Sizing and Partitioning Strategies for Burst-Buffers to Reduce IO Contention

Guillaume Aupy, Olivier Beaumont, Lionel Eyraud-Dubois

Sizing and Partitioning Strategies for Burst-Buffers to Reduce IO Contention

Guillaume Aupy*, Olivier Beaumont* and Lionel Eyraud-Dubois*

*Inria and Univ. Bordeaux, Talence, France

guillaume.aupy@inria.fr, olivier.beaumont@inria.fr, lionel.eyraud-dubois@inria.fr

Abstract—Burst-Buffers are high throughput and small size storage which are being used as an intermediate storage between the PFS (Parallel File System) and the computational nodes of modern HPC systems. They can allow to hinder to contention to the PFS, a shared resource whose read and write performance increase slower than processing power in HPC systems. A second usage is to accelerate data transfers and to hide the latency to the PFS. In this paper, we concentrate on the first usage. We propose a model for Burst-Buffers and application transfers.

We consider the problem of dimensioning and sharing the Burst-Buffers between several applications. This dimensioning can be done either dynamically or statically. The dynamic allocation considers that any application can use any available portion of the Burst-Buffers. The static allocation considers that when a new application enters the system, it is assigned some portion of the Burst-Buffers, which cannot be used by the other applications until that application leaves the system and its data is purged from it. We show that the general sharing problem to guarantee fair performance for all applications is an NP-Complete problem. We propose a polynomial time algorithms for the special case of finding the optimal buffer size such that no application is slowed down due to PFS contention, both in the static and dynamic cases. Finally, we provide evaluations of our algorithms in realistic settings. We use those to discuss how to minimize the overhead of the static allocation of buffers compared to the dynamic allocation.

I. INTRODUCTION

The I/O bottleneck is one of the major issues in current HPC systems.

On the one hand, the architectural changes in supercomputers move in the direction of an increasing bottleneck: when Los Alamos National Laboratory moved from Cielo to Trinity, the peak performance moved from 1.4 Petaflops to 40 Petaflops ($\times 28$) while the I/O bandwidth moved to 160 GB/s to 1.45TB/s (only $\times 9$) [1]. Similar trends can be seen at Argonne National Laboratory: their home machine went from Intrepid (0.6 PF, 88 GB/s) to Mira (10PF, 240 GB/s) and to Aurora (expected 180PF and 1TB/s) [2].

On the other hand, the applications running on the supercomputers also require increasingly more I/O exchanges. First, in the framework on the convergence between HPC and BigData [3], HPC systems are now also used to run BigData applications. One main characteristic of BigData workload is that they are dominated by read operations. Second, the MTBF (Mean Time Between Failures) of HPC systems is decreasing [4], [5] and Checkpoint/Restart (C/R) strategies are necessary to enforce reliable computations in a failure prone system. C/R strategies mostly induce write operations. Third,

HPC applications themselves consume a lot of I/O bandwidth (see Section III-B) as they alternate read, compute and write phase to the PFS, that cannot be overlapped. We consider in the present paper both read and write accesses to the PFS. In typical HPC systems, very few applications (usually one) are enough to saturate the bandwidth to the PFS, so that delays are experienced if the transfers are not coordinated. Mechanisms such as Clarisse [6] have thus been introduced to reorganize transfers to the PFS at system level.

When running several such applications, even if the overall bandwidth is enough to cope in the long term with required data transfers, the bursty nature of both read and write operations and the lack of synchronization between applications induces I/O peaks, that in turn degrade the aggregated bandwidth, as noted in [7]. In this context, in order to cope with the limited I/O bandwidth of HPC system, Burst-Buffers have emerged as a promising solution [8], [9], [10], either as a cache between the computational nodes and the PFS so as to accelerate all data transfers (at the price of a limited lifetime [11]), and by acting as an intermediate storage used to delay write operations and to prefetch read operations, in order to avoid access conflicts and to hide contentions to the user by dealing smoothly with I/O peaks.

There is still no clear consensus on Burst-Buffers architecture (see Section II-A). In this paper, we consider the simplest model where the Burst-Buffers are not distributed and act as a potential intermediate centralized layer, with a higher I/O bandwidth but a smaller capacity than the PFS. It can be partitioned between the different applications, ensuring that each application has a dedicated allocation. Furthermore, we consider a set of applications running independently on dedicated computational nodes belonging to the same machine, where the allocation of nodes has been done a priori using a batch scheduler such as SLURM [12]. In order to deal with BigData, Checkpoint/Restart and HPC applications, we consider that the pattern of I/O and processing phases is known in advance, typically through monitoring and historical data [13]. Our goal is then both to dimension the Burst-Buffers and to partition it between the different applications so as to limit the application slowdown experienced due to the limited I/O bandwidth to the PFS.

The main contributions of this work are the following. Section III presents a precise and complete model of the platform and applications. We propose several uses of Burst-Buffers (static allocation and dynamic allocation). Section IV

proves the intractability of the problem in the general case: for a given buffer size, find an allocation that ensures that all applications are treated fairly. Section V introduces the necessary concepts used in our algorithm, and shows how to dimension the Burst-Buffers size for a single application. Section VI presents an optimal algorithm for the problem of minimizing the Burst-Buffers size to ensure that no application is delayed because of inter-application competition to the PFS bandwidth (optimal stretch). Section VII provides several evaluations with bounded buffer size to compare the performance of the static and dynamic buffer allocations. We discuss these strategies. We also present in Section II the related work on Burst-Buffers architecture, bandwidth allocation and HPC applications models, and propose concluding remarks and perspectives in Section VIII.

Note that all our results are validated through thorough evaluation. The source code and scripts for those evaluations are available at https://gitlab.inria.fr/ordo-bdx/io-peak.

II. RELATED WORK

A. Burst-Buffers Architectures and models

There are many implementations of Burst-Buffers. The two most studied characteristics are the location of the buffers and whether they are shared between multiple applications or dedicated. A typical architecture consists in locating the Burst-Buffers between the compute nodes and the Parallel File System (PFS). This is the case of DDN IME [8], [14] and Cray DataWarp [9], [15], [16]. In this *pseudo-centralized* architecture, the Burst-Buffers are often colocated with the I/O nodes. Several management strategies have been proposed. Mubarak et al. [15] study the case where the buffers are shared between the different applications running onto the platform and used both to accelerate transfers and to prevent I/O congestion. On the contrary, in Schenck et al. [14] and Daley et al. [16], applications decide the size of the buffer that should be dedicated to them.

Another solution is a *distributed* version of Burst-Buffers where buffers are allocated closer to the compute nodes [17], [18]. A solution consists in allocating the distributed buffers to the application using compute nodes close to buffers [19]. Other strategies focus on how to share them between the different applications [17], [18]. In [15], the interaction between the placement of Burst-Buffers and high-radix interconnect topologies is studied. In the context of fault-tolerance, using a buffer on a different node can allow the implementation of hierarchical checkpointing strategies that provide more resilience than in-node buffer strategies [18]. Furthermore, in the case where the number of buffers in the machine is limited due to their cost, one must choose on which node they should be deployed and between which subset of applications they should be shared.

B. Algorithms to deal with Burst-Buffers

When it comes to using Burst-Buffers, several solutions have been proposed. We present and discuss the most common ones. A natural idea is to use Burst-Buffers as a cache to improve the I/O-performance of applications [14]. For instance, DDN [8] announces bandwidth performance 10-fold that of PFS using their Burst-Buffers. The idea is to move the I/O to the Burst-Buffers as a temporary stage between compute nodes and the PFS (whether the data is incoming or outgoing). Thanks to the higher bandwidth of the Burst-Buffers, this improves the I/O transfer time while pipelining the (slowest) phase of sending/receiving data from the PFS with the compute phase of the application. However, as noted by Han et al. [11], this idea may not be viable, as (i) Burst-Buffers are based on technologies that are extremely expensive with respect to hard drives and (ii) they are currently based on SSD technology, that is known to have a limited rewrite lifespan [11]. Thus, the large number of I/O operations in HPC applications would decrease their lifespan too fast. We consider a solution where not all data transfers go through the Burst-Buffers but only those necessary to avoid I/O congestion.

The second natural idea proposed in the literature is indeed to use Burst-Buffers to prevent I/O congestion [20], [21] while maintaining their lifespan. To achieve this goal, the applications use the direct link to the PFS (see Fig. 1) when its bandwidth B is not exceeded. When the bandwidth is exceeded by the set of transfers, then the higher bandwidth of the Burst-Buffers is used to complement the bandwidth to the PFS. This is one of the solutions advocated by DDN in [8]. The intuition behind this strategy is that the average use of PFS bandwidth is usually small enough, but that Burst-Buffers are crucial to deal with the simultaneous bursts of applications. This corresponds to the model depicted in Fig. 1 that will be used throughout this paper. In our previous work [22], we have studied management strategies using a very simplistic random application model. Using Markov chain, we were able to show that the bandwidth B_{BB} to the Burst-Buffers (see Fig. 1) does not have to be very large compared to the bandwidth B to the PFS (less than 10x is enough), which is generally the case [8]. We showed that strategies that do not empty as soon as possible the buffer but wait until the buffer is at least 20% full do not add contention to the system while avoiding write operations. Even though we were able to give general performance trends, the problem of efficiently sizing the buffer remains open due to lack of a precise application model, which is the purpose of the present paper. Tang et al. [7] showed experimentally that the reactive draining strategy that empties the Burst-Buffers as soon as possible can lead to a severe degradation of the aggregate I/O throughput. They advocate for a proactive draining strategy, where data is divided into draining segments which are dispersed evenly over the I/O interval, and the burst buffer draining throughput is controlled through adjusting the number of I/O requests issued each time. This work [7] however does not consider the dimensioning nor the optimal partitioning of the Burst-Buffers, but concentrates on dynamic strategies to actually perform transfers.

Finally, a large part of the literature on Burst-Buffers studies how to use them with a specific application workflow [16], [14]. Specifically, systems where applications have dedicated
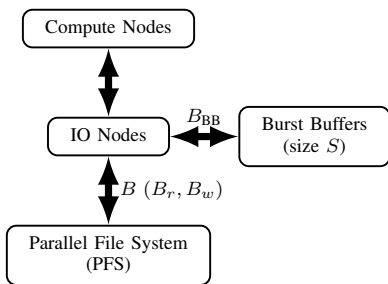
Figure 1: Modeling of the pseudo-centralized platform.

and pre-allocated Burst-Buffers are considered, when the application can explicitly control its data transfers and the use of its Burst-Buffers. This must be done for each application and is very platform dependent. In practice, only few applications have the human-power to implement such solutions. By opposition, our work is only architecture dependent and does not require any additional work from application developers. However, we believe our results can also be used by applications developers if they want to estimate the size of Burst-Buffers that they would need based on their application characteristics (see Section V). In [16] and [23], the authors analyze workflows used in HPC system (CAMP and SWarp in [16], CyberShake in [23]) in order to model their accesses to the storage system and to identify opportunities to leverage the capabilities of the Burst Buffer of NERSC's Cori system based on Cray DataWarp [9].

III. MODELS

A. Machine model

We assume that we have a parallel platform made up of N identical unit-speed nodes, composed of the same number of identical processors. We model the long-term storage system (Parallel File System or PFS) as a single file server with input bandwidth B (in a system with several file servers, B would represent their aggregate bandwidth). For the sake of completeness, we also consider B_r and B_w the maximum read and write bandwidths from/to the PFS. In general, $B_r = B_w = B$. In addition to this file server, the platform is equipped with Burst-Buffers of input bandwidth B_{BB} and size S. Fig. 1 depicts a schematic view of this model.

In present work, we assume that the Burst-Buffers can be partitioned either statically or dynamically between applications. In the case of static partitioning, once a new application enters the system, the scheduler decides on the share of the buffer allocated to it. It cannot be modified as long as the application has not left the system. In the dynamic model, the share of each application can change at runtime. Formally, we write these two strategies:

1/ **STATIC**: when an application $\mathcal{A}_k$ enters the system, it is allocated a volume S_k of the Burst-Buffers. S_k remains constant throughout its execution, and must respect the following constraint: at any time, if $\{\mathcal{A}_k\}_{k \leq n}$ are running on the system, then $\sum_{k=1}^{n} S_k \leq S$.

2/ **DYNAMIC**: at any time t, if $\mathcal{A}_k$ is running on the system

it can use a volume $S_k(t)$ of the Burst-Buffers. The same constraint on the total buffer used holds: at all time, if $\{\mathcal{A}_k\}_{k \leq n}$ are running on the system, then $\sum_{k=1}^{n} S_k(t) \leq S$.

Note that part of this model has been verified experimentally to be consistent with the behavior of Intrepid and Mira, supercomputers at Argonne [24] and Jupiter, a machine at Mellanox [13]. In order to be compliant with the strategies described in [25], we introduce (see Section VI for details) the possibility of progressively providing Burst-Buffers resources to an application before it starts (to prefetch its input data) and to progressively remove Burst-Buffers resources from an application after it ends (to release processing resources as soon as possible).

B. Application Model

We consider scientific applications running simultaneously onto a parallel platform. In the present study, there is no interaction with the batch scheduler and we assume that the set of processing resources provided to the application is given. With respect to I/Os, applications consist in a sequence of three consecutive (and possibly nil) actions: (i) data fetching from disks (read); (ii) computations (compute); and (iii) data uploading on disks (write).

Formally, application $\mathcal{A}_k$ is released at time r_k and consists of n_k iterations. Iteration $i \leq n_k$ of $\mathcal{A}_k$ consists of three consecutive *non-overlapping* phases: a read phase, where $R_{k,i}$ denotes the volume of data read, at read bandwidth b_k^r; a compute phase, where $l_{k,i}$ denotes the compute time; and a write phase, where $W_{k,i}$ denotes the volume of data to be written at write bandwidth b_k^w. We assume that the phases cannot be overlapped for a given application: reading must be finished before the computation can start, and similarly the computation must be finished before starting to write. This constraint is representative of many applications, whose memory requirements prevent to fetch data for the next phase in advance when the data for the previous phase still occupies the memory. We however assume that the input data of the reading phases can be prefetched in a burst buffer if its size allows it: this data does not depend on the results of the previous computations. A more generic model taking data dependencies into account is out of the scope of this paper.

In practice, b_k^r and b_k^w depend on the resources allocated by the batch scheduler and are given for $\mathcal{A}_k$. Hence, an application can be written as:

$$\mathcal{A}_k = \left(r_k, b_k^r, b_k^w, \Pi_{i=1}^{n_k}\left(R_{k,i}, l_{k,i}, W_{k,i}\right)\right). \tag{1}$$

We also denote by

$$C_k^{\min} = r_k + \sum_{i=1}^{n_k} \frac{R_{k,i}}{b_k^r} + l_{k,i} + \frac{W_{k,i}}{b_k^w} \tag{2}$$

the earliest an application could finish given its parameters and assuming that the system is not slowing it down. In practice, this bound is hard to achieve on a machine: while the computations are done independently because each application makes use of dedicated nodes, the applications compete for sending and receiving data during their I/O phase on a shared
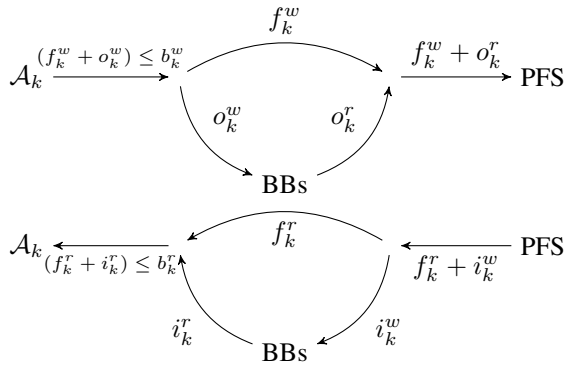
Figure 2: Schematics of the bandwidth used for *output* (top) and *input* (bottom).

I/O network, what results in congestions and delays between I/O nodes of the platform and the PFS. We discuss the conditions necessary to reach this lower bound in Section V-B.

Execution Model: In the execution of a schedule, there are notable events. Specifically, for each phase $(R_{k,i}, l_{k,i}, W_{k,i})$ of application $\mathcal{A}_k$, we can define three events:
(i) The beginning of the read *to the application* following the end of the previous write *from the application* denoted $t_{k,i}^r$ (with $t_{k,1}^r = r_k$).
(ii) The beginning of the compute of the application following the end of the read *to the application* denoted $t_{k,i}^c$.
(iii) The beginning of the write *from the application* folllowing the end of the computation phase denoted $t_{k,i}^w$.

We consider that above phases coincide without loss of generality, since allocated I/O resources can vary with time and can be zero at the beginning and the end to encompass for delays. Finally, we denote by C_k the end of the last write phase (coinciding with the end of the execution of $\mathcal{A}_k$). For each application $\mathcal{A}_k$, the following set of functions (defined at each instant t) describe data movements (see Fig. 2): (i) part of its output (write) data is sent to the PFS at rate f_k^w, and the rest to the Burst-buffers at rate o_k^w, this is a part of the $W_{k,i}$ phase. (ii) part of its input (read) data is collected from the PFS at rate f_k^r, and the rest from the Burst-buffers at rate i_k^r, this is a part of the $R_{k,i}$ phase.

By definition, the following properties on $i_k^r + f_k^r$ (resp. $o_k^w + f_k^w$) hold: (i) it is only non zero on the intervals $[t_{k,i}^r, t_{k,i}^c]$ (resp. $[t_{k,i}^w, t_{k,i+1}^r]$); (ii) it is bounded by b_k^r (resp. b_k^w); and (iii) $\int_{t_{k,i}^r}^{t_{k,i}^c} f_k^r(t)dt = R_{k,i}$ (resp. $\int_{t_{k,i}^w}^{t_{k,i+1}^r} f_k^w(t)dt = W_{k,i}$).

Independently of the current phase of the application, the buffer itself can prefetch or write data from/to the PFS. We denote by i_k^w and o_k^r the function of time expressing the rate at which this is done.

C. Optimization problem

Let us now propose a performance model for these applications. Let us consider application $\mathcal{A}_k = (r_k, b_k^r, b_k^w, \Pi_{i=1}^{n_k}(R_{k,i}, l_{k,i}, W_{k,i}))$ and let us first assume that it is running alone on the machine. In order to perform the I/O operations $(R_{k,i}, W_{k,i})$, several strategies can

be used:
(i) Without burst-buffers, the I/O operations take a time of $\frac{R_{k,i}}{\min(B, b_k^r)}$ and $\frac{W_{k,i}}{\min(B, b_k^w)}$.
(ii) With Burst-Buffers (and no size constraints), to execute $R_{k,i}$, the buffer can prefetch the data at rate B while $\mathcal{A}_k$ is performing a previous iteration. When $\mathcal{A}_k$ is done with $W_{k,i-1}$, the data can be obtained in $R_{k,i}/b_k^r$ units of time. Similarly, to process $W_{k,i}$, the application sends data on the buffer at rate b_k^w, then the buffer sends it on the PFS at rate B. With capacity constraints on Burst-Buffers, $\mathcal{A}_k$ follows a mix of above two behaviors, depending on how much data can be stored into the buffers.

Finally, let us define the stretch of $\mathcal{A}_k$ ($s(\mathcal{A}_k)$) in a schedule. Given C_k, the end of the execution of $\mathcal{A}_k$ in the schedule, and given $C_k^{\min}$, the earliest date when $\mathcal{A}_k$ may finish (as defined by Eq.(2)), the stretch of $\mathcal{A}_k$ is given by

$$ s(\mathcal{A}_k) = \frac{C_k - r_k}{C_k^{\min} - r_k}. \tag{3} $$

For both strategies $X \in \{\text{STATIC}, \text{DYNAMIC}\}$, we consider two different problems:

Definition 1 (X-BUFFER-SIZE(ρ))**.** Find a schedule that minimizes the total size S of the Burst-Buffers with strategy X, for a maximum stretch of ρ ($\forall k, s(\mathcal{A}_k) \leq \rho$).

Definition 2 (X-STRETCH(S))**.** Find a schedule that minimizes the maximum stretch ($\max_k s(\mathcal{A}_k)$) with strategy X, where the total buffer size is bounded by S.

D. Dominant Schedules

A schedule is defined by the list of functions $(f_k^w, f_k^r, i_k^w, i_k^r, o_k^w, o_k^r), \forall k$ which describe the rates of data transfers. The description of these functions over time is not a priori polynomial in the size of the input problem. In this section, we prove that we can focus on strategies where each function is constant between the different events of the schedule (as defined above). We call such schedules *Dominant Schedules*. Hence, a schedule can be fully described by the set of events $(t_{k,i}^r, t_{k,i}^c, t_{k,i}^w)_{k,i}$, and the values of functions $(f_k^w, f_k^r, i_k^w, i_k^r, o_k^w, o_k^r), \forall k$ at these events, what provides a polynomial size description whose correctness with respect to resource limitations can be checked in polynomial time. In the rest of the paper, we therefore restrict the search to Dominant Schedules.

Theorem 1. *Given a schedule $\mathcal{S} = (f_k^w, f_k^r, i_k^w, i_k^r, o_k^w, o_k^r)_k$, there exists a (dominant) schedule $\tilde{\mathcal{S}} = (\tilde{f_k^w}, \tilde{f_k^r}, \tilde{i_k^w}, \tilde{i_k^r}, \tilde{o_k^w}, \tilde{o_k^r})_k$ such that (i) all applications have the same stretch as $\mathcal{S}$; (ii) the total buffer size used is the same as $\mathcal{S}$; (iii) between any two events $(t_{k,i}^r, t_{k,i}^c, t_{k,i}^w)_{i,k}$ of $\tilde{\mathcal{S}}$, all functions $f \in \tilde{\mathcal{S}}$, are constant.*

Due to lack of space, we only provide the intuition of the proof, which is available in the companion report [26].

Proof. Given $e_0 < \cdots < e_n$ the list of events of $\mathcal{S}$ (beginning of read, compute and write phases of each applications). For

$f \in \mathcal{S}$, let us define $\tilde{f} \in \tilde{\mathcal{S}}$ by: $\forall i, \tilde{f} : x \in [e_i, e_{i+1}] \mapsto \frac{\int_{e_i}^{e_{i+1}} f(t)dt}{e_{i+1}-e_i}$. Overall, we show that the transformation has the same events set than $\mathcal{S}$ and that for these events, it satisfies the following constraints:

1/ Application-specific constraints: (i) All the data transfer necessary for a read/write phase is performed during those phases; (ii) The maximum application bandwidths of $\tilde{\mathcal{S}}$ are never larger than b_k^r and b_k^w;

2/ PFS-specific constraint: (i) The total PFS bandwidth is never larger than B;

3/ Buffer-specific constraints: (i) The total buffer peak is never larger than S ($X = \text{DYNAMIC}$); (ii) The sum of individual buffer peaks is never larger than S ($X = \text{STATIC}$). (iii) The data leaving the buffer is not larger than the data entering the buffer.

Satisfying these constraints ensures that the solution $\tilde{\mathcal{S}}$ is valid, and that its performance is identical to the one of $\mathcal{S}$ (same time events, same buffer sizes). $\qquad\square$

IV. COMPLEXITY RESULTS

We provide a NP-hardness proof for X-STRETCH and X-BUFFER-SIZE, using a reduction from the well-known 3-Partition problem (3-PART).

Theorem 2. X-STRETCH(0) and X-BUFFER-SIZE(ρ) for any fixed ρ such that $1 < \rho \leq 2$ are NP-complete.

Proof. Let us consider the associated decision problem: given a set of K applications $\mathcal{A}_k$ and a platform, is there a schedule of stretch at most ρ without using any buffer?

We have shown that by considering *Dominant Schedules*, the problem belongs to NP. We use a reduction from 3-PART. Consider an arbitrary instance $\mathcal{I}_1$ of 3-PART: given an integer B and $3n$ integers $a_1, \ldots, a_{3n}$, s.t. $\sum_{i=1}^{3n} a_i = nB$, can we partition the set of $3n$ integers into n triplets $I_1, \cdots, I_n$, each of sum B?

We build the following instance $\mathcal{I}_2$ of X-STRETCH(0) and X-BUFFER-SIZE(ρ): the maximum bandwidth of the I/O system is B, there are $3n$ applications released simultaneously ($r_k = 0$) with one phase each ($n_k = 1$), $R_{k,1} = W_{k,1} = a_k$, $l_{k,1} = d$, where $d = \frac{n+1-2\rho}{\rho-1}$. The maximum bandwidth of application $\mathcal{A}_k$ is $b_k^r = b_k^w = a_k$, so that for each application, its communication phase takes time at least 1. Note that $C_k^{\min} = d + 2$. We study whether there exists a solution of buffer size $S = 0$ and with a stretch not greater than ρ (by definition of d, we have $\rho = \frac{d+n+1}{d+2}$), or equivalently, is there a schedule such that all applications finish before time $d+n+1$. Note that the definition of d also ensures that $d \geq n - 1$ as long as $\rho \leq \frac{2n}{n+1}$ (which holds if n is large enough).

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

Let us first assume that $\mathcal{I}_1$ has a solution: let us denote by $I_1, \cdots, I_n$ the n triplets of $\mathcal{I}_1$. By definition, $\sum_{i \in I_t} a_i = B$. We can build the following solution for instance $\mathcal{I}_2$: if $k \in I_t$, then $R_{k,1}$ is scheduled from time $t - 1$ to time t at maximum bandwidth a_k. Then, $W_{k,1}$ is scheduled from time $t + d$ to time $t + d + 1$ at maximum bandwidth a_k (see Fig. 3). It is a valid solution for $\mathcal{A}_k$ with respect to the read, compute and
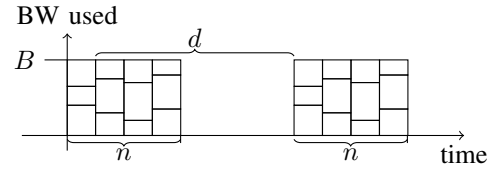

Figure 3: Communication schedule obtained from a positive 3-PART instance

write constraints. The stretch of $\mathcal{A}_k$ is $t + d + 1/d + 2 \leq \rho$. Furthermore, since for all t $\sum_{i \in I_t} a_i = B$, it is also a valid solution with respect to the I/O bandwidth constraint.

Assume now that $\mathcal{I}_2$ has a solution: By definition of the stretch, the latest date an application can terminate is $d+n+1$. Moreover, there cannot be any I/O movement between time n and $d+1$ since (i) write data are not ready yet: the minimum time needed is $d+1$ units of time for any application and (ii) read data should be over: the minimum time needed once data is read is $d + 1$.

Since the total I/O volume is $\sum_k R_{k,1} + W_{k,1} = 2nB$, then the I/O bandwidth must be used at full capacity from time 0 to time n, and from time $d + 1$ to time $n + d + 1$.

Lemma 1. *In a solution to instance $\mathcal{I}_2$ of length $d+n+1$, if an application reads some data between time t and time $t+1$, then its read phase finishes at time $t + 1$.*

Proof. All read phases end at time $t \leq n$ since it takes a minimum time of $d+1$ to do the rest of the computations. We show the result by contradiction. Let us assume that the claim is not true. Let t be the first time such that an I/O transfer occurred between time t and $t + 1$ but did not complete by time $t + 1$. Let $V > 0$ be the volume of this transfer from time t to $t+1$. Since the total volume of I/O transferred from time 0 to $t + 1$ is at most $B(t + 1)$, the amount of *finished* read phases is at most $B(t + 1) - V$.

Because of the structure of $\mathcal{I}_2$, the amount of write data available before time $d + t + 2$ is at most $B(t+1) - V$. This contradicts the fact that the I/O bandwidth has been used at full capacity from time $d + 1$ to time $d + t + 2$, hence ending the proof. $\qquad\square$

Let us consider any time interval $[i, i + 1]$ for $i \leq n - 1$, then the communication link must be fully occupied, and communications must take time 1. This implies that the read phase of $\mathcal{A}_k$ is performed at maximum rate a_k. Therefore, partitioning the applications according to the interval in which their read phase takes place provides a valid solution to the 3-PART problem. $\qquad\square$

Theorem 3. $\forall S \geq 0$, STATIC-STRETCH(S) *is NP-complete.*

Proof. We can extend previous reduction from 3-PART with an additional application. With the same notations as above, let us introduce another application $\mathcal{A}_{n+1}$ with release time n, a single write phase of size $S + B\rho$, and a bandwidth $b_k^r = S + B\rho$. Furthermore, let us enforce $d \geq n + 1$, which holds

as long as $\rho \geq \frac{2n+2}{n+3}$. The minimum execution time of $\mathcal{A}_{n+1}$ is $C_{n+1}^{\min} = 1$. To achieve a stretch at most ρ, this application must therefore complete within time $n + \rho$. Since $d \geq n + 1$, this happens between the read and the write phases of other applications. However, during this time, only a volume $B\rho$ of data can be sent to the PFS; and the rest of the write phase of $\mathcal{A}_{n+1}$ must be sent to the burst buffer. Thus, in a solution of stretch ρ and Burst-Buffers size S, the whole buffer size must be allocated to $\mathcal{A}_{n+1}$. In the STATIC model, this implies that no buffer remains available for the other applications, and the proof of the previous theorem allows to conclude. $\qquad \square$

V. BURST-BUFFERS LOWER BOUNDS FOR THE EXECUTION OF A SINGLE APPLICATION

In Section III, we provided a lower bound on the execution time of a single application given its characteristics as $C_k^{\min} = r_k + \sum_{i=1}^{n_k} \frac{R_{k,i}}{b_k^r} + l_{k,i} + \frac{W_{k,i}}{b_k^w}$. In general, this lower bound is not reachable. Indeed, to reach it, $\mathcal{A}_k$ needs to read and write at maximum bandwidth during all its read and write phases. This is not typically doable for example if $b_k^r > B_r$ or $b_k^w > B_w$. As noted in [25], Burst-Buffers are expected to accelerate applications by (i) accelerating the transfers to and from the PFS by using the Burst-Buffers as a cache for writing (buffering) and reading (pre-fetching) data and (ii) enabling to reorganize the communications to the PFS in order to avoid contention when accessing it. In this section, we focus on a single application running on the platform, and show how to optimally dimension the Burst-Buffers to minimize its runtime.

A. Description of a solution achieving optimal makespan

Let us consider application $\mathcal{A}_k$. Let $C_k^{\min}$ denote the lower bound of C_k, the makespan of $\mathcal{A}_k$, as defined in Eq (2). Let us study a solution that achieves $C_k^{\min}$. Let us consider the case where there are no constraints on the bandwidth to the PFS, i.e. $B = B_r = B_w = +\infty$. The minimal time for the read, compute and write phases are respectively $R_{k,i}/b_k^r$, $l_{k,i}$ and $W_{k,i}/b_k^w$. Since these phases cannot overlap, then for each iteration i:
(i) $t_{k,i}^r = r_k + \sum_{j=1}^{i-1} \frac{R_{k,j}}{b_k^r} + l_{k,j} + \frac{W_{k,j}}{b_k^w}$. The read phase must take exactly $R_{k,i}/b_k^r$ units of time, hence it is performed at bandwidth b_k^r. This situation is depicted on the solid line in Fig. 4 and in what follows, we denote by $R_k^\infty(t)$ the value of the solid line at time t. The slope of $R_k^\infty(t)$ is therefore either 0 (during compute and write phases) or b_k^r during read phases.
(ii) $t_{k,i}^c = r_k + \sum_{j=1}^{i-1} \frac{R_{k,j}}{b_k^r} + l_{k,j} + \frac{W_{k,j}}{b_k^w} + \frac{R_{k,i}}{b_k^r}$. The compute phase lasts $l_{k,i}$ units of time.
(iii) $t_{k,i}^w = r_k + \sum_{j=1}^{i-1} \frac{R_{k,j}}{b_k^r} + l_{k,j} + \frac{W_{k,j}}{b_k^w} + \frac{R_{k,i}}{b_k^r} + l_{k,i}$. This situation is similar to the read phase. We denote by $W_k^\infty(t)$ the value of the solid line at instant t (Fig. 4). The slope of $W_k^\infty(t)$ is therefore either 0 (during read and compute phases) or b_k^w during write phases.

B. Minimum Burst-Buffers size to achieve $C_k^{\min}$ in isolation.

In the case where both $b_k^w \leq B_w$ and $b_k^r \leq B_r$ and if $\mathcal{A}_k$ is running alone on the platform, then the solution that performs transfers as soon as possible and at maximal
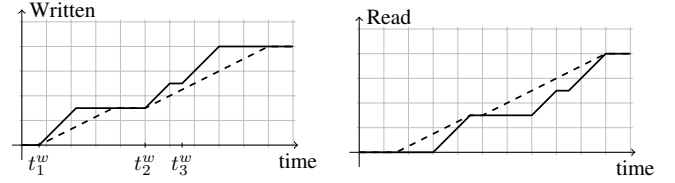

Figure 4: Data written (left) and read (right) by $\mathcal{A}_k$ (when running in isolation) onto/from the PFS when assuming $B = +\infty$ (solid line) and when using a Burst-Buffers (dashed line)

bandwidth trivially achieves $C_k^{\min}$ and Burst-Buffers are only needed if several applications are running simultaneously (case studied in Section VI).

This is not the case if the application can write (resp. read) at speed $b_k^w > B_w$ (resp. $b_k^r > B_r$) on the Burst-Buffers. In this case, in order to achieve $C_k^{\min}$, $\mathcal{A}_k$ must use Burst-Buffers resources, and our goal is to find the minimal Burst-Buffers size, denoted as P_k, so as to achieve optimal execution time. As it is generally the case in practice, we assume that $\mathcal{A}_k$ read operations do not depend on previous write operations and can be stored from the PFS to the Burst-Buffers in advance (see Section III).

Let us first concentrate on the write operations onto the PFS of a single application in presence of a limited bandwidth B_w to the PFS. The dashed line in Fig. 4 depicts the volume of data written to the PFS. Writing the data of the first write phase to the PFS cannot start before time step $t_1^w = r_k + R_{k,1}/b_k^r + l_{k,1}$ and must last at least $W_{k,1}/B_w$ since B_w is an upper bound on the achievable bandwidth to the PFS. In order to achieve makespan $C_k^{\min}$, the second read phase must start at time $t_2^r = r_k + R_{k,1}/b_k^r + l_{k,1} + W_{k,1}/b_k^w$. At that time, given the limited bandwidth B_w to write data onto the PFS, the Burst-Buffers is used to store the data that could be written to the Burst-Buffers (at rate b_k^w) but not on the PFS (at rate B_w). At any time step, the solid line represents the overall volume of data sent by $\mathcal{A}_k$ (either to the Burst-Buffers or the PFS) and the difference between the solid and dashed plots represent the minimal volume of data that must be stored onto the Burst-Buffers. After time $r_k + l_{k,1} + R_{k,1}/b_k^r + W_{k,1}/b_k^w$, the Burst-Buffers is being emptied onto the PFS, at rate B_w following the model described in Section III. This transfer stops either when the Burst-Buffers is empty (i.e. solid and dashed plots cross) or at time $r_k + l_{k,1} + R_{k,1}/b_k^r + W_{k,1}/b_k^w + R_{k,2} + l_{k,2}$ (when a new write operation must start). From then, and until time $r_k + l_{k,1} + R_{k,1}/b_k^r + W_{k,1}/b_k^w + R_{k,2} + l_{k,2} + W_{k,2}/b_k^w$, the solid and dashed plots diverge again, meaning that the minimal amount of Burst-Buffers storage increases during this time interval at rate $b_k^w - B_w$. Therefore, following this algorithm, it is possible to determine the minimal volume of Burst-Buffers for write operations that enables to process $\mathcal{A}_k$ within the deadline $C_k^{\min}$, and that corresponds to the maximal difference between the solid and dashed plots. The situation for read operations is analogous and is depicted in Fig. 4, where the solid plot depicts the volume of data that must be read by the
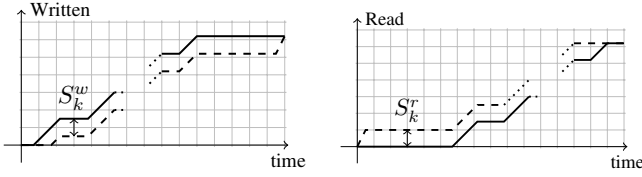
Figure 5: Data written (left) and read (right) to and from the PFS (solid line without contention, dashed line with a BB).

nodes running $\mathcal{A}_k$ and the dashed plot depicts the volume of data that must be read from the PFS (and sent either to $\mathcal{A}_k$ nodes or prefetched on the Burst-Buffers). The algorithm to build the dashed plot is very similar to the algorithm described above in the case of Fig. 4, and the necessary Burst-Buffers size for read operations to run $\mathcal{A}_k$ in isolation with minimal makespan is given by the maximal difference between the plots on Fig. 4.

In above derivations, we have assumed that it is possible to fully use bandwidth B_w (resp. B_r) when writing on (resp. reading from) the PFS. Let us now consider the case where we add an additional constraint stating that the overall bandwidth (the aggregation of incoming and outgoing bandwidth) is bounded by B. This problem can be solved in polynomial time by solving the following linear program in rational numbers, where the goal is to minimize P_k under the constraints:

$$
\begin{cases}
\forall l & w_k^{\mathrm{PFS}}(e_l^k) \leq W_k(e_l^k) \\
\forall l & r_k^{\mathrm{PFS}}(e_l^k) \geq R_k(e_l^k) \\
\forall l & r_k^{\mathrm{PFS}}(e_{l+1}^k) - r_k^{\mathrm{PFS}}(e_l^k) & \leq B_r \cdot (e_{l+1}^k - e_l^k) \\
\forall l & w_k^{\mathrm{PFS}}(e_{l+1}^k) - w_k^{\mathrm{PFS}}(e_l^k) & \leq B_w \cdot (e_{l+1}^k - e_l^k) \\
\forall l & w_k^{\mathrm{PFS}}(e_{l+1}^k) - w_k^{\mathrm{PFS}}(e_l^k) + & r_k^{\mathrm{PFS}}(e_{l+1}^k) - r_k^{\mathrm{PFS}}(e_l^k) \\
& & \leq B \cdot (e_{l+1}^k - e_l^k) \\
\forall l & W_k(e_l^k) - w_k^{\mathrm{PFS}}(e_l^k) + & r_k^{\mathrm{PFS}}(e_l^k) - R_k(e_l^k) \leq \mathrm{P}_k
\end{cases}
$$

In the above LP, e_l^k denotes the ordered events (see Section III for a formal definition), *i.e.* instants where a read I/O phase, a processing phase or a write I/O phase starts (plus time $e_0^k = r_k$). According to Theorem 1, we only need to specify the amount of data transferred to and from the PFS at each of these events. Let us thus denote by $w_k^{\mathrm{PFS}}(e_l^k)$ (resp. $r_k^{\mathrm{PFS}}(e_l^k)$) the overall volume of data written by the application to the PFS (resp. read from the PFS either to the Burst-Buffers or the application) before time e_l^k. Let us also denote by P_k the size of the Burst-Buffers necessary to handle both write and read operations to the Burst-Buffers. Then, the first two constraints ensure that the amount of data written (resp. read) from the PFS is smaller (resp. larger) than the maximal (resp. minimal) volume to achieve $C_k^{\min}$. The following three constraints enforce that neither the read, the write nor the overall bandwidth constraints are exceeded. At last, the last constraint states that the Burst-Buffers size P_k is enough to store both the amount of data written to the Burst-Buffers but not yet to the PFS and the amount of data read from from the PFS but not yet transmitted to the computation nodes.

VI. Burst-Buffers Size to compensate for contention between multiple applications

We have seen in Section IV that X-Buffer-Size(ρ) is NP-complete for $1 < \rho \leq 2$. In this section, we provide a polynomial-time algorithm to solve X-Buffer-Size(1), both in the Static and Dynamic cases. It is computed via a Linear Program whose constraints are detailed in Section VI-B.

A. Data transfers with the PFS in presence of a Burst-Buffers

Let us now compute the minimal Burst-Buffers size S^* necessary to achieve completion time $C_k^{\min}, \forall k$, even if all applications compete for the bandwidth to the PFS. In this case, the Burst-Buffers is also used to avoid contentions to the PFS, by prefetching or storing data that will eventually be read by the application or written to the PFS.

The solid plot on the left of Fig. 5 is analogous to the solid plot in Fig. 4 and depicts the evolution with time of the data volume that must be written by $\mathcal{A}_k$, either to the PFS or to the Burst-Buffers, and is denoted by $W_k^\infty(t)$. On the other hand, the dashed plot in Fig. 4 depicts the evolution of the data actually written to the PFS in presence of a Burst-Buffers of size S_k^w, and is denoted by $W_k^{S_k}(t)$. Indeed, since at any time the amount of data sent by the application must be $W_k^\infty(t)$, and at most S_k^w of it can be stored in the Burst-Buffers, at least $W_k^\infty(t) - S_k^w = W_k^{S_k}(t)$ must have been written to the PFS. After time $C_k^{\min}$, the data to be written to the PFS that still reside in the Burst-Buffers must eventually be transferred to the PFS to release space on the Burst-Buffers. The corresponding amount of Burst-Buffers storage is progressively released by $\mathcal{A}_k$ during this transfer, and can be used by other applications. This emptying strategy is very similar to the ultimate draining strategy described in [25].

Let us now consider any increasing function $W_k(t)$ whose plot remains between $W_k^{S_k^w}(t)$ and $W_k^\infty(t)$ (*i.e.* the solid and dashed plots on Fig. 5) and whose slope is always at most $\min(b_k^w, B_w)$. Then, $W_k(t)$ represents the volume of data written to the PFS in a valid strategy that makes use of a Burst-Buffers of size at most S_k^w. Indeed, if $W_k^{BB}(t)$ denotes the volume written on the Burst-Buffers at time t, *i.e.* $W_k^{BB}(t) = W_k^\infty(t) - W_k(t)$, then the slope of $W_k^{BB}(t)$ is no more than b_k^w, the maximal bandwidth to the Burst-Buffers, since $W_k(t)$ is increasing and the slope of $W_k^\infty(t)$ is at most b_k^w. Furthermore, at any any instant, the sum of the slopes of W_k^{BB} (that may be either positive, when the Burst-Buffers is filled, or negative, when the Burst-Buffers is emptied to the PFS) and $W_k(t)$ is equal to the slope of $W_k^\infty(t)$, so that the strategy is valid and transfers exactly the same volume of data from $\mathcal{A}_k$ as for $W_k^\infty(t)$.

Let us now consider the read phases of application $\mathcal{A}_k$. The situation without Burst-Buffers is depicted in the solid plot at the right of Fig. 5 (identical to the solid plot in Fig. 4) and we denote by $R_k^\infty(t)$ the corresponding value, that represents the minimal amount of data that must be read from the PFS in order to achieve $C_k^{\min}$ when there is no constraint on the bandwidth to the PFS ($B_r = +\infty$). Then,

the dashed plot, denoted as $R_k^{S_k^r}(t)$ in what follows, represents the maximal amount of data that can be read from the PFS if $\mathcal{A}_k$ benefits from a Burst-Buffers of size S_k^r. Then, as for write operations, any increasing function $R_k^{BB}(t)$ such that $R_k^\infty(t) \leq R_k^{BB}(t) \leq R_k^{S_k^r}(t)$ and whose slope is at most $\min(b_k^r, B_r)$ can be associated to a valid reading strategy and at any time t, $R_k^{BB}(t) - R_k^\infty(t)$ represents the amount of data that resides in the Burst-Buffers at time t.

B. Linear Program to Compute the Optimal Burst-Buffers size

Static Case: As already noted, we can consider two different strategies to partition the Burst-Buffers. In what follows, we consider a Burst-Buffers for $\mathcal{A}_k$ of size S_k that can be arbitrarily split between read and write operations. In the STATIC case, we assume that the buffer allocated to $\mathcal{A}_k$ is progressively released from time 0 (size 0) to time r_k (size S_k) and is progressively removed from time C_k (size S_k) to the end of the schedule, to be compliant with the model of [25].

Let us denote (see Section III for a formal definition) the set of points in Fig. 5 at which the slope of any function $R_k^\infty(t)$ or $W_k^\infty(t)$ can change. Let us first remark that all these events, and thus their relative ordering, do not depend on the values of S_k^r and S_k^w, $\forall k$. As previously, we denote by e_l the time of the l-th event, *i.e.* $e_l \leq e_{l+1}$. According to Theorem 1, it is enough to specify the values $R_k(t)$ and $W_k(t)$ for $t \in \{e_l\}$ to be able to rebuild a valid solution. We are now ready to provide all the constraints on these values $R_k(e_l)$ and $W_k(e_l)$ that guarantee a valid solution. We denote this set of constraints $\mathcal{S}_{\text{STATIC}}$ and we use it to optimize FINDOPTIMALSIZE$^{\text{STATIC}}$.
Minimize S under the constraints $\mathcal{S}_{\text{STATIC}}$:

$$\begin{cases} \forall k, l & R_k^\infty(e_l) \leq R_k^{BB}(e_l), \\ \forall k, l & W_k^\infty(e_l) \geq W_k^{BB}(e_l), \\ \forall k, l & 0 \leq R_k^{BB}(e_l) \leq R_k^{BB}(e_{l+1}), \\ \forall k, l & 0 \leq W_k^{BB}(e_l) \leq W_k^{BB}(e_{l+1}), \\ \forall l, & \sum_k (R_k^{BB}(e_{l+1}) - R_k^{BB}(e_l)) \leq B_r(e_{l+1} - e_l), \\ \forall l, & \sum_k (W_k^{BB}(e_{l+1}) - W_k^{BB}(e_l)) \leq B_w(e_{l+1} - e_l), \\ \forall l, & \sum_k (W_k^{BB}(e_{l+1}) - W_k^{BB}(e_l)) + \\ & \sum_k (R_k^{BB}(e_{l+1}) - R_k^{BB}(e_l)) \leq B(e_{l+1} - e_l), \\ \forall k, l, & W_k^{BB}(e_l) \geq W_k^\infty(e_l) - S_k^w(e_l), \\ \forall k, l, & R_k^{BB}(e_l) \leq R_k^\infty(e_l) + S_k^r(e_l), \\ \forall k, l, & S_k^w(e_l) + S_k^r(e_l) = S_k^l, \\ \forall k, l \in \mathcal{Z}_k & S_k^l = S_k, \\ \forall l, & \sum_k S_k^l \leq S \end{cases}$$

In the last constraints of $\mathcal{S}_{\text{STATIC}}$, $\mathcal{Z}_k$ denotes all the events in the interval $[r_k, C_k^{\min}]$, *i.e.* when processing nodes are actually allocated to $\mathcal{A}_k$. During this interval the amount of Burst-Buffers allocated to $\mathcal{A}_k$ is exactly S_k. On the other hand, as previously stated, the Burst-Buffers is progressively allocated to $\mathcal{A}_k$ before r_k and progressively released from $\mathcal{A}_k$ after $C_k^{\min}$.

Dynamic Case: The linear program to compute the optimal Burst-Buffers size in the DYNAMIC case is very similar. We obtain $\mathcal{S}_{\text{DYNAMIC}}$ by removing from $\mathcal{S}_{\text{STATIC}}$ the constraint $\forall k, l \in \mathcal{Z}_k, S_k^l = S_k$, that states that the size of the Burst-Buffers allocated to $\mathcal{A}_k$ cannot change and remains equal to

Workflow	EAP	LAP	Silverton	VPIC
Frequency	65	21	8	6
Number of cores (thousands)	16	4	32	30
Checkpoint size (GB)	3,200	2,000	44,800	3,750
Typical Walltime (hours)	16	4	32	30

Table I: Characteristics of the applications in APEX data set.

S_k when nodes are actually allocated to $\mathcal{A}_k$. This leads to the optimization problem FINDOPTIMALSIZE$^{\text{DYNAMIC}}$: Minimize S under the constraints $\mathcal{S}_{\text{DYNAMIC}}$.

VII. SIMULATION RESULTS

We report extensive simulations to evaluate our linear programming formulations, and compare them with a classic fair sharing approach. To do so, we instantiate our evaluation based on characteristics of the Intrepid platform, and based on a set of applications as described in the APEX report [27] and our previous work [22].

a) **Setup:** We consider a set of 4 applications described in the APEX report [27] which represent the majority of the load at LANL. The characteristics of these applications are provided in Table I. We simulate the execution of these applications on a platform similar to the Intrepid Blue Gene/P supercomputer, used by the Argonne National Laboratory between 2008 and 2014, which was ranked number 3 on the June 2008 Top 500 list. This platform has 96,000 cores, the bandwidth to the file system is $B = 160$GB/s, and the bandwidth per core is $b = 0.02$GB/s. We assume that most of the I/Os for these applications come from periodic checkpoints. We estimate the checkpointing period using the checkpoint optimal period given by $P = \sqrt{2C\frac{\mu}{\#\text{nodes}}}$, following [28]. In this formula, C denotes the checkpointing duration and μ denotes the MTBF (Mean Time Between Failure) of the individual nodes of the platform. In the simulations, we consider different possible values for the MTBF, ranging from 5 years to 50 years. To build the actual workload trace, we select a set of 30 applications, where each application is picked from the four application models described in Table I, with a probability proportional to its usage ratio as reported in [27] (Frequency in Table I). These applications are scheduled in FIFO order on the cores, what provides starting and ending times for each application. In order to compare results, we consider several target values for the *IO load* of the applications (namely 20%, 50% and 80%), defined as follows. An application with checkpoint size s and period P induces an average bandwidth load of $\frac{s}{P}$ over the course of its execution. Thus, when an application starts or ends, the total required bandwidth is updated, and the maximum value over time (normalized by B) provides the I/O need induced by running applications. Once this I/O need is evaluated, the checkpoint sizes for all applications are multiplied by a constant factor (and checkpointing periods are adequately recomputed) so as to obtain the targeted I/O load. To number of cores needed by each application can be read in Table I and it is used to determine the maximal bandwidth b_k at which a given application can communicate with the Burst-Buffers

and the PFS (see Section III). To model the processing time between two checkpoints, we add to the checkpointing period an additional 15% random variability.

Therefore, using the values from the APEX data set [27] summarized in Table I and the description of the Intrepid platform, we are able to instantiate all platform and application parameters needed by the linear programs defined in Section VI. By changing the MTBF (from 5 years to 50 years) and by scaling checkpoint sizes, we are able to study different hardware characteristics and different system loads, while keeping realistic application characteristics. In turn, the linear programs of Section VI compute an optimal buffer size S_{OPT} (both in static and dynamic cases) and its partitioning between applications (in the static case) to process all the applications with a stretch of 1, *i.e.* as if the bandwidth to the PFS was enough to cope with all data transfers at all time. In order to evaluate the influence of the size of the Burst-Buffers, we consider several Burst-Buffers sizes, ranging from 0 to $3 \times S_{\mathrm{OPT}}$. To compare the results of the optimal solutions computed in Section VI to what could be achieved using a dynamic system level strategy, we introduce a greedy strategy, that shares the bandwidth to the PFS in the following way: an application is *write-active* if it is in a write phase or it has output data in its buffer, and the write bandwidth is shared equally between all write-active applications (in the limit of their own output rate). A similar policy is used for sharing the read bandwidth. Additionally, we assume that the Burst-Buffers (of size $0 \leq S \leq 3S_{\mathrm{OPT}}$) is partitioned between applications proportionally to their respective share in the optimal solution computed by the LP. Then, the maximum and average stretch for the applications measure the slowdown induced both by non-optimal bandwidth sharing policy and sub-optimal Burst-Buffers size.

b) **Results:** The performance of the above greedy system level approach is depicted in Figure 6 for both the Max Stretch and Average Stretch metrics, for different MTBF values (as noticed before, a larger MTBF induces rarer and larger checkpoints) and different system loads (either 20%, 50% or 80%). All points in Figure 6 corresponds to 10 executions with different sets of applications, where the length of the processing phases between checkpoints vary from an execution to another, and the darker area shows the typical variability of the results. As expected, the stretch decreases when the Burst-Buffers increases, but only up to a certain size. If a size of S_{OPT} is enough to achieve a solution with stretch 1 for all applications when using the solution of the LP proposed in Section VI, a size of $1.5 \times S_{\mathrm{OPT}}$ is in general required to achieve the lowest stretch using the greedy strategy. Moreover, in particular for large MTBF and load values, the limit value for the stretch is larger than 1, up to 1.15 for the maximum stretch when the load is 80% and the MTBF is 50 years. Nevertheless, even for high loads, the greedy strategy (with the optimal partitioning computed by the LP) is able to achieve close to optimal results when the MTBF is smaller than 10 years, what is in general considered as a reasonable assumption. At last, since the stretch is much higher than 1
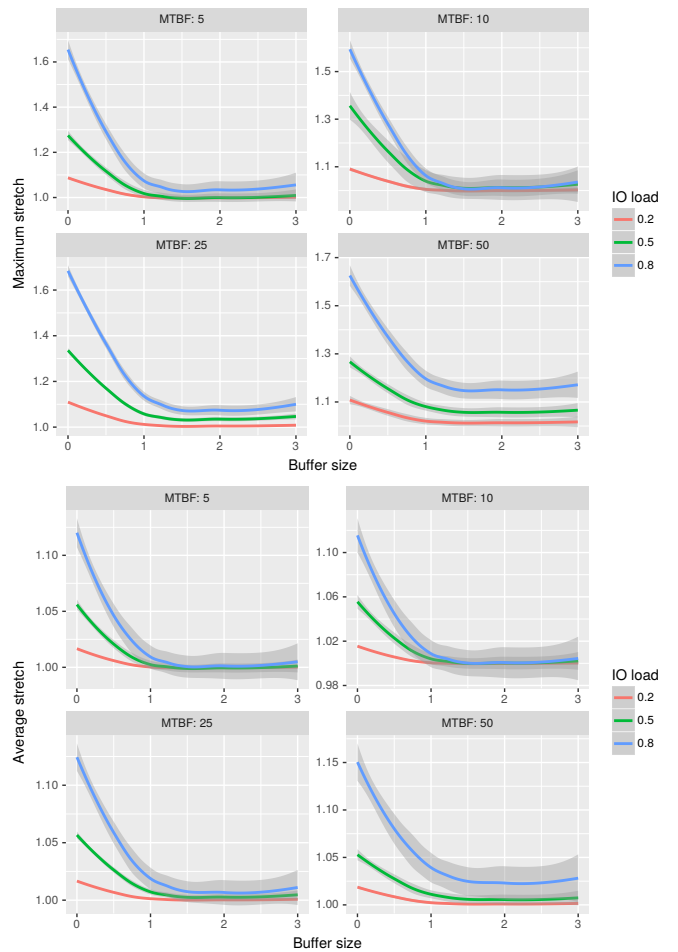

Figure 6: Maximum stretch results (top) and average stretch results (bottom) of fair sharing when buffer size varies, for different MTBF and load values.

for any buffer size below $1 \times S_{\mathrm{OPT}}$, these plots show that the value S_{OPT} returned by the LP is crucial in order to set the size of the Burst-Buffers to a value that does not induce large stretch values and limits the hardware cost.

The cost induced by opting for a static partitioning is analyzed in the following table, that displays the ratio between the Burst-Buffers size required when using a static partitioning and the Burst-Buffers size required when using a dynamic partitioning, for different values of the MTBF (from 5 to 50 years) and for different load levels (from 20 to 80%). These results show that a static partitioning of the Burst-Buffers induces an overhead in size of 25 to 40% for most settings. This overhead is to be compared to the increased simplicity of deployment, in particular with respect to security and dynamic management issues.

Load	5 y	10 y	25 y	50 y
20%	1.32	1.31	1.42	1.67
50%	1.33	1.28	1.26	1.47
80%	1.23	1.26	1.25	1.35

VIII. Conclusion

We consider the problem of sizing and partitioning Burst-Buffers in the context of HPC and Data Science applications running on a supercomputer and competing for the access to the PFS. Our goal is to minimize the slowdown experienced by the applications. Given the characteristics of the platform and of the applications, we first prove a negative result stating that the problem of minimizing the stretch given a Burst-Buffers size is in general NP-Complete. Nevertheless, we provide a polynomial time solution for the special case, of clear practical interest, where the goal is to find the minimal Burst-Buffers size and how to partition it between applications so as to compensate both the limited bandwidth to the PFS and the contention in the access to it. At last, we prove that it is possible to derive from this optimal solution a simple runtime strategy, that can be easily implemented, and that is able to achieve low stretches for most settings. Our study also enables to precisely assess the cost of partitioning the Burst-Buffers between applications as opposed to using it as non-dedicated resource shared between all applications. This work opens several important perspectives. First, if the behavior is well understood when the Burst-Buffers is large enough, the problem of finding efficient strategies when the Burst-Buffers is too small to achieve an optimal stretch is still open. Along the same direction, it would be of great practical importance to be able to assess the good behaviour of the dynamic strategy based on the optimal solution of optimal stretch, both in theory and through a larger set of experiments. Other research directions include extending the model for a more precise data management: taking into account data reuse throughout the execution, and/or considering temporary checkpoint data that could remain on the Burst-Buffers until the next checkpoint if space allows it, instead of being written to the PFS.

References

[1] "The trinity project," http://www.lanl.gov/projects/trinity/.

[2] "I/o at argonne national laboratory," https://wr.informatik.uni-hamburg.de/_media/events/2015/2015-iodc-argonne-isaila.pdf.

[3] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[4] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[5] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni, "Unified model for assessing checkpointing protocols at extreme-scale," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 17, pp. 2772–2791, 2014.

[6] F. Isaila, J. Carretero, and R. Ross, "Clarisse: A middleware for data-staging coordination and control on large-scale hpc platforms," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE, 2016, pp. 346–355.

[7] K. Tang, P. Huang, X. He, T. Lu, S. S. Vazhkudai, and D. Tiwari, "Toward managing hpc burst buffers effectively: Draining strategy to regulate bursty i/o behavior," in *MASCOTS*. IEEE, 2017, pp. 87–98.

[8] DDN Storage, "BURST BUFFER & BEYOND, I/O & Application Acceleration Technology," https://www.ddn.com/download/resource_library/brochures/technology/IME_FAQ.pdf, 2014.

[9] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, "Architecture and design of cray datawarp," *Cray User Group CUG*, 2016.

[10] C. S. Daley, D. Ghoshal, G. K. Lockwood, S. S. Dosanjh, L. Ramakrishnan, and N. J. Wright, "Performance characterization of scientific workflows for the optimal use of burst buffers." in *WORKS@ SC*, 2016.

[11] J. Han, D. Koo, G. K. Lockwood, J. Lee, H. Eom, and S. Hwang, "Accelerating a burst buffer via user-level i/o isolation," in *Cluster Computing, International Conference on*. IEEE, 2017, pp. 245–255.

[12] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[13] G. Aupy, A. Gainaru, and V. L. Fèvre, "Periodic i/o scheduling for super-computers," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, Cham, 2017.

[14] W. Schenck, S. El Sayed, M. Foszczynski, W. Homberg, and D. Pleiter, "Evaluation and performance modeling of a burst buffer solution," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 1, pp. 12–26, 2017.

[15] M. Mubarak, P. Carns, J. Jenkins, J. K. Li, N. Jain, S. Snyder, R. Ross, C. D. Carothers, A. Bhatele, and K.-L. Ma, "Quantifying i/o and communication traffic interference on dragonfly networks equipped with burst buffers," in *Cluster Computing, International Conference on*. IEEE, 2017, pp. 204–215.

[16] C. S. Daley, D. Ghoshal, G. K. Lockwood, S. Dosanjh, L. Ramakrishnan, and N. J. Wright, "Performance characterization of scientific workflows for the optimal use of burst buffers," *Future Generation Computer Systems*, 2017.

[17] D. Kimpe, K. Mohror, A. Moody, B. Van Essen, M. Gokhale, R. Ross, and B. R. De Supinski, "Integrated in-system storage architecture for high performance computing," in *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2012, p. 4.

[18] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: high performance fault tolerance interface for hybrid systems," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. ACM, 2011, p. 32.

[19] L. Cao, B. W. Settlemyer, and J. Bent, "To share or not to share: comparing burst buffer architectures," in *Proceedings of the 25th High Performance Computing Symposium*. Society for Computer Simulation International, 2017, p. 4.

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

[21] S. Herbein, D. H. Ahn, D. Lipari, T. R. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer, "Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 69–80.

[22] G. Aupy, O. Beaumont, and L. Eyraud-Dubois, "What size should your buffers to disks be?" in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 660–669.

[23] R. F. da Silva, S. Callaghan, and E. Deelman, "On the use of burst buffers for accelerating data-intensive scientific workflows," in *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science*. ACM, 2017, p. 2.

[24] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the i/o of hpc applications under congestion," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 1013–1022.

[25] "Trinity / NERSC-8 Use Case Scenarios," Los Alamos National Laboratory, Sandia National Laboratories, Research Report SAND 2013-2941, Feb. 2013. [Online]. Available: https://www.nersc.gov/assets/Trinity--NERSC-8-RFP/Documents/trinity-NERSC8-use-case-v1.2a.pdf

[26] G. Aupy, O. Beaumont, and L. Eyraud-Dubois, "Sizing and Partitioning Strategies for Burst-Buffers to Reduce IO Contention," Inria; Univ. Bordeaux, Research Report RR-9213, Oct. 2018.

[27] S. LANL, NERSC, "Apex workflows," https://www.nersc.gov/assets/apex-workflows-v2.pdf, Technical report, LANL, NERSC, SNL, Tech. Rep., 2016.

[28] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, "Checkpointing algorithms and fault prediction," *Journal of Parallel and Distributed Computing*, vol. 74, no. 2, pp. 2048–2064, 2014.