Scheduling the I/O of HPC applications under congestion

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

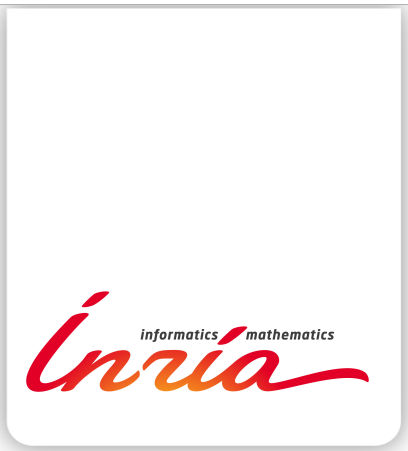
RESEARCH REPORT

N° 8519

October 2014

Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--8519--FR+ENG



Scheduling the I/O of HPC applications under congestion

Ana Gainaru*†, Guillaume Aupy‡§†, Anne Benoit‡¶, Franck Cappello‖§, Yves Robert‡§¶**, Marc Snir*‖

Project-Team ROMA

Research Report n° 8519 — October 2014 — 33 pages

Abstract: A significant percentage of the computing capacity of large-scale platforms is wasted due to 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. As a consequence, 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.

Key-words: IO, HPC, experiment, bandwidth, congestion, scheduling, online

* University of Illinois at Urbana Champaign, USA
† These authors contributed equally to this work
‡ LIP, École Normale Supérieure de Lyon, France
§ INRIA
¶ Institut Universitaire de France
‖ Argonne National Laboratory, USA
** University of Tennessee Knoxville, USA

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Ordonnancement d'applications HPC sous contrainte de congestion d'I/O

Résumé : Dans ce travail, nous proposons des algorithmes efficaces pour pallier aux problèmes de congestion lors des transferts des données de type I/O. Nous les évaluons sur des machines haute performance.

Mots-clés : IO, HPC, bandwidth, congestion

1 Introduction

With the advent of computationally demanding applications, parallel computers have continued to evolve towards post-petascale computing. At the same time, storage systems struggle to match the data generated by the computation of all running applications. According to Biswas et al. [1], when systems grow 10 times, memory bandwidth needs to grow by at least 20 times so that applications can run efficiently. 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 [2, 3]), 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 between simple "first-come first-served" strategies for each storage server to more elaborated strategies. Recently, non-work-conserving disk schedulers, like anticipatory scheduling [4] and the CFQ scheduler [5], were 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 simplest to more advanced ones, deal with low-level requests, without any information from the applications; they cannot take advantage of particular properties or behaviors of each application. As a consequence, current I/O schedulers 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, namely each individual application size and computation-to-I/O ratio, but also when they start performing I/O with regard to one another. An analysis of the Intrepid system at Argonne shows that congestion can cause up to a 70% decrease in the I/O efficiency seen by an application (Figure 1). 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 to 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

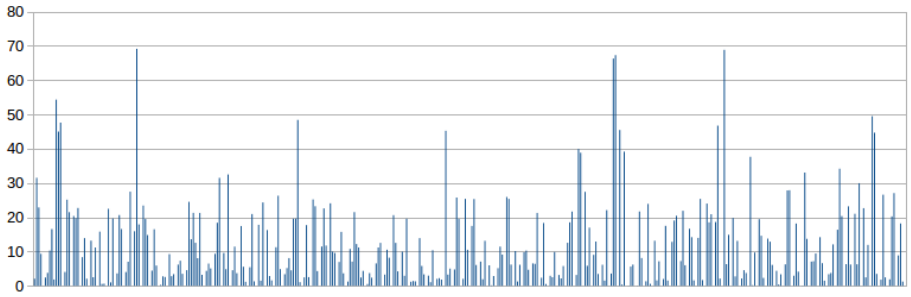


Figure 1: I/O throughput decrease (percentage per application, over 400 applications).

the global scheduler on Argonne's Vesta computer and test its results in 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 are much better when using our implementation than 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.

The rest of the paper is organized as follows. We introduce the application model and optimization problems in Section 2. We derive online scheduling heuristics in Section 3. Through a full set of simulations in Section 4, we thoroughly evaluate and compare these heuristics, before reporting actual execution times on Vesta in Section 5. We give some background and related work in Section 6. We provide concluding remarks and hints for future research directions in Section 7.

2 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.

2.1 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. For simplicity, we assume the I/O and communication network are

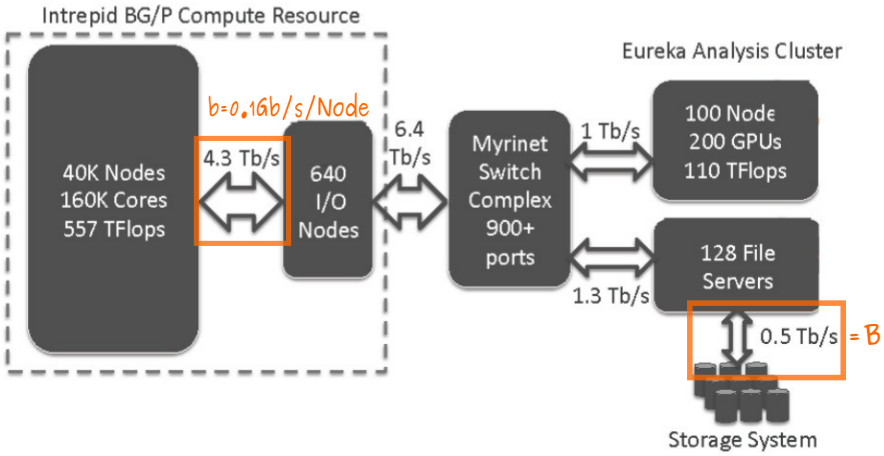


Figure 2: Model instantiation for the Intrepid platform.

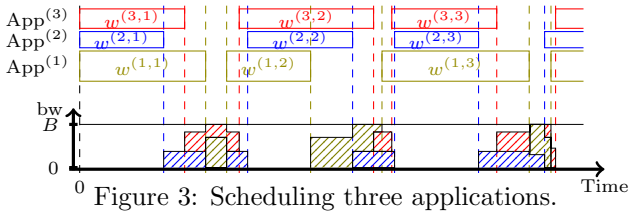


Figure 3: Scheduling three applications.

separated, so that network congestion caused by inter-node communications does not interfere with I/O transfers.

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. Finally, let d_k be 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. On the contrary, 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. Assume that $\text{App}^{(k)}$ needs to transfer $\text{vol}_{\text{io}}^{(k,i)}$. In that case, let γ be the I/O bandwidth used by each processor of $\text{App}^{(k)}$ during this time-interval. 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 constraints that: (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

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

However, 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 well 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. On the top part of Figure 3, we can see the applications doing computations without any constraint. However at the end of their computations, all applications need to transfer some volume of I/O and share the I/O total bandwidth B (bottom part of the figure). When these three applications want to execute some 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 very flexible, and only assumes 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. Again, in the end, the total volume of I/O transfers for each instance $\mathcal{I}_i^{(k)}$ of $\text{App}^{(k)}$ must be $\text{vol}_{\text{io}}^{(k,i)}$, and the rules of the game are simple: never exceed the individual bandwidth b of each processor, and never exceed the total bandwidth B of the I/O system. Formally, if instance $\mathcal{I}_i^{(k)}$ of application $\text{App}^{(k)}$ does its computation from t_1 to $t_2 = t_1 + w^{(k,i)}$, and the computation of the next instance starts in t_3, then the volume of I/O transferred for $\text{App}^{(k)}$ during the interval $[t_2, t_3]$ should be equal to $\text{vol}_{\text{io}}^{(k,i)}$.

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 can make sure that they do not exceeding the time-out existing in the I/O system.

Periodic applications In this work, we further consider a special kind of applications: *Periodic applications*. Periodic applications follow a pattern which is repeated over time: for all instances $\mathcal{I}_i^{(k)}$ of $\text{App}^{(k)}$, we have $w^{(k,i)} = w^{(k)}$ and $\text{vol}_{\text{io}}^{(k,i)} = \text{vol}_{\text{io}}^{(k)}$. There are many examples of periodic applications in the HPC community. A simple example would be an application that does not perform any I/O calls, but implements a periodic checkpoint for reliability constraints [6]. We detail in Section 4.1 other examples of periodic applications.

2.2 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

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

Due to 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 we aim to maximize the performance of the platform, i.e., 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, as the schedule will give priority to compute-intensive applications with large $w^{(k,i)}$ and small $\text{vol}_{\text{io}}^{(k,i)}$ values.

• Dilation: We aim 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, as it gives each application a guarantee on its relative progress rate.

3 Schedules

We propose two distinct techniques to implement the global scheduler. In the first one, 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. Such schedules are *online* schedules. When considering only periodic applications, the system has additional information. *Periodic* schedules take this global information and compute a schedule over a period of length T, which includes several instances of each periodic applications $\text{App}^{(k)}$. Once a schedule is available, all applications are started and their executions follow the prearranged timetable, which repeats every period. In the following, we describe online and periodic schedules, along with the heuristics that we have designed and implemented.

3.1 Online schedules

The online scheduler monitors I/O requests from all applications running in the system. 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 due to the need to call the scheduler each time they need to perform their I/O. We show in Section 5 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 2.2. 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. Here are the strategies that we experiment with.

• 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 low values of $\beta^{(k)}\tilde{\rho}^{(k)}(t)$.

• The MinMax-γ scheduler favors applications with low values of $\beta^{(k)}\tilde{\rho}^{(k)}(t)$, 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

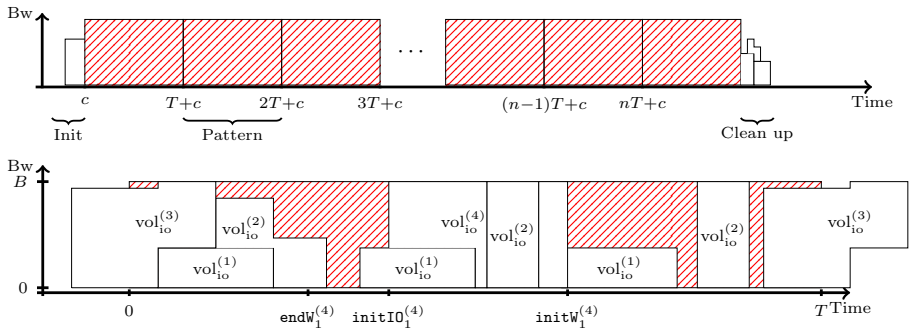


Figure 4: A periodic schedule (above), and the detail of one of its regular period (below) where $(w^{(1)} = 3.5; \mathrm{vol}_{\mathrm{io}}^{(1)} = 240; n_{\mathrm{per}}^{(1)} = 3)$, $(w^{(2)} = 27.5; \mathrm{vol}_{\mathrm{io}}^{(2)} = 288; n_{\mathrm{per}}^{(2)} = 3)$, $(w^{(3)} = 90; \mathrm{vol}_{\mathrm{io}}^{(3)} = 350; n_{\mathrm{per}}^{(3)} = 1)$, $(w^{(4)} = 75; \mathrm{vol}_{\mathrm{io}}^{(3)} = 524; n_{\mathrm{per}}^{(3)} = 1)$.

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 there may be an additional cost incurred by restarting the I/O of an application after an interruption, due to breaking disk locality. Breaking disk locality by alternating multiple applications accessing the device affects their performance and decreases the overall efficiency of the system [7]. Solid-state drives do not present the problem described above since they do not contain any moving mechanical components. This means that 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.

3.2 Periodic schedules

Unlike online schedulers, periodic schedulers have information about all the applications that are expected to run in the system.A periodic schedule of period T repeats the same operations for each application every T units of time. In fact, the first and last period are different, as they correspond to initialization and clean-up, respectively. The other periods are called *regular* and correspond to steady-state operation. Within each regular period of length T, the schedule executes $n_{\mathrm{per}}^{(k)}$ instances of App$^{(k)}$. Of course the first of these instances can overlap with the previous period, and the last one can overlap with the next period, but overall the operations are cyclic. An example of a periodic schedule for 4 applications is presented in Figure 4.

The overhead of periodic schedulers is paid only at the beginning of the execution of each application, after which the entire schedule is fully defined. Note that if we assume that the impact of the initialization and clean-up phases have a negligible impact on the application efficiency, the efficiency of an application becomes[1]:

$$\tilde{\rho}^{(k)}(d_k) = \frac{n_{\text{per}}^{(k)} w^{(k)}}{T} \tag{1}$$

Finally, note that if new applications arrive and leave the platform at certain times, one can recompute a periodic schedule.

In this Section we formally define a periodic schedule in Section 3.2.1. With this definition we show the hardness to compute a periodic schedule in Section 3.2.2. Finally we present heuristics for periodic schedules in Section 3.2.3.

3.2.1 Formal definition of a periodic schedule

Let us now formally define a periodic schedule. Consider any regular period and shift time origin so that this period corresponds to the time-interval $[0, T[$. We need to introduce some definitions to describe the schedule during the period. Consider first a given application $\text{App}^{(k)}$ throughout the period. During this period, there are $n_{\text{per}}^{(k)}$ instances of $\text{App}^{(k)}$ that are scheduled, and we number them from $\mathcal{I}_1^{(k)}$ to $\mathcal{I}_{n_{\text{per}}^{(k)}}^{(k)}$:

- The first instance $\mathcal{I}_1^{(k)}$ starts with the first occurrence of a computing interval of length $w^{(k)}$, from time $\texttt{initW}_1^{(k)}$ to time $\texttt{endW}_1^{(k)} = \texttt{initW}_1^{(k)} + w^{(k)}$ (recall that because resources are dedicated to each application, there is never any slowdown nor interruption in computing phases)
- If $n_{\text{per}}^{(k)} > 1$, the second instance $\mathcal{I}_2^{(k)}$ starts with the second occurrence of a computing interval of length $w^{(k)}$, from time $\texttt{initW}_2^{(k)}$ to time $\texttt{endW}_1^{(k)} = \texttt{initW}_2^{(k)} + w^{(k)}$
- During the interval $[\texttt{endW}_1^{(k)}, \texttt{initW}_2^{(k)}[$, the application is either performing I/O activity or stalled. This means that all I/O activity for instance $\mathcal{I}_1^{(k)}$ takes place within this interval, but possibly with different bandwidths during different parts of the interval (including no bandwidth at all, which means the application $\text{App}^{(k)}$ is stalled).

[1] The rationale behind this can be seen on Figure 4. If $\text{App}^{(k)}$ is released at time r_k, and the first period starts at time $r_k + c$, that is after an initialisation phase, then the main pattern is repeated n times (until time $n \cdot T + r_k + c$), and finally $\text{App}^{(k)}$ ends its execution after a clean-up phase at time $d_k = r_k + c + n \cdot T + c'$. If we assume that $n \cdot T \gg c + c'$, then $d_k - r_k \approx n \cdot T$. Then the value of the $\tilde{\rho}^{(k)}(d_k)$ for $\text{App}^{(k)}$ is:

$$\tilde{\rho}^{(k)}(d_k) = \frac{\left(n \cdot n_{\text{per}}^{(k)} + \delta\right) w^{(k)}}{d_k - r_k} = \frac{\left(n \cdot n_{\text{per}}^{(k)} + \delta\right) w^{(k)}}{c + n \cdot T + c'} \approx \frac{n_{\text{per}}^{(k)} w^{(k)}}{T}$$

where δ can be 1 or 0 depending whether $\text{App}^{(k)}$ was executed or not during the clean-up or initialization phase.

- We proceed likewise for each instance and define $\texttt{initW}_i^{(k)}$ and $\texttt{endW}_i^{(k)}$ for $\mathcal{I}_i^{(k)}$ where $i < n_{\text{per}}^{(k)}$. The last instance may be a little different: indeed, the last part of the computing interval may overlap with the next period. But then the 'missing part' appears in the beginning of the period, and we can still define $\texttt{endW}_{n_{\text{per}}^{(k)}}^{(k)} = \texttt{initW}_{n_{\text{per}}^{(k)}}^{(k)} + w^{(k)} \mod T$. In that case we have to enforce that there is no overlap between the end of the last computing interval and the first complete computing interval: in other words, if $\texttt{initW}_{n_{\text{per}}^{(k)}}^{(k)} + w^{(k)} > T$ then we must have $\texttt{endW}_{n_{\text{per}}^{(k)}}^{(k)} < \texttt{initW}_1^{(k)}$.

The complicated part is to enforce all the constraints on I/O activity. A simple approach is the following:

- With the beginning and termination of each computing interval, we have defined $2n_{\text{per}}^{(k)}$ dates for application $\text{App}^{(k)}$. Altogether, we have $S \leq 2\sum_{k=1}^{K} n_{\text{per}}^{(k)}$ distinct dates. We call these S dates the S *events* of the period, and we sort them to partition the period into S intervals Int_1 to Int_S (possibly wrapping the end of the last interval into the beginning of the first one).
- During each interval Int_s, of duration D_s, we assume that each processor of application $\text{App}^{(k)}$ is assigned a constant bandwidth amount $\gamma_s^{(k)}$. Intuitively, this is because there is no reason to update the bandwidth in the absence of a new event. Of course we must enforce that (i) the outgoing capacity of the processors is not exceeded: $\gamma_s^{(k)} \leq b$ for $1 \leq k \leq K$; and (ii) the total capacity of I/O system is not exceeded: $\sum_{k=1}^{K} \beta^{(k)}\gamma_s^{(k)} \leq B$.
- There remains the final constraint on total I/O volume for each instance. Consider the i-th instance of $\text{App}^{(k)}$, for $1 \leq i \leq n_{\text{per}}^{(k)}, 1 \leq k \leq K$. Its I/O transfers are included in the interval $[\texttt{endW}_i^{(k)}, \texttt{initW}_{i+1}^{(k)}[$, with the wrapping convention for the last instance (and $i+1$ taken as 1 if $i = n_{\text{per}}^{(k)}$). We partition this interval according to the events:

$$[\texttt{endW}_i^{(k)}, \texttt{initW}_{i+1}^{(k)}[= \bigcup_{s=s_1}^{s_2} \text{Int}_s$$

and we derive the condition

$$\text{vol}_{\text{io}}^{(k)} = \sum_{s=s_1}^{s_2} D_s \gamma_s^{(k)} \beta^{(k)}$$

- Finally, we add a notation $\texttt{initIO}_i^{(k)}$, that is the time when the I/O transfer from the i^{th} iteration starts, i.e., the first instant after $\texttt{endW}_i^{(k)}$ for which $\gamma_s^{(k)} \neq 0$. Formally, if $[\texttt{endW}_i^{(k)}, \texttt{initW}_{i+1}^{(k)}[= \bigcup_{s=s_1}^{s_2} \text{Int}_s$, then $\texttt{initIO}_i^{(k)} = \tilde{s} \geq s_1$ such that $\gamma_{\tilde{s}}^{(k)} \neq 0$ and forall s s.t. $s_1 \leq s < \tilde{s}, \gamma_s^{(k)} = 0$.

From a theoretical perspective, the size of the description of a regular period is $O(K + \sum_{k=1}^{K} n_{\text{per}}^{(k)} + T)$. Indeed, it makes sense to consider only periods large enough so that one instance of each application can take place if there were no

contention, i.e., $T \geq \max_{1 \leq k \leq K} \left(w^{(k)} + \text{vol}_{\text{io}}^{(k)} \right)$. Of course, it may well be the case that we have to resort to longer periods to be able to schedule at least one instance of each application when I/O constraints are taken into account. Furthermore, because we look at semi-periodic schedules it is important to fully describe the regular period T, otherwise it may be that the problem does not belong to NP.

The main optimization problem is:

Definition 1 (PERIODIC)**.** We consider a platform of N processors, a set of application $\cup_{k=1}^{K}(\text{App}^{(k)}, \beta^{(k)}, w^{(k)}, \text{vol}_{\text{io}}^{(k)})$, a maximum period $T_{\max}$, we want to find a periodic schedule $\mathcal{S}$ of period $T \leq T_{\max}$, in order to minimize one of the following objectives:

1. SYSEFFICIENCY
2. DILATION

3.2.2 Intractability of a periodic schedule

Theorem 1. PERIODIC *is NP-complete for both objectives.*

Proof. We consider the associated decision problem: given a set of applications, a maximum period $T_{\max}$, a target μ, does there exist a periodic schedule of period smaller than $T_{\max}$, such that the objective considered is not smaller than μ?

The problem clearly belongs to NP, given the size of the period T, the set of $S = \sum_{k=1}^{K} n_{\text{per}}^{(k)}$ dates, and the values $\gamma_s^{(k)}$ for all $(k, s) \in \{1 \cdots k\} \times \{1 \cdots S\}$, and we can verify in polynomial time whether it is a valid schedule and if it matches any objective.

We use a reduction from 3-PARTITION. Consider an arbitrary instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3n$ integers $a_1, \ldots, a_{3n}$, can we partition the $3n$ integers into n triplets $I_1, \cdots, I_n$, each of sum B? We can assume that $\sum_{i=1}^{3n} a_i = nB$, otherwise $\mathcal{I}_1$ has no solution. The 3-PARTITION problem is NP-hard in the strong sense [8], which implies that we can encode all integers (a_1, $\ldots$, a_{3n}, B) in unary. We build the following instance $\mathcal{I}_2$ of PERIODIC: the maximum bandwidth of the I/O system is $B \cdot b$, there are $3n$ applications such that, for $\text{App}^{(k)}$, we define:

$$\begin{cases} \beta^{(k)} & = a_k \\ w^{(k)} & = n - 1 \\ \text{vol}_{\text{io}}^{(k)} & = a_k \cdot b \end{cases}$$

Note that since $B \geq \max_k a_k$, then for all $\text{App}^{(k)}$, we have $\text{time}_{\text{io}}^{(k)} = 1$. We want to find out if there exists a solution such that

1. SYSEFFICIENCY is not smaller than $\frac{n-1}{n}$;

2. DILATION is not smaller than 1;

These objectives are equivalent to $\forall k \leq 3n, \tilde{\rho}^{(k)} = \rho^{(k)}$: (i) it is clear than all those objectives are exactly matched when $\forall k, \tilde{\rho}^{(k)} = \rho^{(k)}$; furthermore (ii), if there exists k such that $\tilde{\rho}^{(k)} > \rho^{(k)}$, then the objectives are not matched.

We now prove that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Assume first that $\mathcal{I}_1$ has a solution. Let us call $I_1, \cdots, I_n$ the n triplets of $\mathcal{I}_1$. Let us consider the following schedule of period $T = n$: For all $1 \leq k \leq 3n$, let i be the triplet to which a_k belongs, i.e., $a_k \in I_i$. We schedule exactly one instance of $\text{App}^{(k)}$ ($n_{\text{per}}^{(k)} = 1$), such that $\texttt{initW}_1^{(k)} = i+1$ and $\texttt{endW}_1^{(k)} = i$. Then, during interval $[\texttt{endW}_1^{(k)}, \texttt{initW}_1^{(k)}[$, we consider that $\text{App}^{(k)}$ transfers I/O with bandwidth b.

We can show that this schedule is correct:

- The bandwidth constraint is respected, during each interval $[i, i+1[$, the total bandwidth used is exactly B: during interval $[i, i+1]$, only applications $\text{App}^{(k)}$ such that $[i, i+1[\cap [\texttt{endW}_1^{(k)}, \texttt{initW}_1^{(k)}[\neq \emptyset$ can do I/O transfers. This is true only for $k \in I_i$, and by definition, $\sum_{k \in I_i} a_k b = Bb$;
- $\forall k \leq 3n, \tilde{\rho}^{(k)} = \rho^{(k)}$: indeed, $\tilde{\rho}^{(k)} = \frac{w^{(k)}}{T} = \frac{w^{(k)}}{n} = \rho^{(k)}$.

Finally, we constructed in polynomial time a valid solution to $\mathcal{I}_2$.

Assume now that $\mathcal{I}_2$ has a solution. Let T be the period of this solution, and let $(q, r) \in \mathbb{N}^2$ such that $T = q \cdot n + r$ and $0 \leq r < n$. Since $\mathcal{I}_2$ is a solution, then

$$\forall k \leq 3n, \frac{n_{\text{per}}^{(k)} w^{(k)}}{T} = \tilde{\rho}^{(k)} = \rho^{(k)} = \frac{w^{(k)}}{w^{(k)} + \text{time}_{\text{io}}^{(k)}},$$

that is $T = n_{\text{per}}^{(k)} \left(w^{(k)} + \text{time}_{\text{io}}^{(k)} \right) = n_{\text{per}}^{(k)} n$. Finally, we have $r = 0$ and for all k, $n_{\text{per}}^{(k)} = q$.

Let $k \leq 3n$, we further have:

- $\forall i \leq n_{\text{per}}^{(k)}, \texttt{initW}_{i+1}^{(k)} - \texttt{endW}_i^{(k)} = \text{time}_{\text{io}}^{(k)} = 1$:
  1. $\forall i \leq n_{\text{per}}^{(k)}, \texttt{initW}_{i+1}^{(k)} - \texttt{endW}_i^{(k)} \geq \text{time}_{\text{io}}^{(k)} = 1$: recall that we should schedule all $\text{vol}_{\text{io}}^{(k)}$ in the interval $[\texttt{endW}_i^{(k)}, \texttt{initW}_{i+1}^{(k)}[$, and that this cannot be done in a time smaller than $\text{time}_{\text{io}}^{(k)}$.
  2. recall that $[0, T[= \cup_{i=1}^{n_{\text{per}}^{(k)}} [\texttt{initW}_i^{(k)}, \texttt{endW}_i^{(k)}[\cup [\texttt{endW}_i^{(k)}, \texttt{initW}_{i+1}^{(k)}[$. By definition, for all $i \leq n_{\text{per}}^{(k)}$, $\texttt{endW}_i^{(k)} - \texttt{initW}_i^{(k)} = w^{(k)}$ and $T = n_{\text{per}}^{(k)} w^{(k)} + \sum_{i=1}^{n_{\text{per}}^{(k)}} \texttt{initW}_{i+1}^{(k)} - \texttt{endW}_i^{(k)} = n_{\text{per}}^{(k)} n$.
- $\forall i \leq n_{\text{per}}^{(k)}, \gamma^{(k)} = b$ on $[\texttt{endW}_i^{(k)}, \texttt{initW}_{i+1}^{(k)}]$, that is a direct consequence of the previous item.
- $\text{App}^{(k)}$ is n-periodic: we have seen that we can decompose its schedule: for all $i \leq n_{\text{per}}^{(k)}$, $[\texttt{initW}_i^{(k)}, \texttt{initW}_{i+1}^{(k)}]$ consists in $w^{(k)}$ units of time where

App$^{(k)}$ does $w^{(k)}$ units of computation, followed by one unit of time where App$^{(k)}$ does $\text{vol}_{\text{io}}^{(k)}$ units of I/O transfers at maximum bandwidth b.

Finally, since each application has a pattern n-periodic in the previous schedule, the new schedule of period $T = n$ which consists in taking the n first units of time of the previous schedule is a valid schedule.

For $1 \leq i \leq n$, let us consider

$$I_i = \left\{ k \,\middle|\, i \leq \texttt{initIO}_1^{(k)} < i+1 \right\}.$$

We have the following properties:

- $\forall i, \sum_{k \in I_i} a_i b \leq Bb$: indeed, if we let $k' \in I_i$ be the application of I_i with the latest $\texttt{initIO}_1^{(k')}$. Then at time $\texttt{initIO}_1^{(k')}$, all applications of I_i are doing I/O transfers at maximum bandwidth b, indeed, we have for all $k \in I_i, \texttt{initIO}_1^{(k)} \leq \texttt{initIO}_1^{(k')} < i+1 \leq \texttt{initIO}_1^{(k)} + 1$.
- $\sum_{i=1}^{n} \sum_{k \in I_i} a_i b \geq nBb$ since all applications are executed once ($\tilde{\rho}^{(k)} = \rho^{(k)}$).

Finally we conclude that for all $i \leq n$, $\sum_{k \in I_i} a_i = B$, and $I_1, \cdots, I_n$ as defined above is a valid solution to $\mathcal{I}_1$ constructed in polynomial time. □

3.2.3 Heuristics for periodic schedules

In the following, we introduce two polynomial-time periodic scheduling heuristics, one for each objective. The first decision is to choose the length T of the period. We start from $T = \max_k \left(w^{(k)} + \text{time}_{\text{io}}^{(k)} \right)$; while T is smaller that $T_{\max}$, the period is incremented by a factor $(1+\varepsilon)$, and a solution is re-computed. We take the best solution over all the periods. Here both ε and $T_{\max}$ are parameters whose definitions have an impact on the quality of the results and on the number of increments: the larger $T_{\max}$ and the smaller ε, the better the results, but the longer the execution time of the heuristics.

Given a period, the heuristics greedily add instances of the applications until no more instance can fit within the current schedule. Adding greedily an instance of application App$^{(k)}$ into the schedule means that the heuristic tries to find the first instant in the period where $\text{vol}_{\text{io}}^{(k)}$ can be executed contiguously with a constant bandwidth while matching the various constraints.

- Insert-In-Schedule-Throu aims at maximizing the SysEfficiency objective. It sorts the applications by non-decreasing $\frac{w^{(k)}}{\text{time}_{\text{io}}^{(k)}}$ ratios. It schedules as many instances as possible of the first application before moving on to the second one, and so on.

- Insert-In-Schedule-Cong aims at maximizing the Dilation objective. It dynamically sorts the applications by non-increasing values of $n_{\text{per}}^{(k)} \left(w^{(k)} + \text{vol}_{\text{io}}^{(k)} \right)$ and always picks the largest one. Here $n_{\text{per}}^{(k)}$ is the current number of instances of App$^{(k)}$ already scheduled within the period, and it is incremented each time App$^{(k)}$ is selected.

4 Simulations

In this section, we report extensive simulations to assess the performance of the **online** heuristics presented in Section 3.1. 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.

4.1 Applications

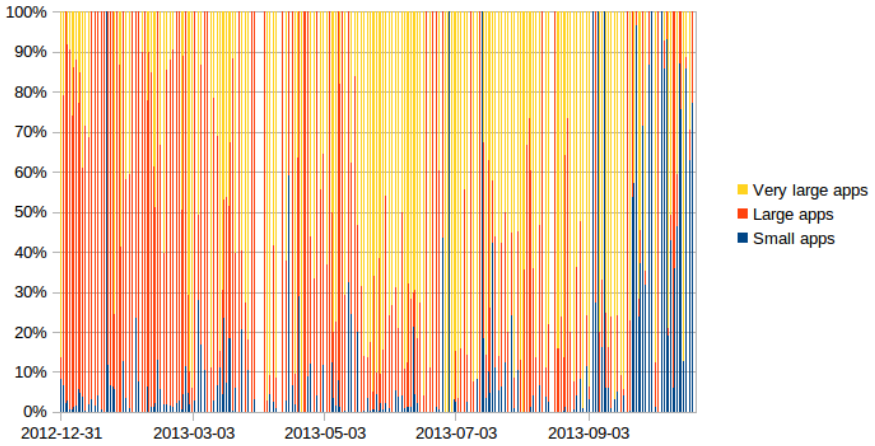
Intrepid is a BlueGene/P supercomputer used by the Argonne National Laboratory between 2008 and 2014 and was ranked number 3 on the June 2008 Top 500 list. Consisting of 48 racks, 786,432 processors, and 768 terabytes of memory, Mira is a 10-petaflops IBM BlueGene/Q system, 20 times faster than Intrepid and currently ranked number 5 on the November 2013 Top 500 list. A wide range of science and engineering applications have run on BlueGene 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 and/or for checkpointing. Intrepid uses separate networks for communication and I/O, which makes it the perfect system to study the effects of congestion on application and system efficiency.

We use Darshan [9], an application level I/O characterization tool developed at Argonne, to capture the behavior of applications running on Intrepid. It intercepts I/O function calls in user space and records access pattern information before the I/O operations are interpreted by the operating system or file system. We analyzed the traces provided by this tool and divided the applications into the following categories [10]:

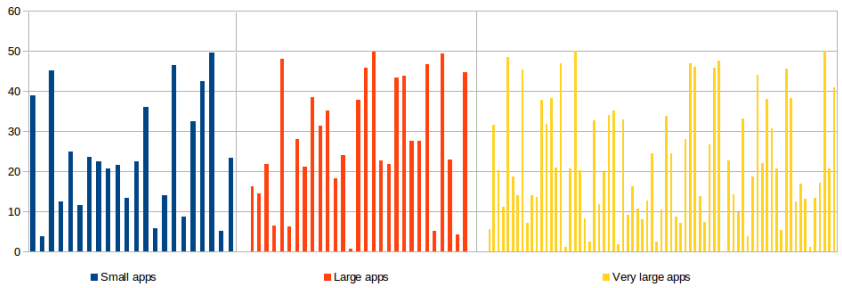
- *small applications* are applications that run on less than 1,284 nodes, that is less than 20,544 FP cores, or, less than 41,088 integer cores;
- *large applications* are applications that run on more than 1,285 nodes, that is more than 20,560 FP cores, or, more than 41,120 integer cores;
- *very large applications* are applications that run on more than 4,584 nodes, that is more than 123,344 FP cores, or, more than 146,688 integer cores.

Figure 5 shows how many applications from each type ran on Intrepid during one year from December 2012 to December 2013, how much time each spent doing I/O, and their utilization of the platform. We use this information for generating the simulation scenarios.

In this section, we mainly focus on scheduling periodic applications under I/O bandwidth constraints. Periodic applications follow a pattern which is repeated over time: for all instances $\mathcal{I}_i^{(k)}$ of $\text{App}^{(k)}$, we have $w^{(k,i)} = w^{(k)}$ and $\text{vol}_{\text{io}}^{(k,i)} = \text{vol}_{\text{io}}^{(k)}$. There are many examples of periodic applications in the HPC



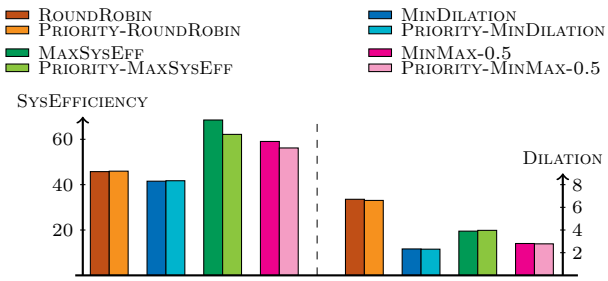
(a) System usage per day for each application type



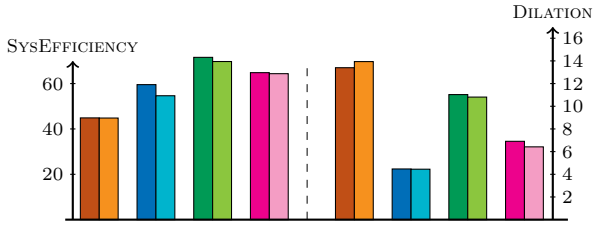
(b) Percentage spent doing I/O per application type

Figure 5: Characteristics of application running on Intrepid in 2013.

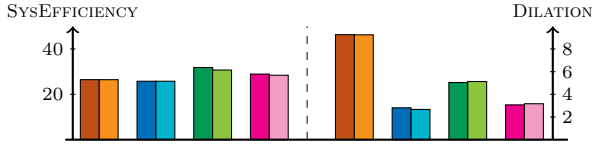
community. A simple example would be an application that does not perform any I/O calls, but implements a periodic checkpoint for reliability constraints [6]. Carns et al. [9] use the Darshan I/O characterization tool to capture an accurate picture of I/O patterns in Petascale workloads. In particular, 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 4.3 we discuss the impact of application periodicity and show that results are the same for non-periodic applications.



(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 6: Objectives for different mixes of applications and I/O computation ratios.

4.2 Assessment of the heuristics

By inspecting the mix of applications that ran on Intrepid (Figure 5a), 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 6 presents the corresponding results. Simulations were run 200 times on different applications mixes that simulate real scientific applications running on Intrepid, and 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 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 6a), 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 $\gamma = 0.27$) 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 6b), with many small applications and a few large ones. In Figure 6c, 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.

4.3 Impact of periodicity

As mentioned, based upon the literature and our own verifications on Intrepid, we have assumed so far that applications are periodic. We now discuss the impact of having non-periodic applications in the system. We define the sensibility of an application as $\text{Sens}_w^{(k)} = \frac{\max_i w^{(k,i)} - \min_i w^{(k,i)}}{\max_k w^{(k,i)}}$ and $\text{Sens}_{io}^{(k)} = \frac{\max_i \text{vol}_{\text{io}}^{(k,i)} - \min_i \text{vol}_{\text{io}}^{(k,i)}}{\max_k \text{vol}_{\text{io}}^{(k,i)}}$. 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 7, 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 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.

4.4 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 8 and 11). However the results for the non-PRIORITY variant of MAXSYSEFF and MINDILATION heuristics can be found on Figures 10 and 13, and the comparison of the PRIORITY-MINMAX-γ with the PRIORITY variant of MAXSYSEFF and MINDILATION heuristics can be found on Figures 9

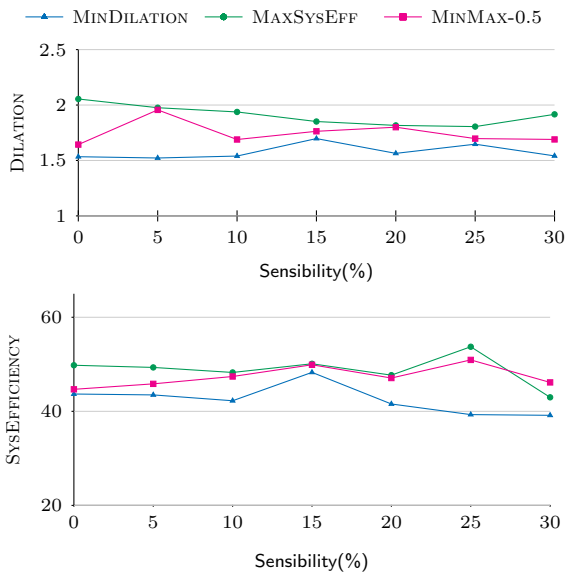


Figure 7: Impact of the sensibility of the computations over SysEfficiency and Dilation of all heuristics.

and 12. We sum up the average in Tables 1 and 2. While the non-Priority variant of all heuristics always outperforms the results of Intrepid and Mira (as can be seen in Tables 1 and 2), in the following we only present results of the Priority variant of the heuristics because Intrepid and 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 to supplement the I/O bandwidth. It is important to see that 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 only give information about the total execution time and the total amount of I/O performed by the applications, but do not say anything about their 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 fix number is chosen so that to simulate the characteristics we have seen for applications running on Intrepid. Recall that Section 4.3 has shown that the sensibility does not impact the results, so this hypothesis is not binding. Another limitation with Darshan logs is that they only record 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 infor-

	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

Table 1: The averages are done over 56 different congested moments on intrepid.

mation 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.

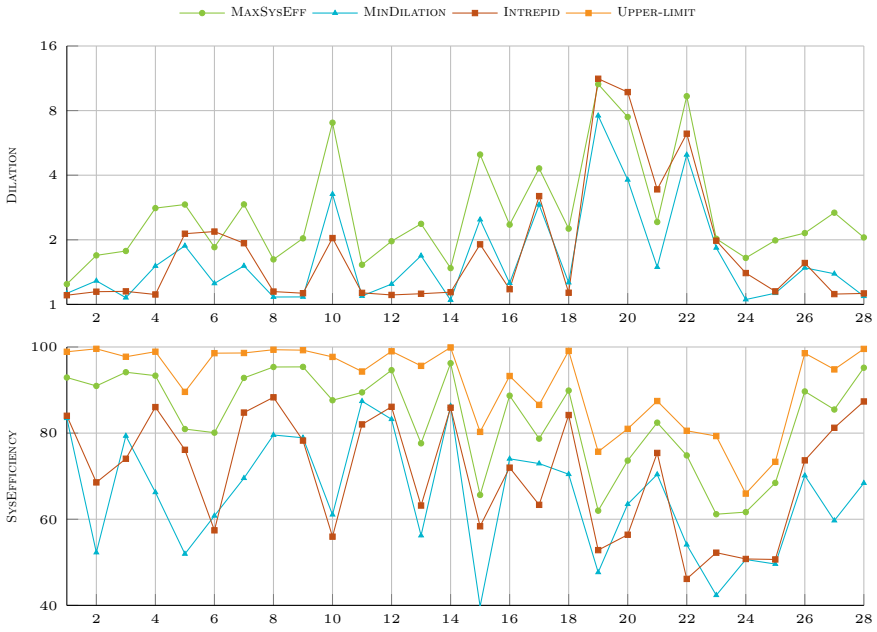


Figure 8: Comparison of the Priority heuristics over the current Dilation and SysEfficiency of Intrepid.

On Figures 8 and 11, 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 testcases where applications are IO 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

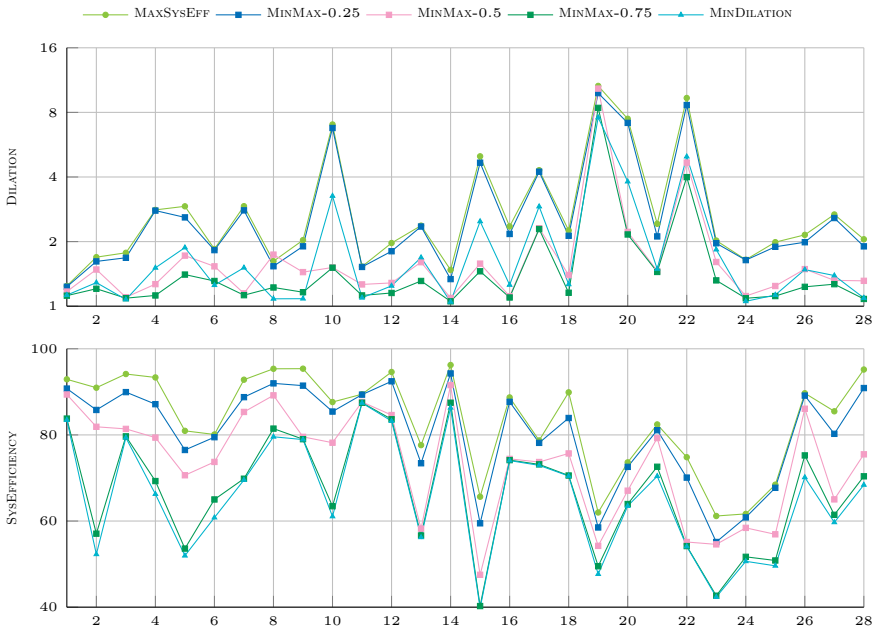


Figure 9: Comparison of the Priority heuristics.

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 this is not the case (e.g., the 15th testcase) and we observed that these test cases present a very 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, since 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 improves on average 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 show improvement compare to the scheduler used by Mira as well: MinDilation improves on average 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

Figure 10: Comparison of the NON-Priority heuristics over the current Dilation and SysEfficiency of Intrepid.

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 5 that the results obtained in simulation accurately describe what would be obtained if Intrepid or Mira was using our heuristics.

5 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 [18] is a developmental platform for Mira. Its architecture is the same as Mira's except that it has two compute racks (Mira has 48). A rack has 32 node boards, each of which holds 32 compute cards. Each compute card comprises 16 compute cores of 1600 MHz PowerPC A2 processors with 16GB RAM (1GB/core). In total, Vesta has 2,048 nodes (32,768 compute cores). Applications running on this machine are electrically isolated from each other. This means that even if there are other applications running on the system, their communications will not impact our experiments. Our focus in this section is directed towards write/write interference between multiple applications.

	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

Table 2: The averages are done over 11 different congested moments on Mira.

5.1 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 [19] 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 one single process is representing the scheduler and 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 is using hard disks and is influenced by the locality of disk access, we implement 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 tradeoff 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 implement the heuristics as an additional layer on top of the Vesta I/O scheduler, so that we can 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 14 presents the overhead of adding the scheduling thread

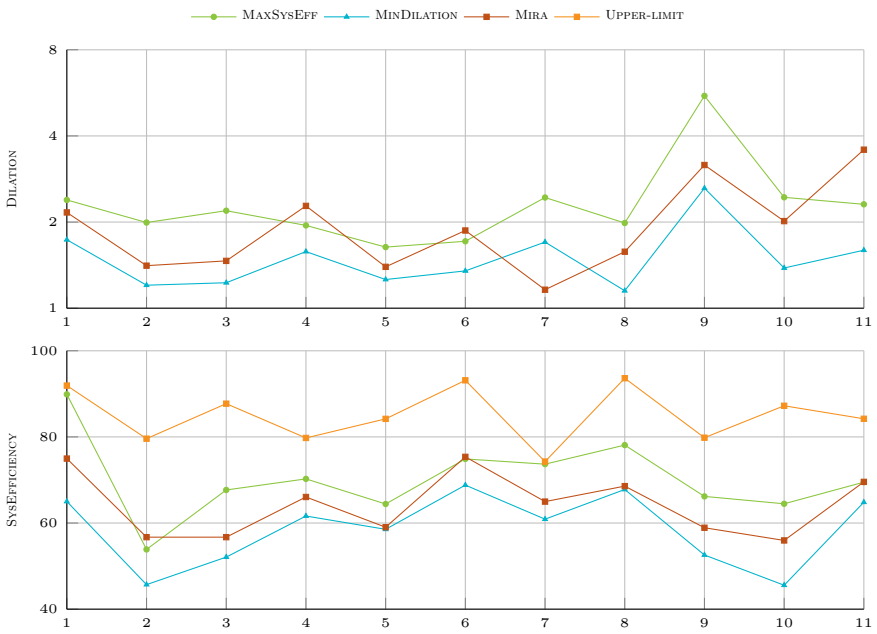


Figure 11: Comparison of the Priority heuristics over the current Dilation and SysEfficiency of Mira.

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 between 1% to 5.3%. In general, for a larger number of applications, the execution time overhead remains under 3%. We account for this idle time as well as the I/O throughput and application delays when computing the system efficiency and application dilation in Section 5.2.

5.2 Results

Figure 15 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 there are two applications 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

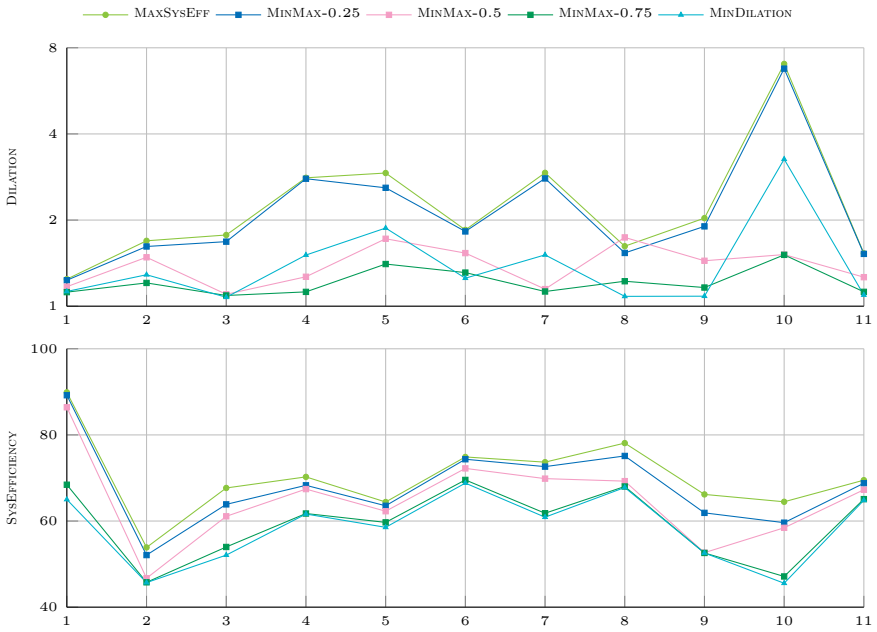


Figure 12: Comparison of the Priority heuristic.

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 very small decrease in system efficiency (and increase in the max dilation) compared to running the IOR benchmark without any modification.

The results when running multiple applications are very similar to what was seen simulating Mira and confirm what we have observed with the simulations: our heuristics perform very well, better than Vesta's I/O scheduler when congestion occurs. Furthermore, the main result of this experimental setup is that with more than 3 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, 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.

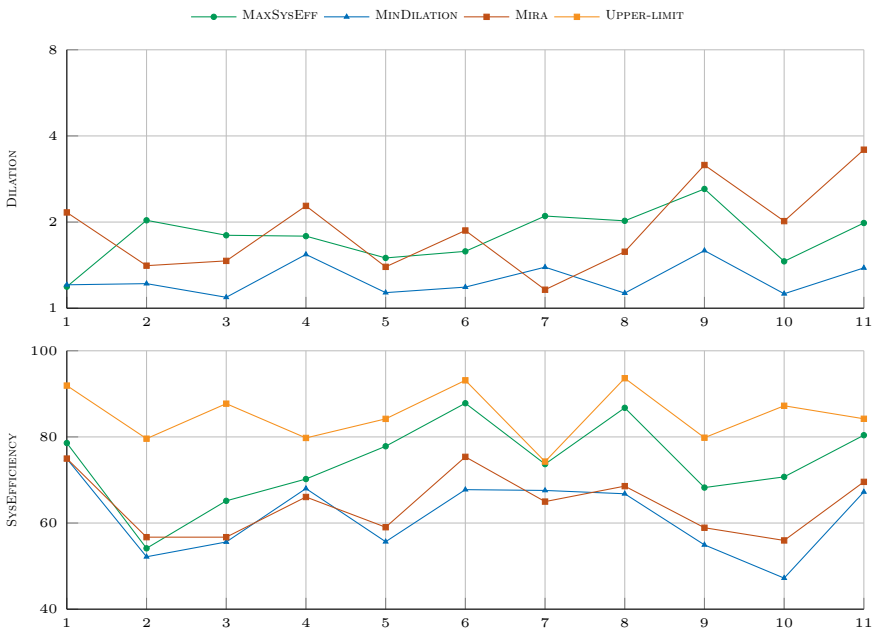


Figure 13: Comparison of the NON-Priority heuristics over the current Dilation and SysEfficiency of Mira.

Figure 15 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 the big ones when using the MaxSysEff heuristic, having an increase in their dilation value of 36% compared to 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 to the ones 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.

6 Related work

Application performance variability can significantly detract from both the overall performance realized 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. There are numerous papers that analyze this problem for clouds [24, 25, 26]. [25] presents 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 [26], the authors investigate the sensitivity of measured performance in relation to consol-

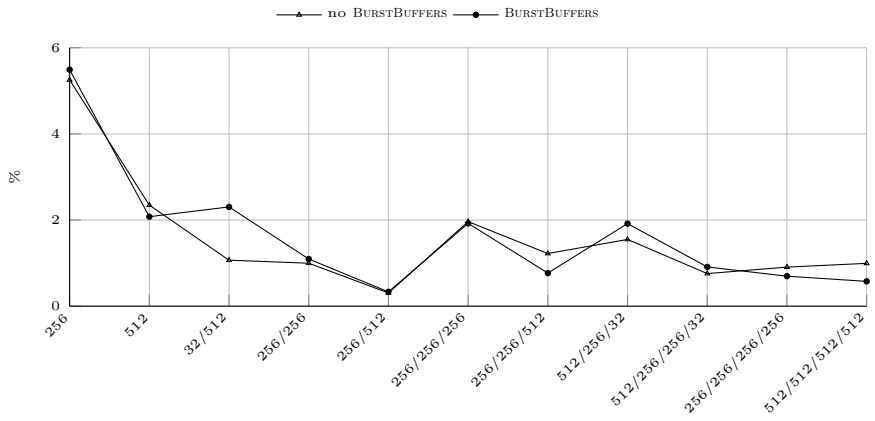


Figure 14: Execution time overhead of our implementation of the IOR benchmark.

idated 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 [1, 27], few paper focus on finding solutions at the platform level. Some papers consider application-side I/O scheduling [2, 3]. In particular, recently, several works focused on using machine learning for auto-tuning and performance studies [20, 21]. 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 recently studied in several articles [28, 29, 30]. The study in [28] evaluates the performance degradation in each application program when 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 it proposes ways of gaining performance by reducing variability, minimizing application interference is still left open. [22] is a more general study that 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 of these studies highlight the impact of having application interference on HPC systems without, however, offering a solution.

[7] studies 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; this can seriously degrade I/O per-

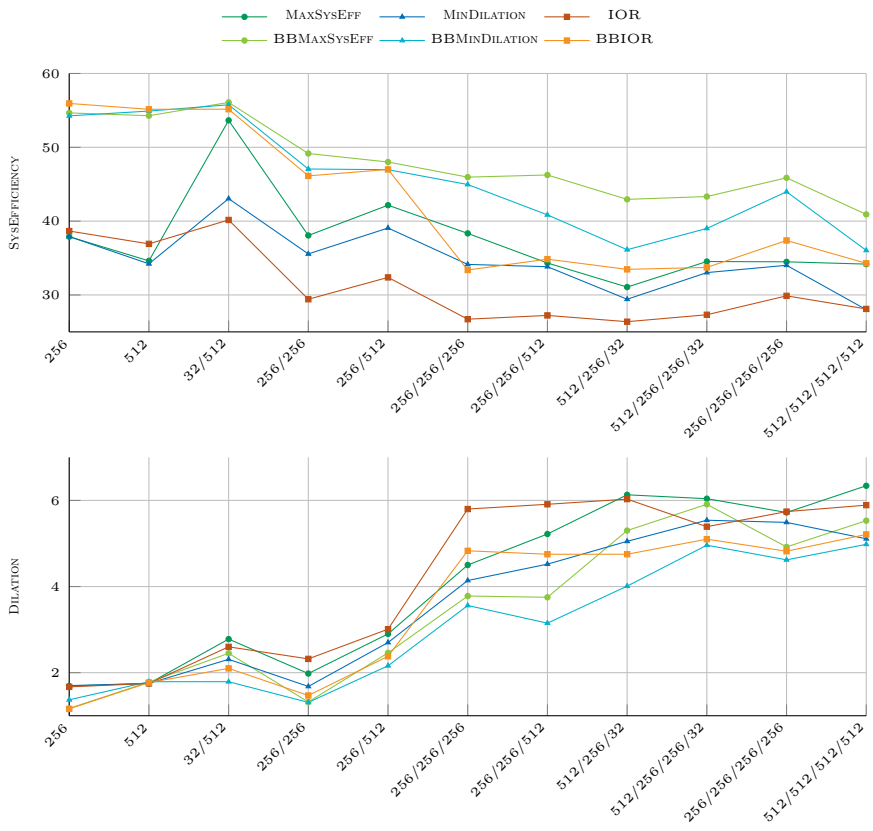
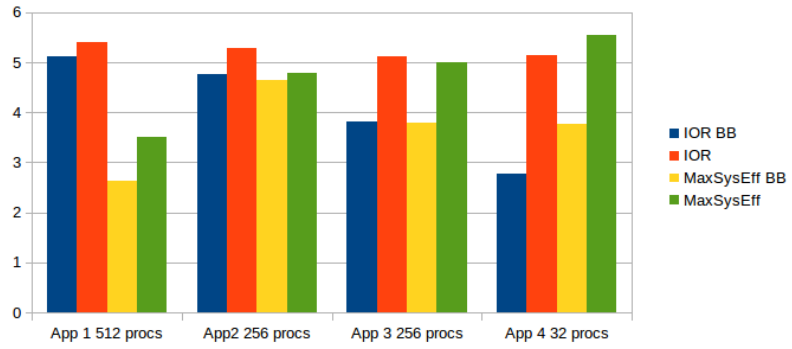


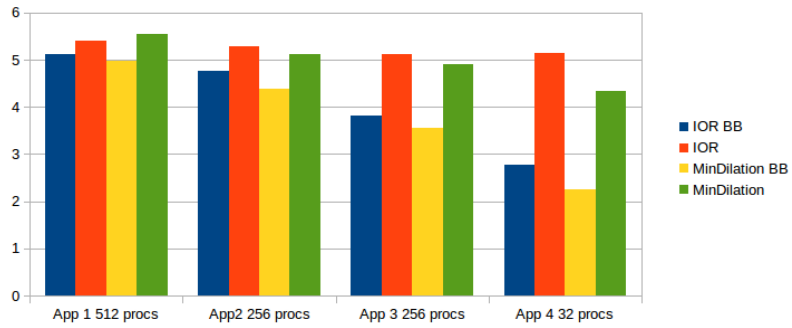
Figure 15: System efficiency and dilation for different scenarios on Vesta.

formance 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 [23]. 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 their particular machine.



(a) MaxSysEff heuristic



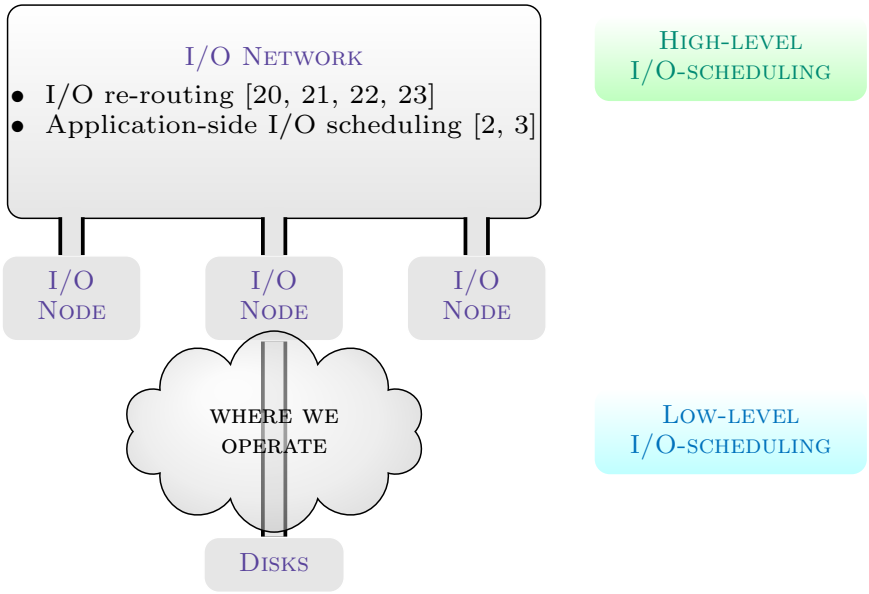
(b) MinDilation heuristic

Figure 16: Dilation values for the applications from 512/256/256/32 scenario.

7 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 the performance degradation. Our global scheduler has a global view of the system and on the past behavior of all applications running at a given time, and 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 very 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 periodic scheduler might give even better results than the one proposed in this paper. Periodic schedules would have to be implemented inside the system's job scheduler. It would have a more accurate global view and would be able to compute a complete schedule over a period of given length in advance for all applications, which in return would give a more flexible way of controlling the behavior of the applications. 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 to online schedulers, and their robustness with respect to the periodicity hypothesis.

Finally 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 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; (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 also supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357, and the ANR Rescue project. A. Benoit and Y. Robert are with Institut Universitaire

de France.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[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] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[9] 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.

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

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

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

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

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

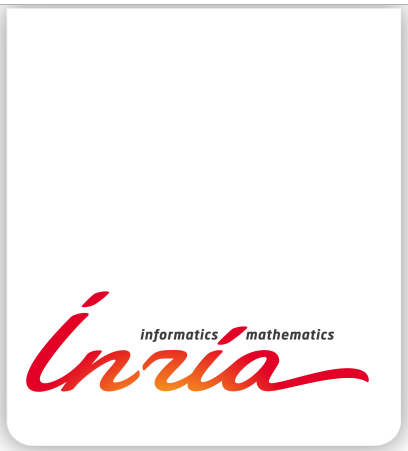
[26] Y. Kanemasa, Q. Wang, J. Li, M. Matsubara, and C. Pu, "Revisiting performance interference among consolidated n-tier applications: Sharing is better than isolation," *IEEE International Conference on Services Computing (SCC)*, pp. 136–143, 2013.

[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.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399