HAL
open science

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

Guillaume Aupy, Olivier Beaumont, Lionel Eyraud-Dubois

To cite this version:

Guillaume Aupy, Olivier Beaumont, Lionel Eyraud-Dubois. Sizing and Partitioning Strategies for Burst-Buffers to Reduce IO Contention. [Research Report] RR-9213, Inria. 2018. hal-01904032v2

HAL Id: hal-01904032

https://inria.hal.science/hal-01904032v2

Submitted on 6 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

Guillaume Aupy, Olivier Beaumont, Lionel Eyraud-Dubois

RESEARCH REPORT
N° 9213
October 2018
Project-Team TADaaM and RealOpt

ISSN 0249-6399 ISRN INRIA/RR--9213--FR+ENG

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

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

Project-Team TADaaM and RealOpt

Research Report n° 9213 — October 2018 — 25 pages

Abstract: Burst-Buffers are high throughput and small size storage which are being used as an intermediate storage between the Parallel File System (Parallel File System) and the computational nodes of modern HPC systems. They can allow to hinder to contention to the Parallel File System, 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 Parallel File System. 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 give a polynomial time algorithms for the special case of finding the optimal buffer size such that no application is slowed down due to Parallel File System 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.

Key-words: scheduling, IO, PFS, Burst-Buffers

* Inria & Labri, Univ. Bordeaux

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Dimensionnement de Burst-Buffers pour réduire la contention Entrées-Sorties

Résumé : Nous nous intéressons à l'utilisation de Burst-Buffers en temps qu'espace de stockage intermédiaire entre les nœuds de calcul et le Système de Fichiers Parallèles (PFS).

Ce dimensionnement peut être statique (à l'arrivée d'une application dans le système), ou dynamique (en fonction des demandes Entrées-Sorties).

Nous montrons que le problème général de partager équitablement les buffers entre applications est NP-complet. Nous montrons que dans le cas particulier où l'on cherche à minimiser la taille totale du buffer pour qu'aucune application ne soit ralentie est résolvable en temps polynomial. Pour résoudre ce problème nous proposons un programme linéaire.

Finalement nous proposons des évaluations à taille de buffer fixé pour montrer la performance de certains algorithmes naifs communs.

Mots-clés : ordonnancement, IO, PFS, Burst-Buffers

1 Introduction

The I/O bottleneck is becoming a major issue in current HPC systems. This is especially striking when considering their recent evolution. For instance, 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) [2]. The same kind of results can be observed at Argonne National Laboratory when moving from Intrepid (0.6 PF, 88 GB/s) and to Mira (10PF, 240 GB/s) and to Aurora (expected 180PF and 1TB/s) [1]. The main storage technology is still based on hard drives, that have shown a better capacity to scale up in terms of storage capacity than speed. On the other hand, the major issue is still on handling I/O peaks, and the overall average usage is still relatively low. For instance, taking the former supercomputer Intrepid at Argonne National Laboratory as an example, the aggregate I/O throughput is lower than one-third of the peak I/O bandwidth for 99% of the time [11, 12]. Nevertheless, applications running regularly perform I/Os, to read their initial data, to write intermediate and final results on reliable storage and to enforce reliability using checkpoint restart techniques. In typical HPC systems, very few applications (usually one) are enough to saturate the bandwidth to the Parallel File System, so that delays are experienced if the transfers are not coordinated. Mechanisms such as Clarisse [21] have thus been introduced to reorganize transfers to the Parallel File System at system level.

On the other hand, the usage of HPC systems makes I/O more and more crucial and complex. First, in the framework on the convergence between HPC and BigData [26], 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 [10, 8] 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 3.2) and they alternate read, compute and write phase to the Parallel File System, that cannot be overlapped. We consider in the present paper both read and write accesses to the Parallel File System.

When running several such applications, even if the overall bandwidth is enough to cope in the long term with the 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 [28]. In this context, in order to cope with the limited I/O bandwidth of HPC system, Burst-Buffers have emerged as promising solution [16, 19, 15].

Burst-Buffers are first used to accelerate transfers and to hide the latency and the limited bandwidth to the Parallel File System, by acting as a cache between the computational nodes and the Parallel File System. In practice, this use is arguable since on the technological side, the use of NVRAM or SSDs makes it possible to achieve much higher bandwidth than hard disks, but at the price of a limited lifetime [18]. Another potential use of Burst-Buffers is to enable to delay and better schedule the I/O operations to the Parallel File System, 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 2.1). In this paper, we consider the simplest model where the Burst-Buffers are not distributed and acts as a potential intermediate centralized layer, with a higher I/O bandwidth but a smaller capacity than the hard disk storage system. It can however be partitioned between the different applications, ensuring that each application has a dedicated allocation.

In the present paper, we therefore consider a set of applications running independently on different computational nodes belonging to the same machine, where the allocation of nodes has been done a priori using a batch scheduler such as SLURM [29]. 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 [5]. Our goal is both to dimension the Burst-Buffers and to partition it between the different applications in order to limit the application slowdown experienced due to the limited I/O bandwidth to the Parallel File System.

The main contributions of this work are the following:

- A precise and complete model of the platform and applications. We propose several uses of Burst-Buffers (static allocation and dynamic allocation).
- 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.
- 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 Parallel File System bandwidth (optimal stretch).
- Finally we provide several evaluations with bounded buffer size to compare the performance of the static and dynamic buffer allocations. We discuss these strategies.

Compared to our previous work [4], we consider other types of usage of the Burst-Buffers. On the system side, we consider the possibility of partitioning the Burst-Buffer either statically (give a fixed share of Burst-Buffer to each application) or dynamically (the share of an application can vary with time). We consider a precise application model, where the different phases (in terms of I/O and computation volumes) are known in advance, whereas [4] relied on random transfer patterns, where only the probability of being involved in a transfer at any point in time was known. This allows us to provide a more precise estimation of the required Burst-Buffer size, by considering actual events when the buffers are emptied and when the I/O operations between applications interfere.

The rest of this paper is organized as follows. In Section 2, we present the related work on Burst-Buffers architecture, bandwidth allocation and HPC applications models. Then, we detail in Section 3 the application model described above and the architecture of the Burst-Buffers. In Section 4, we prove that the problem of partitioning the Burst-Buffers between several applications is NP-Complete in the strong sense. Despite this result, we prove in Section 6 that a special case can be solved in polynomial time, where the goal is to find both the optimal size of the Burst-Buffers and its optimal partitioning between applications so that the completion time of all the applications is exactly the same as what it would be if each application was running alone with a full access to the Parallel File System. We then compare static and dynamic settings through simulation results in Section 7, and discuss the performance of a classic greedy fair sharing strategy. At last, we discuss the current limitations of our model and possible extensions of the present work and provide concluding remarks in Section 8.

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/parted-buffers`.

2 Related Work

2.1 Burst-Buffer Architectures and models

There are many implementations of Burst-Buffers. The two most studied characteristics are the location of the buffers and whether the buffers are shared between multiple applications.

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 [16, 27] and Cray DataWarp [19, 25, 14]. 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. [25] study the case where the buffers are shared between the different applications on the platform and used to accelerate transfers and to prevent I/O congestion. On the contrary, in Schenck et al. [27] and Daley et al. [14], applications decide the size of the buffer that should be dedicated to them.

Another solution is a *distributed* version of Burst-Buffers where the buffers are allocated closer to the compute nodes [22, 7]. A solution consists in allocating the distributed buffers to the applications using the compute nodes close to buffers [9]. However, other strategies focus on how to share them between the different applications [22, 7]. In [25], 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 [7]. Furthermore, in the case where, because of their costs, the number of buffers in the machine has to be limited, one must choose on which node they should be deployed and between which subset of applications they should be shared.

In [28], the authors consider the use of Burst-Buffers to serve the I/O bursts of HPC applications. They prove that a basic reactive draining strategy that empties the Burst-Buffer as soon as possible can lead to a severe degradation of the aggregate I/O throughput. On the other hand, 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. Contrarily to our present work, [28] does not consider the dimensioning nor the optimal partitioning of the Burst-Buffer, but concentrates on dynamic strategies to actually perform transfers.

2.2 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 [27]. For instance, DDN [16] announces bandwidth performance 10-fold that of PFS using their Burst-Buffers. The idea is to move the I/O to the Burst-Buffer as a temporary stage between compute nodes and the Parallel File System (whether the data is incoming or outgoing). Thanks to the higher bandwidth of the Burst-Buffers, this has the advantage of improving 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 was noted by Han et al. [18], this idea is not viable, (i) Burst-Buffers are based on technologies that are extremely expensive with respect to hard drives, (ii) they are currently based on SSD technology, that is known to have a limited rewrite lifespan [18]. Thus, the large number of I/O operations in HPC applications would decrease their lifespan too fast. In the solution we propose, not all data transfers go through the Burst-Buffer 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 [24, 20] while maintaining their lifespan. To achieve this goal, the applications use the direct link to the PFS (see Figure 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-Buffer is used to complement the bandwidth of the PFS. This is one of the solutions advocated by DDN in [16]. 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 applications' (simultaneous) bursts.

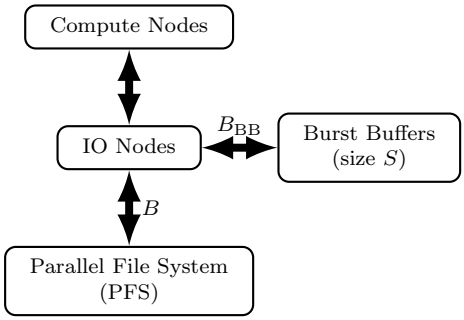

Figure 1: Modeling of the pseudo-centralized platform.

This corresponds to the model depicted in Figure 1 that will be used throughout this paper.

Finally, a large part of the literature on Burst-Buffers shows how to use them with a specific application workflow [14, 27]. Specifically, they consider systems where applications have dedicated and pre-allocated Burst-Buffers, and where the application can explicitly control its data transfers and the use of the Burst-Buffer. This must be done for each application and is very platform dependent. In practice, only few applications have the human-power to implement this. 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. In [14] and [13], the authors analyze workflows used in HPC system (CAMP and SWarp in [14], CyberShake in [13]) 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 [19].

3 Models

In this section we present the models used in this work.

3.1 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). In addition to this file server, the platform is equipped with Burst-Buffers of input bandwidth B_{BB} and size S. Figure 1 provides a schematic view of this model.

In this 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 this application. 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, super-computers at Argonne [17] and Jupiter, a machine at Mellanox [5]. In order to be compliant with the strategies described in [3], we introduce (see Section 6 for details) the possibility of progressively providig Burst-Buffer resources to an application before it starts (to prefetch its input data) and to progressively remove Burst-Buffer resources from an application after it ends (to release processing resources as soon as possible).

3.2 Application Model

We consider scientific applications running simultaneously on 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:

1. Data fetching from disks (read);
2. computations (compute); and
3. data uploading on disks (write).

Formally, the 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 can not 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 if 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 uses its own nodes, the applications compete for sending and receiving data during their I/O phase on a dedicated I/O network, what results in congestions and delays between I/O nodes of the platform and the Parallel File System. We discuss the conditions necessary to reach this lower bound in Section 5.2.

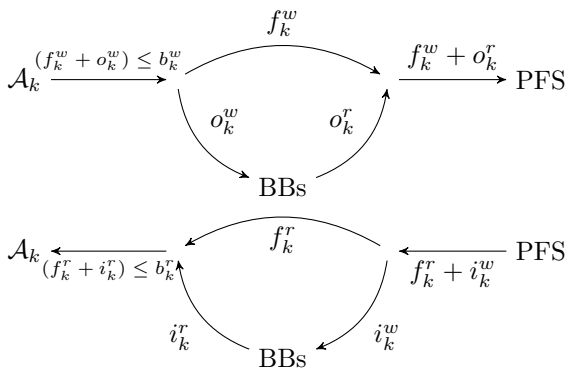

Figure 2: Schematics of the bandwidth used for *output* (top) and *input* (bottom).

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:

1. The beginning of the read *to the application* that coincides with the end of the previous write *from the application* denoted $t^r_{k,i}$ (with $t^r_{k,1} = r_k$).

2. The beginning of the compute of the application that coincides with the end of the read *to the application* denoted $t^c_{k,i}$.

3. The beginning of the write *from the application* that coincides with the end of the computation phase denoted $t^w_{k,i}$.

We consider that above phases coincide without loss of generality, since the amount of 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 Figure 2):

- Part of its output (write) data is sent to the PFS at rate f^w_k, and the rest to the Burst-buffers at rate o^w_k. This is a part of the $W_{k,i}$ phase.

- Part of its input (read) data is collected from the PFS at rate f^r_k, and the rest from the Burst-buffers at rate i^r_k. This is a part of the $R_{k,i}$ phase.

By definition, the following properties on $i^r_k + f^r_k$ (resp. $o^w_k + f^w_k$) hold:

1. It is only non zero on the intervals $[t^r_{k,i}, t^c_{k,i}]$ (resp. $[t^w_{k,i}, t^r_{k,i+1}]$);

2. It is bounded by b^r_k (resp. b^w_k); and

3. $\int_{t^r_{k,i}}^{t^c_{k,i}} f^r_k(t)dt = R_{k,i}$ (resp. $\int_{t^w_{k,i}}^{t^r_{k,i+1}} f^w_k(t)dt = W_{k,i}$).

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

3.3 Optimization problem

We are now interested by the performance model of our 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:

- Without burst-buffers, the I/O operations take a time of

$$\frac{R_{k,i}}{\min(B, b_k^r)} \text{ and } \frac{W_{k,i}}{\min(B, b_k^w)}$$

- 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 its previous iterations. 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 execute $W_{k,i}$, the application sends the 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

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

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.

3.4 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. In general, the description of these functions over time is not polynomial in the size of the input problem. In this section, we prove that we can focus on strategies where each of these functions 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.

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:*

- *all applications have the same stretch as $\mathcal{S}$;*
- *the total buffer size used is the same as $\mathcal{S}$;*
- *between any two events$(t_{k,i}^r, t_{k,i}^c t_{k,i}^w)_{i,k}$ of $\tilde{\mathcal{S}}$, all functions $f \in \tilde{sched}$, are constant.*

Proof. Let us denote by $e_0 < \cdots < e_n$ the list of events of $\mathcal{S}$. For $f \in \mathcal{S}$, let us define $\tilde{f} \in \tilde{\mathcal{S}}$ by: $\forall i, \tilde{f} : x \in [e_i, e_{i+1}] \mapsto \frac{\int_{e_i}^{e_{i+1}} f(t)dt}{e_{i+1}-e_i}$. This transformation satisfies the following constraints:

- Application-specific constraints:
  - All the data transfer necessary for a read/write phase is performed during those phases;
  - The maximum application bandwidths of $\tilde{\mathcal{S}}$ are never larger than b_k^r and b_k^w;
- PFS-specific constraint:
  - The total PFS bandwidth is never larger than B;
- Buffer-specific constraints:
  - The total buffer peak is never larger than S ($X = \textsc{Dynamic}$);
  - The sum of individual buffer peaks is never larger than S ($X = \textsc{Static}$).
  - The data leaving the buffer is not larger than the data entering the buffer.

Let us first note that $\forall f \in \mathcal{S},\ \forall j$:

$$\max_{[e_j, e_{j+1}]} \tilde{f} \leq \max_{[e_j, e_{j+1}]} f. \tag{4}$$

since $\tilde{f}$ is the average of f on those intervals.

Application-specific constraints To check that the solution is valid, we need to check that for all $R_{k,i}$, all necessary data for the read phase is actually read:

$$\int_{t_{k,i}^r}^{t_{k,i}^c} \left(\tilde{f_k^r}(t) + \tilde{i_k^r}(t)\right) dt = R_{k,i}.$$

Denote j_0 and j_1 s.t. $e_{j_0} = t_{k,i}^r$ and $e_{j_1} = t_{k,i}^c$.

$$\begin{aligned}
\int_{e_{j_0}}^{e_{j_1}} \left(\tilde{f_k^r}(t) + \tilde{i_k^r}(t)\right) dt &= \sum_{i=j_0}^{j_1-1} \int_{e_i}^{e_{i+1}} \tilde{f_k^r}(t) + \tilde{i_k^r}(t)dt \\
&= \sum_{i=j_0}^{j_1-1} \int_{e_i}^{e_{i+1}} \frac{\int_{e_i}^{e_{i+1}} f_k^r(x) + i_k^r(x)dx}{e_{i+1}-e_i} dt \\
&= \sum_{i=j_0}^{j_1-1} \int_{e_i}^{e_{i+1}} f_k^r(x) + i_k^r(x)dx \\
&= R_{k,i}
\end{aligned}$$

The last equality comes from the fact that $\mathcal{S}$ is a valid solution. Similarly we obtain that for $W_{k,i}$ we need $\int_{t_{k,i}^w}^{t_{i+1}^r} \tilde{f_k^w}(t) + \tilde{o_k^w}(t)dt = W_{k,i}$.

In addition, from Eq. (4) and because $\mathcal{S}$ is a valid schedule, one can check that for all j:

$$\max_{[e_j, e_{j+1}]} \tilde{f_k^r} + \tilde{i_k^r} \leq \max_{[e_j, e_{j+1}]} f_k^r + i_k^r \leq b_k^r$$
$$\max_{[e_j, e_{j+1}]} \tilde{f_k^w} + \tilde{o_k^w} \leq \max_{[e_j, e_{j+1}]} f_k^w + o_k^w \leq b_k^w$$

PFS-specific constraint From Eq. (4), and because $\mathcal{S}$ is a valid schedule, one can verify that for all j:

$$\max_{[e_j,e_{j+1}]} \sum_k \tilde{f_k^r} + \tilde{i_k^w} + \tilde{f_k^w} + \tilde{o_k^r} \leq \max_{[e_j,e_{j+1}]} \sum_k f_k^r + i_k^w + f_k^w + o_k^r \leq B$$

Buffer-specific constraint The occupation of the buffer at time $x \geq e_0$ due to data from $\mathcal{A}_k$ is given by

$$\int_{e_0}^{x} \left(\left(\tilde{o_k^w}(t) - \tilde{o_k^r}(t)\right) + \left(\tilde{i_k^w}(t) - \tilde{i_k^r}(t)\right)\right) dt$$

For any j, let us define by $S_k^{(j)}$ the volume of data in the buffer of $\mathcal{A}_k$ at time e_j:

$$S_k^{(j)} = \int_{e_0}^{e_j} \left(o_k^w(t) - o_k^r(t)\right) + \left(i_k^w(t) - i_k^r(t)\right) dt.$$

Let $x \in [e_{j_0}, e_{j_0+1}]$:

$$\begin{aligned}
&\int_{e_0}^{x} \left(\left(\tilde{o_k^w}(t) - \tilde{o_k^r}(t)\right) + \left(\tilde{i_k^w}(t) - \tilde{i_k^r}(t)\right)\right) dt \\
&= \sum_{j=0}^{j_0-1} \int_{e_j}^{e_{j+1}} \left(\left(\tilde{o_k^w}(t) - \tilde{o_k^r}(t)\right) + \left(\tilde{i_k^w}(t) - \tilde{i_k^r}(t)\right)\right) dt + \int_{e_{j_0}}^{x} \left(\left(\tilde{o_k^w}(t) - \tilde{o_k^r}(t)\right) + \left(\tilde{i_k^w}(t) - \tilde{i_k^r}(t)\right)\right) dt \\
&= \sum_{j=0}^{j_0-1} \int_{e_j}^{e_{j+1}} \left(\left(o_k^w(t) - o_k^r(t)\right) + \left(i_k^w(t) - i_k^r(t)\right)\right) dt + (x - e_{j_0}) \frac{\int_{e_{j_0}}^{e_{j_0+1}} \left(\left(o_k^w(t) - o_k^r(t)\right) + \left(i_k^w(t) - i_k^r(t)\right)\right) dt}{e_{j_0+1} - e_{j_0}} \\
&= S_k^{(j_0)} + \frac{x - e_{j_0}}{e_{j_0+1} - e_{j_0}} \left(S_k^{(j_0+1)} - S_k^{(j_0)}\right).
\end{aligned}$$

In the Dynamic case, one can check that $\forall x \in [e_j, e_{j+1}]$,

$$\begin{aligned}
\sum_k \int_{e_0}^{x} \left(\left(\tilde{o_k^w}(t) - \tilde{o_k^r}(t)\right) + \left(\tilde{i_k^w}(t) - \tilde{i_k^r}(t)\right)\right) dt &= \sum_k \left(S_k^{(j)} + \frac{x - e_j}{e_{j+1} - e_j} \left(S_k^{(j+1)} - S_k^{(j)}\right)\right) \\
&\leq \max\left(\sum_k S_k^{(j)}, \sum_k S_k^{(j+1)}\right) \leq B.
\end{aligned}$$

Similarly, in the Static case, for any application k, $\forall x \in [e_j, e_{j+1}]$:

$$\int_{e_0}^{x} \left(\left(\tilde{o_k^w}(t) - \tilde{o_k^r}(t)\right) + \left(\tilde{i_k^w}(t) - \tilde{i_k^r}(t)\right)\right) dt \leq \max\left(S_k^{(j)}, S_k^{(j+1)}\right) \leq S_k.$$

With the same reasoning, one can also check that $\forall x \geq e_0$:

$$\int_{e_0}^{x} \left(\tilde{o_k^w}(t) - \tilde{o_k^r}(t)\right) dt \geq 0$$

$$\int_{e_0}^{x} \left(\tilde{i_k^w}(t) - \tilde{i_k^r}(t)\right) dt \geq 0$$

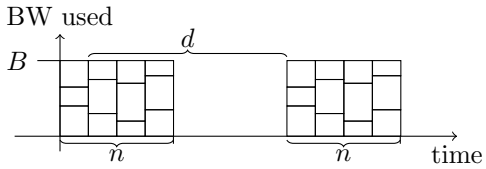

Figure 3: Communication schedule obtained from a positive 3-PART instance

ensuring that in the new schedule, the data does not leave the buffer before it enters it.

This concludes the proof that the solution $\tilde{\mathcal{S}}$ is valid, and that its performance is identical to $\mathcal{S}$ (same time events, same buffer sizes). □

In the future, we therefore restrict the search to Dominant Schedules.

4 Complexity Results

In this Section, 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.*

The reduction we consider is adapted from [17] where it is applied to a the problem of scheduling periodic applications.

Proof. We consider the associated decision problem: given a set of K applications $\mathcal{A}_k$ and a platform, does there exist 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 $3n$ 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.

Assume first that $\mathcal{I}_1$ has a solution. Let us call $I_1, \cdots, I_n$ the n triplets of $\mathcal{I}_1$. By definition we have $\sum_{i \in I_t} a_i = B$.

We construct the followint 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 rate a_k. Then $W_{k,1}$ is scheduled from time $t+d$ to time $t+d+1$ at maximum rate a_k (see Figure 3).

It is a valid solution for $\mathcal{A}_k$ with respect to the read, compute and 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 IO bandwidth constraint.

Assume now that $\mathcal{I}_2$ has a solution. By definition of the stretch, the latest an application can terminate is at time $d + n + 1$.

There cannot be any I/O movement between time n and $d + 1$:

- Write data are not ready yet: the minimum time needed is $d + 1$ units of time for any application;
- 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 finish at time $t \leq n$ since it takes a minimum time of $d + 1$ to do the rest of the computation.

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 occured 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 showing the result. □

Let us consider any time interval $[i, i + 1]$ for $i \leq n - 1$, then the communication link must be fully used, 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 happens provides a valid solution to the 3-Part problem. □

Theorem 3. *For any $S \geq 0$,* Static-Stretch(S) *is NP-complete.*

Proof. We can extend the previous reduction from 3-Part with an additional application. With the same notations as above, we 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, we ensure that $d \geq n + 1$, 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 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 the other applications. However, during this time, only an amount $B\rho$ of data can be sent to the PFS ; the rest of the write phase of $\mathcal{A}_{n+1}$ needs to be sent to the burst buffer. Thus in a solution of stretch ρ and burst buffer size S, all the buffer size needs to 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. □

5 Burst-Buffer Lower Bounds for the Execution of a Single Application

In Section 3, we have provided a lower bound on the execution time of a single application given its characteristics:

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

In general this lower bound is not reachable. Indeed, to be able to reach it, $\mathcal{A}_k$ needs to read and write at maximum bandwidth during 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 [3], Burst-Buffers are expected to accelerate applications by (i) accelerating the transfers to and from the Parallel File System 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 Parallel File System 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.

5.1 Description of a solution achieving optimal makespan

Let us consider application $\mathcal{A}_k$. Let $C_k^{\min}$ be the lower bound of C_k, the makespan of $\mathcal{A}_k$ as defined in Eq (2). We study a solution that achieves the optimal makespan $C_k^{\min} = r_k + \sum_{i=1}^{n_k} \frac{R_{k,i}}{b_k^r} + l_{k,i} + \frac{W_{k,i}}{b_k^w}$.

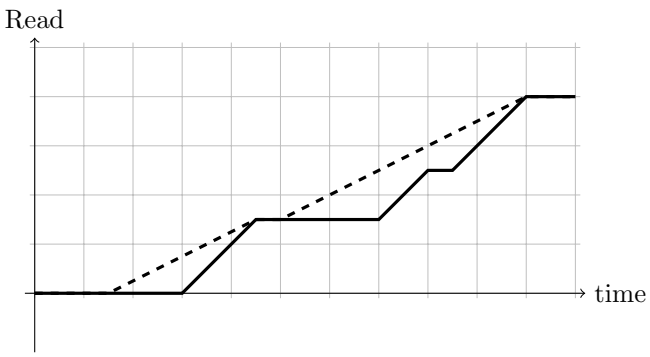

Figure 4: Data read by $\mathcal{A}_k$ (when running in isolation) from the Parallel File System when assuming $B = +\infty$ (solid line) and when using a Burst-Buffer (dashed line)

Let us consider the case where there are no constraints on the bandwidth to the Parallel File System, *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:

1. $t_{k,i}^r = r_k + \sum_{j=1}^{i-1} \frac{R_{k,j}}{b_k^r} + l_{k,j} + \frac{W_{k,j}}{b_k^w}$. 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.

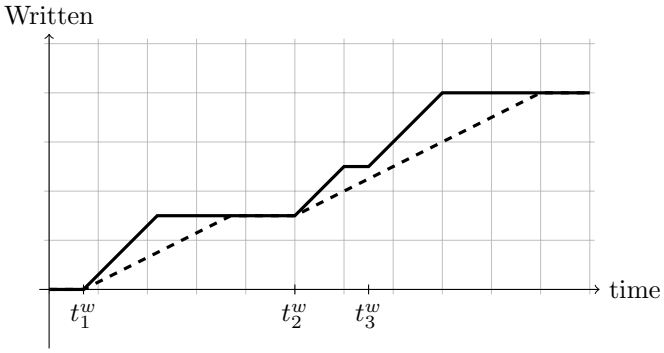

Figure 5: Data written by $\mathcal{A}_k$ (when running in isolation) onto the Parallel File System when assuming $B = +\infty$ (solid line) and when using a Burst-Buffer (dashed line)

2. $t_{k,i}^c = r_k + \sum_{j=1}^{i-1} \frac{R_{k,j}}{b_k^r} + l_{k,j} + \frac{W_{k,j}}{b_k^w} + \frac{R_{k,i}}{b_k^r}$. The compute phase lasts $l_{k,i}$ units of time.

3. $t_{k,i}^w = r_k + \sum_{j=1}^{i-1} \frac{R_{k,j}}{b_k^r} + l_{k,j} + \frac{W_{k,j}}{b_k^w} + \frac{R_{k,i}}{b_k^r} + l_{k,i}$. 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. 5). The slope of $W_k^\infty(t)$ is therefore either 0 (during read and compute phases) or b_k^w during write phases.

5.2 Burst-Buffer size necessary to $\mathcal{A}_k$ 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 rate trivially achieves $C_k^{\min}$ and Burst-Buffer are only needed if several applications are running simultaneously, what will be considered in Section 6).

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-Buffer. In this case, in order to achieve $C_k^{\min}$, $\mathcal{A}_k$ must use Burst-Buffer resources, and our goal is this section is to find the minimal Burst-Buffer size, denoted as P_k, so as to achieve such an exectution 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 Parallel File System to the Burst-Buffer in advance.

Let us first concentrate on the write operations onto the Parallel File System of an isolated application, in presence of a limited bandwidth B_w to the PFS. The dashed line in Figure 5 depicts the volume of data written to the Parallel File System. Writing the data of the first writing phase to the PFS cannot start before time step $r_k + l_{k,1} + R_{k,1}/b_k^r$ 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 $r_k + l_{k,1} + R_{k,1}/b_k^r + W_{k,1}/b_k^w$. At that date, given the limited bandwidth B_w to write data onto the Parallel File System, the Burst-Buffer is used to store the data that could be written to the Burst-Buffer (at rate b_k^w) but not on the Parallel File System (at rate B_w). At any time step, the solid lie represents the overall volume of data sent by $\mathcal{A}_k$ (either to the Burst-Buffer or the Parallel File System) and the difference between the solid and dashed plots represent the minimal volume of data that must be stored onto the Burst-Buffer. After time $r_k + l_{k,1} + R_{k,1}/b_k^r + W_{k,1}/b_k^w$, the Burst-Buffer is being emptied onto the Parallel File System, at rate B_w following the model described in Section 3. This transfer stops either when the Burst-Buffer 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-Buffer 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-Buffer 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 symmetric and is depicted in Figure 4, where the solid plot depicts the volume of data that must be read by the nodes running $\mathcal{A}_k$ and the dashed plot depicts the volume of data that must be read from the Parallel File System (and sent either to $\mathcal{A}_k$ nodes or prefetched on the Burst-Buffer). The algorithm to build the dashed plot is very similar to the algorithm described above in the case of Figure 5, and the necessary Burst-Buffer size for read operations to run $\mathcal{A}_k$ in isolation with minimal makespan is given by the maximal difference between the plots on Figure 4.

In these derivations, we have considered that it is possible to fully use bandwidth B_w (resp. B_r) when writing on (resp. reading from) the Parallel File System. 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. Although this problem is more complicated, it can be solved in polynomial time by solving a Linear Program in rational numbers. Let us denote by e_l^k the ordered set of events (see Section 3 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 $w_k^{\text{PFS}}(e_l^k)$ (resp. $r_k^{\text{PFS}}(e_l^k)$) the overall volume of data written by the application to the Parallel File System (resp. read from the Parallel File System either to the Burst-Buffer or the application) before time e_l^k. Let us also denote by $\textsc{Peak}_k$ the size of the Burst-Buffer necessary to handle both write and read operations to the Burst-Buffer.

Then, our goal is to minimize $\textsc{Peak}_k$ under the following constraints:

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

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, nor the write and the overall bandwidths are exceeded. At last, the last constraint states that the Burst-Buffer size $\textsc{Peak}_k$ is enough to store both the amount of data written to the Burst-Buffer but not yet to the Parallel File System and the amount of data read from from the Parallel File System but not yet transmitted to the computation nodes.

6 Burst-Buffer Size to compensate for contention between multiple applications

We have seen in Section 4 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 detailled in Section 6.2.

6.1 Data transfers from and to the Parallel File System in presence of a Burst-Buffer

Let us now compute the minimal Burst-Buffer size S^* necessary to achieve completion time $C_k^{\min}, \forall k$, even if all applications compete for the bandwidth to the Parallel File System. In this case, the Burst-Buffer will also be used to avoid contentions to the Parallel File System, by prefetching (before later reading) or storing (before later sending) data eventually to be read from or written to the Parallel File System.

The solid plot in Figure 6 is analogous to the solid plot in Figure 5 and depicts the evolution with time of the data volume that must be written by $\mathcal{A}_k$, either to the Parallel File System or to the Burst-Buffer, and is denoted by $W_k^{\infty}(t)$. On the other hand, the dashed plot in Figure 5 depicts the evolution of the data actually written to the Parallel File System in presence of a Burst-Buffer of size S_k^w, and is denoted by $W_k^{S_k}(t)$. Indeed, let us now consider the evolution with time of the minimal volume of data that can be sent to the Parallel File System in presence of a Burst-Buffer of size S_k^w, when achieving the same makespan $C_k^{\min}$. At time $C_k^{\min}$, all data produced by $\mathcal{A}_k$ to be written on the Parallel File System must have been transferred either to the Burst-Buffer or to the Parallel File System, since the nodes allocated to $\mathcal{A}_k$ must be released. Therefore, at least $\sum W_{k,i} - S_k^w$ must have been written to the PFS, since the Burst-Buffer can hold at most S_k^w volume. The same holds at all the instants between $C_k^{\min} - W_{l_k}^k / b_k^w$ and $C_k^{\min}$ and the volume transferred to the Parallel File System between instants $C_k^{\min} - W_{l_k}^k / b_k^w - R_{l_k}^k / b_k^r$ is at least $\sum^{l_k - 1} W_{k,i} - S_k^w$.

Using a trivial induction, we can prove that before instant $C_k^{\min}$, the minimal volume sent to the Parallel File System is given by the dashed plot in Figure 6 and in what follows, we will denote by $W_k^{S_k^w}(t)$ the value of the dashed plot at instant t. After time $C_k^{\min}$, the data to be written to the Parallel File System that still reside in the Burst-Buffer must eventually be transferred to the Parallel File System to release space on the Burst-Buffer. The corresponding amount of Burst-Buffer storage is progressively released by $\mathcal{A}_k$ during this transfer, and can be used by other applications. As previously stated, we search for a solution such that all tasks complete within time $Ckmin$ but note that in our model, the Burst-Buffer is considered as a safe storage and may therefore hold data that should be eventually written to the Parallel File System. This emptying strategy is very similar to the ultimate draining strategy described in [3].

On the other hand, 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 Figure 6) 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 Parallel File System in a valid strategy that makes use of a Burst-Buffer of size at most S_k^w. Indeed, let us denote by $W_k^{BB}(t)$ the volume written on the Burst-Buffer at time t, *i.e.* $W_k^{BB}(t) = W_k^{\infty}(t) - W_k(t)$. Then, since $W_k(t)$ is increasing and the slope of $W_k^{\infty}(t)$ is at most b_k^w, then the slope of $W_k^{BB}(t)$ is no more than b_k^w, that corresponds to the maximal bandwidth to the Burst-Buffer. Then, at any any instant the sum of the slopes of W_k^{BB} (that may be either positive, when the Burst-Buffer is filled, or negative, when the Burst-Buffer is emptied to the Parallel File System) and $W_k(t)$ is equal to the slope of $W_k^{\infty}(t)$, so that at any instant, 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-Buffer is depicted in the solid plot in Figure 7 (identical to the solid plot in Figure 4) and we will denote by $R_k^{\infty}(t)$ the corresponding value, that represents the minimal amount of data that must be read

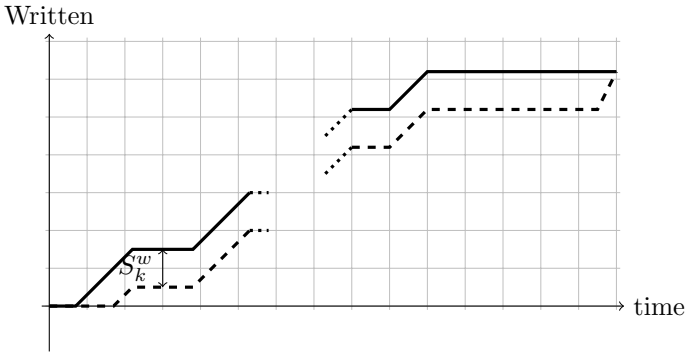

Figure 6: Data transfer to the Parallel File System (solid line without contention, dashed line with a burst buffer).

from the Parallel File System in order to achieve optimal makespan when there is no constraint on the bandwidth to the Parallel File System ($B_r = +\infty$). 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-Buffer 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, $R_k^{BB}(t) - R_k^{\infty}(t)$ represents the amount of data that will eventually by transferred to computing nodes and that resides in the Burst-Buffer at time t.

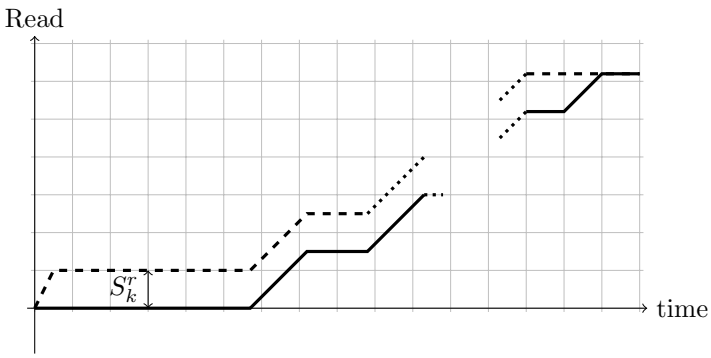

Figure 7: Data transfer from the Parallel File System (solid line without contention, dashed line with a burst buffer).

6.2 Linear Program to Compute the Optimal Burst-Buffer size (Static Case)

As already noted, we can consider two different settings for the partitioning of the Burst-Buffer. In the Static case, a buffer of size S_k is allocated to $\mathcal{A}_k$ during all its lifetime, whereas in the Dynamic case, the amount of Burst-Buffer allocated to $\mathcal{A}_k$ can vary over time. In both cases, we assume that at any given time step, the volume S_k of the Burst-Buffer allocated to $\mathcal{A}_k$ can be shared between prefetch (size S_k^r) and intermediate storage (size S_k^w) usages. In what follows,

we will therefore rather consider a Burst-Buffer 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 [3].

Let us denote (see Section 3 for a formal definition) the set of points in Figures 6 and 7 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 do not depend on the values of S_k^r and S_k^w, $\forall k$. As previously, we will denote by e_l the time of the l-th event, *i.e.* $e_l \leq e_{l+1}$. In what follows, we define a set of linear constraints that must be satisfied by functions $R_k^\infty(t)$ and $W_k^\infty(t)$ when $t \in \{e_l\}$ and we rely on Theorem 1 to rebuild a valid strategy for $R_k^\infty(t)$ and $W_k^\infty(t)$, $\forall t$ by using linear interpolation for all functions between the instants $t \in \{e_l\}$.

Therefore, the following set of linear constraints $\mathcal{S}_{\text{STATIC}}$ describes the set of valid solutions:

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

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-Buffer allocated to $\mathcal{A}_k$ is exactly S_k. On the other hand, as previously stated, the Burst-Buffer is progressively allocated to $\mathcal{A}_k$ before r_k and progressively released from $\mathcal{A}_k$ after $C_k^{\min}$.

We can use $\mathcal{S}_{\text{STATIC}}$ to optimize FINDOPTIMALSIZE^{STATIC}: Minimize S under the constraints $\mathcal{S}_{\text{STATIC}}$.

6.3 Linear Program to Compute the Optimal Burst-Buffer Size (Dynamic Case)

The linear program to compute the optimal Burst-Buffer size in the DYNAMIC case is very similar. We obtain $\mathcal{S}_{\text{DYNAMIC}}$ by removing from $\mathcal{S}_{\text{STATIC}}$ the constraint $\forall k, l \in \mathcal{Z}_k, S_k^l \leq S_k$, that states that the size of the Burst-Buffer allocated to $\mathcal{A}_k$ cannot change and remains equal to S_k when nodes are actually allocated to $\mathcal{A}_k$ (*i.e.* $\forall l \in \mathcal{Z}_k$). This leads to optimization problems FINDOPTIMALSIZE^{DYNAMIC}: Minimize S under the constraints $\mathcal{S}_{\text{DYNAMIC}}$.

7 Simulation Results

In this Section, we report extensive simulations to evaluate our linear programming formulations, and compare them with a classic fair sharing approach. In order to do so we instantiate our evaluation based on characteristics of the Intrepid platform, and based on a set of applications as described in our previous work [4].

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 1: Characteristics of the applications in APEX data set.

7.1 Setup

We consider a set of 4 applications described in the APEX report [23] which represent the majority of the load at the LANL. The characteristics of these applications are provided in Table 1. 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 of these applications come from periodic checkpoints. We estimate the checkpointing period using the checkpoint optimal period given by $P = \sqrt{2 * C * \frac{\mu}{\#\text{ cores}}}$, following [6]. 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 1, with a probability proportional to its usage ratio as reported in [23] (Frequency in Table 1). 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 in the following way. Let us first remark that an application with checkpoint size s and period P induces an average bandwidth load of $\frac{s}{P}$ over the course of its execution. Then, when an application starts or ends, the total required bandwidth is updated, and the maximum value over time (normalized by B) provides the IO need induced by running applications. Once this IO need is computed, the checkpoint sizes of all the applications are multiplied by a constant factor (and the checkpointing periods are adequately recomputed) so as to obtain the targeted IO load. To number of cores needed by each application can be read in Table 1 and it can be used to determine the maximal bandwidth b_k at which a given application can communicate with the Burst-Buffer and the Parallel File System (see Section 3). To model the processing time between two checkpoints, we add to the checkpointing period an additional 15% variability.

Therefore, using the values from the APEX data set [23] and summarized in Table 1 and the description of the Intrepid platform, we are able to instantiate all platform and application characteristics needed by the linear programs of Section 6. By changing the MTBF (from 5 years to 50 years) and by scaling checkpoint sizes, we are moreover able to study different hardware characteristics and different system loads, while keeping realistic application characteristics. In turn, the linear programs of Section 6 compute an optimal buffer size S_{OPT} (both in static and dynamic cases) and its partitioning among applications (in the static case) to process all the applications with a maximum stretch of 1, *i.e.* as if the bandwidth to the Parallel File System was enough to cope with all transfers at any time.

In order to evaluate the influence of the size of the Burst-Buffer, in what follows, we consider

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

Table 2: Ratio of static to dynamic buffer sizes, for different MTBF and load values.

several Burst-Buffer sizes, ranging from $0 \times S_{\text{OPT}}$ to $3 \times S_{\text{OPT}}$. To compare the results of the optimal solutions computed in Section 6 to what could be achieved using a dynamic system level strategy, we introduce a greedy strategy, , that shares the bandwidth to the Parallel File System 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-Buffer (of size $0 \leq S \leq S_{\text{OPT}}$) is partitioned between applications proportionally to their respective share in the optimal solution computed by the LP. We then compute the maximum and average stretch for the applications, to evaluate the slowdown incurred by the non-optimal bandwidth sharing and sub-optimal Burst-Buffer size.

7.2 Results

The performance of the above greedy system level approach, that shares equally the bandwidth to the Parallel File System in case of conflicts to the resource access but partitions the Burst-Buffer as in the solution of the LP, is depicted in Figure 8 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 8 corresponds to 10 executions with different sets of applications, where the length of the processing phases between checkpoints vary from an execution to another (with a 15% maximal variation), and the darker area shows the typical variability of the results. As expected, the stretch decreases when the Burst-Buffer increases, but only up to a certain size. If a size of $1 \times S_{\text{OPT}}$ is enough to achieve a solution with stretch 1 for all applications when using the solution of the LP proposed in Section 6, a size of $1.5 \times S_{\text{OPT}}$ is in general required to achieve the lowest stretch using the greedy strategy. Moreover, in particular for large MTBF (thus larger and rarer checkpoints) 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 for any buffer size below $1 \times S_{\text{OPT}}$, these plots show that the value S_{OPT} returned by the LP is crucial in order to set the size of the Burst-Buffer to a value that does not induce large stretch values while limiting the hardware cost.

The cost induced by opting for a static partitioning is analyzed in Table 2, that displays the ratio between the Burst-Buffer size required when using a static partitioning and the Burst-Buffer 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-Buffer 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.

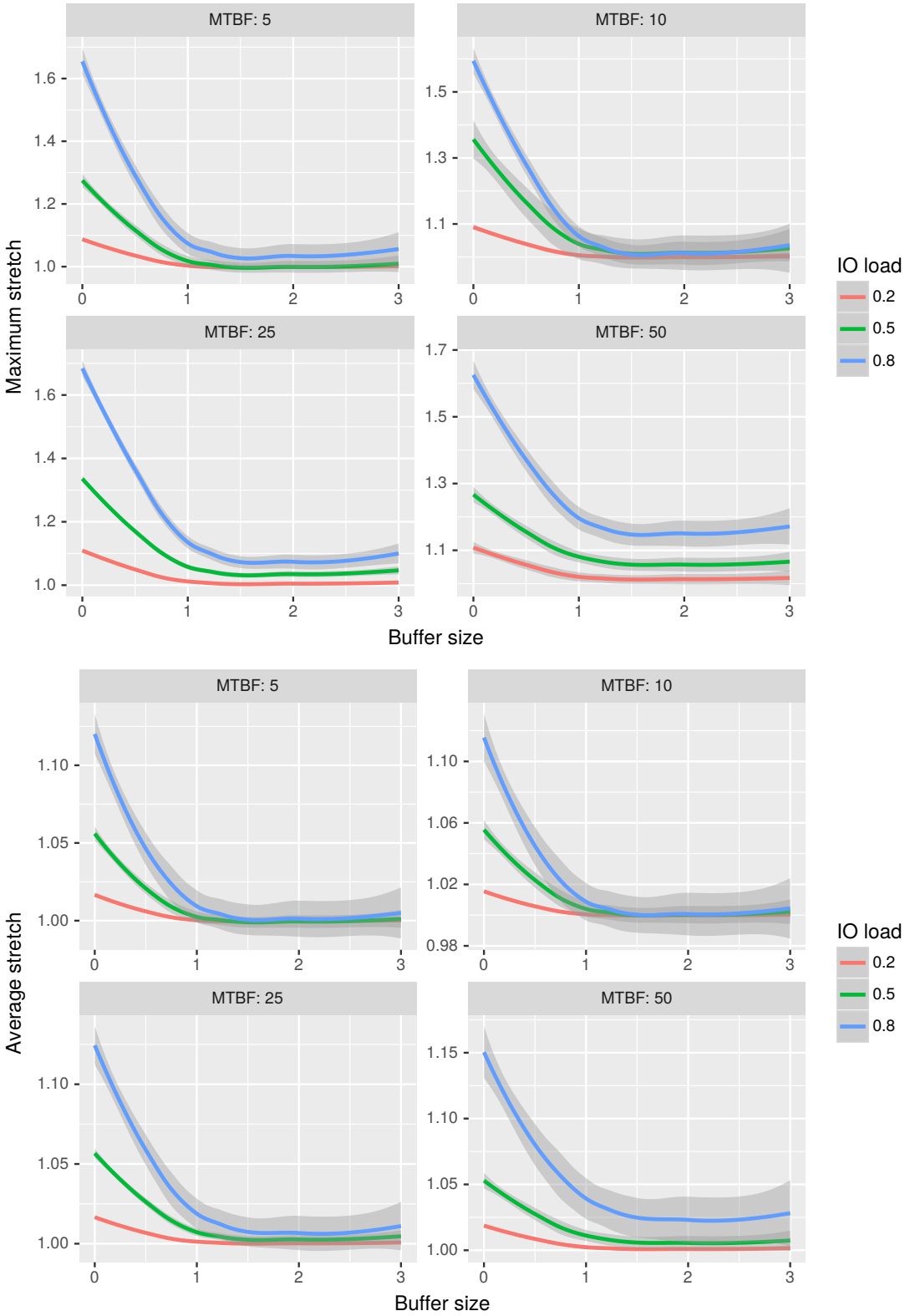

Figure 8: Maximum stretch results (top) and average stretch results (bottom) of fair sharing when buffer size varies, for different MTBF and load values.

8 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 Parallel File System. In this context, our goal is to minimize the stretch experienced by the applications. Given the characteristics of the platform and of the application, we first prove a negative result stating that the problem of minimizing the stretch given a Burst-Buffer size is in general NP-Complete. Nevertheless, we also prove that the special case, of clear practical interest, where the goal is to find the minimal Burst-Buffer size and its partitioning between applications that are able to compensate both the limited bandwidth to the Parallel File System and the contention in the access to it, can be solved in polynomial time. At last, we prove that it is possible to derive from this optimal solution a simple dynamic strategy, that can be easily implemented at runtime, and that is able to achieve low stretches for most settings. Our study also enables to precisely assess the cost of partitioning the Burst-Buffer 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-Buffer is large enough, the problem of finding efficient strategies when the Burst-Buffer 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 behavior 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 buffer until the next checkpoint if space allows it, instead of being written to the PFS.

Acknowledgments

This work was partially supported in part by the French National Research Agency (ANR) in the frame of DASH (ANR-17-CE25-0004), and the "Investments for the future" Program IdEx Bordeaux – SysNum (ANR-10-IDEX-03-02). We would like to thank Gael Goret and his colleagues from the Data Management team at Bull/ATOS Grenoble.

References

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

[2] The trinity project. `http://www.lanl.gov/projects/trinity/`.

[3] Trinity / NERSC-8 Use Case Scenarios. Research Report SAND 2013-2941, Los Alamos National Laboratory, Sandia National Laboratories, Feb. 2013.

[4] 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)*, pages 660–669. IEEE, 2018.

[5] 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.

[6] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni. Checkpointing algorithms and fault prediction. *Journal of Parallel and Distributed Computing*, 74(2):2048–2064, 2014.

[7] 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*, page 32. ACM, 2011.

[8] 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*, 26(17):2772–2791, 2014.

[9] 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*, page 4. Society for Computer Simulation International, 2017.

[10] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[11] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Storage access characteristics of computational science applications. In *MSST*, 2011.

[12] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)*, 7(3):8, 2011.

[13] 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*, page 2. ACM, 2017.

[14] 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.

[15] 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.

[16] DDN Storage. BURST BUFFER & BEYOND, I/O & Application Acceleration Technology. `https://www.ddn.com/download/resource_library/brochures/technology/IME_FAQ.pdf`, 2014. Online; accessed 20 October 2017.

[17] 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*, pages 1013–1022. IEEE, 2015.

[18] 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 (CLUSTER), 2017 IEEE International Conference on*, pages 245–255. IEEE, 2017.

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

[20] 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*, pages 69–80. ACM, 2016.

[21] 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*, pages 346–355. IEEE, 2016.

[22] 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*, page 4. ACM, 2012.

[23] S. LANL, NERSC. Apex workflows. Technical report, Technical report, LANL, NERSC, SNL, 2016.

[24] 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*, pages 1–11. IEEE, 2012.

[25] 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 (CLUSTER), 2017 IEEE International Conference on*, pages 204–215. IEEE, 2017.

[26] D. A. Reed and J. Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, 2015.

[27] 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*, 50(1):12–26, 2017.

[28] 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*, pages 87–98. IEEE, 2017.

[29] 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*, pages 44–60. Springer, 2003.

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

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

ISSN 0249-6399