CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination

Matthieu Dorier*, Gabriel Antoniu†, Rob Ross‡, Dries Kimpe‡, and Shadi Ibrahim†
*ENS Cachan Brittany, IRISA, Rennes, France, matthieu.dorier@irisa.fr
†INRIA Rennes Bretagne-Atlantique, Rennes, France, {gabriel.antoniu, shadi.ibrahim}@inria.fr
‡Argonne National Laboratory, IL 60439, USA, {rross, dkimpe}@mcs.anl.gov

***Abstract*—Unmatched computation and storage performance in new HPC systems have led to a plethora of I/O optimizations ranging from application-side collective I/O to network and disk-level request scheduling on the file system side. As we deal with ever larger machines, the interference produced by multiple applications accessing a shared parallel file system in a concurrent manner becomes a major problem. Interference often breaks single-application I/O optimizations, dramatically degrading application I/O performance and, as a result, lowering machine wide efficiency.**

This paper focuses on CALCioM, a framework that aims to mitigate I/O interference through the dynamic selection of appropriate scheduling policies. CALCioM allows several applications running on a supercomputer to communicate and coordinate their I/O strategy in order to avoid interfering with one another. In this work, we examine four I/O strategies that can be accommodated in this framework: serializing, interrupting, interfering and coordinating. Experiments on Argonne's BG/P Surveyor machine and on several clusters of the French Grid'5000 show how CALCioM can be used to efficiently and transparently improve the scheduling strategy between two otherwise interfering applications, given specified metrics of machine wide efficiency.

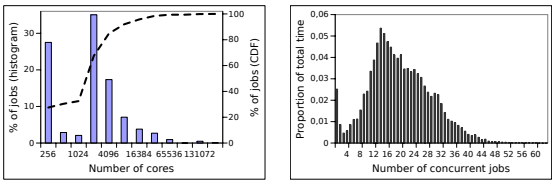
***Keywords*-Exascale I/O, Parallel File Systems, Cross-Application Contention, Interference, CALCioM**

I. Introduction

In 2012 for the first time in history, a supercomputer (LLNL's Sequoia) surpassed a million cores. As of August 2013, the top five supercomputers all have more than 500,000 cores [1]. This tremendous power offers the possibility to run larger, more accurate simulations of scientific phenomena in climate, cosmology, or particle physics. But *sustained petascale* (and *exascale* in a few years from now) is not achieved by running applications one at a time. The real power of a million-core machine comes from the increased number of applications that can run concurrently.

Although computer scientists generally argue that their machines have been designed mainly to run applications at full scale (*i.e.*, large applications), our current machines are already used by many relatively small applications at the same time, as exemplified by Figure 1(a), which shows the distribution of job sizes on Argonne's Intrepid. Half the jobs on this platform indeed run on less than 2,048 cores (*i.e.*, 1.25% of the full machine); this assertion remains true when weighing the jobs by their duration (*i.e.*, half of the machine time is used by applications smaller than 2,048 cores).

An important challenge at such a large scale is dealing with the data deluge coming from these applications;



(a) Distribution of job sizes (b) Number of concurrent jobs

Figure 1. Job sizes and number of concurrent jobs by time unit on Intrepid, extracted from the Parallel Workload Archive [2], using *ANL-Intrepid-2009-1.swg* (8 months of job scheduler's traces, from January 2009 to September 2009).

decades of research have prepared the community for exascale I/O through application-side optimizations, implemented in MPI-I/O or in higher-level I/O libraries (such as HDF5 [3]) and middleware (DataStager [4], Damaris [5], PLFS [6], etc.). These optimizations aim at better organizing the data layout in order to achieve the best possible I/O performance.

Unfortunately, when several concurrent applications access a shared parallel file system in an uncoordinated manner, storage servers have to deal with interleaved requests coming from different sources, which often break the access pattern optimized by each application individually. At the file system level, network requests schedulers impose an order on requests. This order may be simply an independent "first-in-first-out" policy on each storage server or a more elaborated strategy to reach the same order on each server or attempt to service applications one at a time. Lower-level schedulers in storage servers try to minimize disk-head movements by aiming at better data locality [7]. Yet these solutions work on low-level requests without any knowledge from the applications. They may attempt to be fair in the sense that each application should benefit from the same "quality of service." This quality of service is usually defined in terms of sharing of throughput. This notion of "fairness" is not appropriate, however, because it does not take into account the particular constraints (*e.g.*, I/O rate, memory usage, job's duration) of each application [8] nor the global efficiency of the system. As an example, a fair sharing of throughput between two concurrent applications will lead to both applications being slowed down. On the other hand, serializing the access of one application after the other, though unfair, can lead to a higher efficiency from a global system point of view.

While reducing cross-application I/O interference has long been an important challenge in HPC systems, prior work has not taken into account the overall machine wide efficiency. Moreover, the I/O interference varies not only according to the applications size (*i.e.*, small or large applications) but also according to their I/O behavior and the interference time. Hence, it is vital that applications be aware of each other's I/O behavior and therefore coordinate their I/O requests.

This paper aims at mitigating the I/O interferences in HPC systems by exploiting cross-application coordination. *To the best of our knowledge, this is the first study on cross-application coordination in HPC systems*. We specifically study three coordination strategies: interfering, serializing, and interrupting, which are made possible through cross-application communications. We Observed that these strategies are all suboptimal in different contexts yet complement each other in a way that makes a dynamic selection desirable, especially when applications present different I/O behavior and requirements. Therefore, we integrated these strategies into the CALCioM (Cross-Application Layer for Coordinated I/O Management) framework, which can select the most appropriate one for a targeted machine wide efficiency. Using CALCioM, an application will pause its I/O activity for the benefit of another application, wait for another application to complete its I/O, or still access the file system in contention with another application. CALCioM's selection of a scheduling strategy is based on a holistic view of the set of running applications and their respective I/O activity as perceived from each level of the I/O stack, with the aim of optimizing a specified metric of machine wide efficiency.

Our experiments are conducted on Argonne's BlueGene/P Surveyor machine and several Grid'5000 (French national grid testbed [9]) clusters, with a benchmark derived from IOR [10] to simulate multiple interfering applications with a fine control on their I/O patterns.

CALCioM is *radically different* from traditional approaches where applications are optimized individually, disregarding potential cross-application interference, and where interference-avoiding strategies are left to the file system's scheduler, with no information on the constraints or freedom of each application and no way to differentiate I/O requests.

We note that this paper does not aim to provide an absolute solution to solve the I/O interference in HPC systems. Rather, it provides extensive experimental insight on different scheduling options, showing their efficiency and limitations and demonstrating a cooperative solution based on dynamic selection of these strategies at run time.

The rest of this paper is organized as follows. Section II motivates our work through references, trace analysis and experimental insight on cross-application interference. We present the CALCioM approach in Section III, along with its design principles, API, and implementation. Section IV details the performance evaluation. We then discuss our solution and position it with respect to related work in Section V. We conclude in Section VI and open to future work .

II. Interferences in shared storage systems

Interference can be defined as a "performance degradation observed by an application in contention with other for the access to a shared resource". This section gives tools for analyzing interference and examples on real platforms in the context of shared storage systems.

A. Interference as described in the literature

Distributed systems are by nature subject to concurrency. Performance variability as a consequence of resource sharing is a well-known problem in cloud computing, for example. Cloud users share not only network bandwidth, but also the hardware on which their VMs run [11]. In this context, performance guarantees are part of the service-level agreement that also defines the pricing model of the platform; hence, interference has economical consequences. Pu et al. [12], for example, provide a study of interference specifically for I/O workloads in the cloud.

In the supercomputing community however, the lack of an underlying pricing model, along with the fact that computing resources are fully dedicated to a single job at a given moment, did not motivate much analysis of cross-application interference. Yet cross-application contention is mentioned by Skinner and Kramer [13] as one of the five main causes of performance variability in HPC systems, in particular at the level of parallel file systems, which remains the main shared resource of the platform and thus the main point of contention between applications. In their own words, cross-application contention is in fact *one of the most complex manifestations of performance variability on large scale parallel computers*.

Uselton et al. [14] also mention that the high variability observed in the I/O performance of HPC applications is caused by factors coming from both inside and outside the application, which makes its analysis even more challenging.

B. Probability of concurrent accesses

Since a supercomputer is used by several applications at the same time, the number of applications that run concurrently at any given moment can be denoted as a discrete random variable $X \in \mathbb{N}$. Figure 1 (b) shows the distribution followed by X on ANL's Intrepid. The proportion of time spent doing I/O by any application can also be seen as a random variable $\mu \in [0, 1]$. Assuming independence between X and μ (although this assumption is optimistic and does not take into account the fact that interfering applications spend more time in I/O phases), the probability that at least one application is doing I/O when observing the system at an arbitrary moment is

$$\mathbb{P}(\text{another is doing I/O}) = 1 - \sum_{n=0}^{+\infty} \mathbb{P}(X = n)(1 - \mathbb{E}(\mu))^n$$

This formula is simply derived from computing the probability that n applications are running; $\mathbb{P}(X = n)$, and none of them are doing I/O: $(1 - \mathbb{E}(\mu))^n$. It represents a lower bound on the probability of interfering with another application, this probability requiring a more complex mathematical model that is outside the scope of this paper.

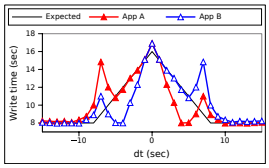


Figure 2. Experiments done on G5K (Nancy site) with PVFS deployed on 35 nodes; two applications of 336 processes each write 16 MB per process in a contiguous collective pattern. A (red) starts at the reference date (0), B (blue) starts at an arbitrary date dt with respected to the reference date.

As an example, assuming that the average portion of time spent in I/O by applications is as small as $\mathbb{E}(\mu) = 5\%$, and using the distribution shown in Figure 1 (b), the probability of concurrent accesses is $\mathbb{P}(\text{another is doing I/O}) = 64\%$, making cross-application interference frequent enough to motivate our research.

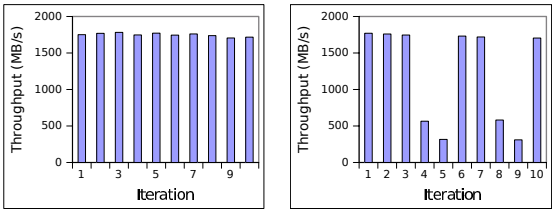
C. Δ-graphs and interference factor

Throughout this paper we will consider two applications A and B. To evaluate the interference between these applications, we introduce the concept of Δ-graphs: Application A starts writing at a reference date $t = 0$, application B starts at a date $t = dt$, and we measure the performance (for example, the time spent in an I/O phase) of A and B. A single experiment with a particular value of dt gives us a point in the graph. A set of experiments with different values of dt allows us to plot the measured performance as a function of dt. If $dt < 0$, B starts before A (as a result, the Δ-graph of the pair of applications (A, B) is the mirror of the Δ-graph of (B, A)). An example of a Δ-graph is shown in Figure 2 (a), which reports experiments done on the Nancy site of Grid'5000 (described in Section IV). Here two instances of the same application run on the same number of cores. From this example, we observe that when two applications compete for access to the file system with the same I/O load, the first one to arrive is favored, although it still observes a degradation of its write time. One can easily compute and display the expected interference as a piecewise linear function, assuming a proportional sharing of resources between the two application. This theoretical performance is also plotted in the figure; and the term "Δ-graph" has, in fact, been chosen after its shape. When considering three applications the Δ-graph becomes a surface in a 3D graph, and is thus more difficult to display.

In the following, we will either consider the I/O time as a reference metric, or use an *interference factor*, defined for a single application as the measured access time divided by the time it would require without the contention with the other application:

$$I = \frac{T}{T_{alone}} > 1$$

I is arguably more appropriate to study interference because it gives an absolute reference for a noninterfering system: $I = 1$. Moreover, it allows the comparison of applications that have different size or different I/O requirements. In a potential extension of this work, I can be computed for other metrics as well, such as the energy consumption: $I = \frac{E}{E_{alone}}$. This metric depends on the application considered but also on the platform and other applications running simultaneously; it is therefore an context-dependent measure.



(a) Without Interference (b) With Interference

Figure 3. Experiments done on the Nancy site of Grid'5000 with 35 PVFS servers across an Infiniband network: (a) one instance of IOR runs on 336 cores and writes every 10 seconds; (b) another instance is started on 336 other cores and writes every 7 seconds (the figure represents the observed throughput of the first instance only).

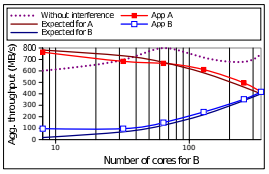


Figure 4. Experiments done on Grid'5000 (Nancy site) with PVFS deployed on 35 nodes; A runs on 336 processes; size of B varies; each process writes 16MB. Both applications start at the same time.

D. Impact of interference on I/O optimizations

Cross-application interference can have a severe impact on I/O optimizations at several levels of the I/O stack.

As an example, Figure 3 shows the consequences of cross-application interference on a caching mechanism. Here two instances of IOR write periodically, one with a 10 seconds delay between each write, the other one with a 7 seconds delay. Kernel caching is enabled in the storage backend, so that applications see a higher throughput than what the disks actually provides. When the two applications happen to write at the same time (iterations 4, 5, 8, and 9), none of them benefits from the cache, and their performance drops dramatically.

E. From diversity to system wide inefficiency

Different applications usually run on different numbers of cores, for different durations. They also have different resource constraints and I/O requirements. For example, the CM1 atmospheric simulation on Blue Waters synchronously writes snapshot files every 3 minutes, for an amount of 23 MB/core. The NAMD chemistry simulation, on the other hand, writes trajectory files of a few bytes per core every second through a designated set of output processors, and in an asynchronous manner[1]. These behaviors and the I/O requirements that they imply cannot be captured by the storage system, which sees only incoming raw requests.

This diversity and lack of knowledge can lead to applications being impacted more than they should by other applications. As an example, Figure 4 shows what happens to the aggregate throughput when a small application interferes with a bigger one. When B runs on 8 cores while A runs on 336, B observes a 6× decrease of throughput compared with B running alone on 8 cores.

[1]This information was gathered through discussions with the Blue Waters PRAC (http://www.ncsa.illinois.edu/BlueWaters/prac.html) users.

More important than the performance of each application individually, cross-application interference lead to a decrease of system wide efficiency. Depending on a given metric to measure this efficiency (for example, the sum of run time of all applications, the number of FLOPs used for actual science, etc.), it is desirable to find ways to decrease these interference factors. Doing so, however, requires some knowledge about each application's I/O behavior and requirements.

III. Mitigating interferences within the CALCioM framework

Having illustrated the I/O interference and shown there frequency as well as their potential performance impact, we propose here strategies to overcome these problems. These strategies are then integrated into the CALCioM framework.

A. Interference-avoiding strategies

Cross-application interference can have a big impact on the performance of some applications, in particular given the diversity of sizes and I/O requirements. This performance impact results in a suboptimal use of the machine. In order to mitigate interference several strategies can be envisioned:

1) ***Serializing accesses on a first-come-first-served basis****:* With this strategy, only the application that arrives second in its I/O phase is impacted in a way proportional to the remaining access time of the first application. This policy requires either giving applications a dangerous `lock` function, to ensure accessing the file system one at a time, or giving them a way to *know that another application is currently doing I/O* and that there will be no advantage in interfering with it.

2) ***Interrupting an application's access****:* In this situation, the application that arrived first is impacted. Indeed, if its access can be paused quickly enough, the second application will immediately get access to the file system and will not be impacted. This strategy specifically requires a way for an application to *be interrupted*, that is, to *know that another application arrived and wants to do I/O*.

3) ***Allowing interference****:* When the interference is low enough (for instance, between two small applications) and the performance decrease can be afforded by all applications involved, then letting the applications interfere can also be a valid choice and lead to better performance than trying to schedule them.

4) ***Adapting dynamically to the best strategy****:* Each strategy having its own advantage and drawbacks, a mechanism can be implemented to select the best option at run time depending on information exchanged between applications. The choice of a strategy over another should be made on the basis of a system wide efficiency metric. For instance, if our goal is to minimize the sum of interference factors $f = \sum_{X \in App} I_X$, we will try to avoid the case of a small application being largely impacted by a big one, by serializing the big one after the small one or by interrupting the big one to favor the small one.

The first three strategies are presented in Figure 5. These strategies all require that an application becomes aware of other applications running on the system, or at least the properties of on going I/O operations, and even have a way to contact other applications to exchange these properties. To this end, we designed a coordination approach illustrated by the CALCioM framework, which includes all these strategies and allows applications to communicate with each other in order to implement them.

Note that these strategies naturally extend to more than two applications. The adaptive strategy would then consist in either choosing a place in a queue of applications that have requested access to the system, or interrupting the one currently accessing it.

B. CALCioM: Design Principles

CALCioM provides a way for applications to communicate with each other in order to make a decision on the best I/O scheduling strategy. Deciding could be done by the applications themselves or enforced by a system-provided entity (this detail is outside the scope of this paper, as our goal is to show the possibilities offered by the sharing of information between applications through a common communication layer). CALCioM seeks the optimization of a set of concurrent applications, rather than optimizing each application individually, and thus considers the set of applications running concurrently rather than each application individually.

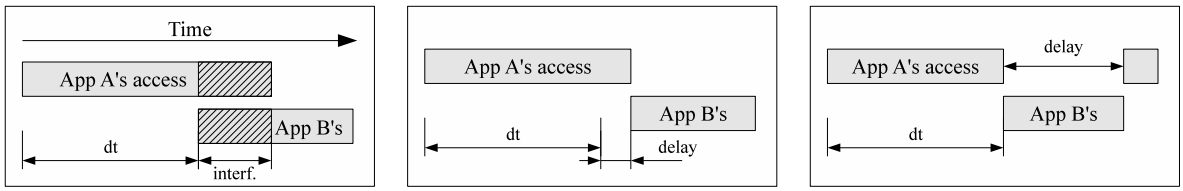
With applications, CALCioM works from knowledge acquired in each layer of the I/O stack, considering the I/O stack as a whole instead of a set of layers (application, I/O library, MPI-I/O, file system) to be optimized individually. For instance, CALCioM will get from the application level how many files (or how many bytes) are intended to be written and from MPI-I/O the series of raw requests to the file system, the targeted storage servers, the number of rounds of collective buffering, and so on.

A design choice central to our approach is that CALCioM *does not* give to the user a dangerous *lock* function to prevent multiple applications from accessing the file system at the same time. Nor does it offer a way for an application to force the interruption of another. CALCioM only provides the means by which applications can communicate. CALCioM can be transparently integrated in the I/O stack of applications and use the information exchanged by different applications to make a decision on the their behavior.

As an example of CALCioM-enabled behavior, consider an application A writing a large amount of data. As another application B starts an I/O phase, it contacts A with some information regarding its expected I/O operations, for example, a well-optimized write of a small amount of data. If, targeting the optimization of a given metric, A (or a centralized entity) considers that stopping and letting B execute its access will lead to better overall performance, it will contact B back with this decision. When B finishes its I/O, A resumes its own operation.

C. Effective implementation and API

The communication between different applications and the gathering of information on I/O behaviors are done only through one process in each application (typically rank 0 in MPI_COMM_WORLD) or a very small set. This



(a) Interference (b) "First-come-first-served" serialization (c) "Interruption-based" serialization

Figure 5. Three possible policies to deal with interference: **(a)** let applications A and B interfere, both will be impacted; **(b)** serialize one application after the other, giving the advantage to the one that started its access first and impacting the second one only; and **(c)** interrupt one application for the benefit of another one, impacting the first one only.

process, called a *coordinator*, is responsible for gathering information from other processes inside the application, for interacting with other applications, and for sending orders back to inner processes on information about accesses to be performed. CALCioM is thus hierarchical in the sense that an application can internally have its own way of managing I/O, and CALCioM acts as the root of each application's particular I/O management system.

CALCioM exposes a simple API to application/libary/middleware developers.

`Prepare(MPI_Info info)` adds more information about the future I/O accesses. In order to be generic, it uses an MPI_Info structure, which contains a set of *(key,value)* pairs, to represent knowledge on the application's I/O behavior. As examples of values that can be leveraged, in Section IV we communicate the number of files, the number of rounds of collective buffering and the amount of data transferred per round. A call to **`Complete()`** will later unstack information.

`Inform()` sends the information to the set of running applications currently doing I/O, as well as applications interrupted or waiting. Suggestions of authorizations are eventually sent back by these applications.

`Check(int* authorized)` checks whether the application is "allowed" to access the file system, based on other applications' responses.

`Wait()` explicitly waits for all the other applications to agree that this application should do its I/O access.

`Release()` ends a step in the I/O access, checks for pending requests from other applications, reevaluates the global strategy (if new information has been sent), and responds to other applications. A new call to **`Inform`** is necessary before the next I/O access.

In coordinators, all these functions perform communications with other applications. Other processes perform communications with the coordinator. Retrieving the list of other running applications is done through communications with the machine's job scheduler when the job starts and finishes. This API is intended to be used at several levels of the I/O stack, from the application level down to the MPI-I/O implementation. `Inform`, `Check`, `Wait` and `Release` can be used at the low level between each atomic request to the file system, surrounding a complex *write* operation or even an entire I/O phase. The reason for also offering these functions to application and library developers is that they can also observe the load of the storage stack at any point in the program and decide to schedule their operations differently (for instance, starting a new iteration of computation and coming back to the I/O phase later). This is, however, beyond the scope of this paper.

The location of these calls gives different degrees of freedom in adapting the I/O behavior; using the functions only between each file access gives less opportunity for the application to be interrupted upon request from another application, for example. Each level can use `Prepare` and `Complete` to provide information that can lead to a better understanding of their I/O behavior and thus to better decisions.

D. Implementation options

CALCioM can be implemented by using MPI as underlying communication layer in order to be platform independent. Indeed MPI already provides functions to build a communicator across multiple applications (`MPI_Comm_{connect,accept}`, `MPI_Port_{open,close}`), as applications start and leave. Yet, since an application cannot know when another one will try to contact it, these connection primitives should be made non-blocking, either by extending the MPI standard with `MPI_Comm_{iconnect,iaccept}` or by calling these functions from a thread (something that is also done in [15]). Such a thread would be required only in coordinator processes, that is, one extra thread per application.

In a production system, one might want to implement these primitives at a system level, in order to improve their security, and to back up the coordination algorithm with a centralized entity to enforce the decisions taken on the basis of the I/O behaviors. For large scale systems it might also be more effective to perform coordination via a separate service running on the system, rather than the peer-to-peer approach used in our prototype. Systems such as BlueGene/Q running the operating system in a spare core [16] would offer a good way of providing fully asynchronous, system-level communications with other applications.

For this work, we implemented CALCioM in pure MPI. We avoided the problem of using `MPI_Comm_accept` by having all our instances launched through the same *mpirun* command and sharing `MPI_COMM_WORLD`. By this way all applications start with a communicator, allowing them to

talk to each other; and we are provided with an easy way of controlling all instances at the same time.

IV. Experimental evaluation

The following experimental campaign aims to present the different policies that CALCioM offers: (1) letting applications interfere, (2) waiting for another application to complete its I/O, (3) interrupting an application's access, and (4) dynamically selecting one of the above policies.

A. Platforms and methodology

The study of cross-application interference requires reserving a full machine in order not to impact (or be impacted by) other applications. We choose the following machines for this purpose.

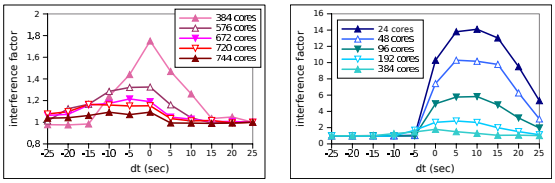
Surveyor is a 4096-core (1024 nodes) BlueGene/P supercomputer at Argonne, running at 13.6 TFlops. It exposes a 4-node PVFS2 shared file system for high-performance I/O. Surveyor consists of one rack of Argonne's Intrepid machine [17] and therefore shares the same architecture. Note that Surveyor's PVFS2 file system is not shared with Intrepid; thus, reserving the full machine ensured that at worst only a user connected to the frontend of Surveyor could interfere with our experiments.

Grid'5000 is the French grid [9] deployed across 10 sites in France and Luxembourg. We mainly used the Rennes site, more specifically the *parapluie* cluster (40 nodes featuring 2 AMD 1.7 GHz CPUs, 12 cores/CPU, 48 GB RAM) and *parapide* (25 nodes featuring 2 Intel 2.93 GHz CPUs, 4 cores/CPU, 24 GB RAM, 434 GB local disk). All nodes of these clusters are connected through a common InfiniBand switch. OrangeFS 2.8.3 was deployed on 12 nodes of *parapide*, using an *ext3* backend file system on local disks with caching disabled in order to avoid the huge performance drop observed in Section II. Reserving these two clusters and deploying our own file system ensured us to be the only users of the IB switch as well as the file system at the time of the experiments.

Using real-life applications to evaluate cross-application interference is arguably not appropriate because (1) it is difficult to differentiate inner and outer causes of performance degradations in applications that exhibit a complex access pattern, (2) they may not be representative of generic interference patterns that applications with perfectly optimized I/O would exhibit, and (3) we need a way to control precisely the moment when these applications perform I/O. Therefore, we developed a benchmark similar to IOR [10] that starts by splitting its set of processes into groups running independently on different nodes. This IOR-like benchmark allows us to control the access patterns of each group of processes (for example, contiguous or strided with a specified number of blocks and block sizes, in a way similar to IOR). For this paper specifically, our study focuses on collective write operations and write/write interference between two applications only.

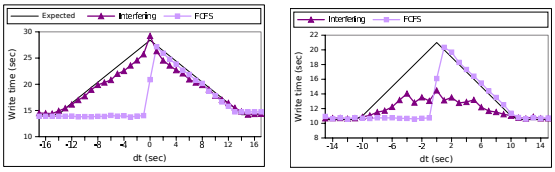
B. Interfering or serializing accesses

Figure 6 completes our study of interference initiated in Section II between applications running on different



(a) $\Delta-$graph of App A (big) (b) $\Delta-$graph of App B (small)

Figure 6. Experiments done on Grid'5000. A total of 768 cores is split into two groups of N (App B) and $768 - N$ (App A) cores, for $N \in \{24, 48, 96, 192, 384\}$. Each application writes 16 MB (8 strides of 2 MB) per process.



(a) 2× 2048 cores (b) 2× 1024 cores

Figure 7. Experiments done on Surveyor. Two applications of the same size write 32 MB per process using a contiguous pattern: **(a)** the applications are big enough to interfere with each other; **(b)** the applications are smaller and the interference is not as high as expected.

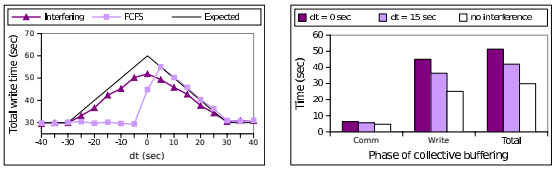
numbers of cores. We observe that the small application (B) is a lot more impacted than the big one, with an interference factor going up to 14 for a 24-core instance competing with a 744-core instance. On the left part of the graphs ($dt < 0$), B manages to write before A starts writing, which prevents them from interfering. On the right part, however, B starts while A is already writing, thus leading to interference. Provided that we try to minimize the sum of write times, or the sum of interference factors, regardless of the number of cores on which the applications run, a smarter strategy consists in having instance A wait for B to have completed its write before starting its own operation (*i.e.*, being on the left side of the Δ-graphs as often as possible). This is possible only if B starts writing before A and A has a way to know that B is writing, in which case the choice of waiting for B to complete is left to either A or a system-provided entity that enforces the decision.

Yet given a time interval $[t_1, t_2]$ during which both A and B are expected to complete exactly one I/O phase, we can show that the probability for B to start writing while A already started (in which case B will either have to interfere with A or be serialized after it) follows:

$$\mathbb{P}(dt < 0) = \frac{T_{A(alone)}}{t_2 - t_1}$$

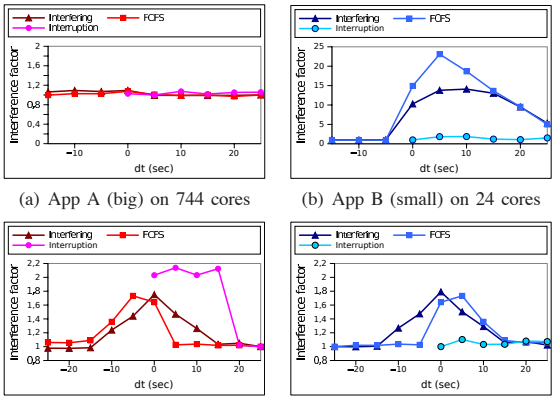
The bigger the difference in size between the applications, the less likely relying only on an FCFS policy will allow us to achieve our target of system wide efficiency.

Figure 7 (a) shows two accesses from instances of the same size serialized one after the other. Contrary to letting these applications interfere, only the application that accesses second is impacted and experiences a performance degradation that is equivalent to that of an interference with



(a) $\Delta-$graph (b) Phases of collective buffering

Figure 8. Experiments done on Surveyor. Two applications of the same size (2048 cores each) write 16 MB per process using a strided pattern (16 blocks of 1 MB per process), triggering the collective buffering algorithm. Figure (a) shows the $\Delta-$graph when interfering and when the applications are serialized one after the other. Figure (b) shows how each of the two phases behave: the communication phase is almost not impacted (it is impacted as a side-effect of the variability in the write phase of different processes), while the write phase is the most impacted.



(a) App A (big) on 744 cores (b) App B (small) on 24 cores

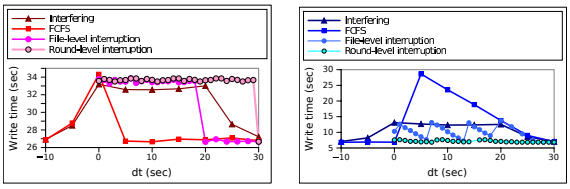
(c) App A (big) on 384 cores (d) App B (small) on 384 cores

Figure 9. Experiments done on G5K Rennes site. Two applications write 8 MB per process using a strided pattern, from a different number of cores (App A runs on 744, 720, 672, 576, and 384 cores, and App B respectively on 24, 48, 96, 192, and 384, for a total of cores of 768). We show how the interference factor behaves for the 3 policies: accepting the interference, serializing one app after the other (very bad for B when B is small, as shown in Figure (b)), and interrupting A (very bad for A if B is of the same size, as shown in Figure (c)).

the first application. The application writing first, however, is not impacted anymore, hence leading to a better overall system performance.

The limitations of the FCFS strategy are however numerous. Figure 7 (b) presents a case where the applications have the same size, but this size being small, the compound A+B tolerates rather well the interference. Serializing the accesses will benefit only the first one, at the expense of the second. In Figure 8 (a), each instance uses a collective, strided access pattern. This access pattern triggers the collective buffering algorithm (also termed "two-phase I/O") that introduces collective communication steps. These communications are less subject to an interference, as shown in Figure 8 (b), and therefor, serializing the accesses has a higher impact on the application arriving second than pure interference.

The experiments presented in Figure 9 show that FCFS serialization has a positive effect when applications have a similar size or similar I/O requirements (Figures 9 (c) and (d)). However, when they have very different sizes



(a) App A (big) (b) App B (small)

Figure 10. Experiments done on Surveyor. App A and B run on 2048 cores each, App A writes 4 files using 4MB per process (contiguous access), App B writes only 1 such a file. These graphs show the interference factor of the two applications depending on the strategy used and on dt.

(Figures 9 (a) and (b)) or very different I/O requirements, the FCFS serialization leads to an important performance degradation of the small application when this application arrives second. As explained above, at equivalent access frequency between the two application, the situation of a small application accessing before the big one is less likely than its opposite. Depending on the global efficiency targeted, it may be desirable in these situations that the big application be interrupted for the benefit of the small one.

C. Interrupting accesses

By calling *Inform/Release* frequently, an application has the possibility to receive information from another applications more often, and also be interrupted at a finer grain. Figure 9 also present the results of experiments where the application accessing second interrupts the one accessing first, regardless of the size or I/O requirements of each application. These experiments are done with a strided write pattern using collective buffering, and *Inform/Release* are called at very fine grain before and after each atomic call to independent contiguous writes in a custom, CALCioM-enabled ADIO layer for ROMIO. The interruption being possible only when $dt > 0$ (e.g., there is someone to interrupt), the curves start at $dt = 0$. These figures shows that, as expected, the interruption strategy has the opposite effect to FCFS serialization; it is effective when a small application interrupts a big one, but it becomes ineffective and even counterproductive when applications have a similar size.

Figure 10 shows results on Surveyor with interference, FCFS serialization and interruptions. In these experiments, applications have the same size; however, A writes four files while B writes only one. The *Infom/Release* functions have been set up in two different levels: in the ADIO layer (between each round of collective buffering) or at the application level between each file. The second case leads to the "saw" pattern because A cannot be interrupted at a fine grain, and is forced to finish writing a file before being interrupted. An implementation in the ADIO layer offers more possibility for A to interrupt its access quickly enough for B not to be impacted.

D. Dynamic choice: interfering, serializing, or interrupting

The previous sections have demonstrated the pros and cons of different policies made possible by CALCioM thanks to cross-application coordination. To close the loop,

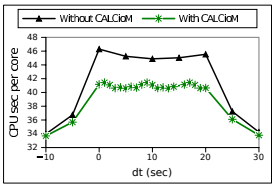


Figure 11. Synthesis on CALCioM's choices and impact on the specified metrics (computational efficiency). The figure shows the CPU seconds per core wasted in I/O under interference, and with CALCioM.

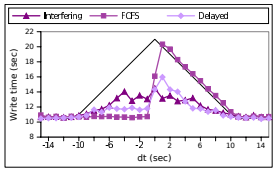


Figure 12. 2× 1024 cores write 32MB/process (contiguous pattern). The interference is not as high as expected. As a consequence, serializing accesses is not a good decision. A tradeoff can be found by slightly delaying one of the writes.

we integrated all three policies in CALCioM and made it select the most appropriate strategy dynamically, based on information exchanged between applications. This selection is based on the targeted machine wide efficiency metric. In this section, we consider an example of such a metric; namely, the total number of CPU hours actually used for doing science, and show how CALCioM can select the best strategy. We will thus aim at minimizing the total number of CPU hours wasted in I/O phases: $f = \sum_{X \in Apps} N_X \times T_X$ where N_X is the number of cores running application X, and T_X is the observed I/O time. Note that this metric does not necessarily favor a small application, since it weights the I/O time with the amount of computing resources. However, it will favor a big application with small I/O requirements.

We consider the scenario presented in Figure 10 where $N_A = N_B = 2048$ cores on Surveyor, and B writing four times less data than A. The case of B starting before A is trivial (A is serialized after B). Thus we consider only $dt > 0$; B either interrupts A or is serialized after it. Using the above definition of f, we can compute the expected cost of each of the policies.

$$\begin{aligned} f_{FCFS} = &\quad 2048 \times (2T_{A_{(alone)}} + T_{B_{(alone)}} - dt) \\ f_{Interrupt} = &\quad 2048 \times (T_{A_{(alone)}} + 2T_{B_{(alone)}}) \end{aligned}$$

A should be interrupted if and only if $f_{Interrupt} < f_{FCFS}$, which translates into $dt < (T_{A_{(alone)}} - T_{B_{(alone)}})$.

As a result, if B starts first, A is serialized after B; if B starts before A finished writing 75 percent of its data (*i.e.*, 3 out of 4 files), A is interrupted; otherwise B is serialized after A.

The result of these decisions on the value of f is summarized in Figure 11 (lower is better) and compared with the situation of applications simply interfering without CALCioM involved. Considering this specified metric of computational efficiency, CALCioM always manages to make a decision that improves this metric; that is, it lowers the global time wasted in I/O per core.

Selecting a policy does not necessarily mean simply choosing between FCFS and interruption. Indeed, in a context where the observed interference is lower than expected, as in Figure 12, we have shown in Section IV-B that serializing (or interrupting) is not a good option. More elaborate decisions could be made, such as delaying an application and allowing some degree of overlap. This decision still depends on the specified system wide efficiency metric to optimize, but this time, it requires a better estimation of the interference by the applications, an estimation outside the scope of this paper.

V. Related work

This section presents the background and related work. We first present how applications usually optimize their I/O through data reorganization and synchronization. We then present approaches that attempt to better schedule I/O requests at single-application-level as well as approaches that take multiple applications into account at the storage side or through forwarding techniques.

A. I/O optimization in HPC applications

Common application-side I/O optimization techniques consist in reorganizing data access patterns in order to read or write large contiguous chunks that potentially match the file's layout in the storage servers. These techniques include two-phase I/O [18] data sieving [19] or list I/O [20] and are now present in most MPI-I/O implementations with specific optimizations for the underlying file systems [21], [22]. Node-level I/O scheduling [23] is also a common approach to prevent multiple processes in the same node from concurrently accessing the network interface. The advantages of these approaches are lost in the context of cross-application interference, as it breaks the data locality individually optimized or leads to different storage servers servicing distinct applications in a different order. Sharing information between applications can prevent from breaking such optimized patterns.

B. Application-side I/O scheduling

Zhang *et al*. [24] propose an approach that couples the I/O scheduler and process scheduler on compute nodes. When an application becomes I/O intensive, processes fork to create new processes that executes the same code only to retrieve information on the future I/O requests that will be issued. These pre-execution processes are then killed and the main processes can leverage knowledge from their own future I/O requests. Their implementation is done under MPI-I/O and in PVFS. This techniques is complementary to our approach; it manages to give a prediction of future I/O behavior that could then be leveraged by CALCioM.

Based on the observation that within a single application, processes already interfere with each other, Lofstead *et al*. [25] propose an adaptive I/O approach in which the processes of an application are gathered in groups. Each group writes in a particular storage target, and one process in each group is chosen to coordinate the accesses issued by all the other in the group. This drastically reduces I/O variability within a single application. CALCioM targets the same goal but at machine scale, between multiple applications, a task inherently more difficult because of the lack of knowledge that applications have from each other and the diversity of I/O workloads.

The use of dedicated I/O cores [5], [26]–[28], threads [29], [30], or dedicated nodes [31], [32] (also termed "staging areas") is becoming more and more common. These strategies overlap I/O with computation by shipping data to dedicated resources, and offer more liberty in delaying actual I/O accesses. These approaches are important to us: dedicated cores and threads provide a good opportunity for an application to analyze its own I/O behavior and make decisions based on a predicted future resources usage.

C. Server-side I/O scheduling

I/O scheduling techniques implemented in parallel file systems aim at lowering disk head movements caused by unrelated requests (*i.e.*, achieving better *data locality*), and better distributing I/O requests across multiple data servers. This objective implies (1) trying to service applications one at a time and (2) trying to force all the data servers to serve the same application at the same time, while keeping fairness across multiple applications. Our experimental evaluation clearly showed that serializing I/O requests without knowledge of the applications' I/O load can lead to machine wide inefficiency.

Qian *et al*. [33] present a network request scheduler built in Lustre [34]. They propose to associate deadlines to requests, as well as their targeted object's identifier, in order to first service requests belonging to the same object while preventing starvation by taking deadlines into account. They propose to dynamically adapt the deadline value depending on the load on the file system, and to add mandatory deadlines for requests that correspond to critical I/O operations (a cache becoming full in the client, for instance). The same goal is achieved by Song *et al*. [35], with an application's id instead of an object id.

Zhang *et al*. [36] propose to coordinate the schedulers of each data server in order to meet QoS requirements set by each application in terms of application run time. The required application run time is converted into bandwidth and latency bounds through machine learning techniques. The application I/O behavior must be extracted from a first run on a dedicated platform. I/O schedulers in data servers then allocate time windows to serve one application at a time in a coordinated manner. Our approach does not require machine learning techniques but requires the sharing of information between application. We also aim at improving machine wide efficiency instead of targeting single applications QoS.

In [7] a "reuse distance" is used 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. In contrast, CALCioM coordinates all running applications without the need for requests to carry an ID, or for the file system to wait arbitrarily for potential new requests to arrive.

Other approaches such as the one from Lebre *et al*. [37] provide multi applications scheduling with the goal of better aggregating and reordering requests, while trying to maintain fairness across applications. The proposed solution does not take into account each application's available resources and required I/O efficiency and does not check the availability of the file system to potentially change the application's behavior.

Closer to our approach is the work from Batsakis *et al*. [8], where the observation is made that different clients with different resource usage should be serviced differently by the file system. In there solution, clients price their requests depending on their ability to delay them (which depends on the memory usage on the client). The server also prices all requests based on its own availability. An auction mechanism is then implemented to chose whether a request should be serviced or delayed. This mechanism is constrained to asynchronous requests and involves communications between the client and the server to set up the auction.

Tanimura *et al*. [38] propose to reserve throughput from the storage system. Their system is implemented in the Papio file system. Applications have to define their requirements in terms of throughput either when submitting a job or at run time, and the level of service is controlled by a centralized manager. Reserving throughput may not be a effective way of improving machine wide efficiency, as it locks resources. We approached the problem in a different way by giving the maximum performance possible to all applications, and to resolve interferences as they occur based on a specified metrics of platform efficiency.

To our knowledge, none of the existing approaches to I/O scheduling and I/O optimizations leverage both the facts that (1) applications can themselves communicate with each other and self-coordinate and (2) applications have different constraints related to their resources usage, I/O load and behavior, which should be taken into account when targeting machine wide efficiency.

VI. Conclusion and future work

Cross-application interference in HPC systems, and more particularly in their I/O system, is an important problem that can affect the efficiency of the entire machine. This problem will be even more important with exascale machines that will allow running more applications in a concurrent manner. In this paper we explore the effect of cross-application contention on their I/O performance; and we propose the CALCioM approach, which provides a mean by which independent applications can communicate with each other in order to coordinate their I/O strategy, targeting system wide efficiency. We illustrate the usefulness of our approach through experiments on two platforms: Argonne's Surveyor and the French Grid'5000 testbed. For example, CALCioM is able to prevent a $14\times$ slowdown of a small application competing with a larger one, at a negligible cost for the latest, by allowing the interruption of its on going I/O operations. CALCioM opens a wide range of new possible scheduling optimizations through the sharing of I/O properties between applications. We intentionally focus our study on interference between two applications only, as displaying interference factors in the context of more than two applications is arguably difficult.

As future work, we plan to take into account the fact that an interrupted application can reorganize some of its internal operations (communications, compression, data processing,

etc.) while waiting for its I/O to be resumed in order to further gain time. Additionally, we will investigate more theoretical models of interference in order for CALCioM to make the right choice of scheduling policies. Finally, we plan to investigate the more complex case of multiple applications accessing overlapping sets of targets in a storage array.

ACKNOWLEDGMENT

This work was supported by the U. S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357. It was done in the framework of a collaboration between the KerData INRIA - INSA - ENS Rennes team and Argonne National Lab within the Joint INRIA-UIUC-ANL Laboratory for Petascale Computing. We also thank Michel Raynal for our fruitful discussion on mutex algorithms, Rob Latham and the PVFS2 and Mpich developers for their help in understanding their tools. Part of the experiments were carried out using the Grid5000/ALADDIN-G5K experimental testbed (see http://www.grid5000.fr/).

REFERENCES

[1] "Top500 supercomputers, http://www.top500.org/."
[2] Parallel Workload Archive, "http://www.cs.huji.ac.il/labs/parallel/workload/."
[3] C. Chilan, M. Yang, A. Cheng, and L. Arber, "Parallel I/O Performance Study with HDF5, a Scientific Data Package," in *TeraGrid Technical Report*, Jul. 2006.
[4] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "DataStager: Scalable Data Staging Services for Petascale Applications," in *ACM HPDC '09*, Jun. 2009, pp. 39–48.
[5] 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 *IEEE Cluster '12*, Sep. 2012, pp. 155–163.
[6] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: a Checkpoint Filesystem for Parallel Applications," in *ACM/IEEE SC '09*, Nov. 2009, pp. 1–12.
[7] X. Zhang, K. Davis, and S. Jiang, "IOrchestrator: Improving the Performance of Multi-node I/O Systems via Inter-Server Coordination," in *ACM/IEEE SC '10*, Nov. 2010, pp. 1–11.
[8] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey, "CA-NFS: a Congestion-Aware Network File System," in *FAST '09*, Feb. 2009, pp. 99–110.
[9] INRIA, "Aladdin grid'5000: http://www.grid5000.fr."
[10] 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.
[11] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing Performance Interference Effects for QoS-Aware Clouds," in *ACM EuroSys '10*, Apr. 2010, pp. 237–250.
[12] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu, "Understanding Performance Interference of I/O Workload in Virtualized Cloud Environments," in *IEEE Cloud '10*, Jul. 2010, pp. 51–58.
[13] D. Skinner and W. Kramer, "Understanding the Causes of Performance Variability in HPC Workloads," in *IEEE Workload Characterization Symposium*, 2005, pp. 137–149.
[14] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker, "Parallel I/O Performance: From Events to Ensembles," in *IEEE IPDPS '10*, Apr. 2010, pp. 1–11.
[15] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi, "Using MPI in High-Performance Computing Services," in *EuroMPI '13*, Sep. 2013, pp. 43–48.
[16] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q Compute Chip," *Micro, IEEE*, vol. 32, no. 2, pp. 48–60, March-April 2012.
[17] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O Performance Challenges at Leadership Scale," in *ACM/IEEE SC '09*, Nov. 2009, pp. 1–12.
[18] "Evaluation of Collective I/O Implementations on Parallel Architectures," *Journal of Parallel and Distributed Computing*, vol. 61, no. 8, pp. 1052–1076, 2001.
[19] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *FRONTIERS '99*, Feb. 1999, pp. 182–182.
[20] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp, "Noncontiguous I/O through PVFS," Sep.
[21] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges, "MPI-IO/GPFS an Optimized Implementation of MPI-IO on Top of GPFS," in *ACM/IEEE SC '01*, Nov. 2001, pp. 17–17.
[22] P. Dickens and J. Logan, "Towards a High Performance Implementation of MPI-I/O on the Lustre File System," *On the Move to Meaningful Internet Systems OTM '08*, 2008.
[23] K. Ohta, H. Matsuba, and Y. Ishikawa, "Improving Parallel Write by Node-Level Request Scheduling," in *IEEE/ACM CCGrid '09*, May 2009, pp. 196–203.
[24] X. Zhang, K. Davis, and S. Jiang, "Opportunistic Data-driven Execution of Parallel Programs for Efficient I/O Services," in *IEEE IPDPS '12*, May 2012, pp. 330–341.
[25] 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 *ACM/IEEE SC '10*, Nov. 2010, pp. 1–12.
[26] M. Li, S. Vazhkudai, A. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman, "Functional Partitioning to Optimize End-to-End Performance on Many-Core Architectures," in *ACM/IEEE SC '10*, Nov. 2010, pp. 1–12.
[27] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *ACM/IEEE SC '10*, Nov. 2010, pp. 1–11.
[28] R. Prabhakar, S. Vazhkudai, Y. Kim, A. Butt, M. Li, and M. Kandemir, "Provisioning a Multi-tiered Data Staging Area for Extreme-Scale Machines," in *IEEE ICDCS '11*, May 2011, pp. 1–12.
[29] J. Fu, R. Latham, M. Min, and C. D. Carothers, "I/O Threads to Reduce Checkpoint Blocking for an Electromagnetics Solver on Blue Gene/P and Cray XK6," in *ROSS '12*, Jun. 2012, pp. 1–8.
[30] X. Ouyang, K. Gopalakrishnan, T. Gangadharappa, and D. Panda, "Fast Checkpointing by Write Aggregation with Dynamic Buffer and Interleaving on Multicore Architecture," in *IEEE HiPC '09*, Dec. 2009, pp. 99–108.
[31] X. Ma, J. Lee, and M. Winslett, "High-Level Buffering for Hiding Periodic Output Cost in Scientific Simulations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, pp. 193–204, 2006.
[32] A. Nisar, W. keng Liao, and A. Choudhary, "Scaling Parallel I/O Performance Through I/O Delegate and Caching System," in *ACM/IEEE SC '08*, Nov. 2008.
[33] 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.
[34] S. Donovan, G. Huizenga, A. J. Hutton, C. C. Ross, M. K. Petersen, and P. Schwan, "Lustre: Building a File System for 1000-Node Clusters," Nov. 2003.
[35] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-Side I/O Coordination for Parallel File Systems," in *ACM/IEEE SC '11*, Nov. 2011, pp. 1–11.
[36] X. Zhang, K. Davis, and S. Jiang, "QoS Support for End Users of I/O-Intensive Applications using Shared Storage Systems," in *ACM/IEEE SC '11*, Nov. 2011, pp. 1–12.
[37] A. Lebre, G. Huard, Y. Denneulin, and P. Sowa, "I/O Scheduling Service for Multi-Application Clusters," in *IEEE Cluster '06*, Sep. 2006, pp. 1–8.
[38] 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 *ACM/IEEE SC Companion*, Nov. 2012, pp. 1384–1384.