On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems

Orcun Yildiz*, Matthieu Dorier†, Shadi Ibrahim*, Rob Ross†, and Gabriel Antoniu*
*INRIA Rennes Bretagne-Atlantique, Rennes, France, {first.last}@inria.fr
†Argonne National Laboratory, IL 60439, USA, {mdorier, rross}@anl.gov

***Abstract*—As we move toward the exascale era, performance variability in HPC systems remains a challenge. I/O interference, a major cause of this variability, is becoming more important every day with the growing number of concurrent applications that share larger machines. Earlier research efforts on mitigating I/O interference focus on a single potential cause of interference (e.g., the network). Yet the root causes of I/O interference can be diverse. In this work, we conduct an extensive experimental campaign to explore the various root causes of I/O interference in HPC storage systems. We use microbenchmarks on the Grid'5000 testbed to evaluate how the applications' access pattern, the network components, the file system's configuration, and the backend storage devices influence I/O interference. Our studies reveal that in many situations interference is a result of bad flow control in the I/O path, rather than being caused by some single bottleneck in one of its components. We further show that interference-free behavior is not necessarily a sign of optimal performance. To the best of our knowledge, our work provides the first deep insight into the role of each of the potential root causes of interference and their interplay. Our findings can help developers and platform owners improve I/O performance and motivate further research addressing the problem across all components of the I/O stack.**

***Keywords*-Exascale I/O, Parallel File Systems, Cross-Application Contention, Interference**

I. Introduction

I/O interference in high-performance computing (HPC) is defined as the performance degradation observed by any single application performing I/O in contention with other applications running on the same platform. This interference, while already present in small clusters, becomes an increasingly important issue as we move closer to exascale, first because larger machines are used by more applications at the same time [1] and second because uncontrolled I/O performance degradations in large-scale applications lead to a larger waste of computation time and energy. Because of the variety of access patterns exhibited by HPC applications, however, it is difficult to predict when this interference will occur and whether and how it will affect each of the applications involved.

For several years researchers have been tackling cross-application I/O interference in the HPC area. The focus has been on causes as diverse as access locality in disks [2], synchronization across storage servers [2], [3], or network contention [4]–[7]. While these solutions get us undeniably closer to solving the I/O interference problem, they all focus on a single potential root cause of interference (e.g., the network) and do not look at the interplay between several potential such causes.

In this paper, we conduct an extensive experimental campaign exploring the root causes of I/O interference in HPC storage systems. We use microbenchmarks on the Grid'5000 [8] testbed to evaluate how the applications' access pattern, the network components, the file system's configuration, and the backend storage devices influence interference. While some of our results follow our intuition, others illustrate unexpected behaviors caused by the interplay between different points of contention. These surprising behaviors include the fact that *decreasing the network bandwidth can in some cases eliminate the interference*.

An important outcome of our study is that in many situations, interference is a result of *bad flow control* in the I/O path, rather than the presence of a single bottleneck in one of its components. We hope that the insights and lessons learned from our experiments will enable a better understanding of I/O interference, help platform administrators diagnose the cause of such issues in their system, and motivate further research addressing the problem, not only at a single level, but also across all components of the I/O stack potentially involved.

The remainder of this paper is organized as follows. Section II gives background for our work. Section III explains our methodology in investigating the root causes of I/O interference, including a description of the benchmark, platform, and software. Section IV presents different sets of experiments highlighting the different causes of I/O interference and then shows cases where it produces unexpectedly high performance degradation. All the experiments are accompanied by lessons learned from the behavior of the system in these scenarios. In Section V we present related work. We conclude in Section VI with a summary and a brief look at future work.

II. Background

In the context of I/O for HPC, the main shared resource that applications contend for is the parallel file system, which comprises components that can all become a point of contention. By considering the design of common HPC storage systems, we identified four potential points of contention in the data path, illustrated in Figure 1.

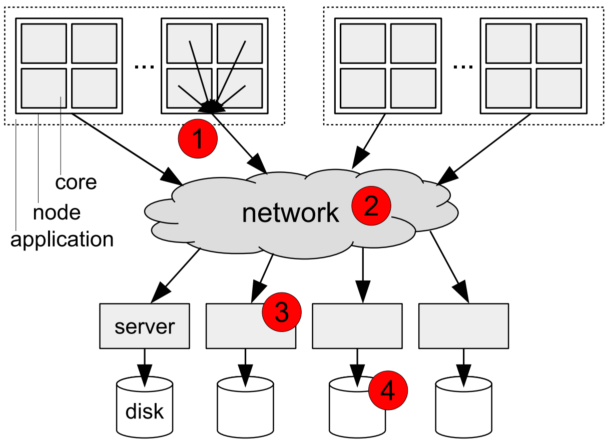


Figure 1. Typical parallel storage system and potential points of contention.

1) As the number of cores per node increases, the network interface shared by all the cores in a node can become a first point of contention [9], [10]. Failure to address this first bottleneck at a single-application level can affect the interference encountered with other applications farther down the I/O path.
2) The network linking computation nodes to storage servers (which we will call "storage network," as opposed to the "computation network" used across computation nodes, and which may be different) is usually the first point of contention between *multiple*, independent applications.
3) The servers running the parallel file system constitute a third possible cause of cross-application interference, because of the limited bandwidth each server provides and because of scheduling decisions being made at this level regarding the order in which I/O requests should be serviced [2]–[4].
4) The disks (or any other backend storage devices) constitute the lowest level at which contention can occur. While the servers serialize requests into actual disk accesses, interleaved requests from different applications can break the locality of disk accesses and degrade performance [11].

Other possible sources of interference include I/O forwarding nodes [12]–[14], RAID technology used in backend storage devices, or the computation network. The platform we use for our experiments does not feature forwarding nodes or RAID technology; and while the computation and storage networks are the same, our benchmark does not perform communications. Thus we will not investigate these potential sources here.

III. Methodology

Investigating the root causes of I/O interference is a challenging task given the complexity of HPC storage systems and the large number of parameters that can contribute to interference. In the previous section, we identified the potential points of contention in the data path. Here we describe our methodology for investigating these parameters in order to draw meaningful and useful conclusions from our experiments.

A. Role of each point of contention

One way of studying the effect of potential points of contention consists of carefully isolating each of them and benchmarking them separately. *This is not the approach we undertake here*. This approach indeed does not capture the interplay between causes, such as the fact that contention at one level can either mitigate or exacerbate interference at another level.

Our approach consists of either ruling out potential causes of interference or modifying their parameters and observing the resulting performance under congestion. This approach has proved much more useful not only in understanding the role of each point of contention but also in evaluating their interactions. More specifically we proceed as follows for each level.

1) The network interface can be ruled out by making a single core on each node issue all the I/O requests of that node.
2) While the network can be ruled out by having clients run on the same node along with a single-server file system, this option gives us little information about the role of the network in a large, multiserver deployment of the file system. We therefore evaluate the impact of the network's bandwidth on the interference as well.
3) The servers can be ruled out by ensuring that each group of processes accesses a distinct set of servers. In this scenario the two groups will interfere at the network level but not in the servers or for the access to the disks.
4) The disks can be ruled out by using much faster devices such as SSDs or local memory or by asking the file system to throw away any incoming data instead of storing it. Another option is to turn off the synchronization of data files in the file system, which allows the servers to keep data cached in memory and flush them to disks later. We use these two methods in our experiments.

B. Microbenchmark and reporting method

To investigate the influence of various parameters on the I/O interference, we follow the methodology used in [1]. We developed a microbenchmark similar to IOR [15]. This application starts by splitting `MPI_COMM_WORLD` into two groups of processes running on two separate sets of nodes. Each group of processes executes a series of collective I/O operations following a specified pattern, simulating two applications accessing the file system in contention. We measure the time taken by each group of processes to complete its set of I/O operations.

The experiments presented in this paper focus on write/write interference only. They use two different access patterns.

Contiguous In this pattern, each process issues a 64 MB write request in a contiguous way in a shared file, at an offset given by $rank \times 64MB$.

Strided We represent the noncontiguous case by a one-dimensional strided access pattern. Each process issues 256 requests of size 256 KB each.

Our experimental evaluation leverages the concept of Δ-graphs introduced in [1]. For a given configuration of the platform and the microbenchmark, we introduce a delay Δ between the beginning of the I/O burst of the first group of processes and the beginning of the I/O burst of the second group. We then plot the time to complete an I/O phase as a function of Δ.

We note that Δ-graphs *do not represent timelines*; each point in a Δ-graph represents a single experiment.

C. Platform description

The experiments were carried out on the Grid'5000 [8] testbed. We used the Rennes site; more specifically we employed nodes belonging to the *parasilo* and *paravance* clusters. The nodes in these clusters are outfitted with two 8-core Intel Xeon 2.4 GHz CPUs and 128 GB of RAM. We leverage the 10 Gbps Ethernet network that connects all nodes of these two clusters. Reserving these two clusters and deploying our own file system ensured that we were the only users of the network switch as well as the file system at the time of the experiments.

The OrangeFS file system (a branch of PVFS2 [16]) version 2.8.3 was deployed on 12 nodes of the *parasilo* cluster. We considered two types of configuration: "Sync ON" and "Sync OFF," which represent whether each request is immediately flushed to the backend storage devices or whether data can stay in kernel-provided buffers, respectively.

We use 60 nodes (960 cores) to run our microbenchmark on the paravance cluster, unless otherwise specified. These cores will always be split into two groups of equal size (30 nodes) and follow the same type of access pattern.

IV. Discussions of Experimental Results

This section presents the results of our investigation. The first part explores the role that each of the components presented above has in the I/O interference. The second part explains further some of the results, in particular the counterintuitive ones, and highlights a flow-control issue at the core of the interplay between components.

A. Insight into the root causes of I/O interference

We discuss several possible causes for I/O interference.

Device	Alone	Interfering	Slowdown
HDD, sync ON	13.4 sec	33.4 sec	2.49×
SSD	2.27 sec	4.46 sec	1.96×
RAM	1.32 sec	2.09 sec	1.58×

Table I. Time taken by an application running on one core to write 2 GB locally using a contiguous pattern, alone and in the presence of another application performing the same access to another file at the same moment.

1) Influence of the backend storage device: To investigate the I/O interference caused by the storage backend (i.e., disk-level interference), we ran our microbenchmark on the same node as the file system, with the file system deployed on this node only. This removes the network from the factors contributing to the interference and highlights disk-level interference. Each application consists of a single client writing 2 GB contiguously in a file (one file for each client). Table I shows the resulting write time and slowdown for different storage backends: HDD, SSD, and RAM.

We confirm that the use of hard disks leads to an important relative performance degradation in the presence of contention. This interference, less present when using SSDs or local memory, may stem from the additional disk-head movements produced by interleaved requests to distinct data files.

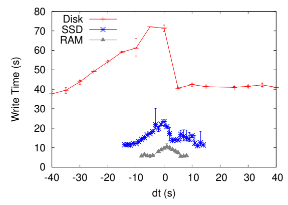
Figures 2 and 3 complement our study of interference at the level of backend storage devices, this time with real parallel applications and file system. In both figures, two applications of the same size (480 cores each) write 64 MB per process, in a contiguous pattern in Figure 2 and in a strided pattern in Figure 3.

As we expect, local memory and SSDs perform better than hard disks. In terms of interference, however, the slowdown is equivalent (up to 2×) regardless of the storage backend for a contiguous pattern.

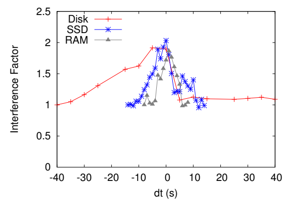
The contiguous scenario with synchronization enabled (Figures 2(a) and 2(b)) shows an interesting result: when using HDDs (and to some extent SSDs), the graph becomes asymmetrical. That is, *the first application entering an I/O phase gets better performance than does the second*, although their I/O patterns are the same. Such unfair interference behavior will appear again in other scenarios throughout this section and will be further explained in Section IV-B.

Experiments with a strided access pattern and sync enabled show that local memory and SSDs have a lower interference factor compared with that of HDDs. This different behavior stems from the greater tolerance of local memory and SSDs to random accesses produced not only by interleaved requests from distinct applications but also by the strided patterns of the applications themselves.

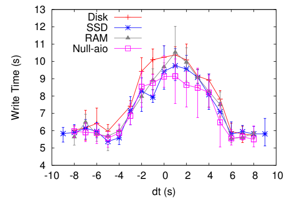
When the synchronization is disabled, we do not observe any significant difference in terms of performance and



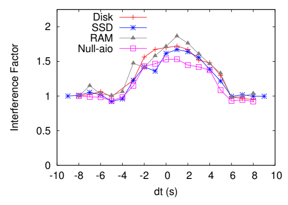
(a) Sync ON (write time)



(b) Sync ON (slowdown)

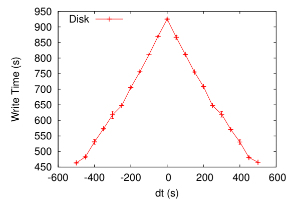


(c) Sync OFF (write time)

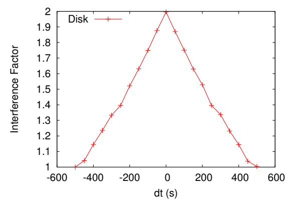


(d) Sync OFF (slowdown)

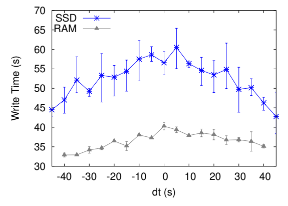
Figure 2. Two applications of the same size (480 cores each) write 64 MB per process using a contiguous pattern. We show how the application behaves for the different storage characteristics: disk, SSD, and RAM. Sync is enabled in (a) and (b) and disabled in (c) and (d). (c) and (d) also display the null-aio method for performing I/O, which simply does no disk I/O at all.



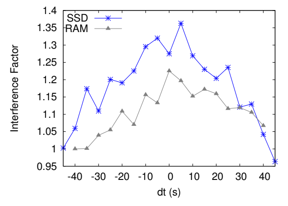
(a) HDD, Sync ON (write time)



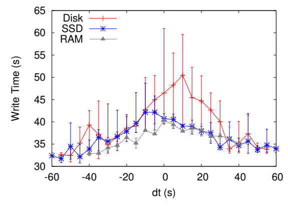
(b) HDD, Sync ON (slowdown)



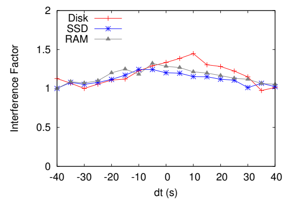
(c) Others, Sync ON (write time)



(d) Others, Sync ON (slowdown)



(e) Sync OFF (write time)



(f) Sync OFF (slowdown)

Figure 3. Two applications of the same size (480 cores each) write 64 MB of data per process to PVFS using a strided pattern. Due to the large write time when using hard disks and synchronization is enabled, we separated the figures for HDDs and other devices. For brevity, only 1 application is shown since both applications have the same size and behavior).

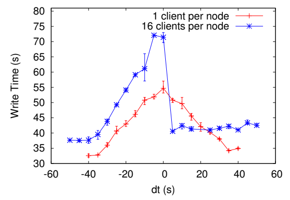


Figure 4. Two applications of the same size (480 cores) write data to PVFS using a contiguous pattern. We show the Δ−graph when all cores participate by writing 64 MB blocks (16 clients per node) and when a single client per node writes 16 blocks of 64 MB.

interference for both access patterns. This is expected since the amount of the generated data is small enough to stay in the local memory when the synchronization is disabled.

Depending on the type of storage device, the access pattern may have an influence on the I/O interference behavior. While the peak interference factor is almost equal for all storage devices with a contiguous pattern, a strided pattern leads to higher interference in HDDs.

Given the regularity and symmetry of the Δ-graph with HDD and synchronization enabled (perfectly triangular figure with an exactly $2\times$ slowdown when both applications start at the same time), we hypothesize that while in these conditions the hard disks are the points of contention, other backends are fast enough to deal with the congestion, and the slowdown observed in these situations comes from another component, such as the network. This could explain the asymmetry in those cases. This hypothesis will be examined in the following two sections.

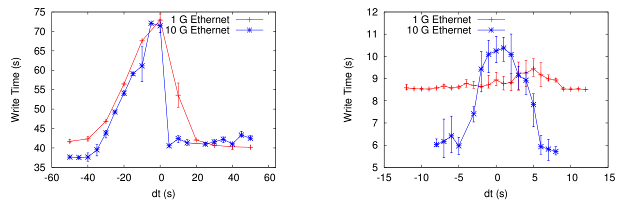
2) Influence of the network interface: The network interface in increasingly multicore nodes is already a limiting factor to single-application I/O performance. We explored its role in cross-application interference. Figure 4 illustrates two scenarios: one in which all cores write 64 MB and one in which one core per node performs an equivalent amount of I/O ($16\times$ 64 MB).

We note that, as expected, the performance without interference is improved by using a single core per node instead of all the cores. This result is in line with the results of our related work focusing on dedicated I/O cores [10].

In terms of interference, having all the cores perform I/O not only produces more interference but also leads to unfairness. Indeed the interference pattern observed in Figure 4 with 16 writers per node is asymmetrical, which shows that the first application entering an I/O burst performs better than the one that follows it.

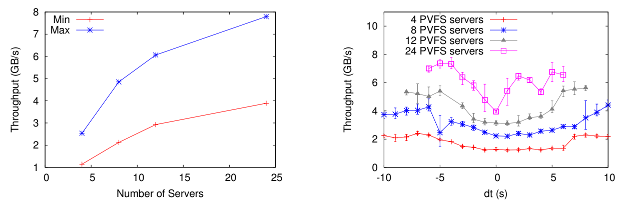
While it was already known that fewer writers per multicore nodes (e.g., aggregators or dedicated I/O processes) improve the I/O performance of a single application, we have shown here that this approach also lowers cross-application interference.

3) Influence of the network: We can hardly rule out the network from the HPC system because it provides the link between the computation and the storage nodes. Hence, we



(a) Sync ON (b) Sync OFF

Figure 5. Two applications of the same size (480 cores each) write 64 MB per process using a contiguous pattern. Sync is enabled in (a) and disabled in the (b). We show the $\Delta-$graph when the network bandwidth is 10 G (default) and adjusted to the 1 G Ethernet.



(a) Scaling (b) Δ-graph

Figure 6. Two applications of the same size write data to the PVFS using a contiguous pattern. Figure (a) shows the maximum throughput achieved (when the application is alone) and the minimum (in contention) as a function of the number of servers. Figure (b) shows the throughput for one of the applications in a Δ-graph, that is, as a function of the delay between applications.

Number of Servers	Interference Factor
4	2.22
8	2.28
12	2.07
24	2.00

Table II. Peak interference factor observed by the application for different numbers of storage servers.

highlighted its role by decreasing the network bandwidth from 10 G to 1 G. Figure 5 shows the results for the different network bandwidths, with two applications writing in a contiguous pattern. Surprisingly, we discover that having a higher network bandwidth neither significantly improves the applications' I/O performance nor helps eliminate the interference. On the contrary, limiting the network bandwidth to 1 G helped eliminate the interference when synchronization was disabled in the disks, as shown in Figure 5(b), and helped regain a symmetrical (fair) interference behavior when synchronization was enabled, as shown in Figure 5(a).

In Figure 5(a), the peak write time in the presence of contention is the same whether we use a 10 G or a 1 G network. The reason is that the performance of the I/O path here is limited by the backend storage devices (HDDs). In Figure 5(b), the data is not synchronized to disks right away when reaching storage servers but stays in buffers. Hence, the network becomes the limiting factor.

The flat Δ-graph observed with a 1 G network stems from the fact that the network is limiting the rate at which each application sends requests to the file system, producing an interference-free behavior. Interesting is the fact that the resulting write time is in some cases smaller than when using a 10 G network, which hints that *constraining the rate at which each application can send data is a valid solution for mitigating interference*.

The fairness regained in Figure 5(a), resulting from the interplay between the storage devices and the network, will be further explained in Section IV-B.

Counterintuitively, a lower network bandwidth may not cause higher interference. On the contrary, it can prevent interference if none of the other components are subject to contention.

4) Influence of the number of storage servers: Intuitively and assuming the network is not a point of contention, using more storage servers increases the aggregate throughput that any single application can achieve. This situation is demonstrated by the maximum throughput achieved in Figure 6(a). In terms of interference, however, it is not clear whether more servers will mitigate the interference.

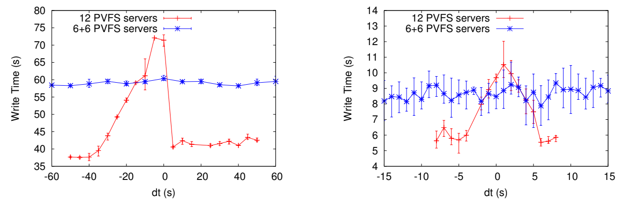
We therefore investigated the role of the number of servers on the interference by deploying PVFS on 24, 12, 8, and 4 nodes with synchronization turned off. Each client writes 64 MB in a contiguous pattern for the first three deployments and writes 32 MB with 4 PVFS servers because of its lower capacity. Figure 6(b) shows the observed throughput for one of the applications depending on the number of PVFS servers used and on Δ. As expected, increasing the number of servers improves the throughput, but it cannot eliminate the interference. I/O interference still exists because each server still has the same number of clients regardless of the number of servers.

Table II summarizes the peak interference factors observed for each number of servers. As we can see, the number of servers does not influence the interference factor much.

Increasing the number of servers does not affect the relative performance degradation generated by cross-application interference.

One could argue that this result would not be true for small applications that cannot get full parallelism from the maximum number of storage servers. Yet as we build larger machines, the number of storage servers tends to get smaller relative to the number of computation nodes, and we tend to run larger applications that become quickly limited by this small number of servers.

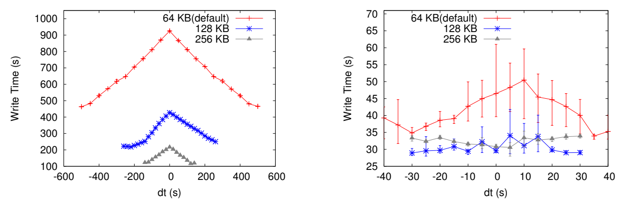
5) Influence of targeted storage servers: In our previous set of experiments, both applications were writing to all



(a) HDD, sync ON

(b) RAM

Figure 7. Two applications of the same size (480 cores each) write 64 MB per process using a contiguous pattern. We show the $\Delta-$graph when both applications use the same set of PVFS servers (12 PVFS servers) and when each application targets different set of PVFS servers (6+6 PVFS servers) for the two different storage backends.



(a) Sync ON

(b) Sync OFF

Figure 8. Two applications of the same size (480 cores each) write 64 MB of data to the PVFS using a strided pattern, with different stripe sizes on the server side. Synchronization is enabled in (a), and disabled in (b).

servers available. In this section, we split the 12 PVFS servers into two groups so that each application targets a different group of servers. Our idea is to remove the servers and disks from the possible points of contention, leaving only the network as a shared component between the two applications.

Figure 7 shows the results for two different storage backends. As expected, using 2× fewer servers decreases the performance of a single application.

The behavior with respect to the interference is more interesting, however. We observe that making each application target a different set of servers removes the interference. In some cases, the interference observed under contention for all 12 servers leads to a higher write time than does using 6 separate servers for each application. This result would motivate approaches that detect potential congestion and partition the storage space across applications instead of letting applications interfere.

Again, the unfairness observed in Figure 7(a) when both applications target all servers is eliminated when they access different sets of servers.

Making distinct applications target distinct sets of servers is a valid solution to at least control if not mitigate the level of interference.

6) Influence of the data distribution policy: Data files are distributed across PVFS servers in a round-robin fashion with a predefined stripe size. Changing this stripe size can have a significant impact on the resulting performance. Hence we wanted to check its influence on the interference as well.

Figure 8 illustrates the interference pattern with stripe sizes of 64 KB, 128 KB, and 256 KB for a PVFS deployment with disk and synchronization enabled (Figure 8(a)) and disabled (Figure 8(b)). Note that 64 KB is the default stripe size and that each application writes 64 MB per process in a strided pattern with 256 KB of blocks.

A stripe sizes higher than the default one leads to significant performance improvements for both cases. However, the interference seems to disappear when using a larger stripe size with synchronization turned off. We hypothesize that this lower interference stems from the smaller number of servers that each request is striped across. When a large request is issued by a client, this request is split into smaller requests sent in parallel to several servers. The operation completes only when all these servers have treated their part of the initial request. Hence, any slowdown experienced by a single server as a result of contention leads to a global slowdown for the entire operation. Provided that two servers decide to serve requests from different applications in a different order, both applications will suffer from a slowdown observed in servers that have not prioritized their request.

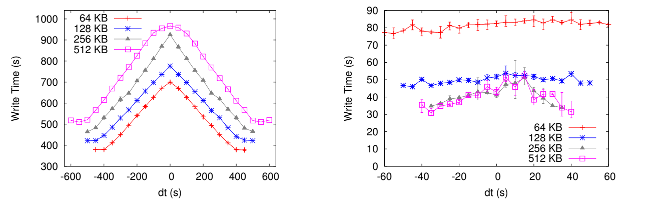
Here each 256 KB request is striped across 4 servers for the default stripe size of 64 KB. This is reduced to 2 servers with a 128 KB stripe size and to 1 with a 256 KB stripe size. Hence, we see the performance improvement for both cases and the removal of I/O interference for the disabled sync case. We believe that interference still exists for the other scenario since disk is still an active component and contributing to the I/O interference.

We confirm that making all servers treat requests from distinct applications in the same order, as done in [3], is an appropriate way of mitigating I/O interference.

7) Influence of the request size: Similarly to the stripe size in the file system, the original request size in applications has an impact on I/O performance. Figure 9 illustrates the interference patterns when each application writes 64 MB in a strided pattern with a block sizes of 64 KB, 128 KB, 256 KB, and 512 KB. The stripe size in PVFS is set to the default of 64 KB.

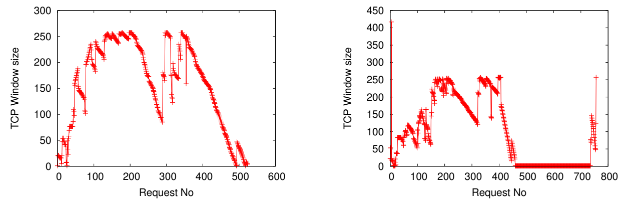
The best performance is achieved for small block sizes when synchronization is enabled (Figure 9(a)), whereas it is achieved with large block sizes when synchronization is disabled (Figure 9(b)).

While the interference pattern shows a fair, proportional sharing of resources for all block sizes when synchronization is enabled (symmetric, triangular figure), when synchro-



(a) Sync ON (b) Sync OFF

Figure 9. Two applications of the same size (480 cores each) write 64 MB of data to the PVFS using a strided pattern with a block sizes of 64, 128, 256 and 512 KB. Synchronization is enabled in (a) and disabled in (b). These graphs show the write time for the application (for brevity, only 1 application is shown since both applications have the same size) depending on the block size used and on dt.



(a) Independent run (b) Interfering

Figure 10. TCP windows sizes of each request during the 64 MB contiguous data write from a client to the server (a) when the application is running independently and (b) interfering with the other application, which also has same size (480 cores each) and started at the same time (dt=0).

nization is disabled the interference pattern disappears for block sizes of 64 and 128 KB. This result is in line with our observations made in Section IV-A6 and the fact that such request sizes have fewer servers involved in each I/O operations. Yet while these small request sizes remove the interference, they are far from optimal for a single application.

The fact that no interference is observed between two applications does not mean that optimal performance is achieved. Our experiments show that, while some request sizes allow cross-application interference to be mitigated because clients interact with fewer servers for each request, these block sizes remain far from optimal from a single-application perspective.

Following this observation, *we warn any researcher proposing solutions to the I/O interference that these solutions should be validated in configurations that are already as good as possible, if not optimal, for a single application alone*. Indeed one can claim that a solution removes the interference, while much higher performance could actually be obtained from each application individually by better optimizing their access patterns.

B. Unexpected behaviors: a flow-control issue

Some of the results of the previous section remain unexplained, such as the unfairness of some scenarios, with the application that starts first getting better performance than the one starting second.

1) The Incast issue: An unfair behavior results from a component that adapts to the workload *over time*. Since PVFS does not implement any particular scheduling mechanism at the server side, there is no reason to think one flow of requests from an application would be prioritized over another. The prioritization of one flow over another cannot result from the backend storage device either, since this storage device sees only serial accesses from a single program: the PVFS server. Hence we suspected that such unfair behaviors stem from the network.

To confirm this hypothesis, we reran the experiment presented in Figure 2(a), in which two applications of 480 cores each write 64 MB per process in a contiguous manner, in a PVFS file system consisting of 12 servers. We examined more closely the TCP packets exchanged between a client of either application and a PVFS server, using *tcpdump*.

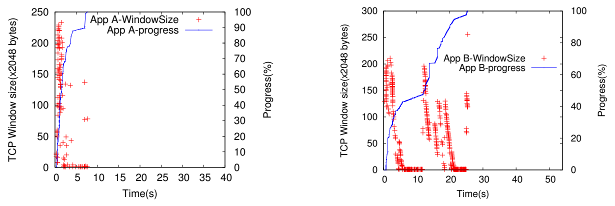
Figure 10(a) shows the evolution of the TCP window size for the sequence of requests issued by one client to one server, when an application runs alone. Figure 10(b) shows the evolution of the TCP window size when the application is interfering with another one. As we can see, the behavior is similar except for the fact that, under contention, the window size drops to nearly 0, making it difficult for the client to eventually send all its data.

The collapse of the TCP window size as a result of contention was shown by Phanishayee et al. [17], who termed it the “Incast problem.” When many clients access to a server, the TCP congestion control mechanism at this server forces the window size to drop in all its opened sockets, leading to an important loss of performance.

Note that this phenomenon does not stem from the network alone (we have seen, by splitting the set of servers into two groups, that the network is not a point of contention). *It comes from the interplay between the network and the disks*, as well as the lack of flow control mechanism in Trove, the component of PVFS that forwards requests from sockets down to the storage devices. Because disks are slow, Trove cannot keep up with the flow of incoming requests and hence relies on the TCP congestion control mechanism to limit the flow the requests from all clients.

The fact that the Incast problem appeared only with HDD and synchronization enabled, but not with RAM or SSD, proves that this type of interference results from the interplay between several components of the I/O path. What appears to be a network congestion issue can actually stem from bad flow-control induced by slow backend storage devices.

2) From Incast to unfairness: This Incast issue explains many of the results presented in the previous section, starting



(a) Application A (b) Application B

Figure 11. TCP windows sizes and progress of the data transfer from one client of each application and one of the servers.

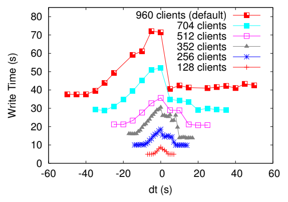


Figure 12. Δ-graph illustrating the appearance of the Incast problem as we increase the number of clients. Each application writes 64 MB per process in a contiguous pattern. PVFS is deployed on 12 servers with hard disks as backend, synchronization enabled. The number of clients shown is the total number of clients.

with the unfairness observed in some scenarios. Figure 11 shows the behavior of two applications, one of which starts 10 seconds after the other. We plot the TCP window size and the progress of the I/O transfer to this server as a function of time, for one client of each application. Whereas the first application starts seeing a slowdown of its progress at around 90 percent, the slowdown can be observed at 40 percent for the second application; indeed, the window size hardly manages to get back to a high value.

The appearance of unfair behavior as a result of Incast is a good way to evaluate the conditions that cause Incast to appear. For example, Figure 12 shows the interference behavior when running different numbers of clients. An interesting observation is that while the unfair behavior benefiting the first application is clear when using 960 or 704 clients in total, the trend seems to reverse at smaller client counts (256 to 512 clients), where the first application is more impacted than the second one.

We hypothesize the following explanation. At large client counts, the TCP window size of the second application immediately collapses as a result of contention, allowing the first application to complete seemingly without contention. At intermediate client counts, the TCP window size is reduced in both applications but does not collapse. Thus the first application is impacted as well. At small process counts, the servers are able to handle all the requests without having to shrink the window size. The interference observed becomes that of the backend storage devices.

As the number of clients increases, we are more likely to observe degenerated flow-control issues as a result of the file system not being able to handle the load.

3) Explanation of counterintuitive results from a flow-control perspective: The interplay between components resulting in a bad flow control can explain both the intuitive and counterintuitive results obtained previously.

Using one core per node instead of all cores to perform I/O, as done in Section IV-A2, reduces the number of sockets involved in an application's I/O phase and forces a serialization of requests at the level of each node. This constrains the rate at which each single node can write and therefore prevents the Incast problem from happening.

Using a low-bandwidth network, as done in Section IV-A3 with a 1 G network instead of a 10 G, also mitigates the Incast problem by constraining the rate at which each client can send requests. By forcing a reduction of bandwidth at the source, the rate of requests becomes sustainable to the backend storage devices and thus the TCP window size does not collapse.

Splitting the servers into two groups completely prevents the interference from happening (Figure 7(a)) because a server has to interact with 2× fewer clients, therefore maintaining flow control on 2× fewer links.

V. Related Work

As we move toward the exascale era, performance variability in HPC systems remains a challenge. Cross-application I/O interference is one of the major causes of this performance variability. A large body of studies have sought to eliminate cross-application I/O interference by focusing on possible sources of this interference. For example, Zhou et al. [18] present an I/O-aware batch scheduler that addresses the interference problem at the level of batch scheduling. The batch scheduler schedules and coordinates the I/O requests on the fly by considering the system state and I/O activities. Gainaru et al. [19] show the performance degradation due to I/O congestion and propose a new scheduler that tries to eliminate this congestion by coordinating the I/O requests depending on the application past behaviors and system characteristics. Boito et al. [11] propose AGIOS, an I/O scheduling library for parallel file systems. AGIOS incorporates the applications' access pattern information into the scheduler based on the traces generated by the scheduler itself and uses this information to coordinate the I/O requests in order to prevent congestion to the file system. As we observed, however, although scheduling-level solutions can help control the level of interference, it does not always lead to improved performance at the same time.

Some works focus on finding solutions at the disk level, the lowest level that I/O interference can occur in the I/O stack. Zhang and Jiang [20] point out that frequent disk head seeks, because of the access interference on each I/O node, can seriously hurt the performance of a system. They propose a data replication scheme, InterferenceRemoval, to eliminate I/O interference. InterferenceRemoval tries to limit the number of the I/O requests served by each I/O node. Although this solution is in parallel with the Incast problem we presented in our work, we observe that it is not present for only a single source (e.g., disk) of interference.

Some research efforts consider network contention as the major contributor to the I/O interference. Bhatele et al. [21] investigated the performance variability in Cray machines and found out that the interference of multiple jobs that share the same network links is the primary factor for high performance variability. Jokanovic et al. [22] introduce the concept of quiet neighborhoods, a job allocation technique based on the job sizes. This technique helps control the fragmentation in the HPC systems and reduces the number of jobs sharing the network, with the aim of minimizing the interference.

Some works study the interference problem with a special emphasis on a single factor. Kuo et al. [23] investigated the influence of the file access pattern on the degree of interference observed. They found out that chunk size can determine the degree of interference and that the interference effect induced by various access patterns in the HPC system can slow the applications by a factor of 5. Our work is different in the targeted objective, since we try to identify all possible sources of interference under various scenarios, as well as their interplay.

Although indeed important, the aforementioned studies –by focusing only on a single potential source– do not necessarily provide a complete solution for the interference problem. In contrast, we consider the possible sources of interference together and conduct an extensive experimental study. Thus, our work can provide useful insights into the I/O interference phenomenon. Furthermore, it can help researchers tackle the interference problem across all components of the I/O system.

VI. Conclusion and Future Work

Cross-application interference in HPC systems is an important problem that can affect the efficiency of an entire machine. This problem will be even more important with exascale machines that will allow more applications to run concurrently. In this work, we investigated the potential root causes of I/O interference. Our findings demonstrate that interference results from the interplay between several components in the I/O stack. For instance, we observe that the impact of the request size on interference varies depending on the configuration of components in the I/O path. Our findings also illustrate many counter-intuitive results besides the intuitive ones. For example, we show that using a low-bandwidth network in some scenarios can eliminate the interference problem, which stems from the interplay between the different points of contention. Hence, we believe that researchers must understand the tradeoffs between several components in the I/O stack and must address the interference problem in its entirety, rather than focusing on any single component.

Several avenues remain open for future work. One is to expand our experimental study by looking at other platforms than Grid'5000, other file systems (e.g., Lustre), other workload types (e.g., read-only) and other types of network (e.g., InfiniBand). As the ultimate goal, by leveraging the knowledge gained in our work, we plan to design event-driven simulators that can accurately model the components subject to interference.

Acknowledgment

We thank Rob Latham and Phil Carns from Argonne National Laboratory for their valuable technical insights into PVFS, and Gail Pieper for her comments on our paper. This work was done in the framework of a collaboration between the KerData joint Inria - ENS Rennes - Insa Rennes team and Argonne National Laboratory within the Joint Laboratory for Extreme-Scale Computing, and the Data@Exascale associate team. The experiments presented in this paper were carried out using the Grid5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several universities as well as other organizations (see http://www.grid5000.fr/). This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-06CH11357.

References

[1] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '14)*, Phoenix, AZ, USA, May 2014. [Online]. Available: http://hal.inria.fr/hal-00916091

[2] X. Zhang, K. Davis, and S. Jiang, "IOrchestrator: Improving the Performance of Multi-Node I/O Systems via Inter-Server Coordination," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.30

[3] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "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. New York, NY, USA: ACM, 2011, pp. 17:1–17:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063407

[4] Y. Qian, E. Barton, T. Wang, N. Puntambekar, and A. Dilger, "A Novel Network Request Scheduler for a Large Scale Storage System," *Computer Science - Research and Development*, vol. 23, pp. 143–148, 2009. [Online]. Available: http://dx.doi.org/10.1007/s00450-009-0073-9

[5] A. Lebre, G. Huard, Y. Denneulin, and P. Sowa, "I/O Scheduling Service for Multi-Application Clusters," in *in Proceedings of IEEE Cluster 2006*, 2006.

[6] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey, “CA-NFS: A Congestion-Aware Network File System,” in *Proceedings of the 7th conference on File and storage technologies*, ser. FAST ’09. Berkeley, CA, USA: USENIX Association, 2009, pp. 99–110. [Online]. Available: http://dl.acm.org/citation.cfm?id=1525908.1525916

[7] Y. Tanimura, R. Filgueira, I. Kojima, and M. Atkinson, “Poster: Reservation-Based I/O Performance Guarantee for MPI-IO Applications Using Shared Storage Systems,” in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, 2012, pp. 1384–1384.

[8] INRIA, “Grid’5000: http://www.grid5000.fr.”

[9] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, “Managing Variability in the IO Performance of Petascale Storage Systems,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–12.

[10] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, “Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-Free I/O,” in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, Sept. 2012, pp. 155–163.

[11] F. Zanon Boito, R. Kassick, P. O. A. Navaux, and Y. Denneulin, “AGIOS: Application-Guided I/O Scheduling for Parallel File Systems,” in *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*. IEEE, 2013, pp. 43–50.

[12] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, “Scalable I/O Forwarding Framework for High-Performance Computing Systems,” in *Cluster Computing and Workshops, 2009. CLUSTER ’09. IEEE International Conference on*, Sept. 2009, pp. 1–10.

[13] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. Papka, R. Ross, and K. Yoshii, “Accelerating I/O Forwarding in IBM Blue Gene/P Systems,” in *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2010, pp. 1–10.

[14] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W. E. Nagel, and S. Poole, “Enabling Event Tracing at Leadership-Class Scale Through I/O Forwarding Middleware,” in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’12. New York, NY, USA: ACM, 2012, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2287076.2287085

[15] H. Shan and J. Shalf, “Using IOR to Analyze the I/O Performance for HPC Platforms,” in *Cray User Group Conference 2007*, Seattle, WA, USA, 2007.

[16] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, “PVFS: a Parallel File System for Linux Clusters,” in *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*. Berkeley, CA, USA: USENIX Association, 2000.

[17] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, “Measurement and Analysis of TCP Throughput Collapse in Cluster-Based Storage Systems,” in *FAST*, vol. 8, 2008, pp. 1–14.

[18] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan, “I/O-Aware Batch Scheduling for Petascale Computing Systems,” in *IEEE International Conference on Cluster Computing*, 2015.

[19] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, “Scheduling the I/O of HPC Applications Under Congestion,” in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2015, pp. 1013–1022.

[20] X. Zhang and S. Jiang, “InterferenceRemoval: Removing Interference of Disk Access for MPI Programs through Data Replication,” in *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 2010, pp. 223–232.

[21] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, “There Goes the Neighborhood: Performance Degradation due to Nearby Jobs,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 41.

[22] A. Jokanovic, J. Sancho, G. Rodriguez, A. Lucero, C. Minkenberg, and J. Labarta, “Quiet neighborhoods: Key to protect job performance predictability,” in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2015, pp. 449–459.

[23] C.-S. Kuo, A. Shah, A. Nomura, S. Matsuoka, and F. Wolf, “How File Access Patterns Influence Interference among Cluster Applications,” in *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2014, pp. 185–193.