Revisiting I/O bandwidth-sharing strategies for HPC applications

Anne Benoit, Thomas Herault, Lucas Perotin, Yves Robert, Frédéric Vivien

Highlights

Revisiting I/O bandwidth-sharing strategies for HPC applications

Anne Benoit[a], Thomas Herault[b], Lucas Perotin[a], Yves Robert[a,b], Frédéric Vivien[a]

- Research highlight 1: Design of novel I/O bandwidth-sharing strategies that go beyond First Come First Served and Fair Share.
- Research highlight 2: First competitiveness analysis of I/O bandwidth-sharing strategies.
- Research highlight 3: Comprehensive simulation campaign using an extensive set of application scenarios.

Revisiting I/O bandwidth-sharing strategies for HPC applications

Anne Benoit[a], Thomas Herault[b], Lucas Perotin[a], Yves Robert[a,b*], Frédéric Vivien[a]

[a] *Laboratoire LIP, ENS Lyon, UCB Lyon, CNRS, Inria, Lyon, France*
[b] *University of Tennessee, Knoxville, USA*

Abstract

This work revisits I/O bandwidth-sharing strategies for HPC applications. When several applications post concurrent I/O operations, well-known approaches include serializing these operations (FCFS) or fair-sharing the bandwidth across them (FairShare). Another recent approach, I/O-Sets, assigns priorities to the applications, which are classified into different sets based upon the average length of their iterations. We introduce several new bandwidth-sharing strategies, some of them simple greedy algorithms, and some of them more complicated to implement, and we compare them with existing ones. Our new strategies do not rely on any a-priori knowledge of the behavior of the applications, such as the length of work phases, the volume of I/O operations, or some expected periodicity. We introduce a rigorous framework, namely *steady-state windows*, which enables to derive bounds on the competitive ratio of all bandwidth-sharing strategies for three different objectives: minimum yield, platform utilization, and global efficiency. To the best of our knowledge, this work is the first to provide a quantitative assessment of the online competitiveness of any bandwidth-sharing strategy. This theory-oriented assessment is complemented by a comprehensive set of simulations, based upon both synthetic and realistic traces. The main conclusion is that two of our simple and low-complexity greedy strategies significantly outperform FCFS, FairShare and I/O-Sets, and we recommend that the I/O community would implement them for further assessment.

Keywords: I/O, bandwidth sharing, scheduling strategy, HPC applications.

*corresponding author

Email addresses: anne.benoit@ens-lyon.fr (Anne Benoit[a]), herault@icl.utk.edu (Thomas Herault[b]), lucas.perotin@ens-lyon.fr (Lucas Perotin[a]), yves.robert@ens-lyon.fr (Yves Robert[a,b]), frederic.vivien@inria.fr (Frédéric Vivien[a])

1. Introduction

HPC applications do not share computing resources: all the nodes assigned to a given application are dedicated to that application throughout its execution. Such a mode of operation is enforced to guarantee a sustained level of performance to all applications that execute concurrently on the platform. However, concurrent applications do share both the interconnexion network and the parallel file system. When several applications request to perform an I/O operation simultaneously, they have to share the resource, which leads to interferences and performance degradation.

Several researchers have already identified and addressed this problem (see [8, 13, 27, 2, 30, 5, 29] among others). Performance degradation due to I/O is already significant for current state-of-the-art platforms and is expected to worsen due to the faster increase in processing speed than in I/O bandwidth [24]. The problem can be partially mitigated by reducing the volume of data transfers, e.g., via compression or in-situ processing. But the main question remains: given several applications executing concurrently and competing for I/O resources, how to orchestrate I/O operations? In other words, scheduling strategies must be designed and evaluated to dynamically assign a fraction of the total I/O bandwidth to individual application transfers. Well-known strategies are FCFS, which gives exclusive I/O access to the first pending I/O operation, and FairShare, which assigns bandwidth proportionally to application transfers.

From a scheduling perspective, which fraction goes to which application at any given time depends upon the optimization metric, such as application progress rate (minimum or average) or platform utilization. When targeting fairness across concurrent applications, a classical objective is to maximize the minimum *yield*, where the yield of an application is the ratio of its actual progress rate over the progress rate that would have been achieved if the application was executing with a dedicated I/O system and always granted the total available bandwidth. We discuss optimization metrics in detail in Section 3.3.

This work focuses on I/O bandwidth-sharing scheduling strategies for HPC applications, revisiting existing strategies and introducing new ones. Our major contributions are described in the following four paragraphs:

General framework. We provide and assess online scheduling strategies that are agnostic of the characteristics of the concurrent applications in terms of processing time and I/O requests. In particular, we do not assume any periodic behavior; several applications execute concurrently and alternate phases of work and phases of I/O operations, whose lengths are not known a priori. Instead, we discover the timing and size of I/O transfers on the fly, as each application posts its operations. We allow for interrupting and resuming on-going I/O operations dynamically, and launching newly posted ones.

Novel strategies. We introduce novel I/O bandwidth-sharing strategies that aim at allocating a fraction of the bandwidth to each application as a function of the current progress of all applications. The main motivation is to maximize the minimum yield that can be achieved each time a scheduling decision is made. These novel heuristics come in several flavors, from simple greedy algorithms to sophisticated decision mechanisms.

Competitiveness analysis. We provide a rigorous framework by focusing on a *steady-state* time window, defined with the following three rules. Throughout the window: (i) several applications, each with a processing history, execute concurrently; (ii) none of them terminates; and (iii) no new application can start. Thus, the window corresponds to a steady-state mode of behavior where each application progresses at the rate enforced by the I/O bandwidth-sharing strategy. Focusing on such a window is key to assess performance. Otherwise, say if some application would terminate before the end of the window, the batch scheduler would likely launch a new application, whose starting time and progress up to the end of the window would depend on all previous scheduling decisions. The same holds if a new application is launched in the middle of the window. Getting rid of the interaction with the batch scheduler, we provide the first complexity results on the performance of several I/O bandwidth-sharing strategies, some old and some new, and for various optimization objectives.

Comprehensive simulation campaign. We compare existing and novel I/O bandwidth-sharing strategies on an extensive set of application scenarios, some generated from realistic traces derived from the APEX workflows report [19], and some with synthetic parameters. A key parameter is the I/O pressure W, defined for a steady-state window $[T_{begin}, T_{end}]$, as the ratio $\frac{V}{B(T_{end}-T_{begin})}$, where (i) V is the total I/O volume (accumulated for all applications) to transfer during the window; and (ii) B is the total I/O bandwidth (see Section 6.1 for details). In a nutshell, if this ratio is close to 1 or even exceeds 1, the set of I/O operations saturates the I/O system, and many I/O operations will have to be delayed. We study how rapidly the performance of each strategy degrades for high I/O pressures, thereby paving the way for a fair bandwidth allocation on future platforms. We point out that simulations are a first but mandatory step to assess the limitations and strengths of all the I/O bandwidth-sharing strategies. Our extensive set of experiments corresponds to several months of platform usage and would have been impossible to deploy on a large-scale platform, even if we had both permission and budget to conduct them. The main conclusion is that two simple and low-complexity greedy strategies significantly outperform FCFS, FairShare and I/O-Sets.

We conclude this section by stating the main limitations of this work (see Section 3 for a detailed list of application and platform parameters):

- We model the storage system as a black box: we assume that it is a monolithic block and that it can offer a fixed bandwidth to all applications regardless of their I/O patterns and

placement on the network. In practice, this is a simplifying assumption because: (1) an HPC storage system is made of hundreds (sometimes thousands) of storage nodes in a complex topology; (2) each compute node does not have the same bandwidth, latency, and number of hops to reach each storage node; and (3) applications do not usually access all the storage nodes, and for those nodes that they do access, they do not necessarily access them in the same manner.

- We assume the existence of an I/O controller, i.e., a centralized entity that: (1) can see all the I/O traffic; and (2) can make split-microsecond decisions on how to handle it.

Both assumptions are introduced to make the problem tractable. As for the first assumption, the recent survey [4] states that *the multi-layered software and hardware HPC I/O stack is complex. To access data in HPC systems, applications issue requests that, while traversing the I/O stack, are reshaped via a series of data transformations. These originate from distinct abstractions and mappings between the data models used in each layer combined with optimization techniques applied before reaching the file system and, eventually, the storage hardware.* Assessing the performance of scheduling algorithms in the framework of the full I/O stack is impossible without extensive experiments conducted on a variety of platforms. As for the second assumption, although an application-level I/O controller is not generally available everywhere, projects like [13] and [8] provide initial implementations on which such a system can be built. In our evaluation, we consider the cost of taking the scheduling decisions, and this cost partly drives our final recommendations. Altogether, the design, analysis and comparison in simulation of all the scheduling algorithms introduced in this work lay the foundations of I/O bandwidth-sharing strategies, and represent a preliminary but mandatory first step before further assessment by the community.

The paper is organized as follows. We first survey related work in Section 2. Then, we detail the application and platform framework in Section 3, together with the optimization objectives. We detail well-known bandwidth-sharing strategies, and introduce new ones, in Section 4. Complexity results are stated in Section 5 in the form of lower bounds for competitive ratios. The experimental evaluation in Section 6 presents extensive simulation results comparing all the strategies. Finally, we conclude and provide hints for future work in Section 7.

2. Related Work

We discuss related work in this section. We survey existing approaches before pointing to a related problem in the scheduling literature.

CALCioM [8]. This pioneering paper introduces and experimentally compares three policies to manage cross-application coordination of I/O operations: (i) *Interference* (called FairShare in this

paper), where the total bandwidth is shared equally[1] among all concurrent operations; (ii) *FCFS-based serialization* (called FCFS in this paper), where I/O operations are serialized based upon an FCFS priority; and (iii) *Interruption-based serialization*, where I/O operations are serialized but preemptive, allowing for another operation B to interrupt the current operation A, which resumes only after the completion of B. Some examples are given to explain when to favor a given strategy, but no general approach is explored. In particular, interruption-based serialization would require to set priorities among applications, which are not detailed in the paper. Altogether, this work presents one of the first comparisons of bandwidth-sharing strategies, and we build upon their ideas to cast a general framework and introduce new strategies.

CLARISSE [13]. This paper introduces a middleware designed to enhance data-staging coordination and control in the HPC software storage I/O stack. Among many other contributions, the CLARISSE middleware enables to directly compare the *no-scheduling* strategy (called FairShare in this paper) with FCFS and reports performance gains for the latter. Intuitively, the superiority of FCFS can be expected as it comes from a classic result in parallel computing: when scheduling two identical communications that can each make use of the full bandwidth, better serialize them than execute them concurrently. Indeed, with serialization, the first communication ends at time t and the second one at time $2t$ (for a duration t, assuming a start at time 0), while in parallel, both communications end at time $2t$. However, our analysis and experiments reveal that this intuition can be misleading and that (i) FairShare prevails over FCFS in many practical scenarios and (ii) more sophisticated policies that account for past history to set priority-based bandwidth assignments perform even better.

I/O-Cop [27]. I/O-Cop is a prototype system aimed at exploring access control mechanisms to manage the shared Parallel File System (PFS) of the platform. This work is motivated by revealing the contention incurred when several applications aim at performing I/O transfers simultaneously. The I/O-Cop prototype is limited to the case when the access controller to the Parallel File System (PFS) provides exclusive access to a single application at a given time, and without allowing for preemption of ongoing I/O operations.

QoS-based and reward-based approaches. In [28], the authors also advocate controlling accesses to the PFS in order to achieve some Quality of Service (QoS) for each application. They envision a system with several I/O storage devices (disks, SSDs or NVRAMs) and aim at load-balancing I/O requests across all storage types to minimize contention. In [26], the authors consider several applications that execute concurrently and post I/O requests. They partition all the I/O requests

[1]More precisely, FairShare shares the total bandwidth in proportion to the size of the concurrent applications, see Section 4.1 for details.

into several queues, one per application, and aim at establishing priorities across the applications. The idea is that after completing some I/O transfer, a given application could be granted access for its next I/O transfer before all the other applications would have completed one I/O transfer themselves. At each time-step, the progress of each application is monitored as the number of I/O transfers that have been granted so far. In a related paper [12], the authors survey I/O capabilities of state-of-the-art supercomputers and enforce QoS constraints for I/O transfers by implementing a token-based bucket algorithm that works similarly to that of [26]. Finally, the authors of [25] target a system with several I/O sub-systems (OST, which stands for Object Storage Target, typically a RAID array of disks). For each application, they aim at the same share of available bandwidth on each OST, because it balances transfers (one needs to wait for the last node to complete its transfer before resuming work). The allocation of nodes (hence applications) to the different OSTs is given by some external mechanism. Then, on a given OST, some application may benefit from an increased bandwidth, which is done by throttling another application. The throttled application is issued a coupon, to be redeemed later. They do not deal with the interplay of successive I/O operations and work phases, and no comparison is made with other strategies. In contrast, our work restricts to a single OST but provides a comprehensive comparison of several bandwidth-sharing strategies.

Periodic applications. A series of papers [10, 1, 2, 7, 14] focus on periodic applications that consist of work phases followed by I/O operations. More precisely, each application repeats a two-phase period with a fixed computing length followed by an I/O of volume. The CPU lengths and I/O volume depend upon the application, but remain the same from one period to the next. The major goal of these works is to orchestrate a global periodic scheme where I/O transfers are meticulously shaped to fill up the smallest possible rectangle that will repeat. While the problem of finding the minimum size rectangle is shown to be NP-complete in the initial work [10], several interesting heuristics have been developed in the subsequent papers. The approach is quite flexible, with I/O transfers possibly split into different sub-transfers, each with a different bandwidth. The main limitation is of course the assumed periodicity of each application. An extension is provided by other authors in [30], where applications still consist of phases with work followed by I/O transfers, but now CPU phases have stochastic lengths taken from some probability distribution, while I/O phases have constant length. As a motivation, for CPU phases, we can think of a constant amount of flops to perform, with some system-dependent or data-dependent noise, while for I/O transfers, we can think of a fixed-size checkpoint operation. In contrast, our approach does not assume any a priori knowledge of the concurrent applications.

I/O-Sets [5]. This recent work can be viewed as an interesting extension of the work in [2] for periodic applications. Each application consists of several iterations, which as above are work

phases followed by I/O operations. Periodicity is no longer assumed. Instead, for each application, they determine the value of ω, which is the average length of an iteration so far. In [5], CPU lengths and I/O volumes are sampled from some probability distributions (that differ for each application), which enables to compute ω with the expectations of these distributions, but one could envision to acquire the value of ω on the fly, as the application progresses. Then, the applications are partitioned into I/O-sets: two applications belong to the same set if they have the same value for $\lceil \log_{10} \omega \rceil$. Each I/O-set is assigned a priority. The I/O bandwidth-sharing strategy is described in detail in Section 4.2. In a nutshell, FCFS is enforced within each I/O-set; hence, at most one application per I/O-set is competing for bandwidth at any time-step. Then some priority-based sharing is enforced across I/O-sets. The motivation for using a mixture of FCFS and FairShare (or more precisely a priority-based variant of sharing) is very interesting: small and large applications (characterized by different orders of magnitude for ω) should not be treated equally by the scheduler. The I/O-sets strategy has several parameters, and we use the same instantiation as in [2], with the same name Set-10. We use Set-10 as a competitor for our novel strategies.

ThemisIO [16]. This recent work introduces an efficient I/O sharing framework for burst-buffers. This full-fledged middleware enables the user to implement their own bandwidth-sharing policies. ThemisIO handles several priority levels, such as users (who may each have several jobs executing concurrently) and groups (each with several users). ThemisIO also deals with several I/O servers, thereby enabling to implement a fair strategy across the whole platform. At the job level, their main strategy is *size-fair*, which is exactly the FairShare strategy in our work: each job receives a fraction of the available bandwidth that is proportional to its size, i.e., its number of nodes. In this work, we introduce novel bandwidth-sharing strategies that go well beyond FairShare, accounting for application progress, not just application size.

A note on the painter problem. In the scheduling literature, the *painter* problem, a.k.a the *scheduling with delays* problem, is the following: (i) several chains of tasks are to be scheduled on a single machine; (ii) for each chain, there is a minimal *delay* to be enforced between the completion of a task and the start of its successor. As for the analogy with a painter: the painter is the machine and has several rooms to paint on its agenda, each with several paint layers (a task is the application of a paint layer); for each room (each chain), there is a delay between the end of a layer and the next one. The tasks are not preemptive. Release times can be simulated by adding delays from a fake source task. This is an offline problem where the objective is to minimize either the makespan (maximum completion time of a task) or the total flow (unweighted or weighted sum of all completion times). The analogy with the I/O problem is clear: the machine is the I/O resource, the task chains are the applications, the tasks are the I/O operations, and the delays are

the computing phases between two consecutive I/O operations. The main differences with the I/O scheduling problem are the following:

1. Execution is not preemptive in the painter problem, while one can pause an on-going I/O operation;
2. A single task is executed at any time step while several I/O operations (from different applications) can share the I/O bandwidth;
3. All chain parameters (task lengths and delay values) are known at the beginning of the execution while the lengths of work phases and the volumes of I/O operations are discovered on the fly in the I/O scheduling problem.

Particular instances of the painter problem have been shown to have polynomial complexity. We refer to the interested reader to [22, 23, 6, 20, 9] for details. A survey of recent results and extensions is available in [18].

3. Framework

In this section, we describe the framework. We start with application characteristics and detail rules for I/O operations and bandwidth allocation in Section 3.1. We discuss the interaction with the batch scheduler and explain why we restrict to steady-state time windows in Section 3.2. We conclude with optimization objectives in Section 3.3. Main notations are summarized in Table 1.

3.1. Applications

3.1.1. Application Characteristics

We consider a very general framework where applications are submitted online to the batch scheduler. Each application $\mathcal{A}_i$ requests p_i nodes and starts executing as soon as the batch scheduler has been able to allocate that many nodes. Thus, each application executes on a dedicated set of

m	number of applications
p_i	size (number of nodes) of application $\mathcal{A}_i$
τ_i	release time of application $\mathcal{A}_i$
$w_i^{(j)}$	duration of work phase number j for application $\mathcal{A}_i$
$v_i^{(j)}$	volume of I/O operation number j for application $\mathcal{A}_i$
B	total bandwidth of the I/O system
b	bandwidth of each platform node
b_i	maximal bandwidth of application $\mathcal{A}_i$: $b_i = \min(p_i b, B)$
α_i^j	fraction of bandwidth assigned to $\mathcal{A}_i$ for I/O operation j (can vary over time)
$[T_{begin}, T_{end}]$	steady-state window
$y_i(t)$	yield of application $\mathcal{A}_i$ at time t

Table 1: Summary of main notations.

nodes throughout its execution, which is the standard approach on large-scale HPC platforms. However, all applications execute I/O transfers (reads and writes) through the I/O controller and share the bandwidth of the I/O system. Our approach is agnostic of the nature of the storage (SSDs, NVRAMs, disks or tapes), and of the organization of the PFS (Parallel File System).

Each application $\mathcal{A}_i$ executes an alternating sequence of work phases and I/O operations, which we represent as follows:

$$\mathcal{A}_i \equiv \quad v_i^{(0)}, w_i^{(1)}, v_i^{(1)}, w_i^{(2)}, \ldots, v_i^{(n_i-1)}, w_i^{(n_i)}, v_i^{(n_i)}, \ldots$$

where $v_i^{(j)}$ stands for I/O volumes, and $w_i^{(j)} > 0$ stands for (parallel) work units. Because computing nodes are dedicated to the application, we can assume w.l.o.g. that one unit of work lasts one second, so that the $w_i^{(j)}$ represent the duration of the work phases; more precisely, within w_i seconds, each of the p_i nodes performs w_i work units. However, because we do not know the bandwidth of I/O operations in advance, we have to express them in volume (amount of bytes) rather than in duration. We detail rules for bandwidth allocation in Section 3.1.2. We will discuss rules for posting and managing I/O operations in Section 3.2.2.

As stated before, the length $w_i^{(j)}$ of each work phase is not known until it terminates, and the volume $v_i^{(j)}$ of each I/O operation is not known until the operation is posted to the I/O controller. Similarly to the related work surveyed in Section 2, we also assume that I/O operations are blocking and coordinated between the different nodes of the application, and that the application does not overlap I/O operations with some work phase. This is typical of HPC applications using a synchronous global interface like MPI-IO [21, 15], which also provides the I/O controller with critical information like the volume of data to transfer.

3.1.2. Bandwidth Allocation

Consider an application $\mathcal{A}_i$ executing on p_i nodes and initiating an I/O operation of volume $v_i^{(j)}$. What are the bandwidth allocation rules for this operation? We let b be the bandwidth of the network card (of interface card) of each node, and B be the total bandwidth of the I/O system.

First, assume for simplicity that the I/O operation is not interrupted, and is granted the same bandwidth from start to completion. The maximal bandwidth that can be granted by the I/O controller is

$$b_i = \min\left(p_i b, B\right). \tag{1}$$

Note that Equation (1) implicitly assumes that each node of $\mathcal{A}_i$ has to transfer (approximately) the same volume of data to/from the PFS. If transfers are unbalanced from one node to another, we should redefine $v_i^{(j)}$ as $v_i^{(j)} = p_i v_{i,\max}^{(j)}$, where $v_{i,\max}^{(j)}$ is the maximum volume of data to be transferred by any of the p_i nodes of $\mathcal{A}_i$. The main rule of the game for the scheduler is to assign a fraction $\alpha_i^{(j)}$ of the maximal bandwidth b_i to the I/O operation $v_i^{(j)}$. The duration of the I/O

operation will then be

$$d_i^{(j)} = \frac{v_i^{(j)}}{\alpha_i^{(j)} b_i}. \tag{2}$$

Of course, if no I/O operation has been posted by another application, the scheduler will enforce $\alpha_i^{(j)} = 1$ to ensure fastest possible completion. In that case, we use the notation

$$d_{i,\min}^{(j)} = \frac{v_i^{(j)}}{b_i} \tag{3}$$

to denote the minimal possible duration of the I/O operation. On the contrary, in the presence of several concurrent I/O operations, the scheduler will resort to some bandwidth-sharing strategies, like the ones studied in this paper.

We are ready to discuss the general case, which will require some additional notations. Intuitively, a given I/O operation will **not** be granted the same bandwidth fraction throughout execution. At any time-step t, some I/O operations that were posted before are granted some bandwidth and executing, while some others may be pending (that is to say their fraction is currently 0). A new I/O operation may be posted at time t, which the scheduler can account for by granting it some bandwidth, at the price of reducing the fraction of other applications. On the contrary, some on-going I/O operation may complete at time t, thereby opening the possibility of a larger fraction to be granted to some applications. We see that bandwidth fractions are granted only for some duration, which we call the *horizon*. Decisions are taken at specific instants, which we call *events*. Typically, an event corresponds to the posting of a new I/O operation, or to the termination of an on-going one. But an event can also be triggered by the I/O scheduler, e.g., for a strategy where additional events are created periodically, say every 10 seconds. The I/O controller takes a new decision at every event, as explained below. The constraints on the number of events, and the cost of bandwidth-sharing strategies, will be detailed in Section 3.2.2.

Consider an event at time t, and let $S(t)$ be the index set of *active* applications, i.e., applications that have posted an I/O operation before time t which is not yet completed, or applications that post a new I/O operation exactly at time t. Among the applications with incomplete I/O-operations, some may be transferring data at some bandwidth fraction and some may be kept waiting. Each active application $\mathcal{A}_i$, $i \in S(t)$, is allotted a bandwidth $\alpha_i^t b_i$ (with some α_i^t possibly 0) so that

$$\sum_{i \in S(t)} \alpha_i^t b_i \leq B. \tag{4}$$

This bandwidth allocation remains valid until the next event at time $t + h$, where h is the horizon. The bandwidth allocation depends upon the bandwidth-sharing strategy, whose inputs are the volume of data that must still be transferred for each on-going I/O operation, the knowledge of the

progress of all active applications so far, and the optimization objective.

We stress that the horizon h is unknown at time t. The next event is triggered either by a new post or a completion, or again by an external decision given to the I/O controller. At time t, after having granted bandwidth fractions to active applications, we only know that h is greater than the time needed to complete the shortest on-going I/O operation, given that no new event (new post or external) will happen before that.

When the next event takes place at time $t + h$, we update the set of active applications, leaving out I/O operations that have completed and including new posts, if any. We also update the remaining volume of data still to be transferred for each active application. The I/O controller applies the bandwidth-sharing strategy for this new set of parameters.

3.2. Steady-State Windows

In this section, we recall the management of HPC applications by the batch scheduler, and explain why we need to restrict to steady-state time windows to assess the performance of bandwidth-sharing strategies.

3.2.1. Interaction with the Batch Scheduler

HPC applications are submitted to the batch scheduler. Each application $\mathcal{A}_i$ has a release time τ_i, a size p_i and a wall-time res_i (length of the reservation slot). Upon release, application $\mathcal{A}_i$ is put in the queue of the batch scheduler and will be allocated resources at time $t_i^{alloc} \geq \tau_i$, which means that p_i nodes are dedicated to the application during the interval $[t_i^{alloc}, t_i^{alloc} + res_i)$. The p_i nodes are released as soon as the application completes its execution or its deadline is reached, whichever comes first.

Each application has dedicated nodes but all applications that execute concurrently share the I/O system. I/O operations are posted by the applications and managed by the I/O controller. If an application posts an I/O operation while another I/O operation has already been granted access, several scenarios can happen, depending upon the bandwidth-sharing policy implemented by the I/O controller. We have already discussed the FCFS and FairShare strategies in Section 2, and will introduce other strategies in Section 4. Whenever the I/O controller makes a decision according to its bandwidth-sharing policy, this decision has an impact on the progress of all active applications. Altogether, the bandwidth-sharing policy will change the termination time of all applications. In theory, some applications may even fail to complete before the end of their reservation due to the bandwidth-sharing strategy being disadvantageous to them. On the contrary, some applications may benefit from the strategy and complete early, thereby releasing their resources early. In summary, the opportunities for decisions of the batch scheduler to allocate new applications will depend upon the bandwidth-sharing strategy applied to the applications that are currently executing. Furthermore, any decision of the batch scheduler changes the mix of

applications that run concurrently and possibly compete for I/O resources. This, in turn, changes the scope and impact of the decisions of the bandwidth-sharing policy implemented by the I/O controller. Altogether, the interplay between the batch scheduler and the decisions of the I/O controller is hard to comprehend.

To the best of our knowledge, none of the papers surveyed in Section 2 has dealt with this difficulty. Instead, these papers consider a fixed number of applications that execute concurrently (each on a dedicated set of nodes) and compete for I/O access. This amounts to consider an execution window $[T_{begin}, T_{end}]$ where all applications start executing at time T_{begin} and do not complete execution before time T_{end}, regardless of the I/O policy that is implemented. In other words, the platform operates in steady-state mode during the window $[T_{begin}, T_{end}]$ with no application terminating nor no new application launched throughout the window. This assumption is never stated in recent papers. Again, the reason why it is assumed that applications do not complete before the end of the window is the following: if an application terminates at time $T < T_{end}$, the batch scheduler might launch another application right after the completion. Because T depends on the bandwidth-sharing strategy that is enforced, it becomes impossible to assess the performance of the strategy by itself.

In this paper, we use a steady-state execution window $[T_{begin}, T_{end}]$ and assume that m applications $\mathcal{A}_i$ $(1 \leq i \leq m)$ execute concurrently throughout the window. To eliminate side effects and deal with a general scenario, we do not assume that the applications start executing at time T_{begin}: on the contrary, the applications may have been launched earlier and have been executing for some time. The history of the applications will be taken into account when evaluating the objective function (see Section 3.3).

3.2.2. Cost Model for Steady-State Windows

Given a steady-state execution window $[T_{begin}, T_{end}]$, assume that m applications $\mathcal{A}_i$ $(1 \leq i \leq m)$ execute concurrently throughout the window. Each application $\mathcal{A}_i$ will execute a series of work phases followed by I/O transfers. If the application $\mathcal{A}_i$ was alone on the platform, all I/O transfers would be granted maximal bandwidth b_i. Let $N_{op}(i)$ be the number of I/O operations that would be initiated from time T_{begin} until time T_{end}, assuming such a dedicated mode.

In concurrent mode, we introduce two *events* for each I/O operation, one when it is posted, and one when it completes. The total number of events due to I/O operations is upper bounded by

$$E = \sum_{i=1}^{m} 2N_{op}(i). \tag{5}$$

Indeed, no application will perform more I/O operations by the end of the window than in dedicated mode, hence the number of events for each application $\mathcal{A}_i$ never exceeds $2N_{op}(i)$, regardless of the

bandwidth-sharing strategy.

The value of E is a key parameter to the size of the problem (other parameters include the binary encoding of work lengths and I/O volumes). We enforce that all bandwidth strategies have a cost polynomial in E, meaning that the number of bandwidth-sharing decisions remains polynomial in E. For instance, if the I/O controller enforces periodic decisions every h seconds, where h is a fixed horizon, the number of additional events $E^{(+)} = \lfloor \frac{T_{end} - T_{begin}}{h} \rfloor$ must remain polynomial in E. We use $E^{(+)} = E$ in the simulations to add equi-spaced decisions across the steady-state window. Note that triggering an external event every second would lead to $T_{end} - T_{begin}$ external events, which is exponential in the problem size (we use a logarithmic encoding for all parameters).

To the best of our knowledge, none of the papers surveyed in Section 2 has discussed how frequently decisions should be taken, nor has included the cost of the bandwidth-sharing strategy each time a decision is taken. We could easily include that cost into the assessment of the performance of the strategies. We do not, because the cost is inherent to the strategy and independent of the actual length of the work phase and I/O operations: if we multiply the latter quantities (and the window size) by a factor 10 or 100, the cost of the strategy remains the same and becomes negligible in front of the execution time of the applications.

3.3. Objectives

In this section, we define the *yield* of an application. The major objective of our novel bandwidth-sharing strategies is MinYield, the maximization of the minimum yield over all applications executing within the steady-state window $[T_{begin}, T_{end}]$. However, we also report performance for two other objectives, Utilization and Efficiency, which we describe at the end of this section.

Consider an application $\mathcal{A}_i$ that is released at time $\tau_i = 0$. Consider a steady-state window $[T_{begin}, T_{end}]$. At any time $t \geq T_{begin}$, we want to monitor the progress of $\mathcal{A}_i$ in terms of work done and data volume transferred. Recall that $\mathcal{A}_i$ executes an alternating sequence of work phases (work) and I/O operations:

$$\mathcal{A}_i \equiv \quad v_i^{(0)}, w_i^{(1)}, v_i^{(1)}, w_i^{(2)}, \ldots, v_i^{(n_i-1)}, w_i^{(n_i)}, v_i^{(n_i)}, \ldots$$

We have assumed unit speed for work phases, and we normalize I/O volumes by the maximal possible bandwidth $b_i = \min(p_i b, B)$. Letting $d_{i,\min}^{(j)} = \frac{v_i^{(j)}}{b_i}$ be the minimum duration for I/O operation number j of volume $v_i^{(j)}$, we rewrite $\mathcal{A}_i$ as

$$\mathcal{A}_i \equiv \quad d_{i,\min}^{(0)}, w_i^{(1)}, d_{i,\min}^{(1)}, w_i^{(2)}, \ldots, d_{i,\min}^{(n_i-1)}, w_i^{(n_i)}, d_{i,\min}^{(n_i)}, \ldots$$

The *ideal progress* of $\mathcal{A}_i$ at time t is the amount of work plus the volume of data transferred

since its release time τ_i and up to time t, when all I/O operations have taken place with no delay and at the maximal possible bandwidth b_i. This corresponds to $\mathcal{A}_i$ progressing at maximal rate, which happens if it executes in dedicated mode on the platform. By definition, at time t, the ideal progress is equal to $t - \tau_i$.

In a concurrent execution, the *actual progress* of $\mathcal{A}_i$ at time t is the amount of work plus the volume of data transferred since its release time τ_i and up to time t. While work phases still progress at full (unit) speed, I/O operations are slowed down by interferences. For any time $t \in [T_{begin}, T_{end}]$, let $W_i^{(done)}(t)$ be the total amount of work done up to time t, and $V_i^{(transferred)}(t)$ be the total volume of data transferred up to time t. The *yield* of $\mathcal{A}_i$ at time t is defined as the ratio of the actual progress over the ideal progress, namely

$$y_i(t) = \frac{W_i^{(done)}(t) + \frac{V_i^{(transferred)}(t)}{b_i}}{t - \tau_i}. \tag{6}$$

As a side note, we show how to compute the value of $V_i^{(transferred)}(t)$ as the concurrent execution goes. We do this computation incrementally, one work phase or I/O operation after another. Consider the I/O operation number j and assume that it has occurred during the interval $[start_i^{(j)}, end_i^{(j)}]$ ($end_i^{(j)}$ is equal to the completion time of this I/O operation and $start_i^{(j)}$ to the completion time of the previous work phase). Let $\alpha_i^{(j)}(u) b_i$ be the bandwidth granted at time $u \in [start_i^{(j)}, end_i^{(j)}]$, where $0 \leq \alpha_i^{(j)}(u) \leq 1$ (and let $\alpha_i^{(j)}(u) = 0$ for u outside this interval). If the I/O operation number j is not complete at time t, i.e., if $t \in [start_i^{(j)}, end_i^{(j)})$, the amount of data volume $V_i^{(j)}(t)$ transferred up to time t is

$$\int_{start_i^{(j)}}^{t} \alpha_i^{(j)}(u) b_i du = V_i^{(j)}(t). \tag{7}$$

In fact, the integral is a discrete sum of at most E components, since we change bandwidth allocation only when a new event takes place. Note that if $t \geq end_i^{(j)}$, we obtain $V_i^{(j)}(t) = v_i^{(j)}$. Equation (7) enables us to compute the actual progress incrementally, from one work phase or I/O operation to the next. Of course, the actual progress depends upon the bandwidth-sharing strategy through the choice of the fractions $\alpha_i^{(j)}(u)$ of the maximal bandwidth b_i allotted at every instant u.

We are ready to state the optimization objectives, together with their initial motivation. Consider a steady-state window $[T_{begin}, T_{end}]$ and m applications. Each application $\mathcal{A}_i$ has a yield $y_i(T_{begin})$ when entering the window. The three target objectives are MinYield, Utilization and Efficiency.

MinYield. The objective is to maximize the minimum yield at the end of the window:

$$\text{Maximize} \min_{1 \leq i \leq m} y_i(T_{end}). \tag{8}$$

This objective aims at enforcing fairness among all the applications, regardless of their characteristics. The intuition is that all applications suffer from the same slowdown factor if they achieve the same yield. As discussed in Sections 1 and 2, previous work has shown the limitations of FCFS and FairShare, which give priority to some applications and severely slow down other ones. MinYield will guide bandwidth-sharing decisions so that all applications exit the window with balanced yields. An application entering the window with a very low yield will be granted more bandwidth to catch up.

Utilization. The objective is to maximize platform utilization throughout the window:

$$\text{Maximize} \frac{\sum_{1\leq i}^{m} p_i \left(W_i^{(done)}(T_{end}) - W_i^{(done)}(T_{begin}) \right)}{(T_{end} - T_{begin}) \sum_{1\leq i}^{m} p_i}. \tag{9}$$

The work $W_i^{(done)}(T_{end}) - W_i^{(done)}(T_{begin})$ done by each application $\mathcal{A}_i$ within the window is weighted by its size p_i. This objective is the classical performance objective from the perspective of the administrator or owner of the platform, because it measures the fraction of time where computing nodes have been used for actual application work. Hence, this objective is natural for HPC applications that perform no or little I/O transfers. However, it may seem ill-suited in a framework focusing on I/O transfers, because it is very sensitive to the ratio of work over data volumes (normalized by maximal bandwidth). For instance, if we multiply all data volumes by, say, 10, platform utilization will plummet, even if we keep the same bandwidth-sharing strategy. This observation leads to introducing the objective Efficiency.

Efficiency. The objective is to maximize the sum of the actual progress of all applications throughout the window:

$$\text{Maximize} \frac{\sum_{1\leq i}^{m} p_i \left(W_i^{(done)}(T_{end}) - W_i^{(done)}(T_{begin}) + \frac{V_i^{(transferred)}(T_{end}) - V_i^{(transferred)}(T_{begin})}{b_i} \right)}{(T_{end} - T_{begin}) \sum_{1\leq i}^{m} p_i}. \tag{10}$$

Comparing Equations (9) and (10), we see that I/O operations are taken into account with Efficiency: this objective aims at optimizing the combined progress of all applications. It can be viewed as a measure of how efficiently platform resources (both compute nodes and the I/O system) are used.

4. Bandwidth-Sharing Strategies

We describe bandwidth-sharing strategies in this section. We start by recalling a few notations and introducing new ones. Consider a steady-state window $[T_{begin}, T_{end}]$ with m applications executing concurrently. Consider an event at time t and let $S(t)$ be the index set of active applications

at time t. Note that applications that are not active are engaged in work phases at time t and progress independently of the decisions made by the I/O controller.

Each active application $\mathcal{A}_i$, $i \in S(t)$, has posted an I/O operation at time $R_i \leq t$ that is not complete at time t. Let $\mathcal{V}_i$ denote the remaining volume still to be transferred for the I/O operation. Each active application is allotted a fraction α_i^t (with some α_i^t possibly 0) of its maximum possible bandwidth $b_i = \min(p_i b, B)$. The bandwidth-sharing strategy consists in determining α_i^t for each active application $\mathcal{A}_i$. Finally, let $\mathcal{BW}_i(t', y)$ denote the bandwidth that should be alloted to application $\mathcal{A}_i$ for it to achieve a yield of at least y at time t'.

We start with some simple greedy strategies, some old and some new, in Section 4.1. Then in Section 4.2, we detail the recent SET-10 strategy proposed in [5]. Finally, in Section 4.3, we sketch an elaborate strategy whose aim is to compute the best horizon for maximizing the minimum yield.

4.1. Greedy Strategies

We discuss below six greedy strategies. The first three strategies do not rely on any (tentative) horizon, while the last two aim at taking some future events into account. Finally, the sixth strategy re-evaluates the current bandwidth allocation at periodic time-steps.

- FAIRSHARE: each active application $\mathcal{A}_i$ with $i \in S(t)$ is allocated $\alpha_i = \min(1, \frac{B}{\sum_{j \in S(t)} b_j})$. Therefore, each application will either saturate its maximal bandwidth b_i, or it will receive a fair share (proportional to its size p_i) of the total bandwidth B. This is the de-facto strategy implemented by the parallel filesystems available in most HPC centers. This strategy does not need to consider what application is requesting the I/O operation, but just how many I/O operations are currently concurrent.

- FCFS: greedily allocate the bandwidth to active applications sorted by non-decreasing R_i. More precisely, up to some re-ordering, let $S(t) = \{1, 2, \ldots k\}$ with $R_i \leq R_{i+1}$ for $1 \leq i < k$. $\mathcal{A}_1$ is granted its maximum bandwidth b_1 (hence, $\alpha_1 = 1$), then $\mathcal{A}_2$ is granted $\alpha_2 b_2 = \min(b_2, B - \alpha_1 b_1)$, and so on until no more bandwidth is available.

- GREEDYYIELD: greedily allocate the bandwidth to active applications sorted by non-decreasing yields $y_i(t)$. The greedy allocation process is the same as for FCFS but with a different criterion, current minimum yield instead of oldest posting time. This strategy gives priority to applications with low yield, so that they can catch up.

- GREEDYCOM: greedily allocate the bandwidth to the applications sorted by non-decreasing ratio $\mathcal{V}_i / b_i$, i.e., by the remaining time to complete the pending I/O operation at maximum possible bandwidth. This strategy gives priority to completing shorter transfers, with the goal of freeing the I/O system as fast as possible and/or give more bandwidth to forthcoming I/O operations.

- LookAheadGreedyYield: for each active application $\mathcal{A}_i$, compute the minimum yield Z_i that can be achieved (over all active applications) if $\mathcal{A}_i$ is given priority and allocated the maximum possible bandwidth b_i, and where the remaining bandwidth $B - b_i$ is allocated following GreedyYield for the other applications in $S(t)$. Then, we retain the allocation that maximizes the minimum yield Z_i obtained with these $|S(t)|$ possible priority choices. The rationale for LookAheadGreedyYield is to look ahead and maximize the minimum yield not at time t, but at time $t + h$, where the horizon h is (tentatively) computed as the end of one ongoing I/O operation.

- PeriodicGreedyYield (δ): this strategy is a variant of GreedyYield where I/O decisions are triggered by external (periodic) events submitted to the I/O controller every δ seconds, in addition to the regular events that correspond to posting and completion of I/O operations. As discussed in Section 3.2.2, we must restrict to a polynomial number of external events. With the notations of Section 3.2.2, we use $E^{(+)} = E$ in the simulations, which leads to choosing $\delta = \frac{T_{end} - T_{begin}}{E^{(+)}}$. At every event, external or regular, bandwidth-sharing decisions are the same as for GreedyYield. The rationale for adding periodic events is to avoid the risk that GreedyYield would apply a bad decision for too long: with several concurrent I/O operations lasting for a long time, greedy decisions are updated every δ seconds, instead of waiting for the first completion of one of these I/O operations.

4.2. Set-10 *Strategy*

This section provides a description of Set-10, the I/O-sets bandwidth-sharing strategy from [5].

Determination of I/O-sets. With the notations of Section 3.3, consider an application $\mathcal{A}_i$ composed of operations

$$v_i^{(0)}, w_i^{(1)}, v_i^{(1)}, w_i^{(2)}, \ldots, v_i^{(n_i-1)}, w_i^{(n_i)}, v_i^{(n_i)}, \ldots$$

Assume that $\mathcal{A}_i$ has just completed the I/O operation $v_i^{(j)}$. Then, the current value of ω^i, the average length of an iteration for $\mathcal{A}_i$, is defined as

$$\omega^i = \frac{1}{j} \sum_{k=1}^{j} (w_i^{(k)} + d_{i,\min}^{(k)}),$$

where $d_{i,\min}^{(j)} = \frac{v_i^{(j)}}{b_i}$, and $b_i = \min(p_i b, B)$. Note that we neglect the initial I/O operation $v_i^{(0)}$ to match the specification of [5]. Then, $\mathcal{A}_i$ is assigned to I/O-set $\mathcal{S}_n$, where $n = \lfloor \log_{10} \omega^i \rceil$, and $\lfloor x \rceil$ denotes the nearest integer to x. Note that an application $\mathcal{A}_i$ may be dynamically reassigned to another I/O-set depending upon the duration of its next work phases and I/O operations. In [5], I/O-set $\mathcal{S}_n$, where $n = \lfloor \log_{10} \omega^i \rceil$, receives a priority $q_n = 10^{-n}$.

Bandwidth assignment. Consider an event occurring at time t, and let $S(t)$ denote the index set of active applications that have a pending I/O transfer at time t. Each participating application $\mathcal{A}_i$, $i \in S(t)$, is allotted a bandwidth $\alpha_i b_i$ computed via the following algorithm [5]:

1. Assume that the applications in $S(t)$ belong to s different I/O-sets $\mathcal{S}_{n_1}, \mathcal{S}_{n_2}, \ldots, \mathcal{S}_{n_s}$.
2. Within each I/O-set, a single application is granted access to the I/O system. In other words, there is exclusive access within sets. If several applications in $S(t)$ belong to the same I/O set, the one with the smallest value of R_i (FCFS, the one that posted its request first) is selected.
3. Now, we have a subset of s applications, one per I/O subset, which will be granted some bandwidth. The intuition is to partition the bandwidth according to the priorities defined above. For simplicity, let us renumber the applications so that $\mathcal{A}_j$ is the application chosen from set $\mathcal{S}_{n_j}$, for $1 \leq j \leq s$. Then, each application $\mathcal{A}_j$ should be granted the fraction $\alpha_j = \frac{q_{n_j}}{\sum_{1 \leq k \leq s} q_{n_k}}$ of the total bandwidth B.
4. As usual, this bandwidth assignment remains valid until the next event.

However, this bandwidth-sharing algorithm implicitly assumes that each application can use the whole system bandwidth: $b_i = B$ for each application $\mathcal{A}_i$. To cope with general scenarios where this is not the case, we have to extend the algorithm. The natural idea is allocate bandwidth to several applications in the same I/O subset, rather than one, while still enforcing the priorities. More precisely, the fraction $\frac{q_{n_j}}{\sum_{1 \leq k \leq s} q_{n_k}}$ of the total bandwidth B is now assigned to several applications from $\mathcal{S}_{n_j}$, chosen greedily in FCFS order. Here is the extended algorithm for bandwidth-sharing:

1. Assume that the applications in $S(t)$ belong to s different I/O-sets $\mathcal{S}_{n_1}, \mathcal{S}_{n_2}, \ldots, \mathcal{S}_{n_s}$.
2. For each I/O-set $\mathcal{S}_{n_j}$, compute the maximum bandwidth fraction that it can receive, namely $\beta_j = \frac{\sum_{k \in \mathcal{S}_{n_j}} b_k}{B}$. As before, let $\alpha_j = \frac{q_{n_j}}{\sum_{1 \leq k \leq s} q_{n_k}}$.
3. We partition the s I/O sets into two categories, those that can receive the fraction α_j and those that are limited by their maximal bandwidth fraction β_j. Let $\mathcal{C}$ be the set of I/O sets of the latter category, i.e., such that $\beta_j \leq \alpha_j$.
4. All the applications $\mathcal{A}_k$ in an I/O set belonging to $\mathcal{C}$ receive their maximal bandwidth b_k.
5. We compute the remaining bandwidth $B_{left} = (1 - \sum_{\mathcal{S}_{n_j} \in \mathcal{C}} \beta_j)B$.
6. We repeat the whole procedure with the remaining I/O-sets and B_{left}, until either there is no I/O-set left, or all remaining I/O-sets have a larger maximal bandwidth than their priority share: $\beta_j \geq \alpha_j$. In the final step, the remaining I/O-sets are granted the fraction α_j of the

remaining bandwidth B_{left}. Within each of these I/O sets, bandwidth is allotted greedily in FCFS order.

Remark on Framework. The I/O-sets strategy [5] does not assume that the total volume of an I/O operation is known when that operation is posted. Instead, they assume that this volume is unknown until the I/O operation ends. They rely on the knowledge of the average length of an iteration for each application, which is acquired from past behavior traces. In our simulations of SET-10, we acquire information on average iteration length on the fly as execution progresses.

As stated in Section 3.1.1, we do assume that the total volume of each I/O operation is known when posted. This knowledge is necessary for GREEDYCOM, LOOKAHEADGREEDYYIELD (described in Section 4.1) and BESTNEXTEVENT (described below in Section 4.3). However, GREEDYYIELD and LOOKAHEADGREEDYYIELD (also described in Section 4.1) do not need any information at all on the applications, they only need to compute application yields on the fly. And of course FAIRSHARE and FCFS do not need any information either.

4.3. Maximizing the Minimum Yield at the Next Event

Given an event at time $t \in [T_{begin}, T_{end}]$, the aim of strategy BESTNEXTEVENT is to find the *best predictable* event in the remainder of the window $]t, T_{end}]$. A predictable event is either the end of the execution window (at time T_{end}) or the first time one of the currently on-going I/O operations is completed, whichever comes first. The best predictable event is the predictable event at which point the minimum yield will be maximized. Of course, if an unpredictable event, such as the posting of a new I/O operation, surges before the best predictable event, the bandwidth-sharing strategy will account for it and recompute the best predictable event from that time on.

A priori, there are infinitely many dates in the interval $[t, T_{end}]$, at which the next predictable event can happen; hence, we cannot test each and every one of them. Instead, we partition the interval $[t, T_{end}]$ into a polynomial (in practice, quadratic) number of sub-intervals. The extremities of these sub-intervals will be either the earliest date at which an I/O operation can complete, or the time at which the characteristic yield functions of two applications intersect (see below for details; the characteristic yield function of an application will be, for instance, its maximum achievable yield at time t', or its yield at time t' if it is allocated no bandwidth, etc.).

Let $t = t_1 \leq t_2 \leq \ldots \leq t_{n_{\text{int}}} = T_{end}$ be the extremities of these sub-intervals. For each sub-interval $[t_i, t_{i+1}]$, we will consider each application $\mathcal{A}_k$ that can define an event in (t_i, t_{i+1}) (hence, each application $\mathcal{A}_k$ such that $t_i \geq \frac{\mathcal{V}_k}{b_k}$). Then, we search for the event defined by $\mathcal{A}_k$ that maximizes the minimum yield in $[t_i, t_{i+1}]$. For that purpose, we start by looking for the best solution at time t_i. Once we have identified that solution, we determine the largest interval $[t_i, t_i'] \subset [t_i, t_{i+1}]$ such that for any $t' \in [t_i, t_i']$ the optimal solution at time t' has the same structure (which applications are allocated bandwidth, which application is allocated its maximal bandwidth, etc.) as the one at

time t_i. If $t_i = t_{i+1}$, we conclude. Otherwise, we call recursively the algorithm on the interval $[t'_i, t_{i+1}]$.

Because application $\mathcal{A}_k$ is defining an event at time t_i, it receives the bandwidth $\frac{\mathcal{V}_k}{t_i - t}$, where $\mathcal{V}_k$ is the remaining volume at time t (hence, the I/O operation completes at time t_i). The remaining bandwidth $B - \frac{\mathcal{V}_k}{t_i - t}$ must be distributed among the other applications. We first compute an upper-bound, y^{UB}, on the maximum minimum yield: y^{UB} is the minimum, over all applications, of the maximum yield achievable by each application at time t_i. We then check whether this upper-bound can be achieved without exceeding the total bandwidth B.

We do not provide all the details and algorithms of this BestNextEvent strategy, which are available in the companion research report [3], in Sections 4.3 and A. Altogether, BestNextEvent is quite complicated, and admittedly too complicated for practical use. But it will serve as a reference to help us assess the quality of the (simpler) greedy strategies of Section 4.1.

5. Lower Bounds on Competitive Ratios

This section provides lower bounds for the performance of the bandwidth-sharing strategies. The results are summarized in Table 2. For instance, the first entry $m^{(1)}$ in the table means that FairShare has a competitive ratio not better than m, and that the proof of this result is given by Example 1. An entry ∞ means that the strategy does not have a ρ competitive ratio for any value of $\rho \geq 1$. All examples are available in Section 5 of the companion research report [3], as well as results on tightness of some bounds. For the sake of conciseness, we only present Example 2 in this paper, which is used to prove several results, in order to illustrate the reasoning used to obtain these lower bounds.

Example 2. We consider a window $[T_{begin}, T_{end}] = [0, 1]$. $m \geq 4$ applications are released at time 0. Each application $\mathcal{A}_i$ verifies $b_i = B = 1$ and $p_i = 1$. We assume that m is even. We suppose that

	MinYield	Efficiency	Utilization
FairShare [8, 13]	$m^{(1)}$	$\frac{m}{4}^{(2)}$	$\infty^{(2)}$
FCFS [8, 13]	$\infty^{(2)}$	$m^{(3)}$	$\infty^{(2)}$
Set-10 [5]	$\infty^{(2)}$	$m^{(3)}$	$\infty^{(2)}$
GreedyYield	$\infty^{(2)}$	$m^{(3)}$	$\infty^{(2)}$
GreedyCom	$\infty^{(2)}$	$\frac{m}{4}^{(2)}$	$\infty^{(2)}$
LookAheadGreedyYield	$\infty^{(2)}$	$m^{(3)}$	$\infty^{(2)}$
PeriodicGreedyYield ($\delta \to 0$)	$2^{(4)}$	$m^{(3)}$	$\infty^{(2)}$
BestNextEvent	$\frac{m}{2} - 4^{(6)}$	$m^{(3)}$	$\infty^{(2)}$
Any strategy	$\frac{3}{2}^{(5)}$	$\frac{m}{4}^{(2)}$	$\infty^{(2)}$

Table 2: Lower bounds for the competitive ratios of bandwidth-sharing strategies.

$m/2$ applications are in category A, i.e., have an I/O operation of volume $\frac{2}{m}$, followed by a work phase of length 1. The other $\frac{m}{2}$ applications are in category B, i.e., have an I/O operation of volume $\frac{2}{m}$ followed by a work phase of length $\alpha = \frac{\varepsilon}{m/2-1}$, where $\varepsilon > 0$ is a small number, and by another I/O operation phase of volume 1.

There are m I/O operations of volume $\frac{2}{m}$ posted at time 0. With a total bandwidth $B = 1$, it is impossible to complete more than $m/2$ of them by time 1. Because the m applications are not distinguishable at time 0, the adversary might force the scheduler to complete only I/O operations of applications of category B at time 1, and have no application of category A having completed its I/O operation by the end of the window. Proceeding with this scenario, only applications of category B may have executed some work at time 1. In fact, the most efficient scenario (which is illustrated on the right side of Figure 1) is to grant the full bandwidth to each application in category B one after the other, so that $m/2 - 1$ of them can complete their work phase by the end of the window; indeed, it is impossible for all $m/2$ applications in category B to terminate their work phase by $t = 1$, and the best, in order to maximize the work done, is to schedule the I/O operations without sharing. Finally, no application of category B can complete its second I/O transfer by $t = 1$. The efficiency at time $t = 1$ is therefore upper bounded as

$$\mathcal{E} = \frac{\sum_{i\in A\cup B} V_i^{(transferred)} + \sum_{i\in B} W_i^{(done)}}{m} \leq \frac{1}{m} + \frac{(m/2-1)\alpha}{m} \leq \frac{1+\varepsilon}{m}.$$

A strategy that would process the I/O operations of the jobs in category A without sharing is illustrated on the left side of Figure 1 and would reach an efficiency

$$\mathcal{E}' = \frac{\sum_{i=1}^{m/2} \frac{i}{m/2}}{m} = \frac{m/2+1}{2m} > \frac{m}{4}\mathcal{E},$$

for ε small enough. Therefore, there is no competitive ratio lower than $\frac{m}{4}$ for EFFICIENCY for any strategy.

Now, if we consider the UTILIZATION objective function, we get $u = \frac{\epsilon}{m}$ with the first scenario and $u' = \frac{\sum_{i=1}^{m/2-1} \frac{i}{m/2}}{m} = \frac{m-2}{4m}$ with the second scenario. Therefore, we can get a competitive ratio

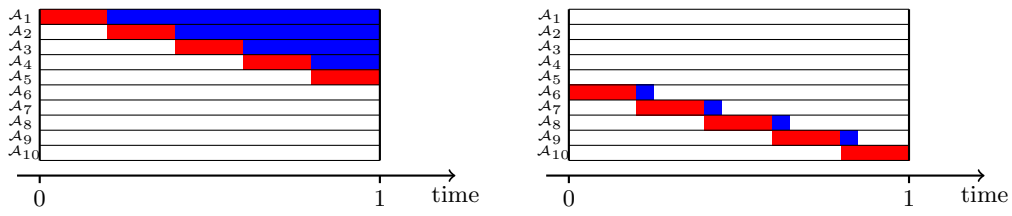


Figure 1: Illustration of Example 2 for the case $m = 10$ and $\epsilon = 1/5$. The best schedule for MINYIELD that completes the initial I/O of each application of category A (resp. of category B) is depicted on the left (resp. on the right).

arbitrarily large by choosing ε small enough.

Finally, for the MinYield objective, the best strategy is sharing I/O bandwidth equally among the $m = 2K$ applications, with gives a minimum yield of $\frac{1}{m}$. Any heuristic that serializes the I/O operations reaches a minimum yield of 0. For this example, this includes FCFS, GreedyYield, GreedyCom, LookAheadGreedyYield and Set-10 (if additionally we assume that all applications belong to the same I/O-set when starting execution at time $t = 0$).

6. Performance Evaluation

We first formally define the main parameter for the experiments, namely the I/O pressure, in Section 6.1. Then, we detail the simulations conducted with synthetic traces in Section 6.2 before discussing results for the APEX workloads in Section 6.3.

6.1. I/O Pressure

For a given a steady-state window $[T_{begin}, T_{end}]$ with m applications, we compute the volume V_i that each application $\mathcal{A}_i$ would be able to transfer if it was executed in dedicated mode throughout the window. The total I/O volume to transfer during the window is $V = \sum_{i=1}^{m} V_i$. The I/O pressure W is then

$$W = \frac{V}{B(T_{end} - T_{begin})}. \tag{11}$$

The I/O pressure W is the ratio of this total volume V over the maximum volume that could have been transferred during the window, assuming that it consists of a single block of data available at T_{begin}. Of course, if W exceeds 1, some transfers will necessarily be delayed. But even if W is lower than 1 but high, say 0.8, it is likely that I/O interferences and delays due to work phases will prevent to transfer the whole data volume V before the end of the steady-state window.

The I/O pressure W is a key parameter for the simulations: most bandwidth-strategies are expected to perform well when W is low, but we aim to assess how much their performance drops when W is high.

6.2. Synthetic Traces

6.2.1. Framework

The synthetic traces follow the methodology of [5] and consist of $m = 60$ applications, each of them being able to saturate the bandwidth (we have $b_i = B = 1$), with an approximate horizon of $h = 2,000,000$. For a given aimed pressure W^{GOAL}, each application $\mathcal{A}_i$ $(1 \leq i \leq m)$ is defined by the three parameters $(\mu_i, \sigma'_i, \nu_i)$: μ_i and σ'_i represents expectation and standard deviation and impact the length of the repetitions for each applications and ν_i determines how much the application differs from one iteration to another. More precisely,

- We generate an iteration duration ω_i for $\mathcal{A}_i$, which corresponds to the sum of a work phase and an I/O phase if the application was alone on the platform. This duration is generated using the two parameters μ_i and σ'_i: ω_i is drawn from the normal distribution $\mathcal{N}(\mu_i, \sigma'_i)$, truncated so that we consider only positive results.

- The number of iterations of application $\mathcal{A}_i$ is $n_i = \left\lceil \frac{h}{\omega_i} \right\rceil$ so its total completion time if it were alone on the platform is close to h.

- All applications are released at time T_{begin}: $\tau_i = T_{begin}$ for each application $\mathcal{A}_i$. In other words, all applications are *fresh* when entering the window and have the same yiled (equal to 0). To avoid having all applications synchronized, we add a work phase $w_i^{(0)}$ whose length is generated in $\mathcal{U}[0, \omega_i]$, so that application $\mathcal{A}_i$ effectively starts at time $w_i^{(0)}$. To simulate SET-10, we put all applications in the same I/O-set with highest priority initially, and hence process them in FCFS order at the beginning of the execution. After that each application has completed its first I/O operation, the duration of each iteration is updated on the fly, and applications get classified into different I/O-sets.

- Next, for each application, we fix the time spent on I/Os vs. on computing, so that the total pressure is around W^{GOAL}. This is done by drawing a value u_k uniformly at random in $\mathcal{U}[0, 1]$ for each application $\mathcal{A}_k$ ($1 \leq k \leq m$), and then by defining the fraction of I/O for application $\mathcal{A}_i$ as $\phi_i = \frac{u_i W^{GOAL}}{\sum_{k=1}^{m} u_k}$. This guarantees that the I/O pressure W is around W^{GOAL}. Indeed, ϕ_i allows us to define the average duration of computing phases $t_{i,cpu} = (1 - \phi_i)\omega_i$ and the average volume of I/O phases: $t_{i,io} = \phi_i \omega_i$. Thus

$$W \approx \frac{\sum_{i=1}^{m} t_{i,io} n_i}{B(T_{end} - T_{begin})} = \frac{\sum_{i=1}^{m} \phi_i \omega_i n_i}{B(T_{end} - T_{begin})} \approx \frac{\sum_{i=1}^{m} \phi_i T_{end}}{B(T_{end} - T_{begin})} = \frac{T_{end} W^{GOAL}}{T_{end}} = W^{GOAL}.$$

  We point out that we cannot enforce exactly $W = W^{GOAL}$ due to the randomness in the generation of instances.

- Finally, for each application $\mathcal{A}_i$, we consider a noise parameter ν_i to generate iterations of different lengths. For all $j \leq n_i$, we draw two variables $\gamma_{cpu}^{(j)}$ and $\gamma_{io}^{(j)}$ from a uniform distribution $\mathcal{U}[-\nu_i, \nu_i]$ and let $w_i^{(j)} = (1 + \gamma_{cpu}^{(j)}) t_{i,cpu}$ and $v_i^{(j)} = (1 + \gamma_{io}^{(j)}) t_{i,io}$.

6.2.2. Results for Synthetic Traces

Still following the methodology of [5], the experiments are conducted by varying four different key parameters for the 60 applications. For the application length, we consider 20 applications of medium size, and then a proportion of smaller and larger applications, as determined by the parameter n_{small} (number of small applications). The standard deviation is dictated by parameter σ, the noise is set to ν, and the pressure is W^{GOAL}. Overall, the applications are as follows:

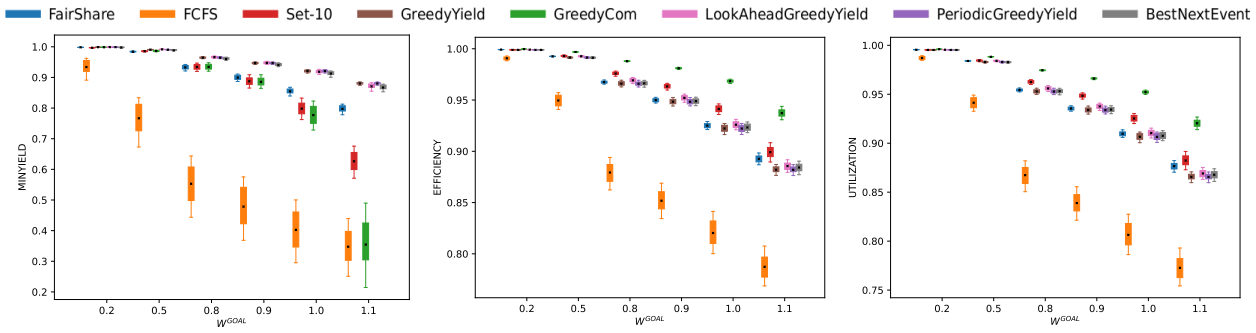
- n_{small} *small* applications with parameters ($\mu = 1\,000, \sigma' = \mu\sigma, \nu$);
- 20 *medium* applications with parameters ($\mu = 10\,000, \sigma' = \mu\sigma, \nu$);
- $40 - n_{small}$ *big* applications with parameters ($\mu = 100\,000, \sigma' = \mu\sigma, \nu$).

The time window is defined as $[T_{begin} = 0, T_{end} \approx h]$, where T_{end} is the smallest time required to complete an application when it is running alone on the platform. Each application is generated in such a way that T_{end} is approximately equal to $h = 2,000,000$. For each set of experiments, we study the results of all the heuristics for the three objectives (MinYield, Efficiency, Utilization).

Finally, for each set of parameters, we generate $K = 200$ instances on which we test all the heuristics presented in Section 4 (including the reference heuristics FairShare, FCFS and Set-10). In the following sections, we vary the parameters one by one and present the results on different figures. Each set of instances is represented by a boxplot of a color associated with the studied heuristic. In these boxplots, the 25th and 75th percentiles of the K instances delimit the box, and the 10th and 90th percentiles are at the end of the whiskers. Finally, the boxplots are connected by a line passing through their means.

Impact of the target I/O pressure (W^{GOAL}). We first set $\nu = \sigma = 0.5$, and $n_{small} = 20$ (20 applications of each category), and we present the results of the experiments for all values of the aimed I/O pressure $W^{GOAL} \in [0.2, 0.5, 0.8, 0.9, 1.0, 1.1]$ on Figure 2. As soon as the pressure increases, we see that the state-of-the-art strategies FairShare, FCFS, and Set-10, along with the new GreedyCom, fail to keep a minimum yield close to 1. The other newly proposed strategies, which all focus on the yield, successfully maintain a very high minimum yield, and achieve a similar performance which very slowly degrades when the I/O pressure increases. LookAheadGreedyYield and BestNextEvent achieve a very slightly worse performance than GreedyYield and PeriodicGreedyYield. This counter-intuitive result may be explained by the fact that an I/O phase is always followed by a computation phase during which the progress rate of an application is perfect. Hence, what heuristics GreedyYield and PeriodicGreedyYield may loose in terms of application yield during an I/O phase may be made up later on in the subsequent computation phase. The fact that GreedyYield, PeriodicGreedyYield and LookAheadGreedyYield achieve a minimum yield no worse than that of BestNextEvent, a costly strategy which exhaustively looks for the best solution, strongly validates these three low-cost strategies.

The classical FCFS strategy also has very poor results in terms of efficiency and utilization, while GreedyCom is actually the best for these objective functions since it will complete short I/Os first, with a risk of starvation for applications with long I/Os. This explains the poor performance of GreedyCom for MinYield for higher values of W^{GOAL}. The yield-based strategies tend to balance the yield of all applications, which optimizes the MinYield. However, not allowing any application to starve requires prioritizing some long I/Os that saturate the bandwidth, which



(a) MinYield (b) Efficiency (c) Utilization

Figure 2: Impact of the aimed I/O pressure (W^{GOAL}).

can negatively impact both Efficiency and Utilization. The underlying tradeoff explains why heuristics achieving a significantly better performance than FairShare for the MinYield usually achieve slightly worse performance than FairShare for Efficiency and Utilization. However, the performance degradation in terms of either Efficiency or Utilization is quite small (under 5%) and only happens for the largest value of I/O pressure. For all but the largest value of W^{GOAL}, LookAheadGreedyYield even achieves better Efficiency and Utilization than FairShare.

Impact of iteration size (ω) and I/O fraction (ϕ). We investigate the impact of ω and ϕ on the yield of each individual application for a fixed set of parameters: $\nu = 0.5$, $\sigma = 0.5$, $n_{small} = 20$, and $W^{GOAL} \in \{0.5, 0.8, 1.1\}$. More precisely, for each experiment $\mathcal{E}_k$, we define two permutations on the index set $\{1, 2, \ldots, 60\}$ to sort the applications by increasing values of ω (permutation π^k_ω) or of ϕ (permutation π^k_ϕ). For each value of W^{GOAL}, we compute the average yield of applications in each position i under each permutation, denoted as $\overline{y}^{(\omega)}_i$ (resp. $\overline{y}^{(\phi)}_i$), for $i \in \{1, 2, \ldots, 60\}$. It is computed as follows:

$$\overline{y}^{(\omega)}_i = \frac{1}{K}\sum_{k=1}^{K} y_{\pi^k_\omega(i)} \quad \text{and} \quad \overline{y}^{(\phi)}_i = \frac{1}{K}\sum_{k=1}^{K} y_{\pi^k_\phi(i)}.$$

We then plot the value of $\overline{y}^{(\omega)}_i$ for i varying in $[1, 60]$ on Figure 3 and the value of $\overline{y}^{(\phi)}_i$ on Figure 4. Therefore, the leftmost point on Figure 3 (respectively, on Figure 4) corresponds to the average yield of the application with the smallest value of ω (resp., of ϕ), while the rightmost point corresponds to the average yield of the application with the largest value of ω (resp., of ϕ).

Impact of iteration size (ω) . On Figure 3, we observe that the differences between the heuristics are more pronounced when W^{GOAL} increases. This is because the increase in W^{GOAL} increases the I/O interferences. For this reason, we now focus on the figure on the right (case $W^{GOAL} = 1.1$). First, we can observe that the variation of ω has little impact on the yields achieved by

FairShare, GreedyYield, LookAheadGreedyYield, PeriodicGreedyYield, and BestNextEvent. GreedyYield, LookAheadGreedyYield PeriodicGreedyYield, and BestNextEvent tend to balance the yield of the different applications, resulting in a constant function. For FairShare, there seems to be no correlation between ω and the yield. This can be explained by the fact that there is no correlation between ω and ϕ in the generated instances.

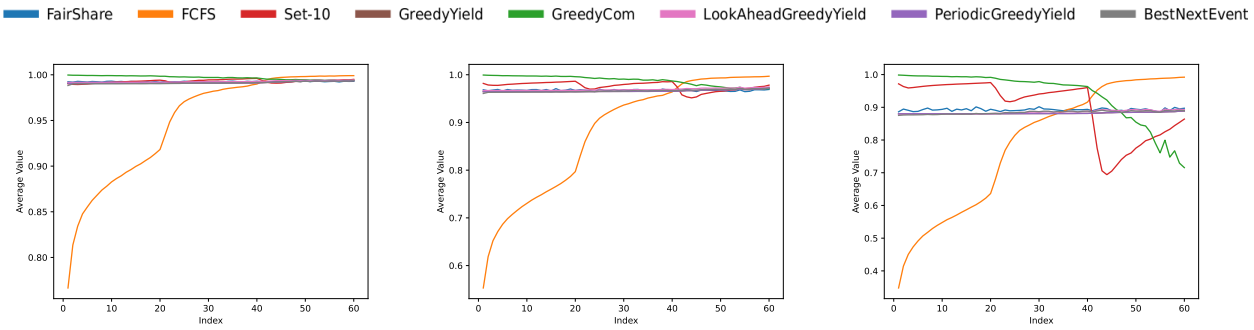
This figure is more enlightening for the other heuristics. First, the yield seems to be positively correlated with ω for FCFS. This is because ϕ is not correlated with ω. Hence, a small value of ω corresponds to short I/O phases. For FCFS, the longest I/Os will saturate the bandwidth more often. Indeed, a single application can saturate the bandwidth, so when a long I/O is executed, all the other applications wanting to performa some I/O are stopped. For an application with a small I/O, the waiting time may be very long compared to its size, and the next waiting phase may also come quickly if some long I/O is posted between two of its I/O phases. Therefore, applications with short I/Os, i.e., a small value of ω, will spend a large part of their time waiting.

We observe the opposite behavior for the GreedyCom strategy since, this time, small I/Os are given priority. As previously mentioned, a low value of ω induces short I/Os; hence, the yield decreases with ω.

Finally, this figure perfectly illustrates the behavior of Set-10. Indeed, we can clearly distinguish the three steps corresponding to the three priority categories in these synthetic traces. Moreover, within each of these steps, we see that the yield increases with ω, just like for FCFS. This is because Set-10 behaves like FCFS inside each of these categories.

Impact of I/O fraction (ϕ) . Figure 4 may appear a bit more cluttered, but illustrates some interesting behaviors. Once again, we only focus on the figure on the right, that is, on the case $W^{GOAL} = 1.1$. First, we can see a difference between GreedyYield (hidden under PeriodicGreedyYield) and LookAheadGreedyYield for small values of ϕ, showing that the best immediate choice is not always the best choice in the long term. We can also see that BestNextEvent favors applications with smaller I/Os so that the next event arrives as soon as possible and the yield do not have the time to significantly decrease (because of I/O interference). FCFS is erratic because an application with a large ω but a small ϕ will still have larger I/O volumes per phase than an application with a small ω but a large ϕ. The same argument also explains the non-monotonic behavior of GreedyCom when ϕ becomes large. The only heuristic that is strongly (negatively) correlated with ϕ is FairShare. Indeed, the larger ϕ, the longer the application will spend performing I/Os, and the lower the yield will be, whereas in a working phase, the instantaneous yield is 1. The linear shape of this curve is related to the uniform distribution of ϕ.

Impact of the other parameters. In Section B.1 of [3], we report on experiments detailing the impact of the other parameters, namely the number of small applications (n_{small}), the standard deviation



(a) $W^{GOAL} = 0.5$ (b) $W^{GOAL} = 0.8$ (c) $W^{GOAL} = 1.1$

Figure 3: Yields sorted by iteration size (ω).

(a) $W^{GOAL} = 0.5$ (b) $W^{GOAL} = 0.8$ (c) $W^{GOAL} = 1.1$

Figure 4: Yields sorted by I/O fraction (ϕ).

(σ), and the noise (ν). The number of small applications (n_{small}) and the noise (ν) do not impact the performance of the strategies. High values of the standard deviation (σ) only impacts the MinYield achieved by FCFS and Set-10, and it impacts them negatively.

6.2.3. Synthesis of the Evaluation on Synthetic Scenarios

For the MinYield objective, the greedy strategies GreedyYield, LookAheadGreedyYield and PeriodicGreedyYield achieve comparable performance, and much better performance than the competitors FCFS, FairShare and Set-10. Furthermore, we stress that the complicated strategy BestNextEvent does *not* turn out to be superior to the simpler ones, which is good news: GreedyYield, LookAheadGreedyYield and PeriodicGreedyYield are all simple to implement and use. Finally, for the Efficiency and Utilization objectives, GreedyCom is the best, FCFS is the worst, and the other strategies achieve close performance in between.

6.3. Evaluation on APEX workloads

6.3.1. Apex Traces [19]

We use the workload and platform described in [19] to evaluate the bandwidth-sharing strategies on realistic scenarios. The table in Figure 3 of [19] describes two very different workloads: the NERSC workload and the TRILAB workload. The NERSC workload contains a large number of small applications (e.g., a single pipeline of the SkySurvey workflow runs over 24 cores for 4 hours, but the set of SkySurvey workflows represents 12% of the overall core-hours used by the workload on the machine), some large applications (GTS spans over 16,512 cores, or 1/8 of the platform, for 48 hours), and some very long running applications (CESM applications run for 10 days over 8,000 cores). The TRILAB workload contains a more homogeneous set of applications (4096 to 32768 cores), and all applications run for a significantly longer time (64 hours for the smallest duration, and up to 12 days for the longest). From this table, we take the application walltime, its number of cores, and the data information to build a possible schedule on the target machine. The table reports how much input, output, and checkpoint data each application uses. The trace does not provide fine-grain information on how the data is consumed or produced. To simulate the schedules, we assume that all inputs happen at the beginning of the application, which then does periodic checkpoints, and eventually outputs all its output data just before its completion. As is often the practice in HPC centers [11], we use a fixed period of 1 hour for the checkpoint interval.

Based on this information, we generate machine schedules using the first-fit strategy. We consider independently NERSC or TRILAB workloads and, for a given workload, we randomly pick applications from this workload, and place them on the schedule, until two conditions are met: 1) the schedule follows the application workload distribution described in the APEX table, and 2) the schedule represents at least 3 months of machine use. For each target machine considered (see below), we generate 100 schedules for the TRILAB workload and 100 schedules for the NERSC workload. In each schedule, we then find the 20 longest windows during which no application is joining or leaving the machine, to fit the analysis conditions with steady-state windows described in Sections 3 and 4. We then assume that each application joined the system at the window start ($\tau_i = T_{begin}$ for each application $\mathcal{A}_i$).

On the Celio system[2], both the NERSC and TRILAB workloads represent a small I/O pressure (about 0.15 on average). However, I/O pressure is a metric that tends to increase as we consider larger platforms and newer systems. In [17], the authors look at the architectural trends and system balance of the top 500 supercomputers. The Parallel File System (PFS) bandwidth is studied for systems that existed between 2009 and 2018. The authors compare the PFS bandwidth with the aggregated memory bandwidth. The different systems have a ratio of aggregated memory

[2]Celio is the platform used for the NERSC and TRILAB workloads [19].

bandwidth by PFS bandwidth between 50 and 17000, with an average of 13,353, without a clear trend in time.

The ratio of aggregated memory bandwidth per computing performance, however, shows a clear diminishing trend. As an example, this ratio decreased by a factor 9 between the No. 1 machine in 2009 and the No. 1 machine in 2018. As a consequence, the ratio between the PFS bandwidth and the computing performance also has a clearly decreasing trend. In [5], the authors note that this ratio has decreased by a factor 24.8 over 20 years. Over long periods of time, it looks like the trend of the PFS bandwidth progresses more slowy than the computing power by a linear factor.

To study how the different algorithms behave with higher values of the I/O pressure, we have considered a set of target machines that are scaled versions of the Celio system. Let C_c and C_{bw} be respectively the total number of cores and system bandwidth of Celio, and let t represent the passing time. The system M_t has $C_c \times 2^{\frac{t}{\alpha}}$ cores (representing a doubling of computing power every α time units, in accordance of the observed progression in [17]), and $C_{bw} \times 2^{\frac{t}{\alpha}}/t$ system bandwidth, following the observation above. M_y, $y > 0$, represents machines built y time units later than the Celio machine, and for each target machine, we compute the schedules and corresponding windows for both workloads. We thus obtain a range of I/O pressures between 0.15 and 1.4, and simulate the behavior of the bandwidth-sharing strategies in each window, to evaluate our metrics as a function of the I/O pressure.

6.3.2. MinYield *of* FairShare *on APEX Scenarios*

We use the FairShare strategy as the basis for our evaluation, so we study first how FairShare behaves as a function of the I/O pressure. Figure 5 presents the MinYield obtained by the FairShare strategy within each of the 2,000 windows obtained during the simulation, as a function of the I/O pressure observed inside each window. The color of points denote on which target platform this I/O pressure and MinYield were observed.

On the NERSC workload, we see that the MinYield stays above 0.8 when the I/O pressure is low (0.4), and the distribution tends to decrease as the I/O pressure increases, with some scenarios that obtain a MinYield under 0.5 when the I/O pressure is 1, and the number of runs that have a low MinYield continue to increase as the I/O pressure continues to increase. The machine scale has some impact on the I/O pressure inside the various windows, but most of the runs present a relatively low I/O pressure, and a MinYield of 1 for FairShare is observed for some runs with high I/O pressures (up to 1.4). We conjecture that this is a consequence of the relatively small windows for the NERSC workload. Small scale, short lived applications constitute the bulk of many windows of the NERSC workload. These applications only do I/O at the beginning and end of their execution, limiting the opportunities for interferences. These I/O are also small (even relative to the short duration of the application), so when they interfere (which is unavoidable when the I/O pressure is higher than 1), they still reduce the MinYield by only a fraction. Only on

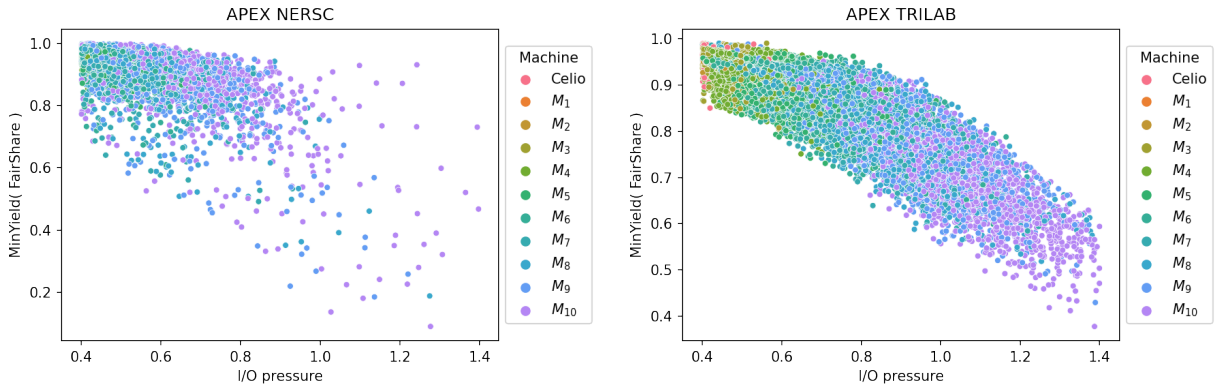


Figure 5: MinYield of the FairShare strategy for the NERSC and TRILAB workloads, as a function of the I/O pressure and of the target platform.

windows that feature the few larger applications and those with costly checkpoints can we observe a measurable decrease of MinYield for FairShare.

This conjecture is corroborated by the measurement of the TRILAB workload. The same trends for this workload are more clearly marked: the larger the machine, the higher the I/O pressure, and the higher the I/O pressure, the lower MinYield for FairShare. Although there are no scenarios where MinYield goes under 0.4, there are also no scenarios with a MinYield close to 1 when the I/O pressure is above 1. The windows are much longer in the TRILAB experiments, and applications have time to checkpoint regularly during these windows. As a consequence, interferences between applications that have overlapping I/O create slowdowns that reduce the MinYield. We note from the left graph of Figure 5 that no NERSC scenario on the Celio platform obtains an I/O pressure of at least 0.5, while some scenarios of the TRILAB workload can saturate the I/O bandwidth. We explore the characteristics of the windows duration, size and utilization in more details in Section B.2 of [3].

6.3.3. MinYield *of All Strategies on APEX NERSC Scenarios*

Figure 6 presents all the scenarios used in Figure 5 for the NERSC workload, and considers the MinYield of each strategy as a ratio of MinYield for FairShare, with an independent graph per strategy. As a reference, the MinYield of FairShare is also presented in a different color. A value of 1 of the ratios means that the target strategy obtains the exact same MinYield as FairShare for the scenario, while a value higher than 1 means that a higher MinYield than FairShare is obtained for this scenario, and a value lower than 1 that on this scenario, the strategy obtains a lower MinYield than FairShare.

There are three classes of graphs in this figure. The strategy GreedyCom presents on average a ratio distributed approximately uniformly between 0.9 and 1.1. This means that this strategy fails to reliably improve the MinYield in at least half of the scenarios. The second set of graphs

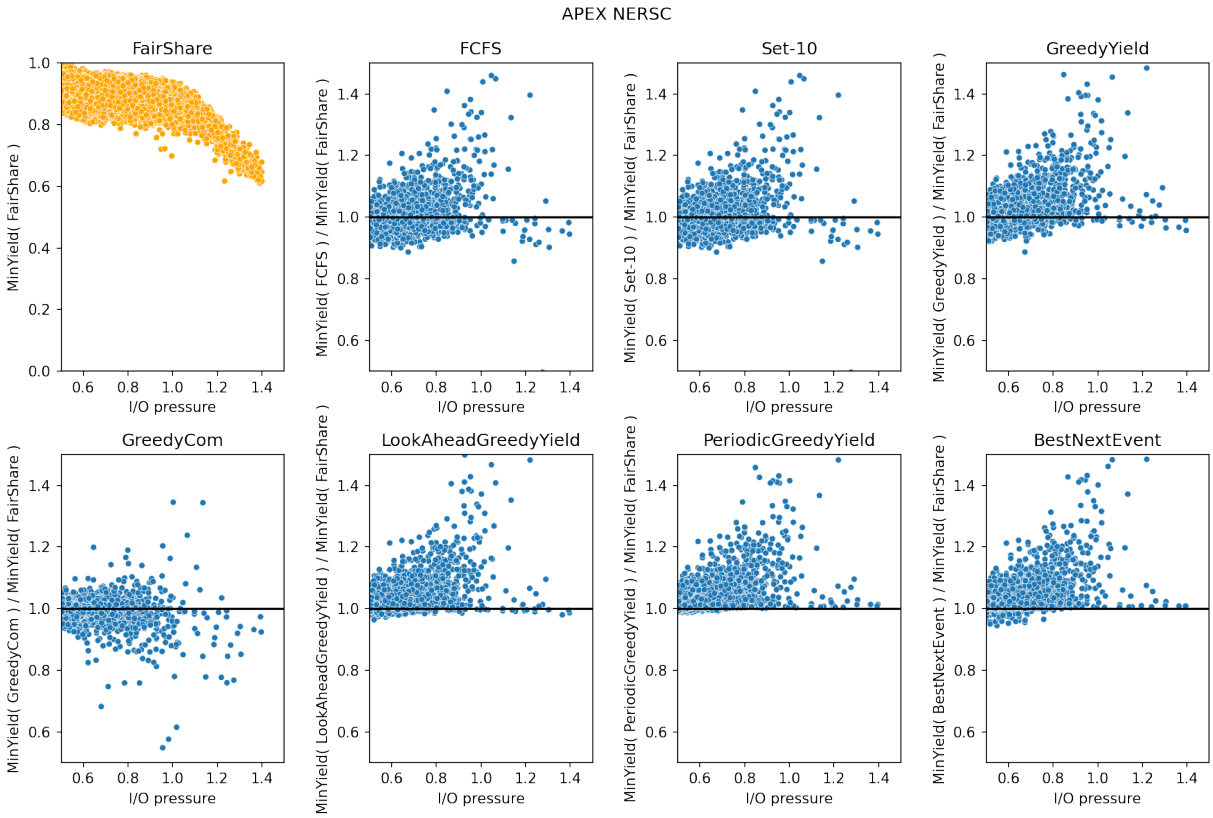


Figure 6: MinYield of all strategies, as a ratio of the MinYield with the FairShare strategy for the same experiment, for the NERSC workload, as a function of the I/O Pressure.

show that FCFS, Set-10, and GreedyYield have a non-negligible set of scenarios where they decrease MinYield compared to FairShare, but as the I/O pressure increases they tend to behave slightly better than MinYield on average (with still a risk of significant performance degradation for all I/O pressures). When they experience gains, the gains are more pronounced for high I/O pressures. Set-10 and FCFS behave strictly identically over the NERSC workload, and this is because the NERSC workload featuring very small windows with typically at most one phase for many applications, Set-10 does not have time to learn the phases, and thus puts all the applications in the same set, behaving as FCFS.

The third set of graphs include LookAheadGreedyYield, PeriodicGreedyYield, and BestNextEvent. These three strategies have a very high probability of increasing MinYield compared to FairShare, and that performance increase tends to be higher as the I/O pressure increases. BestNextEvent is the strategy of this set that features the highest risk of decreasing MinYield (although the decrease is limited to 95% of the MinYield of FairShare in the worst scenario), while PeriodicGreedyYield has almost no scenario with a MinYield lower than FairShare.

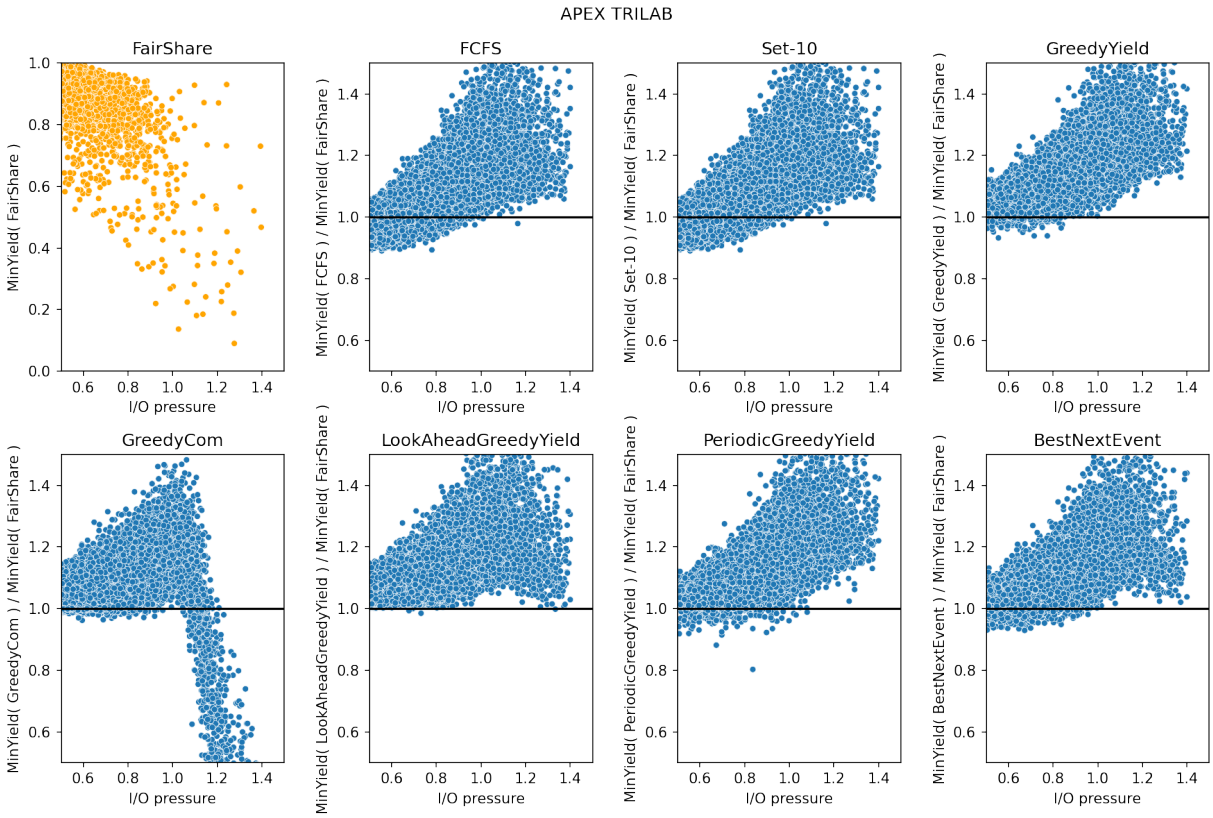


Figure 7: MinYield of all strategies, as a ratio of the MinYield with the FairShare strategy for the same experiment, for the TRILAB workload, as a function of the I/O pressure.

6.3.4. MinYield *of All Strategies on APEX TRILAB Scenarios*

Figure 7 presents the same evaluation, for the TRILAB workload (relative to the experiments shown in the right graph of Figure 5). With this workload, the ratio of MinYield behaves differently than with the NERSC workload. Overall, all strategies tend to behave better (with relatively less scenarios presenting a ratio lower than 1), and the gains over FairShare are on average higher for all strategies at low I/O pressure and for most strategies at high I/O pressure.

GreedyCom presents better behaviors than over the NERSC workload, with only a few scenarios underperforming FairShare, until the I/O pressure reaches a ratio of 1, i.e., until the system reaches saturation of the communication system. Then, the performance of GreedyCom quickly drops dramatically, with eventually all scenarios obtaining a lower MinYield than FairShare.

FCFS, Set-10 and GreedyYield continue to behave similarly, but the trend is more clear, with a significant risk of MinYield degradation for low I/O pressures, but significant gains as the I/O pressure, and consistent gains at I/O saturation (when the I/O pressure is higher than 1). FCFS and Set-10 continue to behave identically. However, this time this is not due to a lack of time to learn the periodicity of the applications: in the TRILAB workload, each application has a

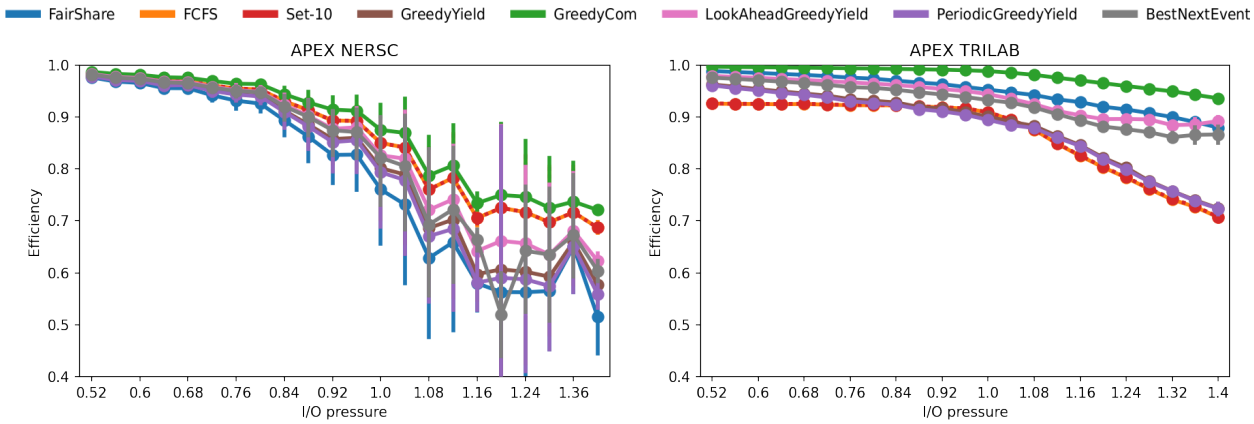


Figure 8: Efficiency of all strategies for the NERSC and TRILAB workloads, as a function of the I/O pressure.

minimum of 5 phases during the window, which is long enough to converge on the phase duration and categorize the application in the appropriate set. Because all applications checkpoint with the same approximate checkpointing period, only the duration of the checkpoint operation can define different categories of phases. The checkpoint duration of the different applications can vary by an order of magnitude or more in the TRILAB workload, but the duration of the slowest checkpointing operation still remains small compared to the checkpointing period. As a consequence, the Set-10 strategy tends to put all applications in the same category, and falls back to applying the FCFS strategy.

Among the three wining strategies for the NERSC workload, LookAheadGreedyYield, PeriodicGreedyYield and BestNextEvent, the trends observed for the NERSC workload are enforced: until the system reaches I/O saturation, PeriodicGreedyYield and BestNextEvent feature a few scenarios where MinYield can be slightly decreased compared to FairShare, but in most scenarios (and in almost all scenarios for LookAheadGreedyYield), these strategies improve MinYield, and that improvement becomes higher as the I/O pressure increases. Contrarily to NERSC traces, GreedyYield performs similarly to PeriodicGreedyYield and BestNextEvent on the TRILAB traces.

On these longer windows, the I/O pressure seems to have a more significant impact than on the smaller windows of the NERSC workload, and as the I/O pressure increases, the gains relative to FairShare tend to increase (see Section B.2 of [3] for detailed results). When the I/O pressure is higher than 1, interference is unavoidable, and the I/O scheduling strategy becomes critical to the performance of applications. Naive strategies, or strategies that are not well suited for the irregular nature of the applications present in these workloads, have then a higher risk of taking the wrong decision and performing worse than FairShare.

6.3.5. Efficiency *of All Strategies on APEX Scenarios*

Figure 8 presents the mean and standard deviation of the Efficiency metric for each strategy as a function of the I/O pressure. To synthetize these graphs, we split the I/O pressure domain in 25 intervals and compute the mean Efficiency value and its standard deviation for all scenarios with an I/O pressure that falls in this interval. The point is presented at the middle of the interval.

The NERSC and TRILAB workloads present both some commonalities and some significantly different features. In the NERSC workload, Efficiency quickly drops as the I/O pressure increases for all strategies, while each strategy seems to hold its Efficiency until the system reaches saturation (I/O pressure of 1) in the TRILAB workload. Once the I/O pressure is above 1, Efficiency drops with the I/O pressure for both workloads, but this drop is more pronounced, and becomes chaotic, for the NERSC workload, while the Efficiency with the TRILAB workload remains stable and supports higher I/O pressures for all strategies.

Efficiency measures the sum of actual progress of all applications throughout the window. As NERSC has on average much smaller windows than TRILAB, the effect of a few bad I/O schedule decisions can be much more impactful on a small window than on a large one. This explains the chaotic Efficiency measurement on the NERSC workload compared to TRILAB.

TRILAB is also a workload on which it is easier, for all strategies, to maintain a high Efficiency compared to NERSC, because the windows feature a lower number of long and large-scale applications, where the I/O is close to periodic per application (mostly driven by fixed-period checkpointing), allowing many opportunities to overlap I/O operations and computation. However, at high I/O pressure, we observe three groups of strategies on the TRILAB workload: GreedyCom, which targets a balance of I/O operation progress, remains the most efficient; FairShare, LookAheadGreedyYield and BestNextEvent provide a similar Efficiency, slightly under GreedyCom; and in the third group, Set-10, FCFS (hidden by Set-10 in the figure), GreedyYield (hidden by PeriodicGreedyYield in the figure), and PeriodicGreedyYield present the worst Efficiency. As the I/O pressure is above 1, contentions are unavoidable, and the strategies that pursue too eagerly an optimization of MinYield fail at providing a good Efficiency. FCFS and Set-10 take I/O scheduling decisions that are detrimental to Efficiency because the I/O that is favored is arbitrary.

In the NERSC workload, the metric is too chaotic at high I/O pressure to define a clear order, but GreedyCom remains the strategy with the highest Efficiency, which is expected as GreedyCom targets this metric.

6.3.6. Utilization *of All Strategies on APEX Scenarios*

Figure 9 presents the mean and standard deviation of the Utilization metric for each strategy as a function of the I/O pressure. We used the same binning approach as for Figure 8 to present trends from individual scenarios.

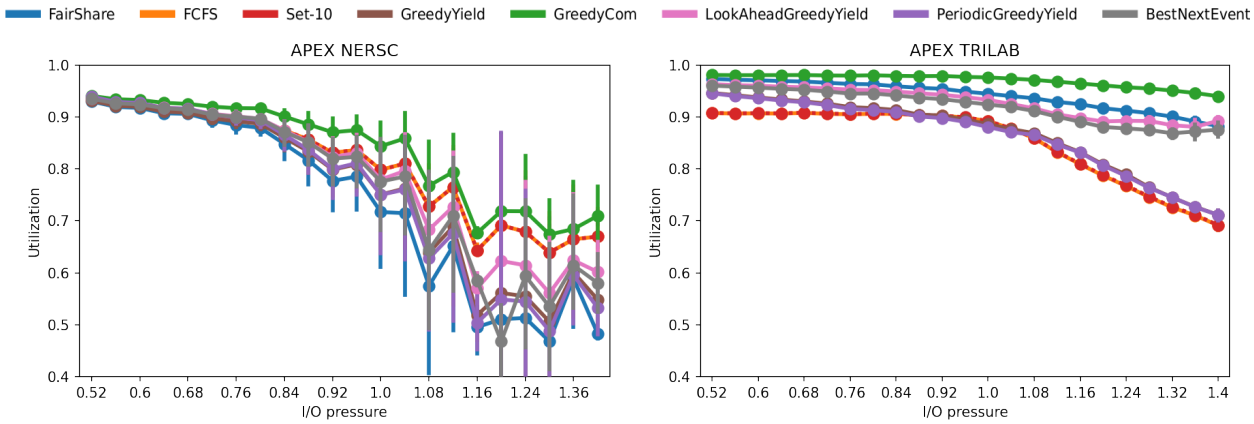


Figure 9: UTILIZATION of all strategies for the NERSC and TRILAB workloads, as a function of the I/O Pressure.

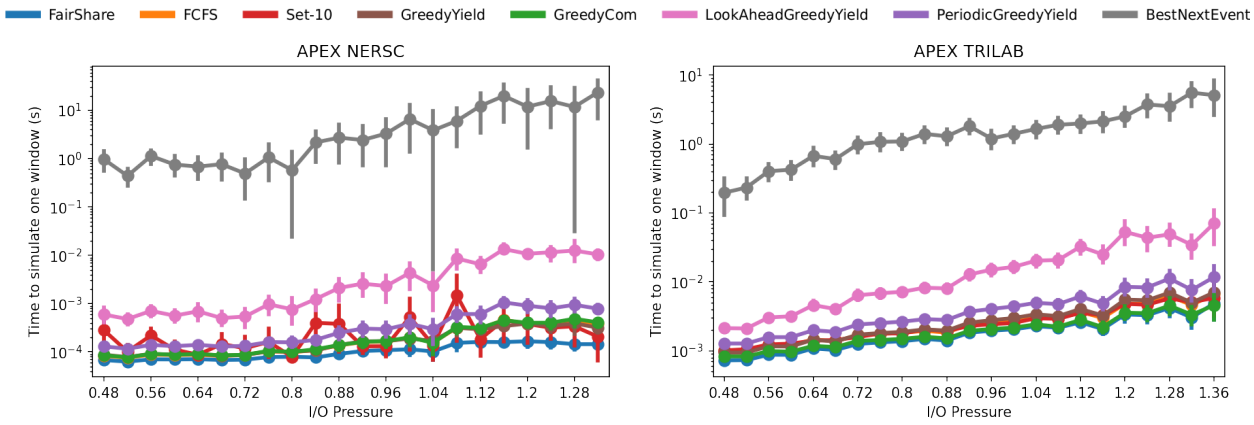


Figure 10: Simulation time of all strategies for the NERSC and TRILAB workloads, as a function of the I/O Pressure.

UTILIZATION is overall lower in the NERSC workload than in the TRILAB workload. This is corroborated by the window characteristics detailed in Section B.2 of [3]: windows in the NERSC workload have on average a lower UTILIZATION than for the TRILAB workload, even without considering I/O interferences.

As the I/O pressure increases and in the saturated domain in particular, I/O interferences reduce even more UTILIZATION, for all strategies and in all scenarios. GREEDYCOM, which targets a balance of I/O operation progress, shines with this metric as well as for EFFICIENCY, at the cost of a worst MINYIELD as illustrated in Figures 6 and 7. On these practical scenarios, EFFICIENCY and UTILIZATION seem to behave very similarly.

6.3.7. Computation Time of All Strategies on APEX Scenarios

Last, we look at the computation time of the different strategies. Each strategy decides to take a scheduling decision at different times, and each scheduling decision impacts the order of

events and when the next scheduling decision will happen. To compare the computation time of the different strategies in a practical setup, we have thus chosen to measure the entire simulation time of a given window, for a given strategy. This time includes the simulation, but also, for each scheduling event, the cost of computing the decision, as an implementation of the strategy would have to do.

The mean and standard deviation of the time to simulate each of the windows is presented in Figure 10. We used the same binning approach as for Figures 8 and 9, in order to present trends. As the different strategies exhibit very different simulation computation times, the time axis is using a logarithmic scale.

BestNextEvent is the only strategy that requires significant computation time, with a few seconds (and never more than 60 seconds) needed to simulate an entire window for both the NERSC and TRILAB workloads. The second highest demanding strategy, LookAheadGreedyYield, only requires 10s of milliseconds to simulate an entire window, and all the other strategies are yet an order of magnitude faster.

Although BestNextEvent is the most demanding strategy in terms of computational complexity, its runtime remains small enough to be considered in practice. The PeriodicGreedyYield strategy, which needs to re-compute regularly the entire schedule, can be called with a very small period (seconds to milliseconds), as its computational demand on realistic scenarios is achieved in a fraction of this time.

6.3.8. Synthesis of the Evaluation on APEX Scenarios

Overall, LookAheadGreedyYield is the strategy that shows the best performance for the MinYield metric on the most variety of scenarios, closely followed by PeriodicGreedyYield and BestNextEvent. PeriodicGreedyYield requires to re-compute goals at a higher frequency, namely twice the frequency of the other greedy strategies with our choice for the periodicity of external events; but LookAheadGreedyYield remains more costly, because each decision requires a set of goal computations, one per active application, and BestNextEvent, with its exhaustive search, is far more computationally demanding. GreedyCom is a strategy that would perform the best on the Utilization and Efficiency metrics, and its MinYield remains reasonable on the TRILAB workload, as long as the I/O pressure is not saturated, but it is a risky choice for the NERSC workload.

7. Conclusion

This work has revisited I/O bandwidth-sharing strategies for concurrent applications. Our main contributions are two-fold. On the theoretical side, we have provided the first competitive ratios for such strategies, owing to a rigorous framework based upon steady-state windows. These

competitive ratios are mostly negative. In particular, the lower bound for MinYield is as high as the (order of) number of applications for all strategies except PeriodicGreedyYield. These results bring new insights on the hardness of the problem, and lay the foundations for the study of its complexity. On the practical side, we have introduced several new greedy heuristics and have compared them to well-established strategies such as FCFS, FairShare and Set-10. We have used a comprehensive set of experiments, some based upon synthetic traces and some based upon an extended version of APEX traces. In both cases, the well-established strategies perform worse, and often much worse, than the new heuristics. As a global conclusion, although there is no absolute winner for all scenarios and objectives, we recommend using LookAheadGreedyYield, which achieves an excellent performance for MinYield on all scenarios, achieves better Efficiency and Utilization than FairShare for the NERSC workload and comparable ones for the TRILAB and synthetic workloads. LookAheadGreedyYield requires knowing the volume of an I/O operation when it is posted. If such an information is not available, one can use PeriodicGreedyYield: it achieves very good MinYield on all scenarios, achieves better Efficiency and Utilization than FairShare for the NERSC workload, comparable ones for the synthetic workload, but worse ones for the TRILAB workload.

We acknowledge that the results of this paper have been obtained by simulation, and that the performance of the new strategies, in particular LookAheadGreedyYield and PeriodicGreedyYield, should be assessed through a deployment on large-scale platforms. Future work includes gaining access to ThemisIO and compare our new bandwidth-sharing strategies with those of ThemisIO. Another direction is to extend these strategies to account for users, groups, and systems with several I/O servers.

Acknowledgements. We would like to thank the editor and the reviewers for their comments and suggestions, which greatly helped improve the final version of the paper.

References

[1] Guillaume Aupy, Ana Gainaru, and Valentin Le Fèvre. Periodic I/O scheduling for supercomputers. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *PMBS workshop*, volume 10724 of *Lecture Notes in Computer Science*, pages 44–66. Springer, 2017.

[2] Guillaume Aupy, Ana Gainaru, and Valentin Le Fèvre. I/O scheduling strategy for periodic applications. *ACM Trans. Parallel Comput (TOPC)*, 6(2):1–26, 2019.

[3] Anne Benoit, Thomas Herault, Lucas Perotin, Yves Robert, and Frédéric Vivien. Revisiting I/O bandwidth-sharing strategies for HPC applications. Research report RR-9502, INRIA, 2023. https://inria.hal.science/hal-04038011.

[4] Jean Luca Bez, Suren Byna, and Shadi Ibrahim. I/O access patterns in HPC applications: A 360-degree survey. *ACM Computing Surveys*, jul 2023.

[5] Francieli Boito, Guillaume Pallez, Luan Teylo, and Nicolas Vidal. IO-Sets: Simple and Efficient Approaches for I/O Bandwidth Management. *IEEE Transactions on Parallel and Distributed Systems*, 34(10):2783–2796, 2023.

[6] Peter Brucker, Sigrid Knust, and Ceyda Oğuz. Scheduling chains with identical jobs and constant delays on a single machine. *Mathematical Methods of Operations Research*, 63(1):63–75, 2006.

[7] Jesús Carretero, Emmanuel Jeannot, Guillaume Pallez, David E. Singh, and Nicolas Vidal. Mapping and scheduling HPC applications for optimizing I/O. In *ICS: International Conference on Supercomputing*. ACM, 2020.

[8] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination. In *Proc. 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, page 155–164. IEEE Computer Society, 2014.

[9] K Ecker and M Tanaś. Complexity of scheduling of coupled tasks with chains precedence constraints and any constant length of gap. *Journal of the Operational Research Society*, 63(4):524–529, 2012.

[10] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the I/O of HPC applications under congestion. In *IPDPS'2015, the 29th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2015.

[11] Thomas Herault and Yves Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks. Springer Verlag, 2015.

[12] Yusheng Hua, Xuanhua Shi, Hai Jin, Wei Liu, Yan Jiang, Yong Chen, and Ligang He. Software-defined QoS for I/O in exascale computing. *CCF Trans. High Perform. Comput.*, 1(1):49–59, 2019.

[13] Florin Isaila, Jesus Carretero, and Rob Ross. CLARISSE: A Middleware for Data-Staging Coordination and Control on Large-Scale HPC Platforms. In *16th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 346–355. IEEE, 2016.

[14] Emmanuel Jeannot, Guillaume Pallez, and Nicolas Vidal. Scheduling periodic I/O access with bi-colored chains: models and algorithms. *J. Scheduling*, 24(5):469–481, 2021.

[15] Qiao Kang, Sunwoo Lee, Kaiyuan Hou, Robert Ross, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. Improving MPI Collective I/O for High Volume Non-Contiguous Requests With Intra-Node Aggregation. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2682–2695, 2020.

[16] Ed Karrels, Lei Huang, Yuhong Kan, Ishank Arora, Yinzhi Wang, Daniel S. Katz, William Gropp, and Zhao Zhang. Fine-Grained Policy-Driven I/O Sharing for Burst Buffers. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '23. ACM, 2023.

[17] Awais Khan, Hyogi Sim, Sudharshan S. Vazhkudai, Ali R. Butt, and Youngjae Kim. An Analysis of System Balance and Architectural Trends Based on Top500 Supercomputers. In *The International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2021, page 11–22, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Yiyo Kuo, Sheng-I Chen, and Yen-Hung Yeh. Single machine scheduling with sequence-dependent setup times and delayed precedence constraints. *Operational Research*, 20(2):927–942, 2020.

[19] LANL, NERSC, SNL. APEX workflows. Technical report, Los Alamos National Laboratory (LANL), National Energy Research Scientific Computing Center (NERSC), Sandia National Laboratory (SNL)., 2016. Available online at `http://www.nersc.gov/assets/apex-workflows-v2.pdf`.

[20] Leonardo Lozano, Michael J. Magazine, and George G. Polak. Decision diagram-based integer programming for the paired job scheduling problem. *IISE Transactions*, 53(6):671–684, 2021.

[21] Message Passing Interface Forum. MPI: A message-passing interface standard, version 4.0. `https://www.mpi-forum.org/`, 2021.

[22] Alix Munier, Maurice Queyranne, and Andreas S. Schulz. Approximation bounds for a general class of precedence constrained parallel machine scheduling problems. In *Integer Programming and Combinatorial Optimization*, pages 367–382. Springer, 1998.

[23] Alix Munier and Francis Sourd. Scheduling chains on a single machine with non-negative time lags. *Mathematical Methods of Operations Research*, 57(1):111–123, 2003.

[24] Sarp Oral, Sudharshan S. Vazhkudai, Feiyi Wang, Christopher Zimmer, Christopher Brumgard, Jesse Hanley, George Markomanolis, Ross Miller, Dustin Leverman, Scott Atchley, and Veronica Vergara Larrea. End-to-End I/O Portfolio for the Summit Supercomputing Ecosystem. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '19. ACM, 2019.

[25] Tirthak Patel, Rohan Garg, and Devesh Tiwari. GIFT: A Coupon Based Throttle-and-Reward Mechanism for Fair and Efficient I/O Bandwidth Management on Parallel Storage Systems. In *Proc. 18th USENIX Conf. on File and Storage Technologies*, page 103–120. USENIX Association, 2020.

[26] Zhipeng Tan, Li Du, Dan Feng, and Wei Zhou. EML: An I/O scheduling algorithm in large-scale-application environments. *Future Generation Computer Systems*, 78:1091–1100, 2018.

[27] Sagar Thapaliya, Purushotham Bangalore, Jay Lofstead, Kathryn Mohror, and Adam Moody. IO-Cop: Managing concurrent accesses to shared parallel file system. In *43rd International Conference on Parallel Processing (ICPP) Workshops*, pages 52–60. IEEE, 2014.

[28] Ying Yang, Xuanhua Shi, Wei Liu, Hai Jin, Yusheng Hua, and Yan Jiang. DDL-QoS: a dynamic I/O scheduling strategy of QoS for HPC applications. *Concurrency and Computation: Practice and Experience*, 33(7), 2021.

[29] Izzet Yildirim, Anthony Kougkas, Xian-He Sun, and Kathryn Mohror. Exploring the Impacts of Multiple I/O Metrics in Identifying I/O Bottlenecks. Poster at SC'23, available at `https://sc23.supercomputing.org/proceedings/tech_poster/poster_files/rpost163s3-file3.pdf`, 2023.

[30] Benbo Zha and Hong Shen. Adaptively Periodic I/O Scheduling for Concurrent HPC Applications. *Electronics*, 11(9), 2022.