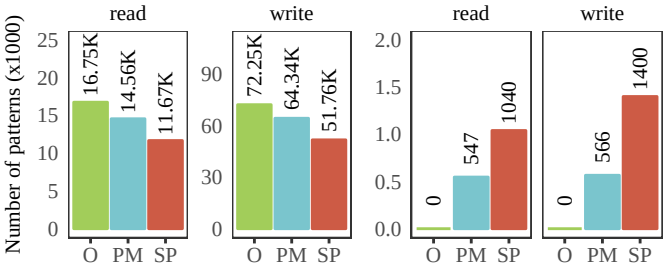
TABLE VI: Best results with **traces from I/O nodes**

	1 I/O node		2 I/O nodes		4 I/O nodes	
	Read	**Write**	**Read**	**Write**	**Read**	**Write**
Precision (%)	92.1	80.1	93	74.6	81.5	73.7
Recall (%)	99.7	99.9	99.7	99.9	98.8	99.8

Fig. 2 shows results for the **server traces**, with estimations of precision and recall obtained with two methods. This data set has executions of benchmarks with the same parameters for spatiality, number of files and request sizes, but with different numbers of processes (64+64 or 64+32 over 64 client nodes). In the first method, we assume patterns obtained with different numbers of processes should *never* match, and hence we obtained low precision. In "method 2", we do not consider the number of processes when evaluating the pattern matching mechanism, i.e. we assume patterns obtained with different numbers of processes should *always* match. The second method reports higher precision. In reality, the right method would be a combination of both where it is assumed *similar* patterns should match. **These results evidence the challenge in evaluating such an approach: it can be hard to say if patterns are similar or not**.

For all results with server traces, the best compression was 100, the best *maxdiff* 25%, and the best threshold 95%. Precision with these parameters was of 50.5% for reads and 61% for writes, and recall 84.5% for reads and 97.9% for writes. If *maxdiff* is increased to 55 for reads, precision decreases to 48.2% but recall increases to 95.8%. We can see in Fig. 2 that these values reflect in fact the representativity of access patterns in the data set, as precision for 1D-strided is always higher than for contiguous patterns.

In general, it was harder to match correctly in the PFS data servers, after going through the striping process, than in the intermediate I/O nodes. Although both precision and recall



(a) *Increased* performance (b) *Decreased* performance

Fig. 3: **Scheduling algorithm selection (first case study)**. *O = Oracle, PM = Pattern matching, SP = Static policy*.

are important, we believe recall is particularly crucial as a low recall would increase the size of the knowledge base. In the case of a learning algorithm being used to each context given by a known pattern, having too many contexts would slow down the learning process. **When designing such a technique, it is important to keep in mind the precision results and make it robust to eventual incorrect matches**.

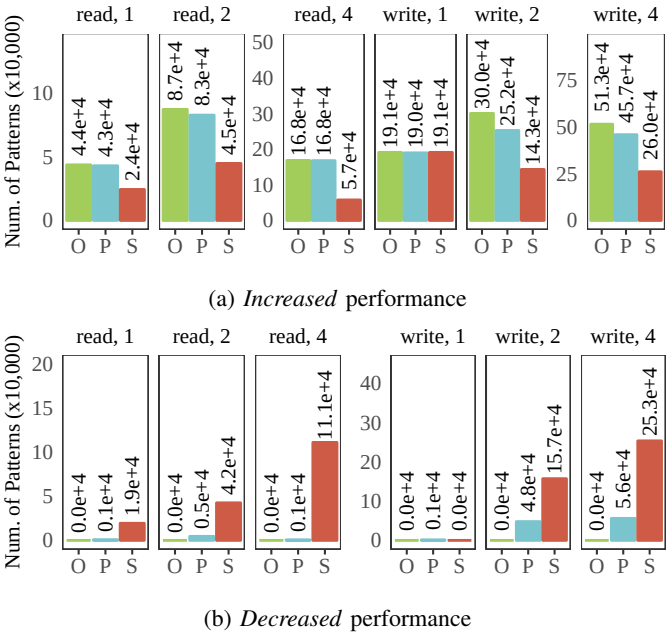
C. Request scheduling algorithm selection for PFS servers

Using the results from the offline evaluation with server traces, presented in Section VI-B, in this section we evaluate the use the of access pattern detection to select a scheduling algorithm for the parallel file system data servers. To evaluate that in an offline fashion, we parsed all experiments: to each new observed pattern A (every one second of accesses), matching A to a previous pattern B represents selecting the best known algorithm for the benchmark that originated B, and not finding a match results in using a fixed policy. Then we can estimate performance results with these selections by using previous measurements that we have for the different combinations of scheduling policies and benchmarks.

By comparing the performance results estimated in this way with a baseline — the performance with the OrangeFS original scheduler — we count the number of decisions (one per second, thus the number of seconds) that resulted in performance improvements or decreases, as presented in Fig. 3. From 92,160 observations (10 repetitions with all patterns), only the ones with performance differences superior to 5% are shown. We compare the results obtained with our pattern matching approach to an oracle that *always* makes the best selection, and the use of a fixed scheduling policy (the overall best among the ones considered). The oracle was able to improve performance for 96.6% of the observations. **The pattern matching approach improved performance for** 88.6% **of the patterns *where the oracle was able to improve it*,** 24.4% ***more* than the static solution. It decreased performance by making inadequate algorithm selections for** 1113 (1.2%) **patterns,** 54.4% ***less* than the static solution**.

D. Parameter tuning at the I/O forwarding layer

Similarly to Section VI-C, we used the pattern matching results, obtained with I/O node traces, to evaluate decisions on the time window duration. In this case, a fixed scheduling



(a) *Increased* performance

(b) *Decreased* performance

Fig. 4: **TWINS parameter tuning (2nd case study)**, with 1, 2 and 4 I/O nodes. *P = Pattern matching, S = Static window*.

policy (TWINS) is used, and the baseline for performance improvements was the use of 1 ms windows, which is a conservative value that decreases (and increases) performance for the least number of scenarios. We compare the pattern matching results with an oracle and a static solution, where 125 μs windows are always used. This value was chosen because it increases performance for the highest number of scenarios. Results are presented in Fig. 4. **The pattern matching approach was able to improve performance for** 91.6% **of the situations where the oracle was able to,** 66% ***more* than the static solution. It decreased performance for** 19% **of the patterns where the static solution did**.

E. Memory footprint and performance of the mechanism

The pattern matching mechanism uses $24N + 72$ bytes to keep each pattern of N requests. That means 23.5 KB per pattern of 1000 requests, or 22.9 MB per pattern of a million. Hence compression is important for the knowledge base size, specially considering it is to be kept in memory during the execution. It is also important for time, as a comparison between patterns of 1000 requests takes 479 μs (and of 100,000 requests, 46.3 ms, median of 20 repetitions). Reaching a decision fast is key to periodically adapt the system to a changing workload, and a number of comparisons have to be done each time. A strategy to limit the number of patterns (and thus the footprint *and* the time to compare) would be to include an occurrence counter per pattern. Then **rare patterns can be periodically discarded**, paying the price of not being able to adapt to these *rare* situations. Furthermore, comparisons to different patterns can be done in parallel.

VII. Conclusions and Future Work

In this paper, we have proposed a pattern matching approach for server-side file-level access pattern detection. Such a de-

tection is important to allow for the adaptation of optimization techniques on the I/O stack to the current workload, since such techniques often provide performance improvements only for some of the possible access patterns, or depend on parameters tuning. In this context, we considered two case studies where adapting to the workload is essential:: selecting the best scheduling algorithm for parallel file system data servers, and tuning a parameter in the I/O nodes from the forwarding layer.

Our approach periodically represents access patterns as a time series of offset distances, combined with the number of read and write requests, and the number of accessed files. New patterns are compared to previously observed patterns using a pattern matching algorithm (FastDTW). We evaluated our proposal using two large data sets of fine-grained traces we generated and made publicly available. Our results showed good matching capabilities, with precision of up to 93% and recall of up to 99%. When used to select the best scheduling algorithm, our mechanism was able to improve performance for 89% of the situations where it was possible to improve it, being up to 24% better than the best static algorithm selection. Used to select the best value for a parameter, it improved performance for 92% of the situations where it was possible, 66% better than the static alternative.

As future work, we plan to explore the idea of building a probabilistic chain between known patterns. It could be used to predict the next pattern and apply optimizations accordingly.

ACKNOWLEDGMENT

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). This research was conducted in the context of the NCSA-Inria-ANL-BSC-JSC-Riken Joint-Laboratory on Extreme Scale Computing (JLESC). It also received funding from the Spanish Ministry of Science and Innovation under the TIN2015–65316 grant; and the Generalitat de Catalunya under contract 2014–SGR–1051. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

This project is funded by the European Union.

REFERENCES

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

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

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

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

[5] F. Z. Boito, R. V. Kassick, P. O. A. Navaux, and Y. Denneulin, "Automatic I/O scheduling algorithm selection for parallel file systems," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 8, pp. 2457–2472, 2016.

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

[7] M. Dorier, S. Ibrahim, G. Antoniu, and R. B. Ross, "Omnisc'IO: A Grammar-Based Approach to Spatial and Temporal I/O Patterns Prediction," in *SC '14 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 623–634.

[8] H. Tang, X. Zou, J. Jenkins, D. A. Boyuka II, S. Ranshous, D. Kimpe *et al.*, "Improving Read Performance with Online Access Pattern Analysis and Prefetching," in *Euro-Par 2014 – Parallel Processing*, ser. Lecture Notes in Computer Science, F. Silva, I. Dutra, and V. S. Costa, Eds. Springer International Publishing, 2014, vol. 8632, pp. 246–257.

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

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

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

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

[13] J. L. Bez, F. Zanon Boito, R. Nou, A. Miranda, T. Cortes, and P. Navaux, "Adaptive Request Scheduling for the I/O Forwarding Layer," Feb. 2019, working paper or preprint. [Online]. Available: https://hal.inria.fr/hal-01994677

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

[15] Y. Yin, J. Li, J. He, X.-H. Sun, and R. Thakur, "Pattern-Direct and Layout-Aware Replication Scheme for Parallel I/O Systems," in *IPDPS '13 Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 345–356.

[16] S. He, X.-H. Sun, B. Feng, X. Huang, and K. Feng, "A cost-aware region-level data placement scheme for hybrid parallel I/O systems," in *CLUSTER '13 Proceedings of the 2013 IEEE International Conference on Cluster Computing*. IEEE, 2013, pp. 1–8.

[17] 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.

[18] X. Zhang, K. Davis, and S. Jiang, "IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination," in *SC '10 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[19] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-side I/O coordination for parallel file systems," in *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM Press, 2011, pp. 1–11.

[20] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series." in *KDD workshop*, vol. 10, no. 16. Seattle, WA, 1994, pp. 359–370.

[21] S. Salvador and P. Chan, "Toward accurate dynamic time warping in linear time and space," *Intelligent Data Analysis*, vol. 11, no. 5, pp. 561–580, 2007.

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

[23] F. Z. Boito, E. C. Inacio, J. L. Bez, P. O. A. Navaux, M. A. R. Dantas, and Y. Denneulin, "A Checkpoint of Research on Parallel I/O for High-Performance Computing," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 23:1–23:35, 2018.