Implementation of an unbalanced I/O Bandwidth Management system in a Parallel File System

Clément Barthélemy, Francieli Zanon Boito, Emmanuel Jeannot, Guillaume Pallez, Luan Teylo

Implementation of an unbalanced I/O Bandwidth Management system in a Parallel File System

Clément Barthélemy, Francieli Boito, Emmanuel Jeannot, Guillaume Pallez and Luan Teylo

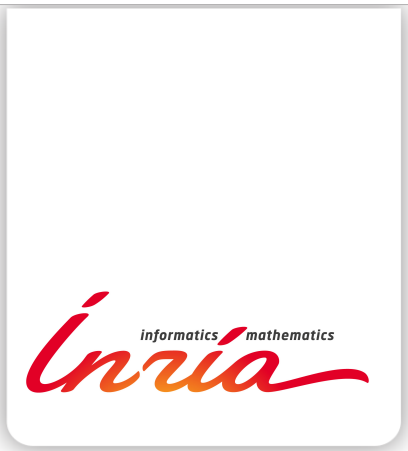
RESEARCH REPORT

N° 9537

January 2024

Project-Team TADaaM

ISSN 0249-6399 ISRN INRIA/RR--9537--FR+ENG



Implementation of an unbalanced I/O Bandwidth Management system in a Parallel File System

Clément Barthélemy, Francieli Boito, Emmanuel Jeannot, Guillaume Pallez* and Luan Teylo†‡

Project-Team TADaaM

Research Report n° 9537 — January 2024 — 15 pages

Abstract: In this report, we discuss a BeeGFS-based implementation of the IO-Sets method for I/O scheduling in HPC systems. I/O scheduling is a technique to mitigate contention in the access to the shared parallel file system. Despite its popularity in the literature, most proposed I/O scheduling techniques are *not* used in practice due to the difficulty of obtaining the needed information and due to overhead concerns. Compared to other techniques, IO-Sets uses little information on the applications, which makes it attractive to be used in practice. In this context, we chose to implement it inside the parallel file system to make it transparent to applications, and *without* a centralized control. That means relaxing the method, but mitigates its limitations regarding overhead and fault tolerance. We discuss the implementation in details and present preliminary results.

Key-words: high performance computing, parallel I/O, I/O scheduling

* Pallez is with INRIA Rennes, Univ. Rennes - KerData
† Barthélemy, Boito, Jeannot and Teylo are with INRIA Bordeaux, Université de Bordeaux, LaBri - TADaaM
‡ e-mail: {clement.barthelemy, francieli.zanon-boito, emmanuel.jeannot, guillaume.pallez, luan.teylo} @inria.fr

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Implémentation d'un ordonnancement déséquilibré d'E/S dans un système de fichiers parallèle

Résumé : Dans ce rapport, nous présentons une implémentation basée sur BeeGFS de la méthode IO-Sets pour l'ordonnancement d'E/S dans les systèmes HPC. L'ordonnancement d'E/S est une technique qui permet d'atténuer la contention dans l'accès au système de fichiers parallèle partagé. Malgré leur popularité dans la littérature, la plupart des techniques d'ordonnancement d'E/S proposées ne sont pas utilisées en pratique en raison de la difficulté d'obtenir les informations nécessaires et des surcoûts. Par rapport à d'autres techniques, IO-Sets utilise peu d'informations sur les applications, ce qui rend son utilisation pratique attrayante. Dans ce contexte, nous avons choisi de la mettre en œuvre à l'intérieur du système de fichiers parallèle pour la rendre transparente pour les applications et sans contrôle centralisé. Cela assouplit la méthode, mais atténue ses limites en termes de surcharge et de tolérance aux pannes. Nous discutons de la mise en œuvre en détail et présentons des résultats préliminaires.

Mots-clés : calcul hautes performances, E/S parallèles, ordonnancement d'E/S

Contents

1 Introduction

Parallel file systems (PFS) are the main storage solution for high-performance computing (HPC) platforms. In these machines, applications share the PFS and can access it at any time, without any control. When multiple applications concurrently access the PFS, their I/O performance will be slowed down: presumably, they get equal shares of the available bandwidth. However, in practice, this ideal of fair sharing is not achieved [10] and not even desirable, as it is not necessarily the most efficient approach [4].

Indeed, as demonstrated by Boito *et al.* [4], in the context of applications that alternate between computing and I/O phases (periodic applications), when fairly sharing the PFS with applications with *long and less frequent* I/O phases, those with *small and frequent* phases experience significant delays, which severely impact their performance. Conversely, allocating more bandwidth to applications with *small and frequent* I/O phases enhances their performance without significantly impacting the *large, less frequent* ones. They proposed IO-Sets, an I/O scheduling framework to arbitrate the available PFS bandwidth among concurrent applications. In IO-Sets, applications are classified into sets in such a way that applications belonging to the same set perform I/O exclusively (one at a time), while applications from distinct sets can access the PFS concurrently. Moreover, each set has a priority that defines how much I/O bandwidth an application can receive when sharing the PFS with others.

Differently from other I/O scheduling techniques found in the related literature, IO-Sets uses only one piece of information about the application: the mean time between I/O phases (also called its *characteristic time*). This offers interesting advantages to IO-Sets in terms of practical use and implementation. **Combined with the compelling results presented in the paper, these advantages motivate us to invest time and effort into implementing it at the PFS level. In this report, we discuss an implementation done with BeeGFS [2] and some preliminary results obtained with it.**

Implementing IO-Sets brings two main challenges:

1. Controlling what applications can access the PFS at any given time.
In [4], the IO-Sets method was evaluated through simulation and a proof-of-concept implementation, where applications were modified to explicitly talk to a centralized scheduler. Such a solution is not suitable for production because i) requiring modifications in applications would significantly complicate the adoption of this technique, whose success depends on all applications going through the scheduler; and ii) a centralized scheduler would limit performance at large scale and be a single point of failure.

That means we need a way of controlling what applications can access the file system without changing them and without having them going through a centralized scheduler. It is important to notice, however, that in practice different applications could want to access different servers of the PFS. In that case, we do not see a reason to prevent them from doing so, even if they are in the same set. Therefore, we propose to do that in the context of each object storage server (OSS) of BeeGFS. On the one hand, that allows for a scalable distributed solution, but on the other hand it represents a deviation from the original IO-Sets method. The impact of this deviation is something we want to study.

2. Priority-based sharing of the available bandwidth.
Part of IO-Sets is that applications from different sets can access the PFS at the same time, receiving portions of the bandwidth that correspond to the priorities of their sets. Some popular PFSs, such as BeeGFS, do not offer any kind of bandwidth partitioning mechanisms, while others, such as Lustre [7], offer some simple QoS mechanisms that give some control over the

amount of bandwidth applications will receive [11]. Nonetheless, such a QoS mechanism is not suitable for implementing IO-Sets: a maximum bandwidth is defined for each application or grouping of applications. Even when additional I/O bandwidth is available (because no other application is using the PFS), it cannot be used. On the other hand, what we need for implementing an effective I/O scheduling technique is that applications use as much of the available resource as possible, but also some priority-based bandwidth sharing scheme to be respected when multiple applications share the PFS. For example, the weighted fair queuing (WFQ) could be used at server side to implement the latter [8, 9]. This intuition motivated our choice of BeeGFS for this implementation, as the data structures used in the servers are suitable for such an implementation.

In the remainder of this report, we present the implementation aspects, challenges, and decisions we made to adapt the ideas from [4] in practice. Moreover, we focus on how we manage certain complexities typically associated with scheduling techniques, such as obtaining information about applications. We also explore how we relax some of the constraints outlined in the IO-Sets paper, enabling us to implement the proposed framework without compromising performance and scalability. This document is organized as follows. In Section 2, we provide the necessary background information. In Section 3, we describe the implementation of bandwidth partitioning and exclusive access in detail. A preliminary performance evaluation is presented in two sections: the methodology in Section 4 and results in Section 5. Finally, Section 6 concludes the report while outlining future directions.

2 Background

2.1 BeeGFS

BeeGFS is a parallel file system that is composed of several components divided into four categories: Client, Management, Metadata, and Storage. Figure 1 presents the file system components and their general organization[1]. In this report, we are interested in components that belong to the Client and Storage categories. Data storage is composed of two components: Object Storage Servers (OSS) and Object Storage Targets (OST). As the name suggests, the OSS is responsible for storing the file's data, while the OSTs handle the actual storage devices through local file systems. As can be seen in Figure 1, each OSS has one or more OSTs.

[1] For more details on all BeeGFS components, see [[3], Section II]

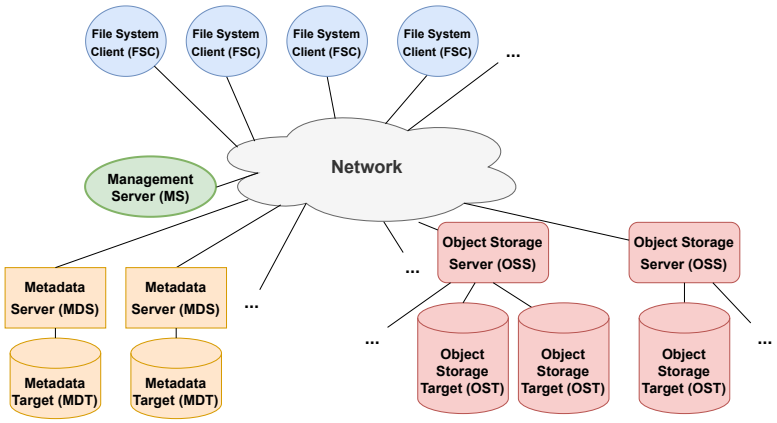


Figure 1: Components of the BeeGFS software architecture (figure from [3], inspired by [5])

The File System Client (FSC) is a kernel module that allows for mounting the remote file system and exposes some useful functions. For instance, it can show real-time information on the amount of I/O operations users are performing on the PFS. The FSC is the component that actually receives the I/O requests from the applications, interacts with the metadata servers to perform the the metadata operations, and submits the data requests to the OSS. In other words, all communication between applications and the PFS occurs through the FSC.

2.2 The IO-Sets method for I/O scheduling

I/O scheduling strategies try to decide algorithmically which application(s) are prioritized (e.g. first-come-first-served or semi-round-robin) when accessing the shared PFS. Previous work [6] thoroughly demonstrated that existing approaches based on either *exclusivity* or *fair-sharing* heuristics showed inconsistent results, with exclusivity sometimes outperforming fair-sharing for particular cases, and vice versa. Based on these observations, Boito *et al.*[4] decided to research an approach capable of combining both by grouping applications according to their I/O frequency. As a result, they designed IO-Sets, a novel method for I/O management in HPC systems.

As said before, in IO-Sets, applications are categorized into *sets* based on their characteristic time, representing the mean time between I/O phases. Applications within the same set perform I/O exclusively, one at a time. However, applications from different sets can simultaneously access the PFS and share the available bandwidth. Each set is assigned a priority determining the portion of the I/O bandwidth applications receive when performing I/O concurrently. Therefore, proposing a heuristic in the IO-Sets method involves addressing two important questions: (i) how to allocate an application to a set, and (ii) how to define the priority of a set.

The authors proposed a heuristic called Set-10. It answers the questions using the *characteristic time* of applications (denoted w_{iter}), defined as the average time between the start of consecutive I/O phases. Then, Set-10 consists of:

- **Set mapping:** Given an application A_{id} with a characteristic time $w_{\text{iter}}^{\text{id}}$, the π function maps it to a set:

$$\pi : A_{\text{id}} \mapsto S_{\lfloor log_{10} w_{\text{iter}}^{\text{id}} \rceil} \tag{1}$$

  Where $\lfloor x \rceil$ represents the nearest integer value of x.

  In other words, an application is assigned to a set that corresponds to the order of magnitude of its w_{iter}. For instance, there will be a set for applications with w_{iter} values such

that $\log_{10} w_{\text{iter}}$ falls between 0.5 and 1.5 (corresponding to w_{iter} values between 4 and 31, assigned to S_1), another set for $\log_{10} w_{\text{iter}}$ between 1.5 and 2.5 (corresponding to w_{iter} values between 32 and 316, assigned to S_2), and so on.

- **Set priority:** the priority p_i for set S_i (which corresponds to jobs such that $i = \lfloor \log_{10} w_{\text{iter}}^{\text{id}} \rceil$) is computed as follows:

$$p_i = 10^{-i} \tag{2}$$

  This means that applications with the smallest w_{iter} receive the highest priority and, consequently, most of the bandwidth. The priorities decrease exponentially: S_1 receives a priority of 1/10, S_2 has a priority of 1/100, and so on.

In the next section, we describe how we implemented the IO-Sets method into the BeeGFS file system. We believe that the PFS is, in fact, the right place for such an implementation, as it offers several advantages. Firstly, at the PFS level, the scheduling approach will be transparent to all applications, and no intervention at either the application code or I/O library level is needed. Secondly, the PFS is already designed to handle a large number of requests and clients concurrently. Therefore, we can leverage its well-designed architecture to develop new QoS mechanisms, thereby avoiding the addition of a new point of overhead and failure in the systems (as would be the case with an external controller).

3 Adding IO-Sets to BeeGFS

In this section, we detail our implementation of IO-Sets in BeeGFS. Requests from the clients, generated by the FSC, reach the storage servers and are queued there for later processing by worker threads. I/O scheduling then becomes a matter of deciding in which order requests are processed. The bulk of our modification was to introduce a new scheduling policy at this point (Section 3.5). Additionally, some modifications were required at the client side to obtain and export required information (Sections 3.1 to 3.4).

The BeeGFS file system was chosen for this because it already provides two policies to handle requests. Neither is precisely the policy needed for IO-Sets, but because the software already provides a choice, these features are cleanly isolated in the code base and it was easy to add a third one. It is important to notice, however, that this behavior of queuing requests at the server for a pool of threads to treat is fairly standard and not specific to BeeGFS. That means a similar strategy we describe here could be used to implement IO-Sets in another PFS.

3.1 Identifying applications

IO-Sets works at the application level, and thus requires I/O requests to be tied to a specific application. Unfortunately, application ID is not a concept as well defined as user ID, and there is no standard system call to get an application ID from the operating system.

In the context of this work, application IDs are set by the job scheduler of a cluster, and made available to applications via the environment, Slurm writes it to the `SLURM_JOB_ID` environment variable for example. Similarly to what is done by Lustre, the BeeGFS client was modified to read the application ID from the environment of the processes of the application, by taking advantage of the fact that the client runs as a kernel module and can access the virtual memory of the process directly using the `access_process_vm` kernel function to read at the `env_start` position. Note however that this trick, not particularly clean, only works for a static environment variable set before the process start and not modified afterwards. Indeed, if a process modifies

its environment, the new variables could be written somewhere else in memory and there is no way for the kernel to find out where.

In an effort to be system agnostic, the environment variable we read is `IOSETS_JOB_ID`, so application scripts are expected to set it, most likely based on the environment variable set by their cluster job scheduler.

3.2 Finding the applications' w_{iter}

The application priority as defined by IO-Sets is based on the periodicity of the application (w_{iter}). Our implementation does *not* include a tool for profiling applications. Instead, an external tool called FTIO [12] is used for that. Note that this is a dynamic piece of information, which can change during the lifetime of the application. Indeed, FTIO can run alongside an application and provide up-to-date predictions of its w_{iter}.

To further simplify our implementation, we assume that applications, or tools running with them, will not only detect w_{iter}, but also compute its priority. Because all applications in the same set have the same priority, the priority is also used to identify the set.

We developed a script that calls FTIO for an application, collects its predictions and computes its priority. In the beginning of the execution, FTIO results may not be available yet (or may be inconclusive). In that case, our script gives the application a unique value for its priority (so it will belong to a set of its own) by taking a default value (in our results, 1/50) and adding a small value derived from its application ID. After the first FTIO prediction is obtained, this priority is updated according to the Set-10 heuristic and kept until new conclusive predictions are available.

3.3 Obtaining application priorities

Once the priority is computed, we need it to be available for the BeeGFS client. Because the priority of each application may change during its execution (as detected values for w_{iter} change), and as discussed in Section 3.1, this rules out the use of the environment to communicate the priority to the BeeGFS client.

This led us to use Configfs, a pseudo file system provided by the Linux kernel to create and manage kernel objects from userspace [1]. Loading the kernel Configfs module mounts it (usually at `/sys/kernel/config`). Our BeeGFS client module initializes a new `iosets` namespace that will contain one directory per application, named after the application ID. Each directory contains one file, named `priority`. On the kernel side, this corresponds to a map of application IDs to their priority, available for the BeeGFS kernel module. Whenever an integer is written to the `priority` file, the corresponding priority is updated. Practically speaking, applications or a userspace helper acting on their behalf are expected to create one directory per application in the Configfs and to add its priority in the `priority` file. Using command-line tools, the process is simply

```
$ mkdir /sys/kernel/config/iosets/${IOSETS_JOB_ID}
$ echo $JOB_PRIORITY > /sys/kernel/config/iosets/${IOSETS_JOB_ID}/priority
```

3.4 Sending information to the servers: the BeeGFS message header

The priority and application ID available at the client need to be attached to every file system request so that the server can use it for scheduling. The BeeGFS message header has thus been extended to fit the application ID and the associated priority, each in a 4 bytes integer. Of course,

these extra 8 bytes means that the message format used by our modified version of BeeGFS is not compatible with the one used by the standard version.

```
struct NetMessageHeader
{
  [...]
  uint32_t       msgUserID;   // system user ID for per-user msg queues,
                              // stats etc.
  uint32_t       msgJobID;    // resource manager job identifier
                              //(IO-SETS addition)
  uint32_t       msgPriority; // scheduling priority (IO-SETS addition)
  [...]
}
```

3.5 Adding a scheduling policy at the BeeGFS server

BeeGFS servers are designed around a queue of network requests and a pool of worker threads that call a function `getAndPopNextWork()` repeatedly, unqueuing a request each time. Different scheduling policies can be defined by changing the way requests are picked from the queue. BeeGFS already provides two queue policies, the standard first-come, first-served or "per-user", the latter intended to improve fairness between users, activated with the option `tuneUsePerUserMsgQueues`. We added a new one for IO-Sets, activated with the new option `tuneUseIosets`.

IO-Sets requires both exclusive access (for applications with the same priority) and unfair bandwidth sharing (for applications with different priority). As detailed in previous sections, application ID and priority are available to the servers, attached to each request received. We defined a two-levels hash-map associating network requests, application ID and priority: the inner level maps an application ID to a list of its requests, and the outer level maps a priority to the list of applications sharing this priority (i.e. the applications belonging to the same set).

Exclusive access within sets: when picking requests from one set/priority, the server will go over the applications, which are ordered by application ID, only moving to the next one when there are no requests for the previous one. This is illustrated in the simple example of Figure 2, where there are two applications: APP1 and APP2.

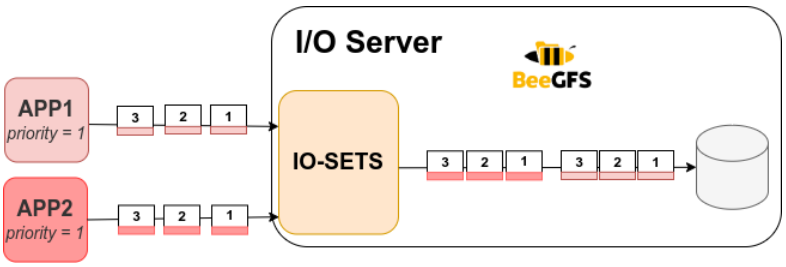


Figure 2: Ensuring exclusive access. All requests for APP1 are processed before APP2.

It is important to notice that moving to APP2, in the absence of requests from APP1, does *not* mean that APP1 finished its I/O phase. This is another way we relax the original IO-Sets, where applications are meant to perform their whole I/O phases before others are allowed to

proceed. In practice, the server has no way of knowing an I/O phase is done and waiting for more requests from an application to arrive would only waste server time. Because the applications are ordered, if new requests from APP1 arrive they will be executed before the ones queued from APP2.

This strategy could lead to starvation of APP2, for example, if APP1 were to continuously generate I/O requests. We consider that to be unlikely, however, because applications that belong to the same set are expected to have similar periodicity, and most applications perform I/O synchronously, meaning they wait for an answer from the server before issuing new requests (which gives time for requests from APP2 to be treated). Moreover, eventually APP1 will finish and leave APP2 to be treated. Still, we plan on further exploring this starvation possibility as future work.

Priority-based bandwidth sharing: Using this priority-application map, we make so that requests are picked using a *credit*-based system similar to weighted fair queuing (see [8]). Each priority is associated with a credit value, scaled so that the smallest priority gets a credit of one. The credit corresponds to the number of requests that will be processed from that priority/set before moving to the next one (which may mean taking requests from multiple applications in that set).

This is illustrated in the simple example from Figure 3, where APP1 and APP2 belong to different sets, and APP2 has twice the priority of APP1. That means that two requests of APP2 will be treated for every request of APP1. In the end, APP2 will have received twice as much bandwidth as APP1.

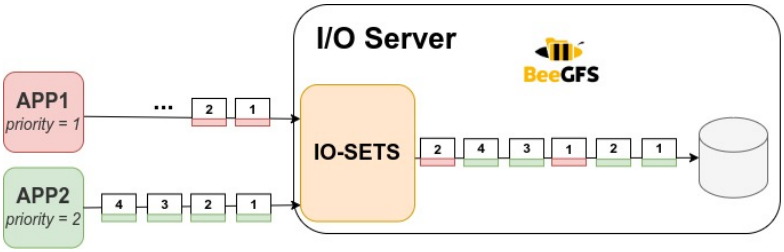


Figure 3: Ensuring unfair bandwidth sharing. Two requests of *APP2* are processed for every one request of *APP1*.

An option *–jobstats* was added to the *beegfs-ctl* tool, to help visualize the instantaneous bandwidth usage per application. That allowed us to confirm the implementation works as expected.

4 Experimental Methodology

This section explains how experiments, whose results are discussed in Section 5, were conducted. We describe the experimental environment in Section 4.1, the application generation and profiling in Section 4.2, and the computation of the reported metrics in Section 4.3. All results and code used to parse and analyze them are available at `https://gitlab.inria.fr/hpc_io/iosets-ftio-experiments`.

4.1 Experimental environment

Experimenting with our implementation involves installing our modified BeeGFS version, and hence requires *root* access to compute nodes. We executed all experiments presented in this document using the Grid'5000 experimental testbed[2], on the Gros cluster of the Nancy site[3]. Nodes were prepared with Debian 11 and our implementation, which is based on BeeGFS 7.3.2.

A single separate node was deployed as the metadata server and management node, while other nodes are either OSS or client nodes (never both at the same time). Each OSS has a single OST, which uses the node-local device, a 480 GB SSD SATA Micron MTFDDAK480TDN, for storage through the ext4 file system. Gros nodes are connected through a 25 Gbps Ethernet network.

4.2 Application generation

In all experiments, IOR was used to create the applications, always using the MPI-IO API. They are all launched at the same time. We use a modified IOR, based on version 3.4.0:

- At the end of each compute phase, and beginning and end of each I/O phase, rank 0 writes the current *epoch* timestamp to the output. This leverages pre-existing barriers in these moments.
- The code includes the TMIO library, which is part of the FTIO implementation [12].
- At the end of each I/O phase, all processes call *iotrace_summary()*, from TMIO, to append the recent activity to the trace.

Each IOR has a dedicated prediction-mode FTIO, running on the same node and watching for its I/O trace, which is updated after every I/O phase (due to the TMIO call). A script parses the FTIO output for the predicted period and confidence, calculates the priority, and writes it to the */sys/kernel/config/iosets/[job id]/priority* file, as discussed in Section 3.3.

4.3 Measuring performance

We report the same metrics described in [4, Section 5.4]: `Max Stretch`, `Utilization`, and `IO-slowdown`, and add to those the `Geometric Mean Stretch` (which takes the geometric mean instead of the maximum of the applications' individual stretch values). As in [4], they are computed over a time frame $[T_{begin}, T_{end}]$ of the execution, aiming at evaluating them at a steady state. Because of the practical scenario, their computation is slightly adapted:

1. All IOR outputs are parsed and the timestamps that mark the boundaries of compute and I/O phases (see Section 4.2) are obtained. Then all timestamps are made relative to the beginning of the first application to start.
2. All phases that are outside of the time frame $[T_{begin}, T_{end}]$ are removed from the analysis. Phases that partially overlap the time frame (because they start before T_{begin} or end after T_{end}) are resized: compute phases simply have their duration adapted to only comprise the time inside the time frame, whereas for I/O phases we recompute the amount of written data according to their new duration and the bandwidth observed for the whole phase. It is important to notice that if an I/O phase starts before T_{end} and finishes after it, then

[2] `www.grid5000.fr`
[3] `https://www.grid5000.fr/w/Nancy:Hardware#gros`

we cannot adapt it, because we do not know its bandwidth (due to not knowing its full duration). In this case, we cannot compute metrics, and the solution is to either change the time frame or repeat the experiment.

3. For each application i, we compute the *effective completed compute work* e_i^{cpu} as the sum of its compute phases, and the *effective completed I/O work* e_i^{io} as the total amount of data moved by its I/O phases divided by the *expected bandwidth of the system* B (which is the bandwidth the application would observe in isolation). In practice, after parsing all I/O phases of all experiments, we set B as the median bandwidth observed for I/O phases that did NOT share the file system. e_i^{cpu} differs from the definition in [4] because we do not use the expected duration of the compute phases, but their actual one (which may vary in practice, despite consisting in a *sleep* call in the IOR code).

4. Then, for each application i, its stretch is given by

$$\frac{T_{end} - T_{start}}{e_i^{cpu} + e_i^{io}}$$

We take the maximum and the geometric mean among the concurrent applications to obtain the `Max Stretch` and `Geometric Mean Stretch` metrics.

5. The I/O slowdown of application i is computed as

$$\frac{T_{end} - T_{start} - e_i^{cpu}}{e_i^{io}}$$

and the geometric mean among the application gives the `IO-slowdown` metric.

6. Finally, `Utilization` is computed as

$$\frac{\sum_j e_j^{cpu}}{N.(T_{end} - T_{begin})}$$

Stretch and `IO-slowdown` are application-centric metrics: they represent how slower the application executed (and respectively how slower its I/O was) when running concurrently with the others, compared to running in isolation. `Max Stretch` represents the application that was slowed down the most, while `Geometric Mean Stretch` gives an overall view of all applications. They have therefore a lower bound of 1. On the other hand, `Utilization` is a system-centric metric that measures how much of the node/core/resource time was spent on compute (and not waiting for I/O), and hence has a value between 0 and 1.

5 Results

The goal of this first experimental campaign was to see if the implementation could provide good results in a situation where Set-10 is expected to. Additionally, we wanted to see if the period provided by FTIO would be a good-enough estimation of w_{iter} so the heuristic would still reach good results. For that, we aimed at recreating one of the experiments shown in [4, Figure 4]: 16 concurrent applications execute, clearly belonging to two separate sets, and inside each set they all present the same period.

In all experiments, we compare our obtained results with two baselines: the original BeeGFS (without Set-10) and a *clairvoyant* version of our Set-10 implementation of BeeGFS *without*

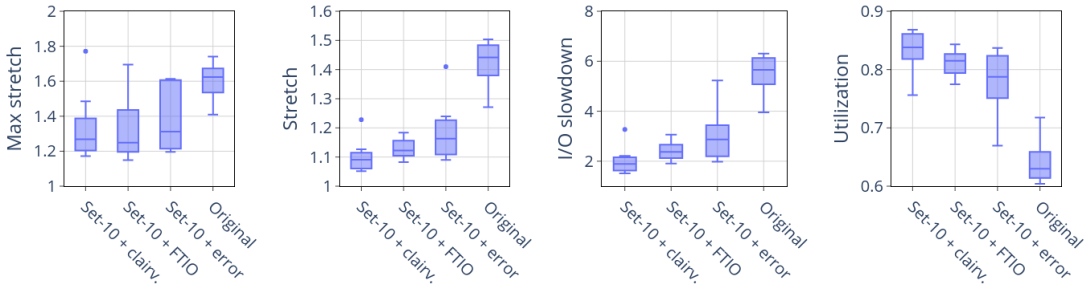


Figure 4: Results from the experiment with 1 application with a period of 19.2 seconds and 15 applications with a 384 s period. Boxplots group ten repetitions of each test. The y-axes are all different and do *not* start at zero.

the FTIO-provided period detection. For the latter, we use the expected period (or w_{iter}) of each application (if executed in isolation, a hardcoded value) to compute its priority and write it to the file in the beginning of the execution. Moreover, we also conducted experiments where error is injected to the output of FTIO before being used by Set-10. The point in doing so was to see to what extent results obtained with FTIO were due to it being accurate, or to Set-10 being robust. In these experiments, each period detected by FTIO during each application's execution is randomly increased or decreased in 50%.

It is important to notice the actual period each application will present during its execution — and that FTIO will aim at detecting — is different from the period it would have in isolation, because its I/O phases may be slowed-down due to sharing the file system (and notably due to the decisions made by Set-10).

IOR was used to create one high frequency application and 15 low-frequency ones: all of them use a single node (which is a shared client node for all 16 applications), 8 processes, and write to file-per-process files using the *fsync* option (*-e*). The high-frequency application has 200 compute phases of 18 seconds, each followed by I/O phases where each process writes 16 MiB. Because the write performance with *fsync* was measured to be of approximately 110 MiB/s, that means this application has a period of approximately 19.2 s. The low-frequency applications, on the other hand, have compute phases of 360 seconds, followed by I/O phases where each process writes 320 MiB (for a period of $\approx$ 384 s). Metrics were computed using in the time frame between 400 and 3200 seconds after the start of the first application, and are shown in Figure 4.

The *clairvoyant* version of Set-10 improves performance over the original BeeGFS: on average `Max Stretch` was decreased in 16%, `Geometric Mean Stretch` in 22%, `IO-slowdown` in 63%, and `Utilization` was increased in 29%. That shows our implementation, even if a relaxed version of the original proposal, can still provide good results.

The results achieved when using FTIO to obtain applications' w_{iter} are close to the clairvoyant version — only $\approx$ 2% worse in stretch and I/O slowdown, and 19% in utilization. In contrast, the version where we inject errors to FTIO results made stretch and utilization worse by 5% and I/O slowdown 27% higher, compared to the "Set-10 + FTIO" version, in addition to presenting higher variability. Thus, we conclude that FTIO hits a sweet spot in performance where a better prediction would not improve the performance observed by the system or the users, however, a worse prediction would increase the variability and impair the performance of the system. That indicates FTIO provides results that are good enough for Set-10.

6 Conclusion and Future Work

In this report, we described a BeeGFS-based implementation of the IO-Sets method for I/O scheduling in HPC systems. The method was previously proposed by Boito et al. [4], and evaluated using simulation and a proof-of-concept implementation. We chose to implement it in a parallel file system so it would be completely transparent for applications. Moreover, in our implementation the scheduling algorithm works in the context of each data server, sacrificing the idea of a centralized control over who can access the parallel file system, which is part of IO-Sets. We believe this is an important adaptation that makes it better suitable to be used in practice, since a centralized scheduler would impose higher overhead and be a single point of failure. Our preliminary results show promising results, where our version significantly improved performance over the baseline (BeeGFS without modifications). As future work, we plan on further exploring the consequences of our implementation choice in results, for example with experiments including multiple data servers.

References

[1] J. Becker. Configfs - Userspace-driven Kernel Object Configuration. `https://docs.kernel.org/filesystems/configfs.html`, 2005.

[2] BeeGFS. The leading parallel file system. `https://www.beegfs.io/`, 2023.

[3] F. Boito, G. Pallez, and L. Teylo. The role of storage target allocation in applications' i/o performance with beegfs. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 267–277, 2022.

[4] F. Boito, G. Pallez, L. Teylo, and N. Vidal. Io-sets: Simple and efficient approaches for i/o bandwidth management. *IEEE Transactions on Parallel and Distributed Systems*, 34(10):2783–2796, 2023.

[5] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu. I/o characterization and performance evaluation of beegfs for deep learning. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[6] E. Jeannot, G. Pallez, and N. Vidal. Scheduling periodic I/O access with bi-colored chains: models and algorithms. *J. of Scheduling*, 24(5):469–481, 2021.

[7] Lustre. The lustre® filesystem. `https://www.lustre.org/`, 2023.

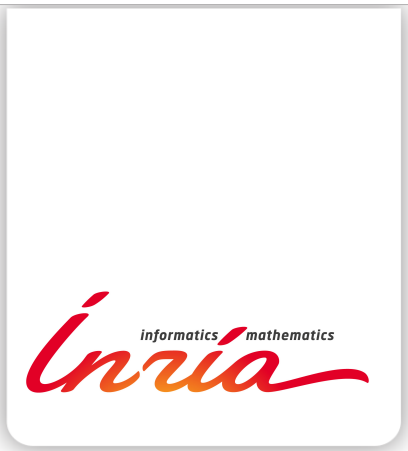
[8] A. Mayer, L. Teylo, and F. Boito. Implementation and Test of a Weighted Fair Queuing (WFQ) I/O Request Scheduler. Research Report RR-9480, Inria, Aug. 2022.

[9] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[10] T. Patel, R. Garg, and D. Tiwari. {GIFT}: A coupon based {Throttle-and-Reward} mechanism for fair and efficient {I/O} bandwidth management on parallel storage systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 103–119, 2020.

[11] Y. Qian, X. Li, S. Ihara, L. Zeng, J. Kaiser, T. Süß, and A. Brinkmann. A configurable rule based classful token bucket filter network request scheduler for the lustre file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[12] A. Tarraf, A. Bandet, F. Zanon Boito, G. Pallez, and F. Wolf. FTIO: Detecting I/O Periodicity Using Frequency Techniques. Working paper or preprint: https://arxiv.org/abs/2306.08601, June 2023.



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