Leveraging Burst Buffer Coordination to Prevent I/O Interference

Anthony Kougkas*†, Matthieu Dorier†, Rob Latham†, Rob Ross†, and Xian-He Sun*
*Illinois Institute of Technology, Department of Computer Science, Chicago, IL akougkas@hawk.iit.edu, sun@iit.edu
†Argonne National Laboratory, Mathematics and Computer Science Division, Lemont, IL {mdorier, robl, rross}@anl.gov

***Abstract*—Concurrent accesses to the shared storage resources in current HPC machines lead to severe performance degradation caused by I/O contention. In this study, we identify some key challenges to efficiently handling interleaved data accesses, and we propose a system-wide solution to optimize global performance. We implemented and tested several I/O scheduling policies, including prioritizing specific applications by leveraging burst buffers to defer the conflicting accesses from another application and/or directing the requests to different storage servers inside the parallel file system infrastructure. The results show that we mitigate the negative effects of interference and optimize the performance up to 2x depending on the selected I/O policy.**

Keywords*—*I/O Interference; Parallel File Systems; I/O Policies; I/O Staging; Burst Buffers

I. Introduction

Large-scale applications already individually suffer from unmatched computation and storage performance, leading to a loss of efficiency in I/O-intensive phases. But another problem appears when several applications compete for access to a common parallel file system, leading to further degradation of I/O performance as a result of contention. Most modern supercomputers have moved from the paradigm of one large application using the entire machine to one where many smaller applications run concurrently. In [1], we see that half of the jobs executed on Argonne's Intrepid machine were using less than 2048 cores (i.e. only 1.25% of the entire available cores); and since many of the same applications were ported to its successor, Mira, we suspect the pattern on this new system to be no different. Consequently, multiple applications commonly run concurrently and share the underlying storage system. This practice, however, can severely degrade the I/O bandwidth that each application experiences. This phenomenon, called *cross-application I/O interference*, stems from diverse sources: network contention at the level of each storage server, poor scheduling decisions within the storage service (i.e., parallel file system) leading to different servers servicing requests from distinct applications in a different order, or additional disk-head movements when interleaved requests from distinct applications reach the same storage device.

The use of *burst buffers* in HPC systems [2], [3] has emerged with the initial goal to relieve the bandwidth burden on parallel file systems by providing an extra layer of low-latency storage between compute and storage resources. The Cori system at the National Energy Research Scientific Computing Center (NERSC) [4], uses CRAY's Datawarp technology [5]. The Los Alamos National Laboratory Trinity supercomputer [6] will also use burst buffers with a 3.7 PB capacity and 3.3 TB/s bandwidth. Intel has also discussed the use of burst buffer nodes under the new Fast Forward storage framework [7] for future HPC systems. One common characteristic in all these use cases is to increase the total I/O bandwidth available to the applications and optimize the input/output operations per second (i.e., IOPS). In addition to serving as a pure storage option, the notion of a burst buffer can make storage solutions smarter and more active.

In this paper, we address the problem of cross-application I/O interference by coordinating burst buffer access to prevent such I/O degradation. We propose several ways to mitigate the effects of interference by *preventing applications from accessing the same file system resources at the same time*. We build on our previous work leveraging cross-application coordination [1] and propose three new strategies to mitigate interference. Two of these strategies are based on the use of burst buffers to delay actual accesses to the storage system when multiple applications are interfering. The third strategy dynamically partitions the parallel file system's resources in order to dedicate distinct subsets of storage servers to each application when contention is detected. In summary, the contributions of this paper are: (i) we propose the coordination of burst buffer access to manage concurrent accesses to the underlying storage system (Section III); (ii) we design and implement several strategies to mitigate the effects of I/O interference (Section IV); (iii) and we evaluate these strategies with several microbenchmarks and show their results (Section V). Section VI presents related work, and Section VII summarizes our conclusions and briefly discusses future work.

II. Background and Motivation

A. I/O Interference

Supercomputers generally are designed for maximum computing power to solve a small number of large, tightly-coupled and compute-intensive problems. While computing and network resources can be shared effectively by state-of-the-art job schedulers, the same cannot be said about the storage resources (i.e., shared parallel file systems). In fact, [8] and [9] suggest that I/O congestion, within and across independent jobs, is one of the main problems for future HPC machines. A significant source of performance degradation seen on the Jaguar supercomputer at Oak Ridge National Laboratory was identified as concurrent applications sharing the parallel file system [10].

The I/O performance degradation is caused by contention for resources. These resources include network hardware used for I/O requests, file system servers that are responsible for metadata operations and other I/O requests, the servers that are responsible for committing the I/O requests to the underlying storage devices, and the storage media itself, as well as virtual resources such as locks that are used to manage distributed accesses [11], [12]. In this study we focus on the file system level. The main cause of interference in this level is how

the parallel file system services I/O requests from multiple applications (i.e., the internal scheduler).

Prior work addressed this contention through storage server coordination, where the basic idea is to serve one application at a time in order to reduce the completion time and, in the meantime, maintain the server utilization and fairness [13]. However, applications still experience a bandwidth reduction since the storage resource is still shared. Other work suggested solutions in the file system scheduler [14], [15], [16]. However, those solutions need to be integrated into the parallel file system's server code. In this paper, we propose that by preventing applications from concurrently accessing the same storage resources, we can exploit the full potential of existing parallel file systems and allow the system to provide higher global I/O throughput across multiple applications.

B. Burst Buffers

Scientific applications often demonstrate bursty I/O behavior [17], [18]. Typically in HPC workloads intense, short phases of I/O activities, such as checkpointing and restart, periodically occur between longer computation phases [19], [20]. New storage system designs that incorporate non-volatile burst buffers between the main memory and the disks are of particular relevance in mitigating such burstiness of I/O [21].

Burst buffers as an intermediate storage tier located between RAM and spinning disks are designed to help scientific applications in many ways: improved application reliability through faster checkpoint-restart, accelerated I/O performance for small transfers and analysis, fast temporary space for out-of-core computations and in-transit visualization and analysis [4]. The most commonly used form of a burst buffer in current HPC systems is dedicated *burst buffer nodes* [2], [22]. These nodes can exist in the I/O forwarding layer or as a distinct subset of the computing nodes (i.e., not part of the compute resources but responsible for acting as burst buffer nodes) or even as entirely different nodes close to the processing resources (i.e., extra resources if available). A a burst buffer can also be located inside the main memory of a computing node or as a fast non-volatile storage device placed in the computing node (i.e., NVRAM devices, SSD devices, PCM devices etc). Besides the above, the main functionality of burst buffers is to quickly absorb I/O requests from the computing elements and asynchronously issue them to the parallel file system (PFS), allowing the processing cores to return faster to computation. In this paper, we propose to use such burst buffers and, by coordinating them, tackle the I/O interference caused by multiple applications running concurrently in the system.

III. Our Approach

Burst buffers are placed between the application and the parallel file system layer. Thus they are naturally suitable to act as *I/O traffic controllers* and prevent applications from accessing the underlying file system resources at the same time. Burst buffers can achieve this objective by making a certain application stage the I/O (i.e., buffer the requests) while another one is accessing the shared parallel file system. We argue that when concurrent accesses from multiple applications are detected, we can avoid the undesirable I/O interference by dynamically changing the data distribution to the PFS, specifically by using burst buffers to direct I/O traffic to the underlying resources in a nonconflicting manner.

By coordinating burst buffers, we can provide the much needed global, system-wide view of running applications with concurrent accesses to the underlying parallel file system and prevent I/O interference in this level by employing the following policies.

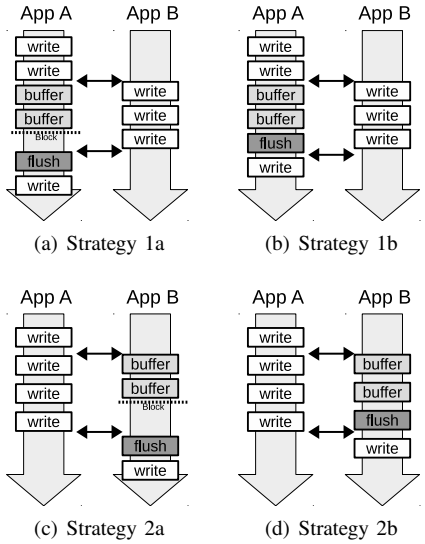


Fig. 1: Buffer-based coordination strategies. Black arrows correspond to cross-application communications that allow them to know when to switch their buffering system on or off.

A. I/O Staging Policies

Previously proposed strategies, which consist of simply blocking one application for the benefit of the other (*first come - first served* order or *interruption* of the running application), have the disadvantage of completely blocking one application while it could actually perform some computation. As an example, if the I/O phase consists of compressing and writing chunks of data (as implemented in HDF5), the application could compress multiple chunks and stage them when there is contention, instead of blocking on a write operation. Another example is in the case of collective I/O [23] (in particular two-phase I/O) where instead of blocking on the first write, multiple rounds of communication could be completed before writing is performed.

We introduce two strategies based on I/O staging to prevent or mitigate I/O interference. Figure 1 demonstrates our approach for the cases where an application has some other options for making progress instead of waiting for another application to complete its I/O operations. Using the burst buffers, an application waiting for access to the file system can actually *stage* its I/O operations locally and execute them later, that is, block only when closing the file or forcing a flush.

The goal for these strategies is to prevent applications from accessing the underlying storage system at the same time while allowing them to do something else and not wait for the resource being blocked. Considering two applications A and B, where A starts its I/O phase before B, we hold the following assumptions: (a) applications notify each other in a timely manner about their respective I/O intentions (e.g., entering an I/O phase); (b) applications, when instructed to start staging their I/O, have some other work to perform (e.g.,

some computation); and (c) while an application is staging, its I/O consists of write-only phases. We propose two strategies with two variations each as follows.

Strategy 1a: Application A starts staging its operations as B enters an I/O-intensive phase. It blocks if necessary and flushes only after B has completed its I/O phase. It continues writing after flushing if its operations are not completed. This strategy is shown in Figure 1(a).

Strategy 1b: Application A starts staging its operations as B enters an I/O-intensive phase. It then flushes its buffer when it has no more available work even if B has not completed its operations yet. This strategy is shown in Figure 1(b).

Strategy 2a: As B starts its I/O phase, it learns that A is already accessing the file system and therefore starts staging its I/O requests. It blocks if necessary and flushes only after A has completed its I/O phase. It continues writing after flushing if its I/O operations are not completed. This strategy is shown in Figure 1(c).

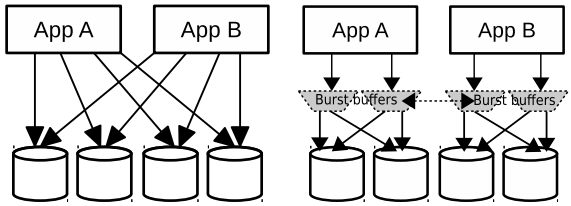
Strategy 2b: When B starts its I/O phase, it discovers that application A is already accessing the file system; therefore, it starts staging its I/O requests. It flushes the buffers as soon as it has no more available work, even if A has not completed its I/O phase. It continues writing if its I/O operations are not completed. This strategy is shown in Figure 1(d).

We propose these strategies having in mind different classes of applications where the I/O phase might be time sensitive (i.e., they cannot wait for some other application to finish) or where more bursty behavior is present in one of them. The two different approaches in each strategy, *stage and block* or *stage and flush*, offer greater flexibility to the system in order to execute concurrent applications and get the maximum I/O throughput from the PFS.

B. Dynamic Partitioning of PFS

The dynamic partitioning strategy involves partitioning in space rather than in time. Figure 2 illustrates our approach. We shift from both applications accessing all available servers to partitioning the PFS into distinct subsets and directing each application's requests to different sets of storage servers. The intuition behind the benefits of this strategy is that the performance of a parallel file system usually does not scale linearly with the number of storage servers accessed by an application. Preventing applications from accessing the same set of servers will prevent interference at the level of storage servers and their disks, leaving only the network as a potential source of contention, however a reduced set of storage servers will offer a lower bandwidth than will the full set. We distinguish two variations of this strategy.

Strategy 3a: We define a static, predefined partitioning where both applications access different storage servers from the beginning until the end of their execution. Even though fewer storage servers can offer less bandwidth to the application, the prevention of I/O interference in the file system level (i.e., exclusive access to the disks) might be enough to outperform the use of the entire PFS installation. Splitting the servers into disjoint sets can be performed according to several criteria; it depends on the available knowledge of the application's I/O needs, scale, priority, and so forth. In this study, we explored proportional sharing of the available storage servers according to the dataset size (i.e., total amount of data) and the application size (i.e., number of MPI processes). For instance, for 8 available storage servers; consider that application A writes 48 MB per process and B writes 16 MB. For the same number of processes, application A would write on 6 servers and B on the remaining 2.



(a) Without partitioning (b) With partitioning

Fig. 2: Partition-based strategy. Instead of accessing all the storage servers, applications communicate and agree to interact with nonconflicting subsets of servers.

Strategy 3b: The second variant is dynamic partitioning of the file system when interference is detected. Burst buffers can make intelligent decisions; and, when appropriate (i.e., when multiple applications try to access the file system at the same time), they redirect I/O into separate and distinct subsets of storage servers for each application. When no contention exists, applications are exposed to the entire set of available servers in order to achieve maximum bandwidth. This strategy is appropriate only for write workloads, since read workloads are tied to the set of servers where the required data is stored. In our previous example, consider that application A starts writing to all 8 servers; when B starts an I/O phase, A directs all its I/O requests to 6 storage servers and B writes to the remaining 2. After B finishes, A goes back to writing on all 8 servers.

IV. Design and Implementation

In this section we present our design and implementation for our proposed solution and how we enable those coordination strategies through our library.

A. Design Overview

To evaluate the various strategies presented in Section III, we implemented a userspace buffering system, called BBIO (Basic Buffered I/O)[1] working under the POSIX and LibC interfaces. BBIO is a library comprising two parts. Its static part *libbbio.a* can be linked to any code and provides the user-level interface to initialize and control buffers. The dynamic part *libbbio_posix.so* can be preloaded to replace existing POSIX and LibC functions (such as `write` and `fwrite`). Hence, the application will buffer its I/O only if the dynamic library is preloaded, making the use of BBIO as simple as setting an environment variable.

The FILE structure provided by the standard C library already provides a local memory buffer. Although the user can control the size of this buffer, the user has no control over *when* the program will decide to flush it. In contrast, BBIO allows the user to control both the size of the buffer allocated to a file where this buffer is located and the moment the buffer can be flushed. It can also leverage local storage devices such as SSD instead of relying on local memory.

[1]Our implementation is available at https://bitbucket.org/mdorier/bbio.

B. Interface and API

Our BBIO library presents the following interface to application developers.

- **BBIO_Init(const char* path, size_t size)**: initializes BBIO, giving it the path to a directory where it can write buffered data (path to an SSD, for instance). Leaving this path NULL instructs BBIO to use RAM as storage. The size provided is the maximum size allowed for a buffer associated with any single file descriptor. Whenever the buffer is filled or if a write is issued with a size that cannot fit the buffer, the buffer automatically resizes if there is available space or is flushed.
- **BBIO_Enable(int fd)**: enables buffering for a particular file descriptor. By default, buffering is enabled for all files outside of system directories.
- **BBIO_Disable(int fd)**: disables buffering for a particular file descriptor. If buffering was previously enabled and some data have been put in the buffer, the buffer is flushed.
- **BBIO_Flush(int fd)**: forces a flush on the buffer associated with the file descriptor.
- **BBIO_Finalize()**: finalizes BBIO. It will flush all the buffers currently managed.
- **BBIO_On_flush(int fd, BBIO_Callback cb)**: installs a callback function that will be called before any flush (whether this flush is triggered manually by BBIO_Flush, BBIO_Disable, or BBIO_Finalize or automatically when the buffer is full). The callback will not be called when trying to flush an empty buffer. The BBIO_Callback object must have the signature `void ()(int fd)`.

Using this interface lets us implement various interference-avoiding strategies based on cross-application communication. For example, using `BBIO_On_flush` can detect some other application's I/O traffic, thus preventing the application from flushing its buffer while another application is accessing the PFS, and wait for the file system to be available again.

C. Implementation Details

When BBIO captures a POSIX or a LibC function call, it first checks whether if the referenced file descriptor has a buffer associated with it. Such a buffer is either an `mmap`-ed file in a local disk or an anonymous memory segment. The buffer associated with a file is *not* a local copy of this file. Instead, the local file is a log of the operations to be performed on a file. For example, a `write` operation will add an operation code identifying it, followed by the size of the write and then a copy of the data. If several contiguous writes are issued, BBIO combines them by updating the size of the first one and appending the data of subsequent ones. Because of this log-structured implementation, BBIO is not yet able to work with files accessed both in read and write modes. BBIO will still associate a buffer to files opened in both read and write modes but will disable it if read operations are issued.

While most parallel file systems provide some ways to control the distribution policy across servers (stripe size and number of servers to stripe across), they usually do not provide a coordinated way to specify which servers should be used among multiple applications. To simulate such a possibility, we deployed separate PVFS instances on different sets of storage servers each, and let each application access its own PVFS instance.

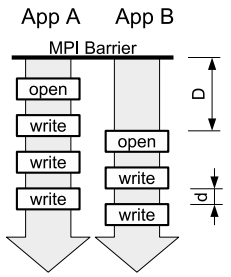


Fig. 3: Overview of our microbenchmark.

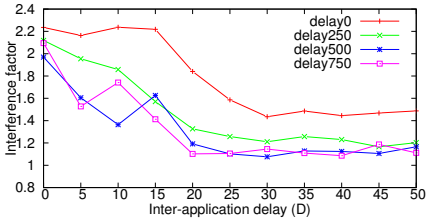
V. Experimental and Evaluation Results

A. Methodology

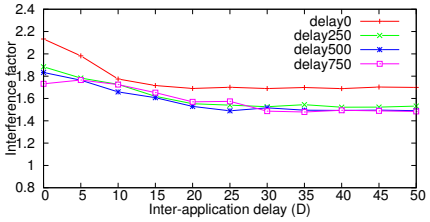
Platform description: All experiments were carried out on a 65-node SUN Fire Linux-based cluster at the Illinois Institute of Technology. The computing nodes are Sun Fire X2200 servers, each with dual 2.3 GHz Opteron quad-core processors and 8 GB of main memory. A 250 GB hard drive and an additional PCI-E X4 100 GB SSD are attached to each node as the storage devices. All 65 nodes are connected with Gigabit Ethernet. The network topology of the cluster consists of three groups of nodes connected to three distinct network switches with adequate capacity (i.e., 22 nodes on a router of 25 Gbits/sec) and a master node. We ran a series of network benchmarks to investigate the network's capabilities, and we found that the Gigabit Ethernet is sufficient to support our experiments without being a bottleneck. A subset of these compute nodes on network switch 1 was used to deploy a PVFS file system [24], and all client processes were dispatched on switches 2 and 3 simulating a supercomputer infrastructure with a separate PFS (i.e., all requests to servers go through the inter-rack network and not through the faster intra-rack connections).

Software used: The operating system is Ubuntu Server Edition 12.04, the parallel file system installed is OrangeFS v2.9.2, and the data distribution policy chosen is the "simple distribution" [25]. We compiled our code using gcc compiler version 4.8. The MPI implementation is MPICH 3.1.4. We developed an IOR-like microbenchmark that starts by splitting its set of processes into two separate sets running on different sets of compute nodes, representing two distinct applications. Each process writes a series of N requests of size S contiguously in a file, using POSIX `fwrite` calls. Between each request, the processes wait (sleep) a given delay d representing computation that could occur between requests. The second group of processes waits a specified amount of time D before starting its own series of I/O operations while the first set begins performing I/O. Both applications have a delay d between each write request. Figure 3 summarizes the behavior of our microbenchmark. The coordination between applications in our benchmark is done by using MPI communication. For example, in strategy 1, to know when B starts its I/O phase, A posts a nonblocking barrier before beginning its series of I/O operations. It then checks (`MPI_Test`) for completion of this barrier before each write. When B starts its I/O phase, it posts the matching nonblocking barrier, allowing A to know that B has started its I/O.

Measurements: For each group of processes acting as a distinct application, we wrapped all I/O operations between



(a) Application A



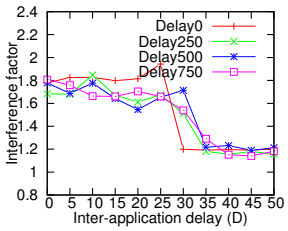
(b) Application B

Fig. 4: Default case with no strategy applied. Interference factor observed by each application as a function of the delay D (sec) and for different values of the inter-request delay d (ms).

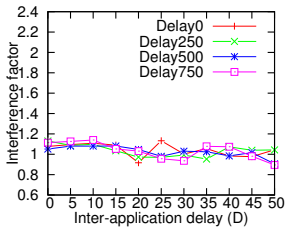
timing calls (i.e., `MPI_Wtime`), and we calculated the total time spent in I/O. We repeated all experiments five times, and we report the average values. We also leveraged the tools introduced in our previous work [1]: Δ-graphs are plots of a given performance metric as a function of the interapplication delay D. We consider only positive values of D (i.e., application B starts its I/O phase *after* application A). We also present the results in terms of an *interference factor*, which is a slowdown with respect to the application running without contention (i.e., an interference factor of 2 means that the duration of the write phase has been multiplied by 2). The interference factor thus is defined as $I_f = \frac{IO\ time_{with\ interference}}{IO\ time_{alone}}$ We always measure the duration of an I/O phase as the time between the moment the file is opened and the moment it is closed; thus the interrequest delay $(N-1) \times d$ is counted in this duration, but the interapplication delay D is not.

B. Experimental Results

1) Default case with interference (no strategies): We first quantify the interference encountered by the applications for different values of d and as a function of D. In this set of experiments, two groups of 256 processes write a series of 32 blocks of 1 MB in individual files, leading to 8 GB of output per application. We do not implement any coordination strategy; instead, we let the two applications interfere. Figure 4 shows the observed interference factor for applications A (a) and B (b). We first observe that when a delay is introduced between I/O requests, the interference factor experienced by each application is lower than when there is no delay at all. A second observation is that although the two applications are identical, the interference factor in both is often larger than 2 (up to 2.3 here) even when the two I/O phases do not start at the same time. This shows that the slowdown produced by interference is larger than what we would expect from a proportional sharing of resources, thus motivating our strategies. We also observe that when the applications start at the same time, the slowdown is significant and cannot be



(a) Application A



(b) Application B

Fig. 5: Strategy 1a. Interference factor observed by each application as a function of the delay D(sec).

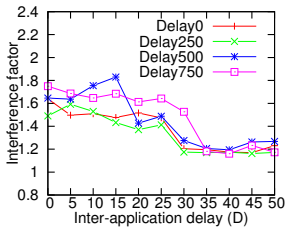
ignored; on the other hand, when the inter-application delay is larger than 25-30 seconds, the interference factor reduces because the first application is almost finished when the second starts accessing the PFS.

2) I/O staging-based strategies: We evaluated our two I/O staging-based strategies with their two respective variations presented in Section III-A using the same configuration as above.

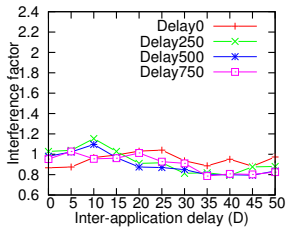
Results with Strategy 1a: Figure 5 presents the results where A buffers when B starts writing. Application A flushes and continues only after B is finished. This strategy is advantageous to B, whose observed interference factor is around 1 (no slowdown) since it enjoys exclusive access to the PFS. From A's perspective, staging its requests instead of interfering with B leads to a lower interference factor, between 1.2 and 1.8, instead of 1.2 to 2.2 in the default case with interference. For B there is no slowdown (since it is prioritized), and for A the interference factor is lower because burst buffers allow it to continue executing until it is absolutely necessary to block and wait for B to finish.

Results with Strategy 1b: Figure 6 depicts the results of the other variation, where A is buffering when it detects that B is writing; but instead of blocking waiting for B to finish, it flushes and continues. The intuition for this variation of the strategy is that we could tolerate some interference to happen in both applications. It lies somewhere between the previous variation of the same strategy and the pure blocking of A until B finishes. For A the behavior is similar to that described earlier, with the difference that interference is less apparent as D increases, because A is allowed to flush the buffer and need not block waiting for B to finish. Specifically for interapplication delay less than 30 seconds, the interference factor for A is kept around 1.5 (lower than the default but slightly lower than strategy 1a) and drops to 1.2 after that. Application B experiences an interference factor of around 1.2, slightly higher than before since the flushing of A happens earlier and forces B to lose the exclusive access to the PFS.

Results with Strategy 2a: This strategy is similar to Strategy 1a but with priority reversed. Here, A writes having exclusive access to the PFS, and B upon entering its I/O phase starts staging the requests until it blocks waiting for A to finish. Application B flushes the buffers and continues writing only after A has finished its I/O operations. Figure 7 shows the results. As expected, A demonstrates an interference factor close to 1, while B starts with around 1.5-1.6 for interapplication delay 0 seconds (i.e., when both applications

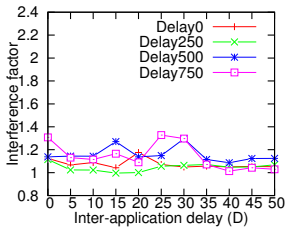


(a) Application A

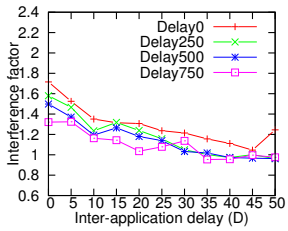


(b) Application B

Fig. 6: Strategy 1b. Interference factor observed by each application as a function of the delay D(sec), when A *buffers* its operations and *flushes* even if B has not finished.



(a) Application A



(b) Application B

Fig. 7: Strategy 2a. Interference factor observed by each application as a function of the delay D(sec).

start at the same time) and drops close to 1 for delays larger than 30 seconds, approximately the time needed to complete the I/O phase.

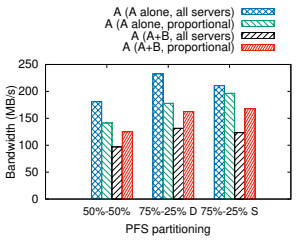
Results with Strategy 2b: As with Strategy 1b, in this case B stages its operations when it enters its I/O phase because A is writing, however B flushes the buffers even if application A is not finished yet. Figure 8 demonstrates the results for both applications. Application A performs stably with interference factor around 1.2 for all values of D. Application B, on the other hand, when it starts at the same time as A, observes an interference factor of 1.6, which gradually decreases as delay D increases. One observation is that this decrease comes with higher interapplication D than Strategy 2a has, because the flushing of the buffers, before A has finished, increases the interference for both applications.

3) Partition-based Strategies: To evaluate the partitioning strategy, we conducted three tests with four configurations each as follows. The first test considers two applications A and B, identical in scale (i.e., same processes) and workload (i.e., data per process), and is referred as 50-50% in the figures. The proportional split of 8 available storage servers is in half for this test case. The second test considers two applications with the same scale, but A writes three times more data per process than does B, and is referred as 75-25% D in the figures. In the third test A has three times more processes than does B, but writes the same data per process and is referred as 75-25% S in the figures. For those two test cases the proportional split is 6 servers for A and the remaining 2 for B. As we presented in Section III-B, we have two cases for the partition-based strategy; static and dynamic partitioning of the PFS, referred to as Strategy 3a and 3b, respectively.

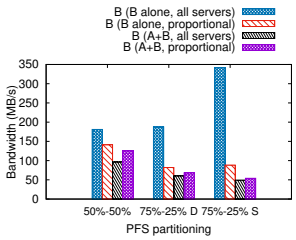
(a) Application A

(b) Application B

Fig. 8: Strategy 2b. Interference factor observed by each application as a function of the delay D(sec), when B *buffers* its operations and *flushes* even if A has not finished.



(a) Application A

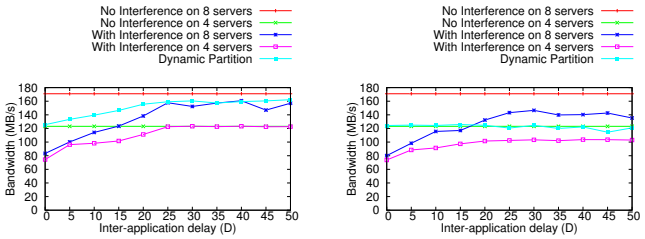


(b) Application B

Fig. 9: Strategy 3a. Throughput observed by the applications when writing alone and in contention to all available servers or to proportional number of servers.

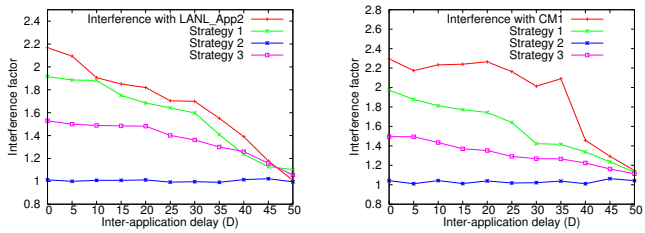
Results with Strategy 3a (static partitioning): The results are reported in Figure 9. We notice that when dividing by 2 the number of servers that an application accesses, the throughput observed by the application is decreased to about 75% of the throughput observed with all the servers, instead of 50%. When two applications concurrently access all 8 servers however, the slowdown is more than $2\times$. By partitioning the file system and letting each application access its own set of servers, we achieve an expected result: the application's throughput lies somewhere between the throughput observed with contention and the throughput observed with proportional servers for an individual application. This shows that by partitioning the file system so that concurrent applications do not access the same set of servers, we are able to mitigate the I/O interference and increase the I/O performance up to 40% relative to the default case where applications interfere while sharing the same storage resources. Note that each application, from the beginning till the end of its execution, has exclusive access to a distinct subset of servers.

Results with Strategy 3b (dynamic partitioning): In this experiment, we modified our benchmark to simulate the dynamic partitioning of the parallel file system's storage servers. There are three different installations of PVFS: one full deployment on all 8 storage servers, and two deployments with the proportional split. For instance, for the identical applications scenario where we split the PFS in half, two new deployments of PVFS on 4 storage servers each were used. Application A opens two files, one on the full and one on the proportional installation of the PVFS. Application B opens a file on the other proportional deployment. When A starts executing, it uses the



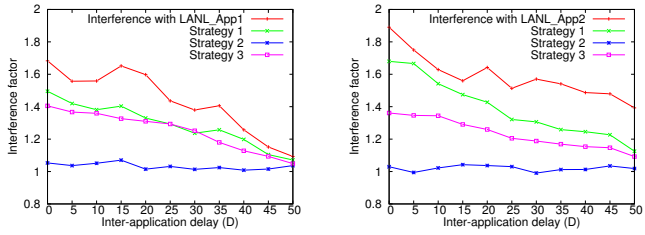
(a) Application A (b) Application B

Fig. 10: Strategy 3b. Throughput observed by the applications when writing alone and in contention to all available servers or to proportional number of servers.



(a) LANL_App1 - LANL_App2 (b) LANL_App1 - CM1

Fig. 11: Interference factor observed by LANL_App1 when interfering with LANL_App2 and CM1 as a function of the delay D(sec).



(a) CM1 - LANL_App1 (b) CM1 - LANL_App2

Fig. 12: Interference factor observed by CM1 when interfering with LANL_App1 and LANL_App2 as a function of the delay D(sec).

full installation on 8 servers. Upon the arrival of B, A redirects all its conflicted I/O requests to the proportional deployment. Thus no interference occurs since the applications are accessing a different set of servers. Once B finishes, A goes back to using all 8 servers.

Figure 10 demonstrates this dynamic partitioning strategy. The results are expressed in terms of bandwidth (MB/s). We can see that when both applications start at the same time in the case of interfering (i.e., blue and purple lines), the performance is around 70-80 MB/s, and it increases as the interapplication delay D increases (i.e., interference is less). After 30 seconds of delay D, there is no more interference since A has finished its operations and the bandwidth reaches maximum values. The dynamic partitioning strategy offers application A a bandwidth of 123 MB/s at delay 0, an increase of 75% with respect to the default case with interference. Bandwidth continues to increase with D since A is taking advantage of a full 8-server PFS deployment until it is forced, when B enters its I/O phase, to use the smaller 4-server PFS resource. On the other hand, B experiences stable performance since it always operates on a separate set of 4 servers, representing applications with smaller I/O bursts. We conducted the same dynamic partitioning strategy for the case 75-25% on data size and job size, and the results are similar. Because of space limitation, however, we do not present the results here. We note that, this partitioning strategy should be selected based on information on the applications' scale, I/O patterns, and the scalability of the file system.

4) Real Scientific Applications Workloads: To evaluate our strategies with real workloads we used three scientific applications, CM1 [26], Anonymous Application 1 and 2 by Los Alamos National Lab (referred to as LANL_App1 and LANL_App2) [27]. These applications are real-world codes that run on current supercomputers (for instance CM1 has been used on NCSI's Kraken and NCSA's BlueWaters). All of them use the POSIX I/O interface and follow the one-file per process logic for the write operations. Their workload comprises mostly by the check-pointing behavior they exhibit (i.e., periodically write their progress to the disk).

For the experiments in this paper, we created a workload generator that takes the I/O trace as an input and *replays* all the operations to the parallel file system. The applications are run on 256 cores on the same testbed. Since the I/O behavior is repetitive, we isolated the I/O access pattern from the traces for only one checkpoint phase and fed this to our workload generator to study the I/O interference. This means that for CM1 each process writes 52 MB, and for LANL_App1 and LANL_App2 each process writes 50 MB. The rest of the tests are similar to the ones presented previously.

Figure 11 shows the slowdown experienced by LANL_App1 (expressed in interference factor). In figure 11(a), when LANL_App1 executes at the same time with LANL_App2 the execution time is 2.2 times higher compared to when the applications execute with no interference. Strategy 1 prioritizes LANL_App2 but also allows LANL_App1 to lower the interference factor to 1.9. On the other hand, strategy 2 offers LANL_App1 exclusive access to the PFS thus there is no slowdown at all. Finally, strategy 3 forces the two concurrent apps to share the PFS in half and the results are similar as before. Note that we used dynamic partitioning for this test. In figure 11(b), LANL_App1 interferes with CM1 and the results are similar.

In Figure 12 (a) and (b), we see the results for CM1. All three strategies help alleviate the negative impact of the I/O interference in the performance of the application. We notice a slightly lower interference factor when CM1 runs at the same time with LANL_App1 and LANL_App2 mostly due to the different access patterns. However, the trend is the same and by using our proposed strategies one can mitigate the performance slowdown.

C. Scaling of Our Strategies

In this subsection, we first present how concurrent applications affect each others performance. By increasing the number of concurrent instances, we see that the execution time increases dramatically. In the following tests, we run

multiple instances of the same applications with the exact same parameters to investigate the relationship between the number of concurrent instances and the increase in the overall execution time (an interference factor of 3 means three times higher execution time). We used our benchmark and the three scientific applications to perform the following test. We varied the number of concurrent instances and measured the overall execution time. Due to the limitations of our testbed, we used 80 processes per instance (320 processes in total when running four instances). For example, we first ran one instance of our benchmark and measured the execution time. We then ran it again but this time interfering with one exact same instance, and then with two same instances and so on. In figure 13, we see the results. The slowdown is significant even with only two concurrent instances with interference factor around 2.4 and the situation gets worse with three and four where the interference factor reached 3.9. This shows that there is great potential to optimize performance by mitigating the I/O interference between concurrent applications.

We achieved this performance optimization by using our proposed solution. We can utilize our strategies one by one as is, or we can use a combination of all three strategies to achieve even better results. We distinguish some different scenarios. For strategies 1 and 2 burst buffers are heavily used to buffer I/O requests, and thus there might be a situation where all of the burst buffers would want to flush the data at the same time. This would create a new contention to the PFS and the benefits of using the burst buffers would be possibly eliminated. To alleviate this issue, we propose three heuristics to coordinate the flushing of the buffers. The first one is *token-based* where each app flushes the data from its burst buffers to the PFS if it holds the token. Upon completion, it simply passes the token to the next application. The second heuristic we propose is based on a *time-window*. In order to avoid "starving" an application by waiting for the token, we allocate a time-window in a round robin fashion to each application that can use it to flush data to the PFS. The third heuristic is a *priority-based* flushing of the buffers. This allows the system administration to assign the priority according to the needs. Thus, BBIO offers, through coordination of the burst buffers, a solution to the interference problem. Our third strategy is able to entirely avoid contention by efficiently sharing the available storage servers but it cannot scale to a large number of concurrent applications. A combination of all three strategies is likely to offer an overall efficient solution.

Consider the following scenario. App A is starting and then App B joins the system. We can utilize strategy 3 and solve the interference by sharing the servers. After a while, App C is also joining. Instead of further dividing the available storage servers we use either one of the other two strategies. Thus, App C uses the burst buffers to divert the I/O traffic from the PFS. When App A finishes, storage servers become available to App C again which flushes the buffers and continues normally.

In Figure 14 we report the execution time for all applications that run concurrently. The delay between applications was set to 0 and we employed our strategies in a first-come first-server fashion. Strategy 1 and 2 are somewhat mirroring the effect of prioritizing a certain application. For strategy 3, App A initially shares the 8 available servers with App B and then App B share its 4 servers with App C in half. Since the applications are exactly the same in terms of size and workload and due the limitation of space we do not present results from all the proposed heuristics on when to flush the buffers.

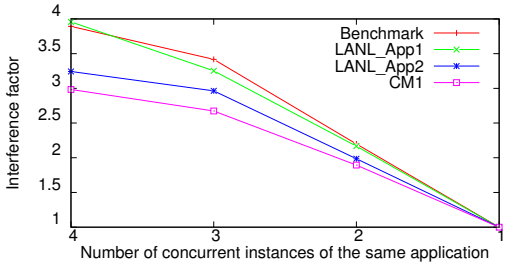


Fig. 13: Slowdown expressed in interference factor due to concurrent instances of the same application.

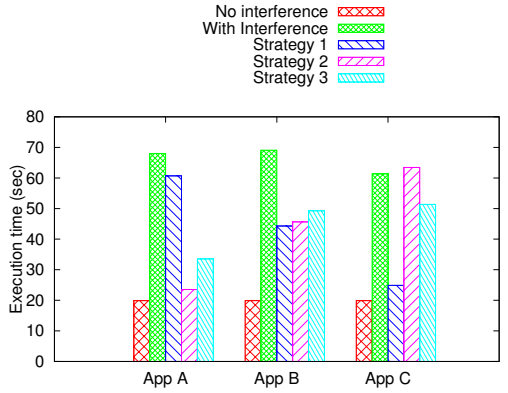


Fig. 14: Execution time (in seconds) for App A, App B and App C when running concurrently with each other. All strategies are activated in a FCFS manner.

D. Benefits and Limitations of Buffering

During our experiments we noticed an effect taking place in certain test cases. Figure 15 shows the performance of Application A for the following test: A is writing with a 750 ms delay between each I/O request (simulating a computation/communication pattern), and B is writing with no delays in between each I/O request. As a base case we include the test where both applications are writing with 0 delay between each I/O operation. When A is running alone with delay 0 it achieves 170 MB/s bandwidth and for delay 750 ms, 162 MB/s. When interfering with B, those values become 75 MB/s and 90 MB/s, respectively. By turning on the buffering system, we would intuitively expect a boost in performance. For delay 0, A achieves 96 MB/s bandwidth, which is 20% better compared with the interference case with buffering off. For delay 750 ms however, application A experiences a bandwidth of 87 MB/s, which is slightly lower than that with buffering off. Additionally, the interference factor between delay 0 is higher than the case with delay 750 ms.

This behavior is further examined and analyzed as follows. We investigated the internal behavior of every process inside each application. We focus on one application. The test comprises our microbenchmark where each process is writing 1 MB of data 32 times with a delay between each I/O operation. In Figure 16 we can see the duration of each I/O request for some randomly selected processes during our tests. When

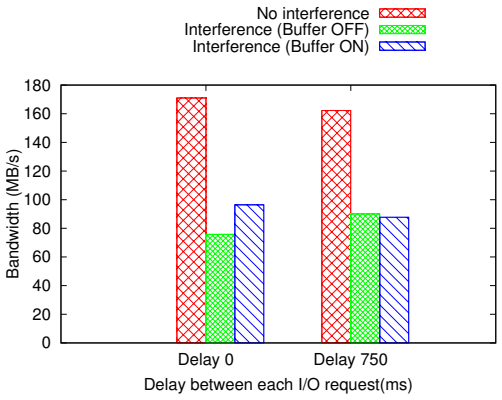
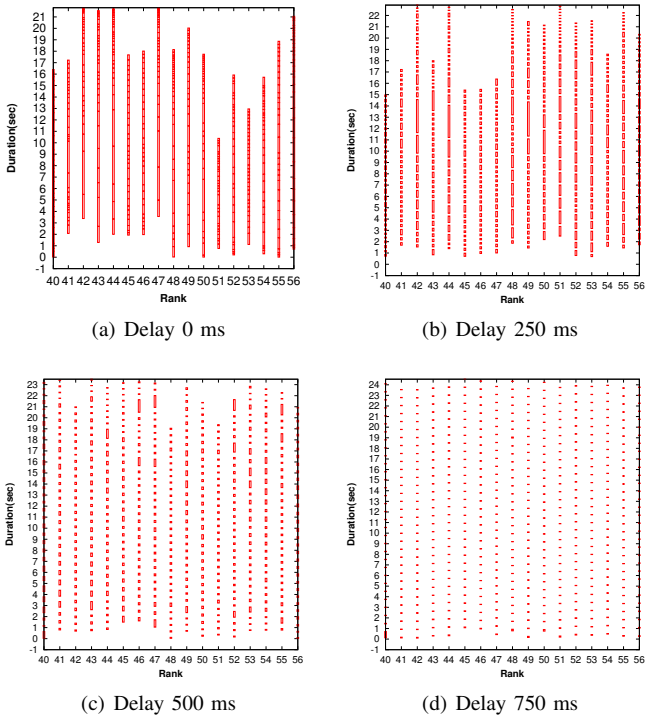


Fig. 15: Throughput observed within one application when buffering is turned on and off. Different workloads representing an I/O burst phase (delay 0) and a computation-I/O intensive alternating (delay 750).



(a) Delay 0 ms (b) Delay 250 ms

(c) Delay 500 ms (d) Delay 750 ms

Fig. 16: Timelines per process with delays between each I/O request.

all processes are writing at the same time with no delay between each I/O request, we see that the duration of those I/O operations is longer because of the interference. As the delay between each I/O request increases (i.e. 250 ms, 500 ms, 750 ms), we see a decrease on the duration of each request, a *self-stabilizing* effect where the computation/communication phase of one process might allow the I/O phase of another one to complete faster. Thus, *turning on the buffer and flushing it at some later point make all processes of a single application interfere internally with one another*. Hence, this solution is worse than allowing the interference with a second application in the first place. Therefore, counterintuitively, staging the I/O of one application to allow another one to exclusively access the shared storage resource and then flushing all the buffers from the delayed application at the same time might actually hurt the overall performance and highly depends on the access patterns the applications have.

VI. Related Work

Many research studies have tried to address the I/O interference issue by scheduling I/O requests at the PFS level. In the network request scheduler presented by Qian et al. [28], requests embed deadlines and the targeted object's identifier. Song et al. [13] achieve the same result with applications' ids instead of objects' ids. AGIOS, proposed by Boiteau et al. [15], also guides the file system's scheduler's decision through additional information that future I/O requests predicted thanks to traces. Zhang et al. [12] leverage a "reuse distance" to state whether it is worthwhile for a data server to wait for an application's new I/O request or to service other applications requests. Lebre et al. [14] provide multi-applications scheduling with the goal of better aggregating and reordering requests, while trying to maintain fairness across applications. Gainaru et al. [16] propose scheduling techniques to optimize the system's I/O efficiency under congestion. In [29], an efficient distributed message queue was used to increase the latency and minimize the interference in the network.

Closer to our work, Batsakis et al. [30] propose a system in which clients price their nonblocking requests depending on the ability to delay the requests, and an auction mechanism chooses which requests should be serviced first. In a way, the ability to delay a request is also present in our proposed Strategies 1 and 2, where requests can be staged in a burst buffer to allow the computation to continue. But our proposed strategies work with any type of request, whereas theirs is tied to nonblocking requests only. Tanimura et al. [31] propose to reserve throughput from the storage system. This approach can be compared with our third strategy based on file system partitioning. However, this reservation is made at job submission in their approach, whereas ours leverages communications between applications in order to understand when and how the file system's resources should be partitioned.

In a recently published work [32], an I/O orchestration framework named TRIO is proposed that also coordinates the timing when burst buffers are moving the checkpointing data to the PFS. By controlling the flushing orders among concurrent burst buffers TRIO tries to alleviate the contention on storage servers. However, it does not consider the application's I/O access patterns, and an alternating computation-I/O behavior is not taken advantage of. TRIO focuses more on when the burst buffers will spill the data to the PFS, whereas we leverage the existence of burst buffers to act as a *traffic controller* and prevent I/O interference while allowing applications to do other *usefull* tasks. We also provide a comprehensive set of strategies to mitigate the negative effects of I/O interference, and we propose other ways to use burst buffer coordination (dynamic partitioning of PFS).

VII. Conclusion

Cross-application I/O interference is becoming an important issue as we move toward exascale. This issue has initiated unconventional approaches in which independent applications

can learn each other's behavior and coordinate their accesses to the shared, parallel file system. In this paper, we have proposed three strategies that applications can employ to better coordinate their accesses. Two rely on burst buffers and on the fact that instead of blocking, applications can stage their I/O requests and reissue them later, when the file system is more available. The third strategy ensures that the applications access distinct sets of storage servers. We have shown the potential of our three strategies with a microbenchmark; the results show performance improvements up to 2x.

As future work, we plan to move from the cross-application coordination scheme, where applications communicate their I/O behavior to each other, to a system wide coordination scheme where a global, centralized entity is responsible for managing all concurrent applications' accesses to the shared underlying storage resources using our strategies. In this direction, we plan to equip this entity with I/O prediction capabilities, using the Omnisc'IO [33] approach, and thus select the best coordination strategy.

ACKNOWLEDGMENT

This material was based upon work supported by the U.S. Department of Energy, Office of Science, under Contract No. 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)*, 2014.

[2] G. Grider, "Exascale FSIO," https://institute.lanl.gov/hec-fsio/conferences/2010/presentations/day1/Grider-HECFSIO-2010-ExascaleEconomics.pdf, LANL.

[3] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 2012, pp. 1–11.

[4] "NERSC's Cori burst buffers," https://www.nersc.gov/users/computational-systems/cori/burst-buffer/.

[5] "CRAY's datawarp technology," http://www.cray.com/sites/default/files/resources/CrayXC40-DataWarp.pdf.

[6] "LANL's Trinity specs," http://www.lanl.gov/projects/trinity/specifications.php.

[7] Intel, "Extreme-Scale Computing R&D Fast Forward Storage and I/O Final Report," Intel, The HDF Group, CRAY, EMC, Tech. Rep., June 2014.

[8] Y. Hashimoto and K. Aida, "Evaluation of Performance Degradation in HPC Applications with VM Consolidation," in *Networking and Computing (ICNC), 2012 Third International Conference on*. IEEE, 2012, pp. 273–277.

[9] J. Lofstead and R. Ross, "Insights for Exascale IO APIs from Building a Petascale IO API," in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 2013, pp. 1–12.

[10] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing Output Bottlenecks in a Supercomputer," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 8.

[11] 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*. IEEE Computer Society, 2010, pp. 1–12.

[12] 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*, 2010.

[13] 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*. ACM, 2011, p. 17.

[14] A. Lebre, G. Huard, Y. Denneulin, and P. Sowa, "I/O Scheduling Service for Multi-Application Clusters," in *IEEE International Conference on Cluster Computing*, Sept 2006.

[15] F. Zanon Boito, R. Kassick, P. Navaux, and Y. Denneulin, "AGIOS: Application-Guided I/O Scheduling for Parallel File Systems," in *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS '13)*, Dec 2013.

[16] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC Applications Under Congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, 2015.

[17] N. Mi, A. Riska, Q. Zhang, E. Smirni, and E. Riedel, "Efficient management of idleness in storage systems," *ACM Transactions on Storage (TOS)*, vol. 5, no. 2, p. 4, 2009.

[18] Y. Kim, R. Gunasekaran, G. M. Shipman, D. Dillow, Z. Zhang, B. W. Settlemyer *et al.*, "Workload characterization of a leadership class storage cluster," in *Petascale Data Storage Workshop (PDSW), 2010 5th*. IEEE, 2010, pp. 1–5.

[19] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, p. 8, 2011.

[20] "Leadership Computing Requirements for Computational Science," https://www.olcf.ornl.gov/wp-content/%20uploads/2010/03/ORNL%20TM-2007%2044.pdf.

[21] "Large Memory Appliance/Burst Buffers Use Case," https://asc.llnl.gov/CORAL-benchmarks/Large_memory_use_cases_llnl.pdf.

[22] D. Brown, P. Messina, D. Keyes, J. Morrison, R. Lucas, J. Shalf, P. Beckman, R. Brightwell, A. Geist, J. Vetter *et al.*, "Scientific grand challenges: Crosscutting technologies for computing at the exascale," *Office of Science, US Department of Energy, February*, pp. 2–4, 2010.

[23] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Frontiers of Massively Parallel Computation, 1999. Frontiers '99. The Seventh Symposium on the*, 1999.

[24] R. B. Ross, R. Thakur *et al.*, "PVFS: A Parallel File System for Linux Clusters," in *Proceedings of the 4th annual Linux Showcase and Conference*, 2000.

[25] "PVFS2 data distribution schemes," http://www.orangefs.org/trac/orangefs/wiki/Distributions.

[26] G. Bryan, "CM1 code," http://www2.mmm.ucar.edu/people/bryan/cm1/.

[27] "LANL anonymous applications trace files," http://institutes.lanl.gov/plfs/maps/, LANL.

[28] 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, 2009.

[29] I. Sadooghi, K. Wang, D. Patel, D. Zhao, T. Li, S. Srivastava, and I. Raicu, "Fabriq: Leveraging distributed hash tables towards distributed publish-subscribe message queues," in *2015 IEEE/ACM 2nd International Symposium on Big Data Computing (BDC)*. IEEE, 2015, pp. 11–20.

[30] 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.

[31] 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.

[32] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu, "Trio: Burst buffer based i/o orchestration," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 2015, pp. 194–203.

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