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[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 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 applied optimizations to the current workload. Compared to existing detection techniques, ours differ by working at runtime 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's parameter. We show our approach has good detection capabilities, with precision of up to 80% and recall of up to 99%, and discuss all involved 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) — such as Lustre [1] or Panasas [2] — 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. 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 [3]–[7].

Institute of Engineering Univ. Grenoble Alpes

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. This was demonstrated to be the case for request scheduling at different levels [8], [9]. In this situation, finding ways of adapting the optimizations is key to achieve good performance. Although the system configuration (number 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, to then make decisions based on this detection.

Extensive work has been dedicated into client-side access pattern detection [10]–[15]. However, at server-side the access pattern is actually the result of the interaction of multiple concurrent applications sharing the file system. That means it is 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). For this scenario, a tuning technique capable of unsupervised learning is more adequate, as we argue in a previous work [16].

Such a technique will use the access pattern as the context to multiple concurrent learning algorithms. Hence the detection of the access pattern directly affects the system's ability to make good decisions. That happens because each context has to be observed multiple times before the system can learn, so learning is slower if too many redundant contexts are represented. Furthermore, if different situations are represented in the same context, that will introduce noise to the learning algorithm and prevent it from converging.

In this paper, we discuss the use of pattern matching for server-side file access pattern detection in the HPC I/O stack. The main goal of using pattern matching is to empower 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 application access patterns.

We propose a pattern matching technique for server-side detection. We 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 proposed techniques and discuss its application for other contexts. The rest of this paper is organized as follows. Section II discusses related work on access pattern detection. Section III discusses our case studies. Our pattern matching technique is detailed in Section IV. The methodology for this research is explained in Section V, and the results are discussed in Section VI. Section VII concludes this paper and points future work perspectives.

II. Related work

Access pattern detection is important for techniques that try to adapt to the workload. One popular way of doing so is through postmortem analysis — metrics are collected from applications during their execution, and the obtained information is used for future executions. Liu et al. [17] 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. [18], the MPI-IO library was modified to generate traces detailing information about each request. This information is later used to guide data replication. He et al. [19] use the IOSIG tool to capture a trace from an application. Data access costs are then calculated for each file region and results are used to optimize future executions of the application.

In this work, we want to detect access patterns at runtime. Making decisions based on application traces make more sense for techniques working at the application-level. When looking at the server side, traces include concurrent requests from multiple applications, a scenario that is harder to reproduce. Moreover, we want to benefit from similarities between different applications. At runtime, techniques can typically only use information from operations already performed by the application in the present execution [20].

Dorier et al. [10] propose a grammar-based approach called Omnisc'IO. Their mechanism intercepts requests in order to build a grammar to predict future accesses. The approach proposed by Tang et al. [11] 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 these techniques to benefit from information available in I/O libraries. Ge et al. [12] collect information from the MPI-IO library: operation, data size, spatiality, if operations are collective, and if operations are synchronous. Liu et al. [13] collect from MPI-IO the number of processes, the number of aggregators, and binding between nodes and processes. Lu et al. [14] use the offsets accessed by each process during collective operations. The processes' access spatiality is also obtained from MPI-IO in the approach proposed by Song et al. [15].

The discussed techniques for runtime detection work at the client side, where information is more easily obtained from I/O libraries, applications, etc. Dong et al. [21] use a time series model to estimate file system server load. The approach by Zhang et al. [22] applies a "reuse distance", defined as the time difference between consecutive requests from the same application to the same server. Compared to these techniques, in our approach we want a more detailed characterization of access patterns, accounting for more aspects.

III. Case studies

In this section, we discuss the two case studies for this work: two situations where online detection of the access pattern is key to improve the performance of the I/O stack.

A. Selecting the best request scheduler for data servers

In a previous work [8], we studied request scheduling policies working in the context of parallel file system data servers. From the five studied policies (TO, TO-agg, aIOLi, MLF, and SJF), we showed the best choice depends on the system and on the 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 performance *decreases* (up to 70%) for other applications. Therefore, in this situation it is important to dynamically select the best policy, adapting to the current workload.

We explore this 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. [23]. With different polices, 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.

B. Selecting the best window size for the TWINS scheduler

The TWINS request scheduler [9] was proposed to work in the I/O nodes — from the forwarding layer between processing nodes and the parallel file system — aiming at decreasing the contention in the access to the PFS data servers. 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 runtime [16]. This technique uses multiple k-armed bandits instances, one per context, and the context is given by the observed 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, allowing this learning technique to be applied 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 [24], or the approximate, linear-time DTW algorithm used in this work, FastDTW [25]).

It is important to notice here we talk about server-side access pattern, which may be different from the application access pattern due to striping over multiple servers and also due to concurrent applications. Therefore we *cannot* simply leverage application-side information, for instance from the I/O library, to characterize these accesses.

A similar pattern matching strategy was used with success in a previous work [26] 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. In this document, we call this metric *offset distance*. That means 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. Since we are working at file level, we do not have 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 Equation 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 for a new pattern, it 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. As such a system — for example a parallel file system or an I/O forwarding framework — is expected to run for years under a multitude of workloads, the maximum distance is expected to reach a stable value as soon as the system has observed non-contiguous access patterns. The impact of the first incorrect detections, which are false negatives because the maximum distance was too small, can be mitigated by periodically re-comparing all known patterns between themselves *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 is another reason why the *infinite* value is used in our solution. If there is no bound to how large the offset distance between consecutive requests can be, there will be no bound to the calculated distances, and the system will *not* converge to a stable knowledge base. In this situation, non-contiguous accesses to a particularly large file would generate high values in the time series and then produce large calculated distances when this time series is compared to others. That would push the maximum distance and thus skew all scores to be large, generating false positives in future comparisons.

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 ← false
2  for i ← 1 to n do
3      if diff(p, P[i]) < maxdiff then
4          dist ← fastDTW(p_ts, P[i]_ts)
5          maxdist ← max(maxdist, dist)
6          score ← 1 − dist/maxdist
7          if score > thres then
8              match ← true
9              pos ← i
10             break
11 if not match then
12     P[n + 1] ← p
13     pos ← n + 1
14 return match, pos
```

C. Coverage of access pattern aspects

Spatiality is used to represent access patterns. It is the aspect more commonly considered because it has a deep impact on performance [20]. Moreover, the offset distance also accounts for request size, another commonly considered aspect of access patterns. **The number of files being accessed is a secondary aspect** as an access pattern to multiple files will generate a highly noncontinuous pattern, which is why the number of files tends to affect *data* access performance. Finally, the intensity of the access (the requests arrival rate, usually modified in experiments by changing the number of concurrent client

processes) 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 (discussed in Section III) and also in the literature, the only one that is not directly 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 Equation 2 by taking into account their number of files (p_f), and read and write requests (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 type (read or write), (ii) the number of calls to the DTW algorithm required to look for a pattern is decreased, as it will only be compared to patterns of similar size and read/write proportion, and (iii) we avoid comparing patterns of very different lengths, what results in very large distances and might skew our scores.

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

D. Compression of patterns

Finally, if we consider the case of a parallel file system server in a large-scale HPC machine, and a period T of a few seconds, on periods of high intensity the patterns could become very long, including millions of requests. That means the time series comparisons take longer, and that the pattern matching mechanism has a larger memory footprint due to having to keep longer patterns. 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

For this research, we generated two data sets of server-side traces using the Grid'5000 [27] experimental platform. There are some publicly available traces from real systems, such as the ALCF I/O Data Repository[1]. Nonetheless, it would not be possible to use those for our evaluation as they contain application-side aggregated statistics about I/O accesses. What we need for this evaluation are server-side fine-grained traces, containing information about each request. Such traces from real systems are not typically captured and shared as they represent intractable volumes of data.

[1]https://www.mcs.anl.gov/research/projects/darshan/data/

We generated multiple benchmarks with the MPI-IO Test benchmarking tool [28] 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 [29] and the IOFSL forwarding framework [30] were in turn enriched with the AGIOS I/O scheduling library [8]. The library was used for its tracing capabilities.

Each experiment was repeated multiple times (6 for server traces and 10 for I/O node traces), and each repetition accesses a different file. The whole set of tests was executed in random order to avoid the impact of aspects not accounted for, and to minimize the impact of cache in the read experiments. Write experiments are not affected by caching because we configured OrangeFS to always sync data to files.

A. Traces from PFS data servers

The traces from parallel file system servers 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 2x10Gb Ethernet links. The two switches are connected to another switch with 2x40Gb Ethernet links each. The latter switch is connected to the Parasilo nodes with 2x10Gb Ethernet links to each node.

28 benchmarks were generated by varying the following 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 execute 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.

B. Traces from I/O nodes

The traces from I/O forwarding nodes were generated in two clusters located at the Nancy site of Grid'5000. PVFS version 2.8.2 was deployed over four nodes from the Grimoire cluster, and 32 clients and one IOFSL node were deployed over separated Grisou nodes. Each node of both clusters is powered with two eight-core Intel Xeon E5-2630 v3, and 128 GB of RAM. A 558 GB SATA Seagate ST600MM0088 HDD was used at the PFS data servers, one for each node. All the nodes have a 10 Gbps Ethernet interconnection, and there is a 10 Gbps link between the clusters. Both clusters were completely reserved to minimize network interference.

We generated benchmarks varying the following parameters: 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. All requests were issued through MPI-IO. In each experiment, a total of 4 GB of data are accessed.

C. Offline evaluation

We implemented our approach as described in Section IV and use it within a code for offline evaluation. This evaluation code reads requests from a trace file and feeds those requests to the mechanism, which outputs its decisions regarding pattern matching. The pattern matching mechanism writes a file to disk at the end of the execution containing its knowledge base. This file is read in the beginning of the next execution, hence the testing code can be called multiple times with different traces to emulate that sequence of applications being executed.

Scripts are used to parse the output to calculate *precision* and *recall*. These metrics come from the number of false and true positives and negatives, which is available during an offline analysis because we know from what access pattern the traces were obtained. Therefore we assume all patterns from the execution of the same benchmark should match, and patterns from different benchmarks should never match. Those are rather conservative assumptions, and make our results *pessimistic* regarding the quality of our results.

We gave traces from different benchmarks (and from different repetitions of each benchmark) in a random order to the offline evaluation system. We repeat this process four times with each group of traces to account for the random aspect. We have multiple parallel groups of traces, obtained from each data server and I/O node. Results are combined by calculating the median of output metrics. All results in this paper were obtained with 1-second long patterns. Previous research [16] indicate the feasibility of taking decisions and adapting the system every few seconds. Hence we have chosen a periodicity which represents a real usage while keeping a large number of patterns in our data sets to better evaluate our approach.

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: *[the repository will be online after the paper is accepted for publication]*

VI. Evaluation of the proposed technique

The server data set contains 9216 patterns, and the I/O nodes data set 23454. Since the patterns have a fixed duration (one second), the number of patterns is *not* the same for each benchmark, it depends on the execution time. The number of patterns represent different benchmark parameters are presented in Table I. Furthermore, the length of patterns is directly related to performance. Server patterns contain a median of 605 requests, and I/O nodes patterns a median of 123. In both cases, read patterns are longer, as seen in Table II because reading is faster than writing.

TABLE I: Representativity of access patterns in the data sets

	Data servers	I/O nodes
Read vs Write	1779 vs 7437	4367 vs 19087
Shared-file vs File-per-process	5063 vs 4153	11571 vs 11883
Contiguous vs 1D-strided	5154 vs 4062	17860 vs 5594

TABLE II: Number of requests per pattern

		Median	Maximum
Data server patterns	**Read**	734	14389
	Write	551	3008
I/O node patterns	**Read**	809	2102
	Write	135	418

A. All-to-all comparisons

Table III summarizes the results obtained by comparing all patterns from the I/O node traces, with the same operation, among themselves. We show the distribution of scores obtained with different levels of compression, and separating the comparisons that should result in a match from the ones that should not. In general we can see a clear separation (a possible threshold value) between the two groups.

TABLE III: Scores between patterns from I/O node traces

	Min.	1st Qu.	Median	3rd Qu.	Max.
			No compression		
No match	0	0.9029	0.9290	0.9736	1
Match	0.8116	0.9995	1	1	1
			Compression of 10		
No match	0	0.8872	0.9397	0.9844	1
Match	0.9673	0.9999	1	1	1
			Compression of 100		
No match	0	0.8731	0.9251	0.9820	1
Match	0.9506	0.9996	1	1	1

However, results without compression were not good for all access patterns. Table IV 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.

As shown in Table II, write patterns are shorter than read patterns. The fact 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-

TABLE IV: Scores between read patterns from I/O node traces. SF = "shared-file"

	Min.	1st Qu.	Median	3rd Qu.	Max.
No compression					
NO MATCH	0	0.4061	0.8543	0.9511	1
MATCH					
SF, contig	0.8116	0.9173	0.9390	0.9896	0.9985
SF, strided	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, contig	0.9673	0.9856	0.9898	0.9979	0.9995
SF, strided	0.9877	0.9991	0.9994	0.9996	1
Compression of 100					
NO MATCH	0	0.5895	0.8992	0.9856	1
MATCH					
SF, contig	0.9506	0.9955	0.9969	0.9987	1
SF, strided	0.9882	0.9990	0.9995	0.9997	1

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 depend on using different compression factors and maximum observed distances. In practice, that can be easily handled by building two time series at the same time (with different compression factors), and then at the end of the pattern keeping only one of them depending on the type of most requests (read or write).

1) Results with traces from PFS servers: Similarly, we compared all server patterns from each one of the four servers, with the same operation, among themselves. We present the score distributions in Table V, separating the comparisons that should result in a match from the ones that should not. Without compression, we can see there is no clear threshold that allows to match read patterns without also making most of incorrect matches. Similarly to what happened with I/O node traces, using compression improves results.

TABLE V: Scores between patterns from PFS server traces

		Min.	1st Qu.	Median	3rd Qu.	Max.
No compression						
No match		0	0.9801	0.9858	0.9973	1
Match	**Read**	0.4412	0.9446	0.9816	0.9990	1
	Write	0.9138	0.9904	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
Compression of 100						
No match		0	0.8882	0.9643	0.9983	1
Match	**Read**	0.5095	0.9838	0.9985	0.9996	1
	Write	0.9045	0.9974	0.9998	0.9999	1

2) The impact of the infinite *value:* As discussed in Section IV-A, an "infinite" value is used to represent the offset distance between requests to different files. If a measured offset distance between requests to the same files is larger than this infinite, it is replaced by this bound. The results presented so far were obtained using an infinite of 10 GB.

We repeated the all-to-all comparisons between *read* patterns using a compression factor of 100. Table VI shows the score distributions among comparisons that should not match, and comparisons that should match separated by spatiality and number of files. 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 if by a factor of ten (from 10 to 100 GB) does not affect results much, indicating **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 number accessed files are similar. Hence the actual results are expected to be better than what was observed in these all-to-all comparisons. For the same reason, the thresholds that worked best for the all-to-all comparisons provide only an indication of what an appropriate value will be.

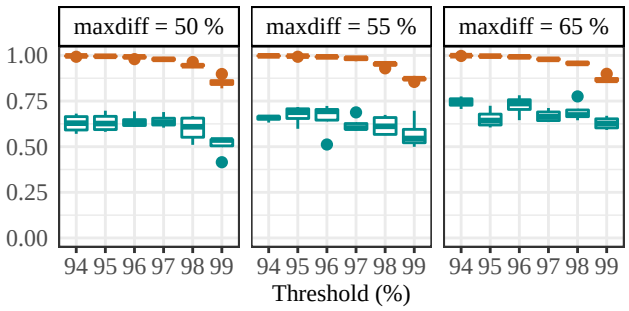
B. Precision and recall from the offline analysis

Figure 1 shows median precision (in cyan) and recall (in orange) from the offline evaluation of our mechanism using the server traces. For this analysis, we set the maximum observed distance to be the one from the all-to-all comparisons, aiming to evaluate an "already trained" system, without the initial impact of learning this parameter.

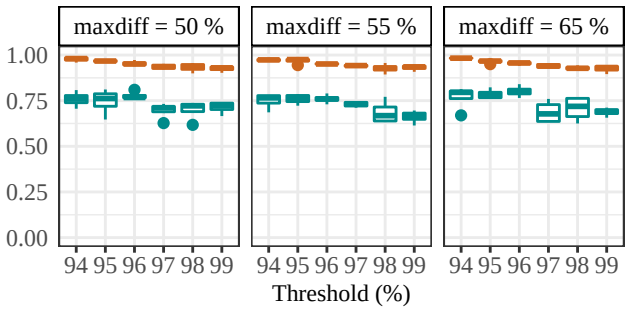
We observed recall of 1 for the lowest tested thresholds. In those situations, the threshold was too low and hence the mechanism gave too many positives, resulting in a small number of patterns in the knowledge base (less than 28, which is the number of different benchmarks). The best results were

TABLE VI: Scores between read server patterns (compression of 100). SF = "shared-file", FPP = "file-per-process"

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

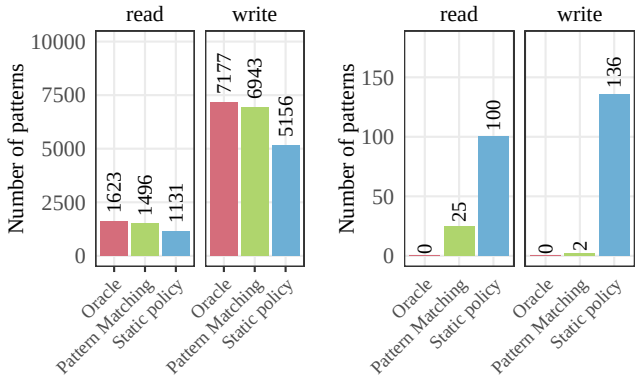


(a) Write experiments (compression of 10)



(b) Read experiments (compression of 100)

Fig. 1: Precision (cyan) and recall (orange) for experiments with server traces



(a) Increased performance (b) Decreased performance

Fig. 2: Number of seconds with performance increases or decreases by scheduling algorithm selection

obtained with a threshold of 96%, where the number of known patterns is between 20 and 50, consistent to what we would expect from the number of access patterns in the benchmarks.

Results were improved, specially for write experiments, by increasing the *maxdiff* parameter up to 65%. That allows for more comparisons to be made, as the tolerance for differences in the number of reads, writes, and accessed files is higher. The median precision with threshold of 96% and maxdiff of 65% was of 73.6% for write experiments and 79.9% for reads. The median recall was 99.2% for writes and 95.7% for reads. Of course both precision and recall are important, but 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 result and make it robust whenever possible 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 results obtained when using the 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 by these algorithm selections from measurements we have for the different scheduling policies with the different benchmarks.

By comparing the performance results estimated in this way with a baseline — the performance with the OrangeFS original scheduler — we counted the number of decisions (one per second, thus the number of seconds) that resulted in performance improvements or decreases. Results are presented in Fig. 2. We compare the results obtained with out 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). To evaluate a system that was already "trained" — that has already observed all the patterns — we did not count the first repetition of each benchmark, leaving 1723 read patterns and 7381 write patterns. From these, the oracle was able to improve performance for 94.2% of the read and 97.2% of the writes. Our pattern matching approach improved performance for 92.2% of the read and 96.7% of the write patterns *where the oracle is able to improve performance*, 32.3% and 34.6% *more* than the static solution. Our approach decreased performance by making inadequate algorithm selections for 25 (1.5%) read patters and 2 (0.03%) write patterns, 98.5% *less* than the static solution.

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 detection 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 the tuning of its parameters. In this context, we considered two case studies where adapting to the workload is essential to achieve good results: 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, the 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 80% and recall of up to 99%. When used to select the best scheduling algorithm, our mechanism was able to improve performance for up to 97% of the situations where it was possible to improve it, being up to 35% better than the best static algorithm selection.

As our approach builds a knowledge base of known patterns, we could easily enhance it to build a probabilistic chain between patterns, and then use this chain to predict the next pattern and apply optimizations accordingly. Exploring this possibility will be the subject of future work.

REFERENCES

[1] SUN, "High-performance storage architecture and scalable cluster file system," 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 *et al.*, "Scalable performance of the panasas parallel file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08. USENIX Association, 2008, pp. 2:1–2:17.

[3] 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.

[4] 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.

[5] 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.

[6] S. Seelam, I.-H. Chung, J. Bauer, and H.-f. Wen, "Masking I/O Latency using Application Level I/O Caching and Prefetching on Blue Gene Systems," in *IPDPS '10 Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2010, pp. 1–12.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot *et al.*, "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *The International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

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

[29] "Orangefs," accessed: March 2019. [Online]. Available: http://www.orangefs.org/

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