Scheduling the I/O of HPC Applications Under Congestion

Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, Marc Snir

Scheduling the I/O of HPC applications under congestion

Ana Gainaru[1*], Guillaume Aupy[2*], Anne Benoit[2], Franck Cappello[3], Yves Robert[2,4], and Marc Snir[1,3]
[1] *University of Illinois at Urbana Champaign, USA, againaru@illinois.edu*
[2] *École Normale Supérieure de Lyon & INRIA, France, {Guillaume.Aupy|Anne.Benoit|Yves.Robert}@ens-lyon.fr*
[3] *Argonne National Laboratory, USA, {cappello|snir}@mcs.anl.edu*
[4] *University of Tennessee Knoxville, USA*

***Abstract*—A significant percentage of the computing capacity of large-scale platforms is wasted because of interferences incurred by multiple applications that access a shared parallel file system concurrently. One solution to handling I/O bursts in large-scale HPC systems is to absorb them at an intermediate storage layer consisting of burst buffers. However, our analysis of the Argonne's Mira system shows that burst buffers cannot prevent congestion at all times. Consequently, I/O performance is dramatically degraded, showing in some cases a decrease in I/O throughput of 67%. In this paper, we analyze the effects of interference on application I/O bandwidth and propose several scheduling techniques to mitigate congestion. We show through extensive experiments that our global I/O scheduler is able to reduce the effects of congestion, even on systems where burst buffers are used, and can increase the overall system throughput up to 56%. We also show that it outperforms current Mira I/O schedulers.**

I. Introduction

With the advent of computationally demanding applications, parallel computers have continued to evolve toward post-petascale computing. At the same time, storage systems struggle to match the data generated by the computations of all running applications. The challenge is particularly obvious when many applications are executed concurrently. Indeed, while many I/O optimizations are available within each application (application-side collective I/O, software such as MPI-IO, and other network and disk-level optimizations [1], [2]), the interferences produced by multiple applications accessing a shared parallel file system in a concurrent manner frequently break these single-application optimizations.

The current server-side scheduling policies used by HPC systems at the file system level range from simple "first-come first-served" strategies for each storage server to more elaborated strategies. Recently, non-work-conserving disk schedulers, such as anticipatory scheduling [3] and the CFQ scheduler [4], have been designed to save the spatial locality with concurrent servicing of interleaved requests issued by multiple processes. This strategy keeps the disk head idle after serving a request of a process until either the next request from the same process arrives or the wait threshold expires. All policies, ranging from the simplest to more advanced ones, deal with low-level requests, without having any information from the applications. Consequently, current I/O schedulers cannot take advantage of particular properties or behaviors of each application and thus are not able to address the global efficiency of the system. As system size continues to increase, schedulers need to have a global view of the I/O requirements of all applications in order to make appropriate decisions.

In this paper, we focus on scheduling applications under I/O bandwidth constraints. Congestion due to I/O interference depends on many factors, from individual application size and computation-to-I/O ratio, to application interference as they start performing I/O with regard to one another. We analyzed the amount of time consumed by I/O operations on the Intrepid system at Argonne and shown in Figure 1 that congestion can cause up to a 70% decrease in the I/O efficiency of an application. We propose a global high-level scheduler that is aware of application I/O past behaviors and that dynamically coordinates I/O accesses to the parallel file system. Our contributions can be summarized as follows. (1) We design a global scheduler that minimizes congestion caused by I/O interference, by considering application past behaviors and system characteristics when scheduling I/O requests. We show that this scheduler reduces I/O delays incurred by each application and increases overall system throughput. (2) We build a simulator in order to test our scheduler in a large variety of scenarios and to assess its performance and limitations. We simulate the Intrepid and Mira systems and show that our heuristics obtain better system throughput and application dilation compared with what happens when congestion occurs. Notably, we report that a simulation of our scheduler without burst buffers achieves a better system throughput than the one observed on Intrepid in congested moments. (3) We implement the global scheduler on Argonne's Vesta computer and test its results when running the IOR benchmark. We validate our simulation model and show that, besides a small increase in the execution time of applications when congestion does not occur, the results of our implementation are much better than those of the current Vesta schedulers. (4) A striking result obtained on Vesta is the confirmation of the simulations: in most scenarios, our scheduler outperforms the use of burst buffers without having the incurred cost.

∗. These authors contributed equally to this work.

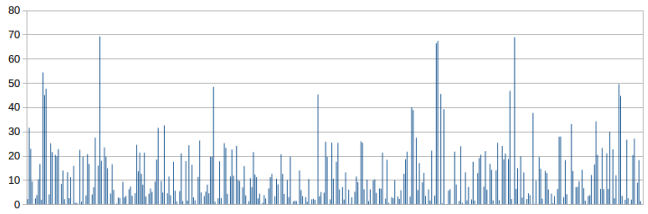


Figure 1: I/O throughput decrease (percentage per application instance, over 400 applications) on Intrepid.

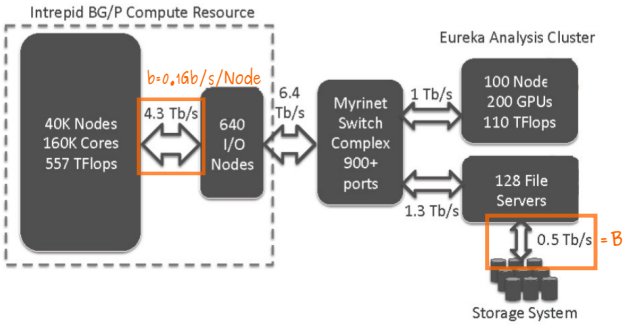


Figure 2: Model instantiation for the Intrepid platform.

The rest of the paper is organized as follows. We introduce the application model and optimization problems in Section II. We derive online scheduling heuristics in Section III. Through a full set of simulations in Section IV, we thoroughly evaluate and compare these heuristics, before reporting actual execution times on Vesta in Section V. We give some background and related work in Section VI. We provide concluding remarks and ideas for future research directions in Section VII.

II. Framework

In this section, we provide a formal description of the application and platform model, and we state scheduling objectives. We target a parallel platform composed of N identical unit-speed processors, each equipped with an I/O card of bandwidth b (expressed in bytes per second). This corresponds to the I/O network from the compute nodes to I/O servers on a typical cluster. We further assume a centralized I/O system with a total bandwidth B (also expressed in bytes per second) from these I/O servers to the disks. Figure 2 shows the model projected over Argonne's Intrepid architecture.

A. Application and platform model

We assume that K applications are running concurrently, each of them being assigned to independent and dedicated computational resources, but competing for I/O. In this work, we assume the I/O and communication network are separated, (i) so that network congestion caused by inter-node communications does not interfere with I/O transfers, and (ii) because it is the case in many systems (for example Mira and Intrepid [5]).

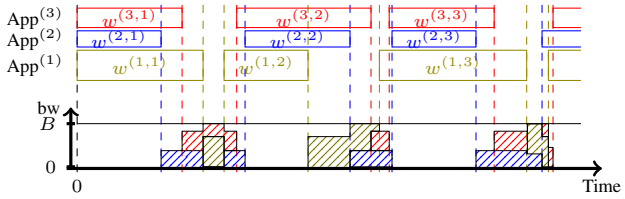


Figure 3: Scheduling three applications.

Each application $\text{App}^{(k)}$ is released on the platform at time r_k, executes on $\beta^{(k)}$ dedicated processors and consists of $n_{\text{tot}}^{(k)}$ *instances* that repeat over time until the last instance is executed. An instance is composed of some chunk of computations followed by some I/O transfer. More precisely, the i-th instance $\mathcal{I}_i^{(k)}$ of $\text{App}^{(k)}$ consists of $w^{(k,i)}$ units of computation (at unit-speed), followed by the transfer of a volume $\text{vol}_{\text{io}}^{(k,i)}$ of bytes to or from the I/O system. We consider that these actions do not overlap. We define d_k as the time when the last instance of $\text{App}^{(k)}$ is completed.

Because computational resources are dedicated, we can always assume w.l.o.g. that the next computation chunk starts right after completion of the current I/O transfers and is executed at full (unit) speed. However, this is not the case for I/O operations. All applications compete for I/O, and congestion will likely occur. The simplest case is that of an application $\text{App}^{(k)}$ using the I/O system in dedicated mode during a time-interval of duration D. For instance $\mathcal{I}_i^{(k)}$, $\text{App}^{(k)}$ needs to transfer $\text{vol}_{\text{io}}^{(k,i)}$ bytes. Let γ be the I/O bandwidth used by each processor of $\text{App}^{(k)}$ during this instance. We derive the condition $\beta^{(k)}\gamma D = \text{vol}_{\text{io}}^{(k,i)}$ to express that the entire I/O data volume is transferred. We must also enforce the following constraints: (i) $\gamma \leq b$ (output capacity of each processor); and (ii) $\beta^{(k)}\gamma \leq B$ (total capacity of I/O system). Therefore, the minimum time to perform the I/O transfers for the current instance of $\text{App}^{(k)}$ is

$$\text{time}_{\text{io}}^{(k,i)} = \frac{\text{vol}_{\text{io}}^{(k,i)}}{\min(\beta^{(k)}b, B)}.$$

In general, many applications will use the I/O system simultaneously, and the bandwidth capacity B will be shared among all these applications. The I/O of some applications may take place during several non consecutive time intervals and use different bandwidths. In Figure 3, we show an example of three applications competing for I/O bandwidth. The top part of Figure 3 shows the applications doing computations without any constraint. At the end of their computations, however, all applications need to transfer some volume of I/O, and thus they have to share the I/O total bandwidth B (bottom part of the figure). When these three applications want to execute I/O at the same time, congestion occurs, and the scheduler needs to choose which bandwidth fraction to assign to each application. The model is flexible and assumes only that at any instant all processors assigned to a given application are assigned the same bandwidth. This assumption is transparent for the I/O system and simplifies the problem statement without being restrictive.

The richness of the model comes from its flexibility for scheduling all the I/O transfers. It corresponds to a practical framework where the central scheduler would assign different I/O bandwidths per time-interval to each application. Depending on how many applications are trying to perform I/O, the scheduler might also decide to prevent some applications from accessing the disk during some time-intervals. This way, the scheduler controls the wait time for all applications and ensures that none of them is subject to excessive starvation.

B. Objectives

We first define $\tilde{\rho}^{(k)}(t)$, the *application efficiency* achieved for each application $\text{App}^{(k)}$ at time t, as

$$\tilde{\rho}^{(k)}(t) = \frac{\sum_{i \leq n^{(k)}(t)} w^{(k,i)}}{t - r_k},$$

where $n^{(k)}(t) \leq n_{\text{tot}}^{(k)}$ is the number of instances of application $\text{App}^{(k)}$ that have been executed at time t, since the release of $\text{App}^{(k)}$ at time r_k. Because we execute $w^{(k,i)}$ units of computation followed by $\text{vol}_{\text{io}}^{(k,i)}$ units of I/O operations on instance $\mathcal{I}_i^{(k)}$ of $\text{App}^{(k)}$, we have $t - r_k \geq \sum_{i \leq n^{(k)}(t)} \left(w^{(k,i)} + \text{time}_{\text{io}}^{(k,i)} \right)$. Without congestion, the schedule would achieve $t - r_k = \sum_{i \leq n^{(k)}(t)} \left(w^{(k,i)} + \text{time}_{\text{io}}^{(k,i)} \right)$, and the optimal application efficiency, where all I/O resources are available in dedicated mode for $\text{App}^{(k)}$, is

$$\rho^{(k)}(t) = \frac{\sum_{i \leq n^{(k)}(t)} w^{(k,i)}}{\sum_{i \leq n^{(k)}(t)} \left(w^{(k,i)} + \text{time}_{\text{io}}^{(k,i)} \right)}.$$

Because of I/O congestion, $\tilde{\rho}^{(k)}(t)$ never exceeds $\rho^{(k)}(t)$. We are ready to present the two optimization objectives, together with a rationale for each of them.

- SysEfficiency: Here the goal is to maximize the performance of the platform, namely the amount of CPU operations per time unit. This objective writes:

$$\text{maximize } \frac{1}{N} \sum_{k=1}^{K} \beta^{(k)} \tilde{\rho}^{(k)}(d_k).$$

Recall that $N = \sum_{k=1}^{K} \beta^{(k)}$ is the total number of processors, and that d_k is the time-step where $\text{App}^{(k)}$ terminates its execution. An upper limit of the system efficiency is $\frac{1}{N} \sum_{k=1}^{K} \beta^{(k)} \rho^{(k)}(d_k)$. The rationale is to squeeze the most flops out of the platform's aggregated computational power. This objective is CPU-oriented, since the schedule will give priority to compute-intensive applications with large $w^{(k,i)}$ and small $\text{vol}_{\text{io}}^{(k,i)}$ values.

- Dilation: Here the goal is to minimize the largest slowdown imposed to each application. This objective writes:

$$\text{minimize } \max_{k=1..K} \frac{\rho^{(k)}(d_k)}{\tilde{\rho}^{(k)}(d_k)}.$$

The rationale is to provide fairness across applications, and it corresponds to the stretch in classical scheduling: each application incurs a slowdown factor due to I/O congestion, and we want the largest slowdown factor to be minimal. This objective is user-oriented, since it gives each application a guarantee on its relative progress rate.

Note that both problems are hard, even in an (easier) offline setting.

Theorem 1. *The problem where all instances of an application are known in advance and are identical is NP-complete for either objective.*

The proof of this theorem, along with additional theoretical results on the offline problem, is available in the companion report [6].

III. Schedules

The scheduler monitors the stream of I/O calls and decides on the fly (as I/O calls appear in the system) which applications are allowed to perform I/O. We define an event as the start or the end of an I/O transfer by some application. At each event, the scheduler looks at the current state of the system, which is represented by the application efficiency and the amount of I/O already performed by each application. Then, based on a given strategy, it chooses a subset of applications and allows them to start or continue their I/O. This scheduler does not require any knowledge of the applications running in the system. Applications pay a supplementary cost because of the need to call the scheduler each time they need to perform their I/O. We show in Section V that this overhead is well paid off by the benefits of minimizing congestion.

Depending on the strategy used by the online scheduler to select applications at each event, the results might benefit either objective described in Section II-B. For each strategy, *favoring* application $\text{App}^{(k)}$ means that $\text{App}^{(k)}$ is executed as fast as possible, with bandwidth $\min\left(b\beta^{(k)}, \text{bw}_{\text{avail}}\right)$, where bw_{avail} is the available bandwidth at the moment the decision is taken. We experiment with several strategies.

- The RoundRobin scheduler favors available applications in a round-robin fashion similar to what the I/O scheduler is doing in HPC systems [7]. This heuristic is useful for comparison. The general idea of scheduling applications is "first-come first-served" (FCFS) with an additional constraint to ensure fairness. More precisely, each time an application needs to transfer some I/O, if there is no congestion, then this application is favored. Otherwise, the application that finished the I/O transfer of its last instance the longest time ago is favored.
- The MinDilation scheduler favors applications with low values of $\frac{\tilde{\rho}^{(k)}(t)}{\rho^{(k)}(t)}$.
- The MaxSysEff scheduler favors applications with high values of $\beta^{(k)} \frac{\rho^{(k)}(t)}{\tilde{\rho}^{(k)}(t)}$.
- The MinMax-γ scheduler favors applications with high

values of $\beta^{(k)}\frac{\rho^{(k)}(t)}{\tilde{\rho}^{(k)}(t)}$, unless there exists an application with a value $\frac{\tilde{\rho}^{(k)}(t)}{\rho^{(k)}(t)}$ below a certain threshold, γ, in which case it favors the application with the lower $\frac{\tilde{\rho}^{(k)}(t)}{\rho^{(k)}(t)}$. This threshold should be defined by the system administrator and depends on the DILATION targeted for the platform.

Note that since $0 \leq \frac{\tilde{\rho}^{(k)}(t)}{\rho^{(k)}(t)} \leq 1$, the MINMAX-$\gamma$ heuristic is exactly MINDILATION if $\gamma = 1$, and MAXSYSEFF if $\gamma = 0$. For all these heuristics, we have also implemented a PRIORITY variant. In this version, the scheduler always chooses applications that already started performing their I/O before favoring any other application. The rationale behind this is that an additional cost may be incurred by restarting the I/O of an application after an interruption, because of breaking disk locality. Breaking disk locality by alternating multiple applications accessing the device, affects performance and decreases the overall efficiency of the system [7]. Solid-state drives do not present this problem because they do not contain any moving mechanical components. Therefore, future clusters that use only SSD can use the original heuristics without paying the extra cost of not being able to choose the best possible applications that avoid congestion.

IV. SIMULATIONS

In this section, we report extensive simulations to assess the performance of the heuristics presented in Section III. In the first set of simulations, we thoroughly study the impact of each heuristic on different scenarios and use multiple applications with similar properties to real applications that ran on the Intrepid system. In the second set, we compare the heuristics to the I/O scheduler of Intrepid and Mira on traces of applications that run on these platforms when congestion occurs.

A. Applications

Intrepid is a Blue Gene/P supercomputer used by the Argonne National Laboratory between 2008 and 2014, it was ranked number 3 on the June 2008 Top 500 list. Consisting of 48 racks, 786,432 processors, and 768 terabytes of memory, Mira is a 10-petaflops IBM Blue Gene/Q system, 20 times faster than Intrepid and currently ranked number 5 on the June 2014 Top 500 list. A wide range of science and engineering applications have run on Blue Gene systems, including those used by the computational science community for cutting-edge research in chemistry, combustion, astrophysics, genetics, materials science, and turbulence. The typical behavior of scientific simulations is defined by alternating computation phases and I/O phases. The I/O phases are in general used either for writing out intermediary results for visualization purposes or for checkpointing. Intrepid uses separate networks for communication and I/O, which make it the perfect system to study the effects of congestion on application and system efficiency.

We use Darshan [8], an application-level I/O characterization tool developed at Argonne, to capture the behavior of applications running on Intrepid. Applications can be divided into the following categories [9]:

- *small applications* (run on less than 1,284 nodes);
- *large applications* (run on 1,285 to 4,583 nodes);
- *very large applications* (run on more than 4,584 nodes).

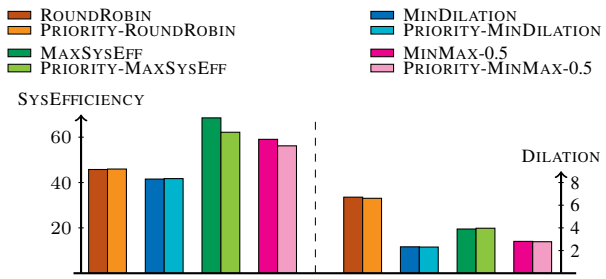
We use this information for generating the simulation scenarios. More details are available in [6].

In this section, we focus mainly on scheduling periodic applications under I/O bandwidth constraints. Periodic applications follow a pattern that is repeated over time: for all instances of $\mathcal{I}_i^{(k)}$, we have $w^{(k,i)} = w^{(k)}$ and $\text{vol}_{\text{io}}^{(k,i)} = \text{vol}_{\text{io}}^{(k)}$. Many examples of periodic applications exist in the HPC community. A simple example is an application that does not perform any I/O calls but implements a periodic checkpoint for reliability constraints [10]. Carns et al. [8] use the Darshan I/O characterization tool to capture an accurate picture of I/O patterns in Petascale workloads. In particular, they show that both the S3D application [11] (an application to simulate turbulent combustion using direct numerical simulation of a compressible Navier-Stokes flow) and the HOMME application [12] (an application to model atmosphere physics using spectral element techniques) periodically write out restart files through MPI-IO. Many other applications are periodic. For instance, we were able to verify that the following applications that run on Intrepid are in fact periodic: the gyrokinetic toroidal code (GTC) [13], Enzo [14], HACC application [15], and CM1 [16]. In Section IV-C we discuss the impact of application periodicity and show that results are similar for non-periodic applications.

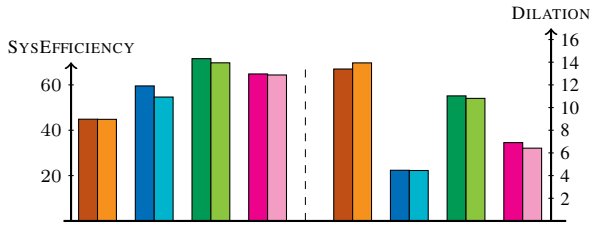
B. Assessment of the heuristics

By inspecting the mix of applications that ran on Intrepid, we observed that two scenarios cover over 95% of the cases: a few large or very large applications running alone on the whole system, or a mix of small and large applications dividing the machine un-uniformly. We compare the results of the different heuristics over different sets of applications (I/O intensive, computationally intensive, or a mix between the two) following these two scenarios. Figure 4 presents the corresponding results. Simulations were run 200 times on different application mixes that simulate real scientific applications running on Intrepid; only the mean values are reported.

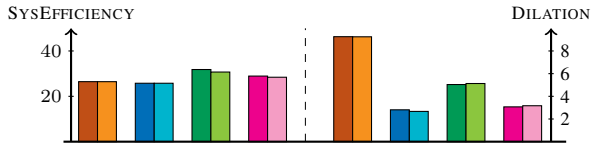
We first observe that the PRIORITY variants are, most of the time, less efficient than the original versions, especially when the number of applications running on the system increases. Adding the PRIORITY constraint lessens the flexibility in choosing the set of applications that would maximize the system efficiency. However, the difference in system efficiency and application dilation is small in all studied scenarios. This shows that the heuristics have



(a) 10 large applications, ratio of 20%



(b) 50 small and 5 large applications, ratio of 20%



(c) 50 small and 5 large applications, ratio of 35%

Figure 4: Objectives for different mixes of applications and I/O computation ratios.

good results even under the PRIORITY constraint, so that systems that use disks (which at this point represent the large majority of supercomputers) can still benefit from our scheduler.

Another (expected) observation is that MINDILATION has better results than MAXSYSEFF for the DILATION objective but worse results for the SYSEFFICIENCY objective. In particular, with 10 large applications and an average I/O ratio over computation of 20% (Figure 4a), the SYSEFFICIENCY of MAXSYSEFF can be up to 50% larger than that of MINDILATION, with a DILATION also up to 50% larger (recall that we want a large SYSEFFICIENCY and a small DILATION). The MINMAX-γ heuristic (run for $\gamma = 0.5$) is a good trade-off and achieves an intermediate result for both objectives. These results are confirmed, although less visible, in the second scenario (Figure 4b), with many small applications and a few large ones. In Figure 4c, the average I/O ratio over computation is 35%; there are 50 small applications and 5 large ones. In that case, the SYSEFFICIENCY of MAXSYSEFF can be up to 25% that of MINDILATION, for a loss in DILATION of 33%. Again, in that case, the MINMAX-γ heuristic is a good trade-off.

C. Impact of periodicity

Based on the literature and our own verifications on Intrepid, we have assumed so far that applications are

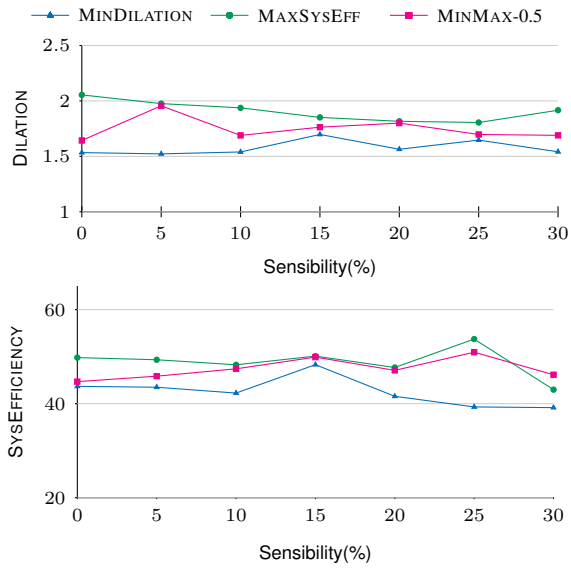


Figure 5: Impact of the sensibility of the computations over SYSEFFICIENCY and DILATION of all heuristics.

periodic. We now discuss the impact of having non periodic applications in the system. We define the sensibility of an application as $\text{Sens}_w^{(k)} = \frac{\max_i w^{(k,i)} - \min_i w^{(k,i)}}{\min_k w^{(k,i)}}$ and $\text{Sens}_{io}^{(k)} = \frac{\max_i \text{vol}_{\text{io}}^{(k,i)} - \min_i \text{vol}_{\text{io}}^{(k,i)}}{\min_k \text{vol}_{\text{io}}^{(k,i)}}$. For example, for a given application $\text{App}^{(k)}$, if the amount of work between two instances varies from 65 to 102 time units, then $\text{Sens}_w^{(k)} = 1 - \frac{65}{102} \approx 36\%$.

In Figure 5, we study the impact of the sensibility of $w^{(k)}$ for the three heuristics without the PRIORITY constraint. To compute each point on the $x\%$ sensibility axis, we have generated applications where the value of the computation has a continuous uniform distribution between $w_{\min}$ and $w_{\min}(1 + x\%)$. We see that this parameter has almost no impact on the results obtained with periodic applications. This result can be explained as follows. The heuristics have no global information about the applications that are being processed; they simply make scheduling decisions according to the information available at each event. We point out that the conclusion is similar when studying the sensibility of the I/O volume.

D. Intrepid and Mira simulations

In this section, we focus on comparing the PRIORITY variant of the MAXSYSEFF and MINDILATION heuristics, with the Intrepid and Mira schedulers as congestion occurs (Figures 6 and 7). Due to lack of space, we report only average results for their non-PRIORITY variants and for the MINMAX-γ heuristic (see Tables I and II). The full results are available in the companion report [6]. While the non-PRIORITY variant of all heuristics always outperforms the results of Intrepid and Mira (as can be seen in Tables I and II), in the following we present results only of the PRIORITY variant of the heuristics because Intrepid and

Table I: Averages over 56 different congested moments on Intrepid.

	DILATION (minimize)	SYSEFFICIENCY (maximize)
MAXSYSEFF	2.46	85.35
PRIORITY variant	3.13	82.98
MINMAX-0.25	2.33	83.08
PRIORITY variant	2.93	80.31
MINMAX-0.5	1.99	77.2
PRIORITY variant	2.43	72.96
MINMAX-0.75	1.69	71.66
PRIORITY variant	2.03	66.94
MINDILATION	1.63	70.45
PRIORITY variant	1.96	65.64
INTREPID	2.55	71.12
UPPER-LIMIT	-	91.59

Mira need disk locality.

Note that Intrepid and Mira use burst buffers to improve the behavior of applications with large bursts of I/O. Burst-buffers are an architectural enhancement that allow the I/O bandwidth to be supplemented. In these simulations we compare our heuristics without burst buffers with that of systems using burst buffers.

We have Darshan logs for every congested moment, describing the applications that were running at a given time. We use this information to create the simulation scenario used by our heuristics. The main limitation of the Darshan logs is that they give information only about the total execution time and the total amount of I/O performed by the applications but do not say anything about the application's actual behavior. Because most of the applications that run on Intrepid are periodic, we choose to enforce application periodicity by considering that these applications have a fixed number of iterations, each of a constant execution time and I/O volume. This fixed number is chosen in order to simulate the characteristics we have seen for applications running on Intrepid (Recall that Section IV-C has shown that the sensibility does not impact the results, so this hypothesis is not binding). Another limitation with Darshan logs is that they record only around 50% of all the applications running in the system. In most cases when congestion occurs, we did not have access to the information related to the other jobs running in the system. However, we did have information about the coverage of Darshan, so we replicated known applications in order to simulate similar conditions to the usage of the system at the moment of congestion.

In Figures 6 and 7, we observe the expected different behavior between MINDILATION and MAXSYSEFF, Intrepid's scheduler and the upper limit given by the characteristics of the applications running at that time. In general, the test cases where applications are I/O intensive (lower upper limit) present lower MINDILATION and higher DILATION values for all heuristics and for the Intrepid scheduler. The congested moments (when the difference between the upper limit and the results with the Intrepid scheduler is high) increase the gap between our heuristics and the Intrepid scheduler. We further investigated the few moments when

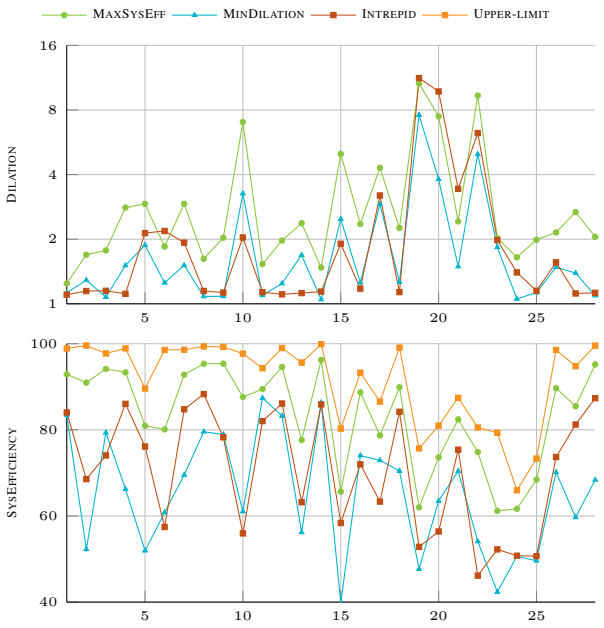


Figure 6: Comparison of the PRIORITY heuristics over the current DILATION and SYSEFFICIENCY of Intrepid.

Table II: Averages done over 11 different congested moments on Mira.

	DILATION (minimize)	SYSEFFICIENCY (maximize)
MAXSYSEFF	1.82	73.96
PRIORITY variant	2.41	70.26
MINMAX-0.25	1.71	71.58
PRIORITY variant	2.29	68.13
MINMAX-0.5	1.51	67.27
PRIORITY variant	1.94	64.88
MINMAX-0.75	1.31	62.24
PRIORITY variant	1.58	59.44
MINDILATION	1.27	61.62
PRIORITY variant	1.53	58.49
MIRA	2.01	64.26
UPPER-LIMIT	-	85.04

this is not the case (e.g., the 15th test case) and we observed that these test cases present a small number of large applications. In such a connect, some contention remains unavoidable.

Overall, the main result here is that without burst buffers, our heuristics have comparable results with those of Intrepid or Mira with burst buffers. This is impressive because burst buffers act as additional bandwidth to disks: when congestion occurs, as long as the burst buffers are not full, the applications can resume their execution right after they transferred their I/O volume to the burst buffer, instead of waiting for the I/O network to be available.

On Intrepid, MINDILATION on average improves DILATION by a 25% factor for a 8% loss in SYSEFFICIENCY while MAXSYSEFF improves SYSEFFICIENCY by 17% for a 20% loss in DILATION. MINMAX-0.5 improves both objectives, by 9.5% for DILATION and 2% for SYSEFFICIENCY. Our heuristics also show improvement compared

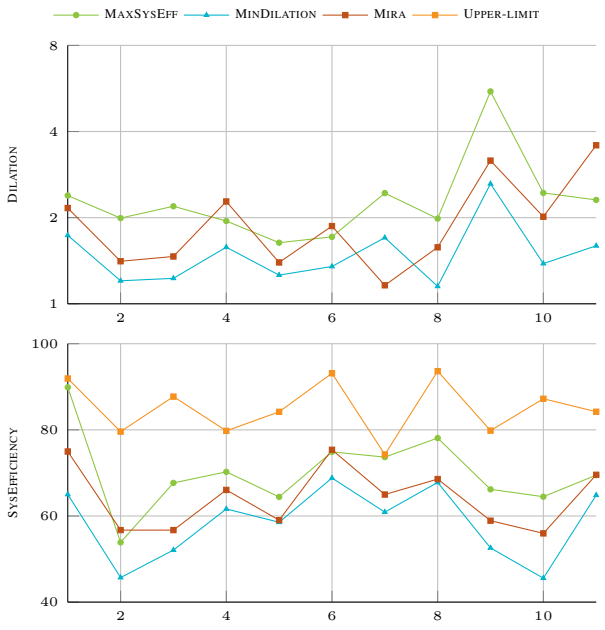


Figure 7: Comparison of the PRIORITY heuristics over the current DILATION and SYSEFFICIENCY of Mira.

with the scheduler used by Mira: MINDILATION improves on average the DILATION by a 24% factor for a 9% loss in SYSEFFICIENCY while MAXSYSEFF improves SYSEFFICIENCY by 9.3% for a 20% loss in DILATION. As before, MINMAX-0.5 improves both objectives, by 5.5% for DILATION and 1% for SYSEFFICIENCY.

Because Darshan is not covering all applications running in the system, and also because our model does not include any overhead induced by synchronizing the applications each time they perform I/O, we further validated the results by implementing our heuristics and running them on a real machine. We show in Section V that the results obtained in simulation accurately describe what would be obtained if Intrepid or Mira were using our heuristics.

V. EXPERIMENTS

The study of cross-application interference requires reserving a full machine in order not to be impacted by other applications running in the system at the same time. We have chosen the Vesta machine at Argonne for this purpose. Vesta [17] is a developmental platform for Mira. Its architecture is the same as Mira's except that it has two compute racks (Mira has 48). In total, Vesta has 2,048 nodes (32,768 compute cores). Applications running on this machine are completely isolated from each other. Thus, even if other applications are running on the system, their communications will not impact our experiments. Our focus in this section is directed to write/write interference between multiple applications.

A. Setup and measurements

The experiments require a way to control the exact moment when all applications perform I/O. Therefore, we modified the IOR benchmark [18] by splitting its set of processes into groups running independently on different nodes, where each group represents a different application. One separate thread acts as the scheduler and receives I/O requests for all groups in IOR. This way, our implementation of the IOR benchmark allows us to control the access patterns of each application. In addition, because IOR applications are communication-free, we modified them to include some inter processor communications at each step, in order to make them more similar to typical HPC applications. The added communication is an MPI_Reduce that adds the number of bytes written in the last iteration by each process and simulates the synchronization between different phases of a HPC application.

We made experiments on the modified IOR benchmark and compared the results with the performance of the original IOR benchmark, with and without using the option of bypassing I/O buffers. One group of a single process represents the scheduler; It is responsible for receiving online requests from the rest of the application processes each time they perform an I/O, and confirmations each time the I/O accesses are finished. Since Vesta uses hard disks and is influenced by the locality of disk access, we implemented the PRIORITY variants of the heuristics.

In this section, we report results only for MAXSYSEFF and MINDILATION. These heuristics correspond to extreme cases when a single objective is under consideration. We can always use the MINMAX-γ heuristic to obtain intermediate results that trade-off between system efficiency and application dilation. A system administrator could tune the threshold set for MINMAX-γ and obtain a variety of results within the range of values achieved by the two extreme heuristics presented here. We implemented the heuristics as an additional layer on top of the Vesta I/O scheduler, so that we could use the burst buffers available on Vesta during our comparison tests.

In the modified implementation of the IOR benchmark, each application process sends a request to the scheduler thread each time it needs to write some I/O volume. Figure 8 presents the overhead of adding the scheduling thread when no congestion occurs for different scenarios. This overhead was computed by comparing the execution time of one application running the original IOR benchmark, with the execution time of our modified version of the IOR benchmark that includes the scheduler. In order to fairly compare the execution time of adding the scheduler without accounting for its benefit in terms of scheduling decisions, in our comparisons, the scheduler always allows all requests to I/O. Depending on the frequency and amount of I/O for each application, the overhead in execution time varies

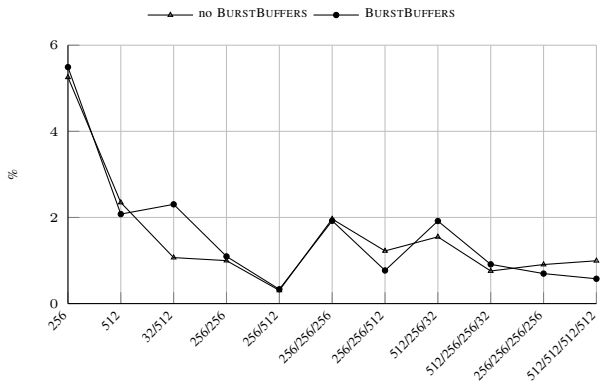


Figure 8: Execution time overhead of our implementation of the IOR benchmark.

between 1% and 5.3%. In general, for a larger number of applications, the execution time overhead remains under 3%. In Section V-B, we account for this idle time as well as the I/O throughput and application delays when computing the system efficiency and application dilation.

B. Results

Figure 9 shows the SysEfficiency and Dilation as seen by all applications running in the system for different scenarios. The horizontal axes present these scenarios in the form x/y/z, where x, y, and z represent the number of nodes used by each application in the system. For example 512/32 means two applications are running, one on 512 nodes and the other on 32. We made experiments without having any heuristic (results for IOR and IOR BB) and with the modified IOR benchmark using either MaxSysEff or MinDilation. For each case, we ran the application mix either bypassing or using the burst buffers (BB in the name).

We have studied the impact of our heuristic's overhead on the system efficiency and dilation by simulating two test cases with only one application running in the system (256 and 512 nodes, respectively). The results show that the overhead is translated into a small decrease in system efficiency (and increase in the max dilation) compared with running the IOR benchmark without any modification.

The results when running multiple applications are similar to what was seen simulating Mira and confirm what we have observed with the simulations: our heuristics perform well, better than Vesta's I/O scheduler when congestion occurs. Furthermore, the main result of this experimental setup is that with more than three applications, when congestion occurs, our heuristics without burst buffers perform similarly to, and sometimes better than, Vesta's current I/O scheduler with burst buffers.

In general, the MaxSysEff heuristic has larger dilation values than those obtained by letting congestion occur. With the MinDilation heuristic, system efficiency values follow the same curves as with the MaxSysEff heuristic but having, on average, values 5.65% lower. The maximum dilation,

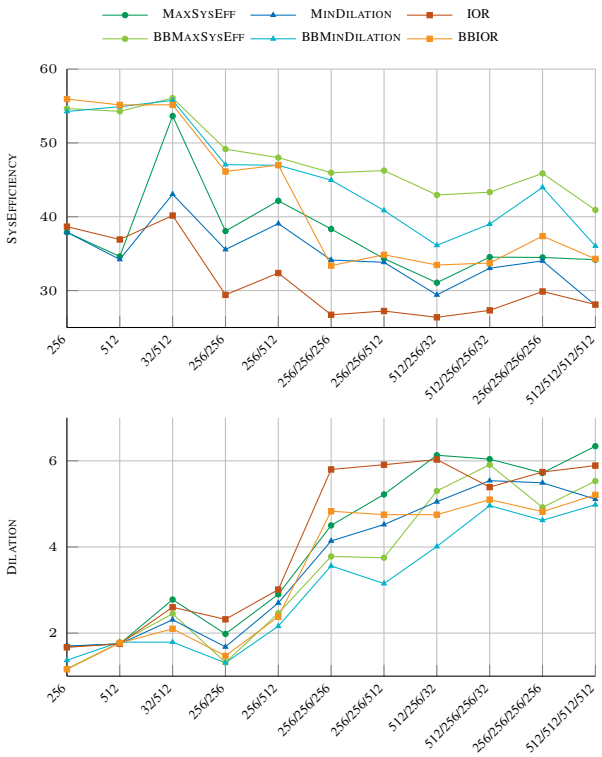


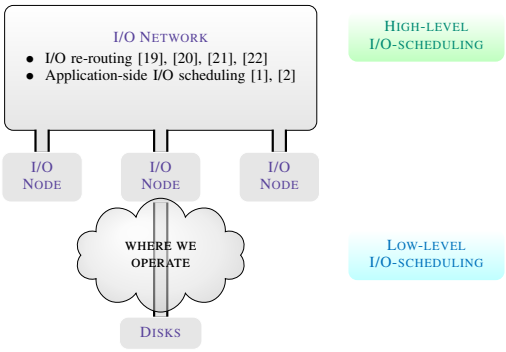
Figure 9: System efficiency and dilation for different scenarios on Vesta. Systematically, heuristics using burst buffers perform better than their counterparts without burst buffers.

however, decreases in all cases, showing values smaller than the congested counterpart in all studied scenarios. In general, the MinDilation heuristic has a more significant decrease for the dilation values than it had in the performance values in scenarios with more uneven applications (512/32 or 512/256/256/32). We study these scenarios further in the next paragraphs.

Figure 9 shows the dilation values for each of the four applications running in one of the analyzed scenarios. The small applications are in general more impacted by congestion than are the big ones when using the MaxSysEff heuristic, having an increase in their dilation value of 36% compared with the congested dilation. The big applications see a decrease in their dilation of over 48%, which is responsible for the good system performance values. When running the same application mix with MinDilation, the results show an almost uniform decrease in all application dilations compared with the results obtained running the benchmark without any heuristic, having on average a decrease of 8.4%, and a maximum decrease of 14.5% for the small application.

VI. Related work

Application performance variability can significantly detract from both the overall performance achieved by parallel workloads and the suitability of a given architecture for a workload. In distributed computing, the problem of having



performance variability due to sharing resources is well-known and studied. Numerous papers analyze this problem for clouds [23], [24], [25]. In [24], Pu et al present a study of interference specifically for I/O workloads in the cloud in order to understand the performance factors that impact the efficiency and effectiveness of resource multiplexing and scheduling among VMs. In [25], the authors investigate the sensitivity of measured performance in relation to consolidated server specification of virtual machine resource availability, and burstiness of n-tier application workload. Their results show that an increasingly bursty workload also increases the performance loss among the consolidated servers, however, without being able to offer a solution.

For the HPC community, while many works suggest that I/O congestion is one of the main problems for future scale platforms [26], [27], few paper focus on finding solutions at the platform level. Some papers consider application-side I/O scheduling [1], [2]. In particular, recently, several works have focused on using machine learning for auto tuning and performance studies [19], [20]. However, these solutions do not have a global view of the I/O requirements of the system, and they need to be supported by a platform-level I/O management for better results. Cross-application contention has been studied recently [28], [29], [30]. The study in [28] evaluates the performance degradation in each application program when Virtual Machines (VMs) are executing two application programs concurrently in a physical computing server. The experimental results indicate that the interference among VMs executing two HPC application programs with high memory usage and high network I/O in the physical computing server significantly degrades application performance. An earlier study in 2005 [29] cites application interference as one of the main problems facing the HPC community. While the authors propose ways of gaining performance by reducing variability, minimizing application interference is still left open. In [21], a more general study analyzes the behavior of the center wide shared Lustre parallel file system on the Jaguar supercomputer and its performance variability. One of the performance degradations seen on Jaguar was caused by concurrent applications sharing the filesystem. All these studies highlight the impact of having application interference on HPC systems, without, but they do not offer a solution.

Zhang et al [7] study the access to disks by multiple applications running in the system by focusing on cases when I/O requests from multiple applications might break the spatial locality of individual programs - a situation that can seriously degrade I/O performance when the data servers concurrently serve synchronous requests from multiple I/O-intensive programs. The authors propose a scheme called IOrchestrator to improve I/O performance of multi node storage systems by orchestrating I/O services among programs when such inter-data-server coordination is dynamically determined to be cost effective. Their tool has a global overview of applications in the system and decides which request to perform and in which order, but they simply choose an FCFS ordering. Our implementation focuses on avoiding application interference and provides a variety of heuristics that take into account application history and system properties.

The research closest to our study is [22]. The authors investigate the interference of two applications and analyze the benefits of interrupting or delaying either one in order to avoid congestion. Our study is much more general. It looks at different application mixes and offers a range of options that give good results for two distinct objectives. These results can be used by a system administrator to configure the best solution for a particular machine.

VII. Conclusion and future work

I/O interference of multiple applications running concurrently in the system is one of the main sources of performance variability in HPC systems. We have studied the effects of congestion on application performance and on total system efficiency, and we propose several solutions that minimize performance degradation. Our global scheduler has a global view of the system and of the past behavior of all applications running at a given time. It dynamically schedules I/O accesses so as to minimize the maximum application dilation and/or to increase the system wide efficiency.

We show through extensive experiments that our scheduler performs better than current solutions for HPC systems. Our two main heuristics, MaxSysEff and MinDilation, are complementary. In particular, MaxSysEff should be favored when the system administrator wishes to optimize the performance of the machine at all cost, while MinDilation should be used when the system administrator wishes to be fair for the users of the machine. The third heuristic, MinMax-γ, is a good trade-off over these two objectives.

HPC applications in general are periodic, and their behavior is in most cases well known in advance. A scheduler that takes this information into account might give even better results than the one proposed in this paper. We have initiated the study of such schedulers (periodic schedulers) in [6]. We expect periodic schedulers to be an interesting

complement to the online schedulers presented in this paper. Future work will be devoted to assessing the additional gain that periodic schedulers may bring in comparison with online schedulers, and their robustness with respect to the periodicity hypothesis.

One of the assumption of this work was a separate I/O and messaging network (which is the case for machines such as Mira and Intrepid). This had the advantage of enabling us to assess more accurately the effects of I/O congestion on application and system efficiency. However, systems with shared networks for I/O and communications (such as Blue Waters) would also benefit from our scheduler. In such systems (i) with congestion caused by communications, execution will slow down with or without our scheduler, but the scheduler is online and will take this congestion into account when measuring application efficiency; and (ii) without congestion, the benefit from using the scheduler will be the same as when using a dedicated I/O system.

Acknowledgments: This research was done in the context of the INRIA-Illinois Joint Laboratory for Petascale Computing. The work was supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357, and the ANR Rescue project. Y. Robert is with Institut Universitaire de France.

REFERENCES

[1] X. Zhang, K. Davis, and S. Jiang, "Opportunistic data-driven execution of parallel programs for efficient I/O services," in *Proceedings of IPDPS12*. IEEE, 2012, pp. 330–341.

[2] 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 SC10*. IEEE Computer Society, 2010.

[3] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O," in *ACM Symposium on Operating Systems Principles (SOSP'01)*, 2001.

[4] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. Ganger, "Argon: Performance insulation for shared storage servers," in *5th USENIX Conference on File and Storage Technologies (FAST'07)*, 2007.

[5] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proceedings of SC09*. ACM, 2009, p. 40.

[6] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling HPC applications under I/O congestion," INRIA, France, Research Report 8519, Oct. 2014.

[7] X. Zhang, K. Davis, and S. Jiang, "IOrchestrator: improving the performance of multi-node I/O systems via inter-server coordination," in *Proceedings of SC12*, 2010.

[8] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *Proceedings of CLUSTER09*. IEEE, 2009, pp. 1–10.

[9] W. Kramer, "Blue waters and the future of scale computing and analysis," in *AICS International Symposium*, 2013.

[10] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *FGCS*, vol. 22, no. 3, 2004.

[11] R. Sankaran, E. R. Hawkes, J. H. Chen, T. Lu, and C. K. Law, "Direct numerical simulations of turbulent lean premixed combustion," in *Journal of Physics: conference series*, vol. 46, no. 1. IOP Publishing, 2006, p. 38.

[12] R. Nair and H. Tufo, "Petascale atmospheric general circulation models," in *Journal of Physics: Conference Series*, vol. 78, no. 1. IOP Publishing, 2007, p. 012078.

[13] S. Ethier, M. Adams, J. Carter, and L. Oliker, "Petascale parallelization of the gyrokinetic toroidal code," *VECPAR: High Performance Computing for Computational Science*, 2012.

[14] G. L. Bryan *et al.*, "Enzo: An adaptive mesh refinement code for astrophysics," *arXiv:1307.2265*, 2013.

[15] S. Habib *et al.*, "The universe at extreme scale: multi-petaflop sky simulation on the BG/Q," in *Proceedings of SC12*. IEEE Computer Society, 2012, p. 4.

[16] G. H. Bryan and J. M. Fritsch, "A benchmark simulation for moist nonhydrostatic numerical models." *Monthly Weather Review*, vol. 130, no. 12, 2002.

[17] "Cetus and Vesta: Test and Development systems." https://www.alcf.anl.gov/cetus-and-vesta.

[18] H. Shan and J. Shalf, "Using IOR to analyze the I/O performance for HPC platforms," *Cray User Group*, 2007.

[19] B. Behzad, L. H. V. Thanh, J. Huchette, S. Byna, R. A. Prabhat, Q. Koziol, and M. Snir, "Taming parallel I/O complexity with auto-tuning," in *Proceedings of SC13*, 2013.

[20] S. Kumar *et al.*, "Characterization and modeling of pidx parallel I/O for performance optimization," in *Proceedings of SC13*. ACM, 2013.

[21] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing output bottlenecks in a supercomputer," *Proceedings of SC12*, pp. 1–11, 2012.

[22] 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 IPDPS14*, 2014.

[23] H. Chiang, R.C.and Huang, "Tracon: Interference-aware scheduling for data-intensive applicationsin virtualized environments," *IEEE TPDS*, vol. 25, pp. 1349–1358, 2014.

[24] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao, "Who is your neighbor: Net I/O performance interference in virtualized clouds," *IEEE Transactions on Services Computing*, vol. 6, pp. 314–329, 2013.

[25] Y. Kanemasa, Q. Wang, J. Li, M. Matsubara, and C. Pu, "Revisiting performance interference among consolidated n-tier applications: Sharing is better than isolation," *IEEE International Conference on Services Computing (SCC)*, pp. 136–143, 2013.

[26] R. Biswas, M. Aftosmis, C. Kiris, and B.-W. Shen, "Petascale computing: Impact on future NASA missions," *Petascale Computing: Architectures and Algorithms*, pp. 29–46, 2007.

[27] J. Lofstead and R. Ross, "Insights for exascale IO APIs from building a petascale IO API," in *Proceedings of SC13*. ACM, 2013, p. 87.

[28] Y. Hashimoto and K. Aida, "Evaluation of performance degradation in HPC applications with VM consolidation," *IEEE International Conference on Networking and Computing (ICNC)*, pp. 273–277, 2012.

[29] D. Skinner and W. Kramer, "Understanding the causes of performance variability in HPC workloads," *IEEE Workload Characterization Symposium*, pp. 137–149, 2005.

[30] 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," *Proceedings of IPDPS10*, pp. 1–11, 2010.