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

brought to you by CORE

provided by INRIA a CCSD electronic archive server

TWINS: Server Access Coordination in the I/O Forwarding Layer

Jean Luca Bez, Francieli Zanon Boito, Lucas Schnorr, Philippe Navaux, Jean-François Méhaut

To cite this version:

Jean Luca Bez, Francieli Zanon Boito, Lucas Schnorr, Philippe Navaux, Jean-François Méhaut. TWINS: Server Access Coordination in the I/O Forwarding Layer. 25th Euromicro International Conference on Parallel, Distributed and Networked-based Processing, Igor Kotenko, Mar 2017, St. Petersburg, Russia. hal-01515047

HAL Id: hal-01515047

https://hal.archives-ouvertes.fr/hal-01515047

Submitted on 27 Apr 2017

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

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

TWINS: Server Access Coordination in the I/O Forwarding Layer

Jean Luca Bez[1], Francieli Zanon Boito[2], Lucas M. Schnorr[1], Philippe O. A. Navaux[1], Jean-François Méhaut[3]
[1]Institute of Informatics – Federal University of Rio Grande do Sul
Porto Alegre, Brazil – Email: {jean.bez, schnorr, navaux}@inf.ufrgs.br
[2]Department of Informatics and Statistics – Federal University of Santa Catarina
Florianópolis, Brazil – Email: francieli.boito@posgrad.ufsc.br
[3]Université de Grenoble Alpes – INRIA & CEA – LIG Laboratory
Grenoble, France – Email: jean-francois.mehaut@imag.fr

***Abstract*—This paper presents a study of I/O scheduling techniques applied to the I/O forwarding layer. In high-performance computing environments, applications rely on parallel file systems (PFS) to obtain good I/O performance even when handling large amounts of data. To alleviate the concurrency caused by thousands of nodes accessing a significantly smaller number of PFS servers, intermediate I/O nodes are typically applied between processing nodes and the file system. Each intermediate node forwards requests from multiple clients to the system, a setup which gives this component the opportunity to perform optimizations like I/O scheduling.**

We evaluate scheduling techniques that improve spatiality and request size of the access patterns. We show they are only partially effective because the access pattern is not the main factor for read performance in the I/O forwarding layer. A new scheduling algorithm, TWINS, is presented to coordinate the access of intermediate I/O nodes to the data servers. Our proposal decreases concurrency at the data servers, a factor previously proven to negatively affect performance. The proposed algorithm is able to improve read performance from shared files by up to 28% over other scheduling algorithms and by up to 50% over not forwarding I/O.

***Keywords*—High Performance Computing; Parallel File Systems; I/O Forwarding; I/O Scheduling; Access coordination;**

I. Introduction

Scientific applications – such as climate and seismic simulations – feed the High-Performance Computing (HPC) field with performance requirements to provide understanding of complex phenomena. These performance requirements justify the appearance of ever increasing large scale parallel platforms. For instance, the *Aurora* supercomputer [1], expected for the next few years, will have over $50{,}000$ processing nodes to achieve 180 petaflops. Such large scale platforms typically include a shared storage infrastructure over a dedicated set of nodes with a parallel file system (PFS) deployment. If all processing nodes were to concurrently access the shared file system servers, contention would compromise performance.

The I/O forwarding technique aims at reducing the number of clients concurrently accessing the file system servers by placing some special nodes (called *I/O nodes*) that receive the processing nodes requests and forward them to the file system [2]. In this schema, the number of I/O nodes is typically larger than the number of file system servers, and smaller than the number of processing nodes. Furthermore, in this scenario, processing nodes may be powered with only a very simplified local I/O stack to avoid its interference on performance [3]. Besides performance improvements, the concept of I/O forwarding has the advantage of creating an additional layer between applications and file system. This layer can work to keep compatibility between both sides and apply optimizations such as request reordering, aggregation and compression.

The I/O scheduling optimization technique has already been successfully applied to the forwarding layer [4], [5]. However, previous scheduling algorithms work to adjust the access patterns generated to the file system. In this work, we evaluate the forwarding layer performance and demonstrate that these techniques are only partially effective because the access pattern is not the main factor that influences the performance of requests through I/O nodes.

We propose a new scheduling algorithm for the I/O nodes which works to decrease contention in the access to the parallel file system data servers. Our algorithm uses time windows and coordinates accesses from intermediate nodes so that at each time window they focus on one of the servers. As far as we know, this is the first work to propose a scheduling technique such as this to the forwarding layer.

Based on an extensive set of experiments, we detect I/O performance improvements with our algorithm over state-of-the-art algorithms. Moreover, our solution provides gains for 1D strided access patterns which are comparable to the use of collective I/O operations, while being completely transparent to applications and I/O library independent.

The rest of the paper is organized as follows. Section II presents related work, comparing our approach against the state of the art. Section III discusses the I/O forwarding framework used in this work. Then, in Section IV we propose our new scheduling algorithm and discuss its implementation and requirements. The experimental methodology is presented in Section V. Section VI evaluates the performance of our new scheduling algorithm for the forwarding layer compared against existing solutions. Section VII concludes the paper and discuss future directions.

II. Related Work

Large scale HPC platforms are typically used by multiple concurrent applications sharing a parallel file system infrastructure. For this reason, applications may observe performance degradation, a phenomenon known as *interference* [6], [7], [8], [9]. To alleviate its effects, a popular choice is the use of I/O scheduling [10], [11], [12], [13], [14], [15]. These techniques, applied at some layer of the I/O stack, decide *where* and *when* requests must be served.

Song et al. [11] proposed a scheduling algorithm for PFS servers. A window-wide coordination concept was employed to make all data servers focus on serving requests from one application at a time. Our proposal to make intermediate I/O nodes dedicate time windows to different data servers was inspired by the work of Song et al. Nonetheless, there are at least three differences between their proposal and ours. First, they target the PFS servers while we change the intermediate I/O nodes behavior. Second, their algorithm coordinates accesses to different applications, while we coordinate server accesses. And third, their algorithm uses the application unique identifier to decide the order of requests inside each time window. Requests from multiple applications may be eventually served in the same window. Differently, we completely dedicate each window to a single server.

Section II-A presents existing techniques to improve the forwarding layer. Section II-B discusses approaches to decrease concurrency at the parallel file system data servers.

A. I/O Forwarding

Considerable research has been focused on improving the I/O forwarding layer performance. Some of them studied the I/O subsystem of an IBM Blue Gene/P supercomputer. In this architecture, the data staging mechanism initially applied multiple threads per I/O node, without any coordination among them. Vishwanath et al. [4] identified some contention-related bottlenecks associated with this design. They improved performance by allowing asynchronous operations in the I/O nodes and by including a simple FIFO scheduler to coordinate accesses from multiple threads. This scheduler alone provided improvements of up to 38%. They also optimized data movement between layers through a topology-aware approach [16]. Isaila et al. [17] proposed a two-level prefetching scheme for this architecture.

Similarly to what was done by Vishwanath et al., Ohta et al. [5] improved performance of the IOFSL framework by using I/O scheduling. They implemented two algorithms: a simple FIFO and a quantum-based algorithm called *Handle-Based Round-Robin* (HBRR). The latter is based on an algorithm successfully applied to parallel file systems data servers [15], [18], [19], that aims at reordering and aggregating requests to improve the generated access pattern.

In this paper, we compare our new scheduling algorithm against FIFO and HBRR. The former provides the simplest coordination, while the latter is a more complex solution to improve the access pattern. They compose our baseline because they represent the state of the art in I/O scheduling for the forwarding layer. Nevertheless, as it is shown in Section VI-A, improving the access pattern is only partially effective to improve performance in this layer.

B. Concurrency at the parallel file system's data servers

Chen et al. [20] proposed a new collective I/O approach which uses the physical data layout information to make each aggregator access as few data servers as possible. A similar approach was presented by Wang et al. [21]. Their technique breaks collective I/O calls into multiple iterations to fit the buffer size. These partitions are optimized so each server is accessed by only one aggregator at each iteration. These initiatives obtain performance by decreasing the number of clients concurrently accessing each server. This approach potentially decreases network contention. Additionally, Yildiz et al. [9] have shown that concurrency at the data servers is one of the key factors for interference.

For these reasons, we propose an I/O scheduling algorithm that seeks to focus an I/O node's accesses to a single data server within each time window. As far as we know, ours is the first work to apply this strategy to the forwarding layer. This design choice has the advantage of making it completely transparent to applications, I/O libraries, and file systems.

III. The I/O Forwarding Scalability Layer

In this paper, we study I/O scheduling techniques at the I/O forwarding layer. Therefore an I/O forwarding framework was necessary to implement and validate our ideas. We have chosen to use the open source IOFSL framework [22] so we could build on previous contributions and effectively compare our new solution with the state of the art.

IOFSL uses the stateless ZOIDFS protocol, the API from the ZOID forwarding infrastructure [23], and the Buffered Message Interface (BMI) network abstraction layer [24] to provide request forwarding over multiple file systems and networks. The IOFSL software stack consists of two main components: a ZOIDFS client library running on the compute

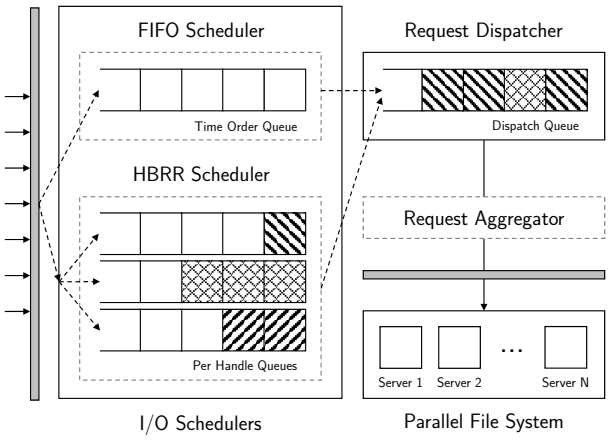


Fig. 1: Flow of requests through the IOFSL I/O node daemon. Requests come from the processing nodes (left) and leave for the file system (bottom right).

nodes and an I/O forwarding daemon running on the intermediate I/O nodes. The client library transparently forwards requests from the compute node to the corresponding I/O node.

In the I/O nodes, multiple threads are created to process the clients' requests. The request scheduler component coordinates these threads' accesses. It offers two options of scheduling algorithms - FIFO and HBRR - to fill a dispatch queue and thus decide the order requests must be processed. As discussed in Section II-A, FIFO is a simple time order algorithm, and HBRR is a handle-based round robin. HBRR employs multiple queues, one per handle, where contiguous requests are aggregated whenever possible. From each queue, a maximum number of requests (defined by the *quantum* parameter) may be served before moving to the next queue [5].

The flow of requests through the IOFSL I/O node daemon is illustrated by Fig. 1. After going through one of the scheduling algorithms, requests are stored in the dispatch queue. From the dispatch queue, requests to the same file and of the same type (read or write) are aggregated before being forwarded to the PFS. Although all clients data and metadata operations go through the IOFSL nodes, only data reads and writes go through the request scheduler component and are affected by scheduling algorithms. Section VI-A will discuss the performance obtained by IOFSL with these algorithms.

IV. TWINS: Decreasing Concurrency at the PFS Data Servers

The previous section described the scheduling infrastructure for the I/O forwarding layer, composed of two scheduling algorithms - FIFO or HBRR - and an aggregator at the dispatch queue. This model focuses on changing the access pattern to improve performance. However, as it is shown in Section VI-A, these schedulers are only partially effective because they do not take into consideration resource contention and data placement on the parallel file system servers. In this section, we present a new scheduling algorithm for the I/O forwarding layer called *Server Time WINdows* (TWINS). The main idea behind TWINS is coordinating intermediate I/O nodes' accesses to the file system so that, at any given moment:

I. an I/O node is focusing its accesses to only one of the parallel file system data servers;
II. the different I/O nodes are focusing on different servers.

TWINS pseudo-code is presented in Algorithm 1. It keeps multiple request queues, one per data server. During the execution, TWINS iterates the different queues following a round robin scheme, respecting a time window that must be dedicated to each server. If server i is the current server being accessed but there are no requests to this server, the scheduler will wait until requests to server i arrive or the time window ends, even if there are queued requests to other servers.

The rest of this section discusses TWINS characteristics and implementation. Section VI-B presents performance improvements provided by our new algorithm.

Algorithm 1 *Server Time WINdows*

Require: $Q[i]$ is the updated list of requests to server i
1: $i \leftarrow 0$
2: **while** $true$ **do**
3: $resetTimer()$
4: **while** $elapsedTime() < windowSize$ **do**
5: **if** $length(Q[i]) > 0$ **then**
6: $processRequest(Q[i])$
7: **else**
8: $timeout \leftarrow windowSize - elapsedTime()$
9: $timedWaitForRequests(Q[i], timeout)$
10: **end if**
11: **end while**
12: $i \leftarrow nextServer(i)$
13: **end while**

A. Implementation with AGIOS

We integrated the *AGIOS* scheduling library [15] in IOFSL as a scheduling option (just like FIFO and HBRR). *AGIOS* can be used by I/O services to manage incoming requests at file level and provides a simple API to the development of new scheduling algorithms. We implemented TWINS through this simple API. It would have been possible to implement the algorithm inside the IOFSL source code. However, doing so with *AGIOS* makes our solution more generic, as it can be used by other I/O services, or by other I/O forwarding frameworks.

The new organization of the I/O node daemon is illustrated in Fig. 2. Using TWINS, requests are added to the per-server queues upon arrival at IOFSL. When the algorithm decides to process a request, a callback function written inside IOFSL simply adds it to the dispatch queue. This ensures requests will be processed in the order dictated by the scheduler.

Differently from FIFO, that uses a single queue, and HBRR, that uses two queues per file handle, TWINS applies one queue per data server. Considering the number of files is typically far superior to the number of servers, the overhead induced by TWINS regarding the management of multiple queues is expected to be lower than what is caused by HBRR.

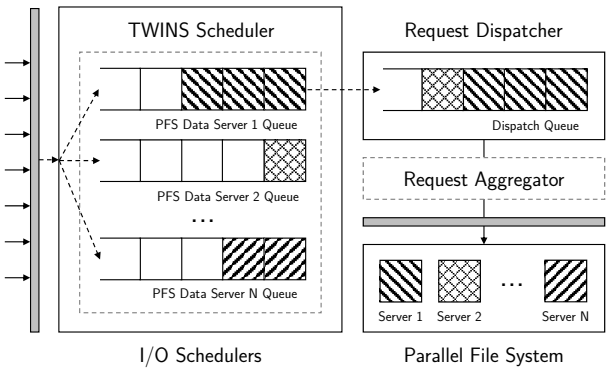


Fig. 2: Flow of requests through the IOFSL I/O node daemon with the new TWINS scheduling option.

B. Finding out to which server a request is

In addition to typically available information about requests - file handle, offset, type, and size - our algorithm requires a server identifier. We have modified IOFSL to collect the file distribution information from the PFS when opening or creating a file. Since this information is easily available to clients in most file systems such as PVFS2 [25] and Lustre [26], our solution can still be considered file system generic.

The distribution information is requested only once per file. The induced overhead is of 54.3ms on our experimental environment (average of 124 observations). This time can be expected to be diluted by longer read and write times.

Using the file distribution information, the starting server for a request is obtained as a function of its starting offset and stripe size - also part of the collected information. This is done when requests arrive, before adding them to the queues.

Considering the described TWINS algorithm, we can notice that simply following this approach would cause all intermediate I/O nodes to focus on the same servers at the same time. To cause the desired distribution effect, we add an extra server identifier translation step. This translation is done according to the I/O node identifier. The N_{th} I/O node will use the N_{th} permutation of the servers list as a translation rule. Therefore, if the number of intermediate nodes is larger than the number of servers, more than one node may access the same server at the same time, but these concurrent accesses are minimized.

For instance, if there are two I/O nodes (N_0 and N_1) and two data servers (S_0 and S_1), N_0 will go through the servers in the order S_0, S_1, while N_1 will use the order S_1, S_0. Therefore the translation function in N_0 maps S_0 to 0 and S_1 to 1, while in N_1 it will map S_1 to 0 and S_0 to 1. This ensures each I/O node will focus on a different server at each time window.

V. Experimental Methodology

All experiments in this paper were conducted in clusters from the Grid'5000 [27] at Nancy. Four nodes from the Grimoire cluster were used as PVFS2 servers (acting as both data and metadata servers) and 32 nodes from the Grisou cluster were used as clients. Four nodes from Grisou were used as IOFSL servers. Hence, each I/O node runs on a separate machine, which is **not** shared with clients or PFS servers.

Each node from Grimoire and Grisou has two 8-core Intel Xeon E5-2630 v3 and 128GB of RAM. A 558GB hard disk is used for storage at the servers. Nodes are interconnected through a 10Gbps Ethernet network, and there is a 10Gbps link between the clusters. Both clusters were completely reserved during the experiments to minimize network interference.

PVFS version 2.8.2 was used with its default parameters, including 64KB stripe size and striping through all four servers. Data servers were configured to perform I/O operations directly to their storage devices, bypassing buffer caches. This was done to avoid a situation where the tests scale would hide the access pattern impact on performance.

The IOFSL dispatcher uses the PVFS2 client library to communicate directly with the file system, allowing direct access to the file system instead of accessing it through the PVFS2 kernel module. The use of IOFSL is transparent to applications, as accesses are forwarded through the ZOID API. An environment variable is set at the processing nodes to determine to where requests must be redirected. Clients are equally distributed among the I/O nodes.

The IOFSL daemon was executed with all its default parameters. The maximum number of requests that can be aggregated from the dispatch queue (batch size) was 16. Minimum and maximum numbers of threads are four and 16, respectively. For the event handler, IOFSL state machines were used.

The MPI-IO Test benchmark was executed by 128 processes to generate requests through the MPI-IO library. Each process generates 1024 read or write requests of 32KB, i.e. 32MB of data is accessed per process, a total of 4GB per test. Tests were executed for the file-per-process approach, where each process contiguously accesses its own independent file, and for the shared-file one. With a shared file, processes either access their own contiguous portions inside the shared file, or follow a 1D strided access pattern. These experiments represent access patterns that are usual among scientific applications.

From each benchmark execution, we take the completion time of the slowest process. We use the makespan as a performance metric because it represents the total time to process a workload from the file system point of view.

Experiments were repeated at least 8 times, and error bars were calculated using a 99.7% confidence interval. Different experiments were executed in a random order to avoid bias imposed by some uncontrolled parameter, or some unexpected effect caused by a specific experimentation order.

VI. Performance Evaluation

This section presents our performance evaluation. First we discuss the performance obtained using the IOFSL framework with the baseline scheduling algorithms. Then, in Section VI-B we describe results with our new TWINS algorithm.

A. Performance of the baseline scheduling algorithms

IOFSL represents a synchronous I/O forwarding approach, i.e., the intermediate I/O nodes are an extra hop between processing nodes and the file system. In this approach, I/O nodes do not act as burst buffers but instead, the clients' expectation of persistent storage in the file system is met.

Figures 3 and 4 present results obtained by read and write tests, respectively, with the three tested access patterns. In all graphs, the first column (in yellow) shows time obtained without IOFSL, and the second and third columns (in red) show time obtained using IOFSL with its base scheduling algorithms. From these, the first bar represents the FIFO scheduling algorithm, while the second bar represents HBRR.

We can see that **read performance is improved up to** 36% **by using IOFSL**, despite the extra transmission cost between clients and the file system. Performance benefits from using the I/O forwarding layer to all tested read access patterns. On the other hand, for write requests, performance is decreased for all situations. This decrease is higher for the file-per-process scenario than for the shared file ones.

One could believe the explanation for the good results observed for read tests is that the gains obtained by aggregating requests before forwarding them to the file system compensates the overhead imposed by the extra hop. Nonetheless, this is not the case. Table I presents the average request size observed at different layers of the I/O stack during the experiments with the IOFSL base schedulers and shared-file approach. In the file-per-process scenario, each file is accessed by a single process one request at a time, so there are no aggregation opportunities. Single median values are presented because values were similar between FIFO and HBRR tests. We can see that the write tests present similar aggregated request sizes (by IOFSL), but still do not achieve the same gains as read tests.

TABLE I: Average request size at different I/O stack levels.

	Contiguous		1D strided	
	READ	**WRITE**	**READ**	**WRITE**
Leaving clients	32KB	32KB	32KB	32KB
Leaving I/O nodes	58KB	58KB	58KB	58KB
Arriving at servers	43KB	44KB	50KB	49KB

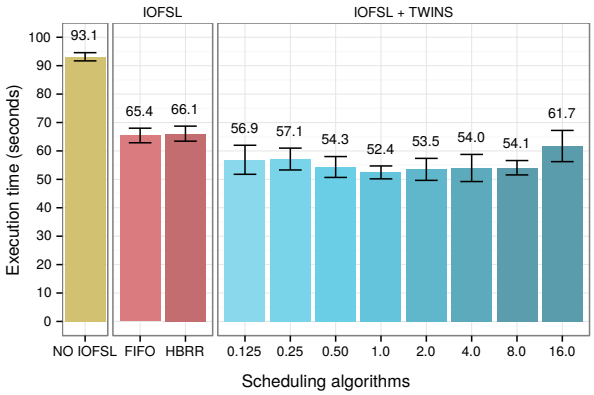
Therefore, **despite requests aggregations usually being helpful for performance, they are not the main factor in the observed read performance improvements**. Another evidence in this direction is that the best gains were observed for the file-per-process scenario, where there are no aggregations. Furthermore, despite making more effort into generating a better access pattern, **the HBRR algorithm does not outperform FIFO**. Results for the two algorithms were not significantly different in any of the situations.

Table II presents the average offset distance of requests leaving the I/O nodes during the read shared-file tests. The offset distance is a spatiality metric calculated by taking the offset difference between every two consecutive requests. The *higher* the distance, the *less contiguous* an access pattern is. The contiguous *local* access pattern presents the highest average offset distances, i.e., it is actually the least contiguous *global* access pattern. This happens because each process accesses contiguously its own portion of the shared file, but different processes are accessing requests that are sparse in the file. During the 1D strided test, requests from different processes are contiguous to each other. The average offset distances during tests with FIFO and HBRR are very similar. Hence, we can notice both algorithms result in very similar access patterns. This explains why they perform similarly.

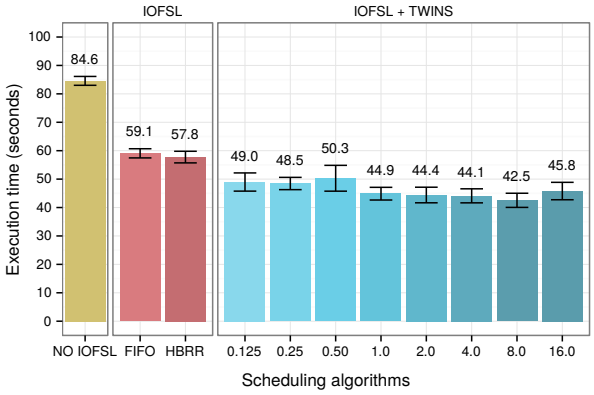
TABLE II: Average offset distance observed during **read** tests.

Contiguous		1D strided	
FIFO	**HBRR**	**FIFO**	**HBRR**
1341.00MB	1340.95MB	45.22MB	44.74MB

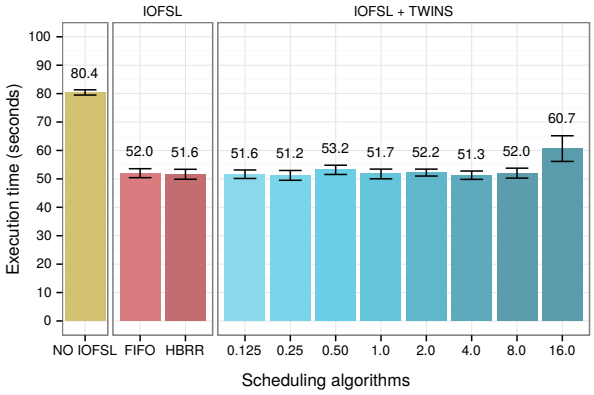
We have measured the time difference between consecutive requests. These values were obtained from four new executions



(a) Shared-file with contiguous access



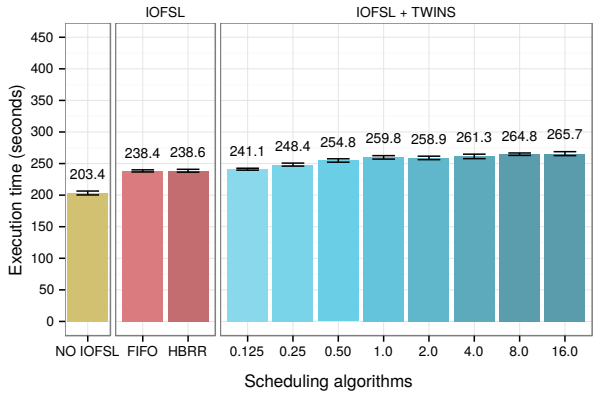
(b) Shared-file with 1D strided access



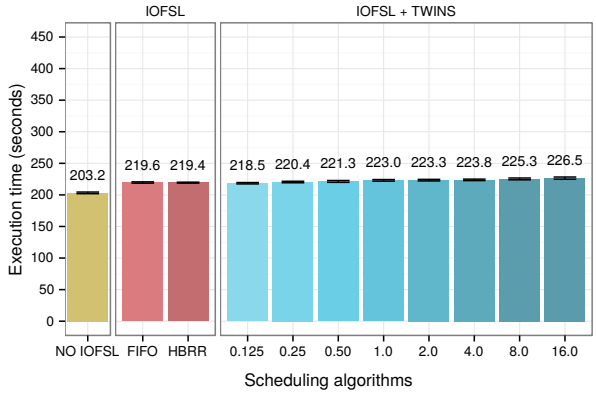
(c) File-per-process access

Fig. 3: Execution time of **read** tests without IOFSL, with IOFSL using FIFO or HBRR, and using the TWINS algorithm with different time window sizes (ms).

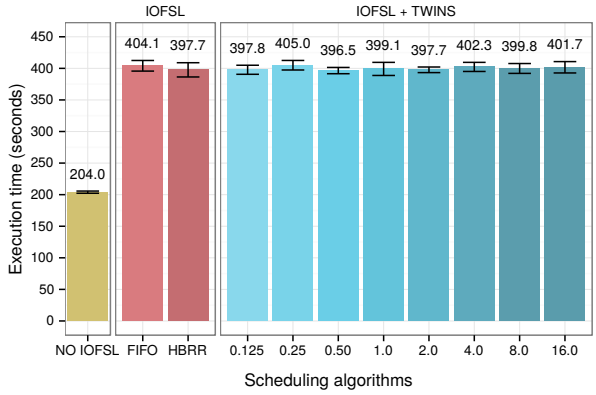
of the 1D strided shared-file test through a single I/O node. In the intermediate I/O node, we have traced all requests' arrival time. We have considered only the first 128, i.e. the first request from each of the 128 processes. Since tests are synchronous, all requests after the first 128 depend on the time it took to process the previous ones, so they are not independent. We have used the median because it is less



(a) Shared-file with contiguous access



(b) Shared-file with 1D strided access



(c) File-per-process access

Fig. 4: Execution time of **write** tests without IOFSL, with IOFSL using FIFO or HBRR, and using the TWINS algorithm with different time window sizes (ms).

sensitive to outliers, and timing values inside each test tend to have high variability. For read requests the median time difference is 26.09μs, whereas for write requests it is 50.92μs.

Since read requests are smaller than writes (they do not carry data), they arrive at a faster pace to the I/O nodes (or to the server if I/O nodes are not present). Hence, the extra hop between clients and PFS works to "funnel" requests and decrease concurrency at the servers. For this reason, **FIFO and HBRR, which work to improve the generated access pattern, are only partially effective in improving read performance**. A more effective strategy would be to work to further decrease contention in the access to the file system. The next section presents results obtained with our new scheduling algorithm, proposed to explore this idea.

B. Performance of the TWINS algorithm

In this section, we evaluate the performance of the new TWINS algorithm, which works to decrease contention in the access to the parallel file system data servers. As described in Section IV, in order to do that, TWINS divides execution in time windows and focuses each IOFSL node's accesses to a different data server during each time window.

The third group of bars from all graphs in Figures 3 and 4 (in blue) present results obtained with our algorithm for different time window sizes. **TWINS provides read performance improvements of up to 28% over the baseline scheduling algorithms and of up to 50% over not using IOFSL. The best results were obtained for the shared-file 1D strided access pattern, and gains for the shared-file contiguous pattern were also observed - up to 20% over the baseline.**

The lower improvements obtained for contiguous access patterns are justified by the requests distribution among the data servers caused by the access patterns. In the 1D strided test, processes start their accesses at different servers and this behavior is kept throughout the execution. Therefore, with this access pattern the scheduler always has requests for all servers and thus has the opportunity to perform meaningful coordination. In the contiguous test, since each process segment has 32MB, which is a multiple of the stripe size $\times$ the number of servers, all processes start their accesses by the same server. In this case, the opportunities to accesses coordination - i.e., the situations where there are queued requests for multiple servers - appear during the execution as the delays induced by TWINS causes some processes execution to advance faster than others. This phenomenon is not guaranteed to happen.

Performance does not benefit from using TWINS in read tests with the file-per-process scenario and in write tests. Even for these cases, performance is not decreased significantly compared to the baseline algorithms as long as a small window size is used. Therefore, **in addition to improving performance of some situations, our proposal does not necessarily harm performance in situations where it is not able to provide improvements**.

1) Impact of the time window parameter: TWINS' behavior is affected by its time window duration. A window that is too small does not allow for an effective coordination of accesses among the data servers because it is not long enough to allow the execution of multiple requests. Moreover, a fast time window does not hold requests to other servers for long enough so requests to the currently accessed data server are out of the dispatch queue to the file system. If requests for different servers are in the dispatch queue at the same time, they could be aggregated before being forwarded to the PFS

and thus the scheduling algorithm work would be undone. On the other hand, a window that is too large imposes overhead as there are not enough requests to each data server to fill a whole window, so the scheduler spends too much time waiting. Another source of overhead, in this case, would be the delay imposed to requests, which could not be compensated by the gains of decreasing concurrency at the data servers.

The best window duration is not the same for all situations where TWINS improves performance. The best results for contiguous shared-file read tests were obtained with a one-second time window, while the best results for 1D strided shared-file read tests were for a eight-seconds window. There is a trade-off to be observed between the induced overhead and how distributed among the servers requests are. Further analysis is required to determine how the scheduler could automatically find the best window duration. This will be the focus of future work.

2) Multi-application scenario: To confirm our algorithm performance on a multi-application scenario, we have conducted additional experiments using the *Ifer* benchmark [28]. This benchmark splits the set of processes into groups running on two different sets of nodes. Each group of processes executes a series of MPI-IO operations, simulating two applications accessing the shared file system in contention. We have modified *Ifer* to perform read operations to previously created files. For these experiments, each application has 64 processes and presents the shared-file 1D strided access pattern.

Fig. 5 presents execution times of the first application (A) in the multi-application experiments. The lines represent different options - not using the forwarding layer, using it with the baseline algorithms and using it with TWINS. The x-axis represents the time difference between start time for applications A and B: when dt is 0 both start at the same time, positive dt means A starts before B, and negative dt means B starts first.

We can see the I/O forwarding layer also improves read performance for the multi-application scenario - up to 35% with the baseline algorithms. The interference experienced by the application is decreased by FIFO and HBRR up to 25%, except when applications start with a 10-seconds difference. The interference factor is calculated as the ratio between the execution time of the application under contention and the time of the application executing by itself. **TWINS improves performance up to 16% over FIFO and HBRR, and up to 45% over not using IOFSL. Interference is further decreased by using TWINS - up to 12% over the baseline algorithms and up to 31% over not using I/O nodes**.

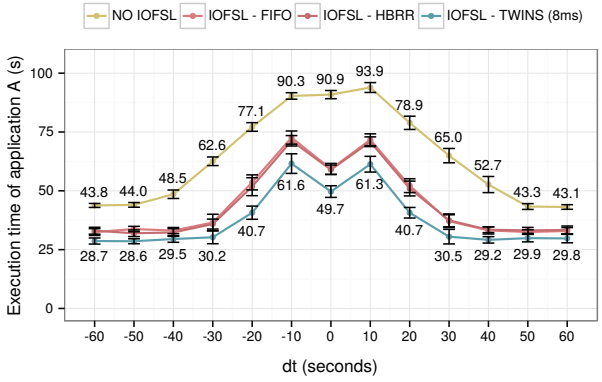


Fig. 5: Execution time for an application under contention caused by another concurrent application.

3) Comparison with collective I/O: The traditional way of improving performance of 1D strided access patterns with small requests is to use collective I/O operations. Fig. 6 compares the performance obtained by TWINS for this access pattern with what is achieved by making the single application perform collective calls. As a reference, times obtained using IOFSL with the baseline algorithms are also presented. We can see TWINS is able to provide as much performance improvement as the use of collective operations for 1D strided read access patterns with small requests.

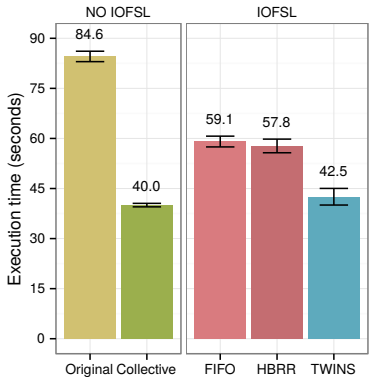


Fig. 6: TWINS vs. collective I/O operations

It is important to notice that TWINS represents a more transparent and generic solution than MPI-IO collective operations. **Because it is applied in the I/O nodes, TWINS is completely transparent to applications and I/O library generic. Therefore applications using any method to perform I/O operations, such as POSIX, can benefit from this optimization.** To the best of our knowledge, this is the first work to propose a scheduler to the I/O forwarding layer which transparently coordinates accesses to alleviate concurrency at the PFS data servers.

VII. Conclusions and Future Work

In this paper, we studied the I/O scheduling technique at the I/O forwarding layer by evaluating algorithms previously applied to this layer - FIFO and HBRR. This evaluation has shown that, despite improving read performance, techniques that work to adjust the access pattern (requests aggregation and reordering) are only partially effective.

We have proposed a new scheduling algorithm for the I/O forwarding layer called TWINS. Our algorithm uses time windows to coordinate the I/O nodes' accesses to different data servers, working to decrease contention in the access to

the servers. To the best of our knowledge, ours is the first work to apply such a technique in the I/O forwarding layer.

TWINS results have shown performance improvements for shared-file read access patterns of up to 28% over the FIFO and HBRR algorithms. Compared to not using I/O forwarding nodes, the gains were of up to 50%. Improvements were also shown for a multi-application scenario, accompanied with a decrease in interference. Even for situations where TWINS is not able to improve performance, it does not harm it. Performance obtained by TWINS for the 1D strided read access pattern was comparable to what can be achieved by making the application use collective operations. We have compared our results with collective I/O because this would be a popular alternative applied to improve performance of the applications with 1D strided access patterns. Nevertheless, compared to collective I/O, our proposal is completely transparent to applications and library independent.

Future work will focus on proposing an automatic mechanism to tune the TWINS' time window duration parameter based on the observed access pattern. Moreover, we will use TWINS to provide performance improvements to real scientific applications.

ACKNOWLEDGMENT

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr/). This research was accomplished in the context of the International Joint Laboratory LICIA. Research has received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E Project, grant agreement n° 689772. This work was also supported by STIC-AmSud/CAPES scientific cooperation program under EnergySFE research project grant 99999.007556/2015-02. The authors would also like to thank Kamil Iskra, Rob Latham, and Rob Ross from the Argonne National Laboratory for insights about the IOFSL framework.

REFERENCES

[1] A. L. C. Facility, "Aurora supercomputer," http://aurora.alcf.anl.gov/, accessed: May 2016.

[2] J. Dongarra *et al.*, "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, p. 3, 2011.

[3] W. Xu *et al.*, "Hybrid hierarchy storage system in MilkyWay-2 supercomputer," *Frontiers of Computer Science*, vol. 8, no. 3, pp. 367–377, 2014.

[4] V. Vishwanath *et al.*, "Accelerating I/O forwarding in IBM blue gene/p systems," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. IEEE, 2010, pp. 1–10.

[5] K. Ohta *et al.*, "Optimization techniques at the I/O forwarding layer," in *Cluster Computing, 2010 IEEE International Conference on*, ser. CLUSTER. IEEE, 2010, pp. 312–321.

[6] A. Bhatele and K. Mohror, "There goes the neighborhood: performance degradation due to nearby jobs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. IEEE, 2013, pp. 41:1–41:12.

[7] A. Shah *et al.*, "Capturing inter-application interference on clusters," in *Proceedings of the 2013 IEEE International Conference on Cluster Computing*, ser. CLUSTER '13. IEEE, 2013.

[8] C.-S. Kuo *et al.*, "How file access patterns influence interference among cluster applications," in *Proceedings of the 2014 IEEE International Conference on Cluster Computing*, ser. CLUSTER '14. IEEE, 2014, pp. 185–193.

[9] O. Yildiz *et al.*, "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems," in *The 30th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS 2016. IEEE, 2016.

[10] X. Zhang *et al.*, "IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. IEEE, 2010.

[11] H. Song *et al.*, "Server-side I/O coordination for parallel file systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. IEEE, 2011.

[12] J. Liu *et al.*, "Hierarchical I/O scheduling for collective I/O," in *Proceedings of the 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGRID '13. IEEE, 2013, pp. 211–218.

[13] M. Dorier *et al.*, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. IEEE, 2014, pp. 155–164.

[14] D. Dai *et al.*, "Two-Choice Randomized Dynamic I/O Scheduler for Object Storage Systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. IEEE, 2014, pp. 635–646.

[15] F. Z. Boito, R. V. Kassick, P. O. A. Navaux, and Y. Denneulin, "Automatic I/O scheduling algorithm selection for parallel file systems," *Concurrency and Computation: Practice and Experience*, 2015. [Online]. Available: http://dx.doi.org/10.1002/cpe.3606

[16] V. Vishwanath *et al.*, "Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. IEEE, 2011, pp. 1–11.

[17] F. Isaila *et al.*, "Design and evaluation of multiple-level data staging for blue gene systems," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 22, no. 6, pp. 946–959, 2011.

[18] A. Lebre *et al.*, "I/O scheduling service for multi-application clusters," in *Cluster Computing, 2006 IEEE International Conference on*, ser. CLUSTER '06. IEEE, 2006, pp. 1–10.

[19] Y. Qian *et al.*, "A novel network request scheduler for a large scale storage system," *Computer Science - Research and Development*, vol. 23, no. 3–4, pp. 143–148, 2009.

[20] Y. Chen *et al.*, "LACIO: A new collective I/O strategy for parallel I/O systems," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. IEEE, 2011, pp. 794–804.

[21] Z. Wang *et al.*, "Iteration based collective I/O strategy for Parallel I/O systems," in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGRID '14. IEEE, 2014, pp. 287–294.

[22] N. Ali *et al.*, "Scalable I/O forwarding framework for high-performance computing systems," in *IEEE International Conference on Cluster Computing and Workshops, 2009*, ser. CLUSTER'09. IEEE, 2009, pp. 1–10.

[23] K. Iskra *et al.*, "ZOID: I/O forwarding infrastructure for petascale architectures," in *in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2008, pp. 153–162.

[24] P. Carns *et al.*, "BMI: a network abstraction layer for parallel I/O," in *19th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '05, 2005, pp. 8 pp.–.

[25] P. H. Carns *et al.*, "PVFS: A parallel file system for linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*. USENIX Association, 2000, pp. 317–327.

[26] I. Sun Microsystems, "LUSTRE file system - high-performance storage architecture and scalable cluster file system," Tech. Rep., 2007. [Online]. Available: http://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf

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

[28] O. Yildiz, "IFER: MicroBenchmark for Studying the Cross-Application I/O Interference," https://team.inria.fr/kerdata/ifer-microbenchmark-for-studying-the-cross-application-io-interference/, accessed: Sep 2016.