Control-based runtime management of HPC systems with support for reproducible experiments

Quentin Guilloteau

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Une approche autonomique à la régulation en ligne de systèmes HPC, avec un support pour la reproductibilité des expériences

Control-based runtime management of HPC systems with support for reproducible experiments

Présentée par :

Quentin GUILLOTEAU

Direction de thèse :

Eric RUTTEN CHARGE DE RECHERCHE, Université Grenoble Alpes	Directeur de thèse
Olivier RICHARD MAITRE DE CONFERENCES, Université Grenoble Alpes	Co-encadrant de thèse

Rapporteurs :

Alessandro PAPADOPOULOS
PROFESSEUR, Mälardalen University (MDU)
Alexandru COSTAN
MAITRE DE CONFERENCES, INSA Rennes, IRISA

Thèse soutenue publiquement le **11 décembre 2023**, devant le jury composé de :

Eric RUTTEN CHARGE DE RECHERCHE HDR, Centre INRIA de l'UGA	Directeur de thèse
Alessandro PAPADOPOULOS PROFESSEUR, Mälardalen University (MDU)	Rapporteur
Georges DA COSTA PROFESSEUR DES UNIVERSITES, Université Paul Sabatier	Examinateur
Fabienne BOYER MAITRESSE DE CONFERENCES, Université Grenoble Alpes	Examinatrice
Noël DE PALMA PROFESSEUR DES UNIVERSITES, Université Grenoble Alpes	Examinateur
Alexandru COSTAN MAITRE DE CONFERENCES, INSA Rennes, IRISA	Rapporteur

Invités :

Olivier RICHARD
MAITRE DE CONFERENCES, Université Grenoble Alpes

Remerciements

(Acknowledgments)

J'aimerais commencer par remercier les membres du jury pour leurs retours, et en particulier Alessandro Papadopoulos et Alexandru Costan pour avoir lu ce document de bout en bout.

Un immense merci à Eric et Olivier pour m'avoir accueilli dès le stage de M2, et de m'avoir supporté pendant presque 4 ans. Deux encadrants, deux styles et domaines d'expertise bien différents, mais tout de même très complémentaires ! Vous m'avez tous les deux tant appris, scientifiquement et humainement. Merci aux équipes CtrlA, DataMove, Polaris pour cet environnement de recherche si sain et stimulant. Merci à Imma, Annie, Maud, sans qui je ne serais pas allé bien loin (géographiquement et administrativement). Merci aux post-docs, ingénieurs, et jeunes chercheuses et chercheurs pour vos connaissances et pour avoir montré la voie à suivre avec autant de bienveillance. Je pense notamment à Raphaël, Sophie, Millian, Danilo, Adrien, Jonathan. Merci à tous les thésard.e.s avec qui j'ai partagé ces trois années et tous ces moments de convivialité autour d'un verre et/ou avec des cartes. Merci à tous les résidents du bureau 443, pour leur sympathie, et avoir supporté toutes mes tentatives au basket. Merci aux brésiliens pour m'avoir fait découvrir vos coutumes culinaires (parfois douteuses). Merci à Grid'5000 sans qui cette thèse aurait été bien fade expérimentalement parlant. Merci à tous les enseignants qui m'ont laissé une marque pendant mon parcours scolaire. Merci à Frédéric et Thomas pour m'avoir fait découvrir la beauté du parallélisme et du distribué, et Arnaud et Jean-Marc pour m'avoir sensibilisé aux subtilités de l'évaluation expérimentale et des statistiques.

Merci à ma famille pour tout leur soutien (Luky, c'est à ton tour maintenant !).

Et enfin, merci infiniment à Sofi pour son support quotidien inconditionnel, et sans qui ces derniers mois de stress auraient été bien plus difficiles à vivre.

Abstract / Résumé

Abstract

High-Performance Computing (HPC) systems have become increasingly more complex, and their performance and power consumption make them less predictable. This unpredictability requires cautious runtime management to guarantee an acceptable Quality-of-Service to the end users. Such a regulation problem arises in the context of the computing grid middleware CiGri that aims at harvesting the idle computing resources of a set of cluster by injection low priority jobs. A too aggressive harvesting strategy can lead to the degradation of the performance for all the users of the clusters, while a too shy harvesting will leave resources idle and thus lose computing power. There is thus a tradeoff between the amount of resources that can be harvested and the resulting degradation of users jobs, which can evolve at runtime based on Service Level Agreements and the current load of the system.

We claim that such regulation challenges can be addressed with tools from Autonomic Computing, and in particular when coupled with Control Theory. This thesis investigates several regulation problems in the context of CiGri with such tools. We will focus on regulating the harvesting based on the load of a shared distributed file-system, and improving the overall usage of the computing resources. We will also evaluate and compare the reusability of the proposed control-based solutions in the context of HPC systems.

The experiments done in this thesis also led us to investigate new tools and techniques to improve the cost and reproducibility of the experiments. We will present a tool named NixOS-compose able to generate and deploy reproducible distributed software environments. We will also investigate techniques to reduce the number of machines needed to deploy experiments on grid or cluster middlewares, such as CiGri, while ensuring an acceptable level of realism for the final deployed system.

Résumé

Les systèmes de calcul haute performance (HPC) sont devenus de plus en plus complexes, et leurs performances ainsi que leur consommation d'énergie les rendent de moins en moins prévisibles. Cette imprévisibilité nécessite une gestion en ligne et prudente, afin garantir une qualité de service acceptable aux utilisateurs. Un tel problème de régulation se pose dans le contexte de l'intergiciel de grille de calcul CiGri qui vise à récolter les ressources inutilisées d'un ensemble de grappes via l'injection de tâches faiblement prioritaires. Une stratégie de récolte trop agressive peut conduire à la dégradation des performances pour tous les utilisateurs des grappes, tandis qu'une récolte trop timide laissera des ressources inutilisées et donc une perte de puissance de calcul. Il existe ainsi un compromis entre la quantité de ressources pouvant être récoltées et la dégradation des performances pour les tâches des utilisateurs qui en résulte. Ce compromis peut évoluer au cours de l'exécution en fonction des accords de niveau de service et de la charge du système.

Nous affirmons que de tels défis de régulation peuvent être résolus avec des outils issus de l'informatique autonomique, et en particulier lorsqu'ils sont couplés à la théorie du contrôle. Cette thèse étudie plusieurs problèmes de régulation dans le contexte de CiGri avec de tels outils. Nous nous concentrerons sur la régulation de la récolte de ressources libres en fonction de la charge d'un système de fichiers distribué partagé et sur l'amélioration de l'utilisation globale des ressources de calcul. Nous évaluerons et comparerons également la réutilisabilité des solutions proposées dans le contexte des systèmes HPC.

Les expériences réalisées dans cette thèse nous ont par ailleurs amené à rechercher de nouveaux outils et techniques pour améliorer le coût et la reproductibilité des expériences. Nous présenterons un outil nommé NixOS-compose capable de générer et de déployer des environnements logiciels distribués reproductibles. Nous étudierons de plus des techniques permettant de réduire le nombre de machines nécessaires pour expérimenter sur des intergiciels de grappe, tels que CiGri, tout en garantissant un niveau de réalisme acceptable pour le système final déployé.

Contents

Introduction

Background

As calculations started to be too complex to be carried out by hand by slow and error-prone humans, scientists developed machines able to help them perform computations. These machines improved though history, from the Abacus, the Pascal's calculator, the Thomas de Colmar's Arithmometer, the tabulating machine for punched cards from Herman Hollerith, or the famous Turing machine.

As every field of Science continued to go forward, and challenges to grow larger and more complex, the amount of computation needed to perform experiments kept increasing, and required the machines to also improve. These improvements were possible with the reduction in size of transistors and circuit boards, as well as the improvements of the raw components (processors, memory, buses, etc.). As computers reached Moore's law, but problem sizes were still increasing, a single machine could not manage the entire copmutational workload. The solution was to link several machines together and distribute the workload on this network of nodes, creating a *computing cluster*. To keep up with the computing demands, these clusters have to add more machines and/or change for newer and more powerful computers. Figure 1 shows the evolution of the numbers of cores, CPU frequency and performance of the clusters of the Top500 list. Even if the CPU frequency has stayed stable for close to 20 years, the performance keeps improving thanks to the increase of parallelism.

Building computing clusters is very expensive, and is usually a cooperative effort between several actors (laboratories, research institutes, universities, etc.). This cooperation leads to shared access to the resulting cluster for the members of the different contributing entities. To ensure *fair* access to the machines, a reservation mechanism is set up. Users have to submit their computations alongside the number of machines, also called nodes, required and the estimated duration to the Resource and Job Management System (RJMS). The RJMS, whose architecture is depicted in Figure 2, is responsible for managing the users computations, scheduling them onto the resources (machines, CPU, or even cores), and managing the resources

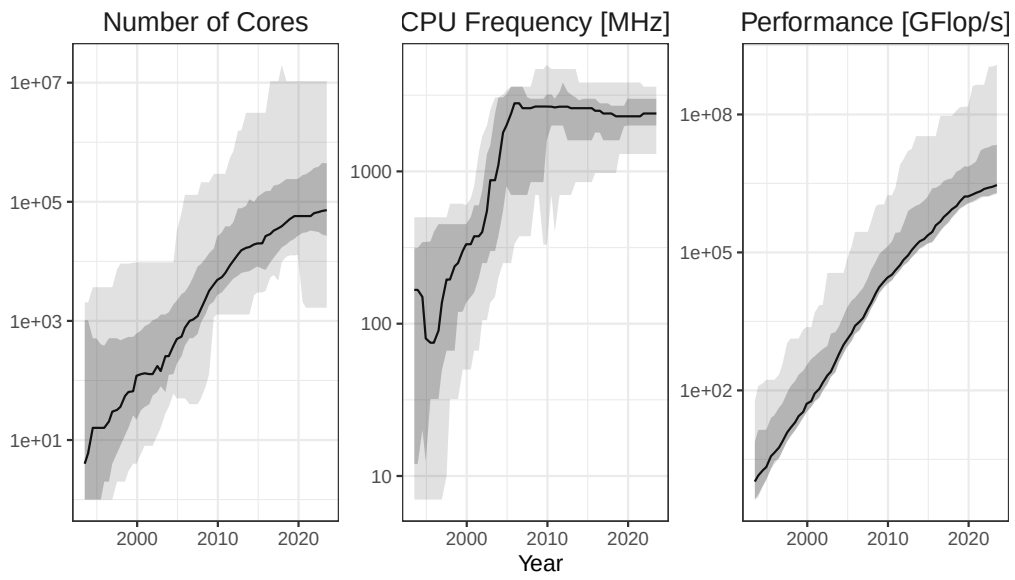


Figure 1.: Evolution of the number of cores, CPU frequency, and the maximum performance of the Top500 machines. The solid line represents the median, and the ribbons the 5% and 95%. Updated from [Cor21] with the data from [Len21].

(deploying and starting the computations, as well as cleaning the environment when they completed).

This reservation process is not perfect, and can lead to the idling of some resources due to the requirements on the resources. While machines are idle, they still consume energy, which is around half of a machine used at full power [Hei+17]. This idling can represent a significant loss of energy and thus money. Turning them on and off is possible but introduces new energetic costs, and degradation of the Quality-of-Service (QoS) if users need the resources. As HPC jobs are large and

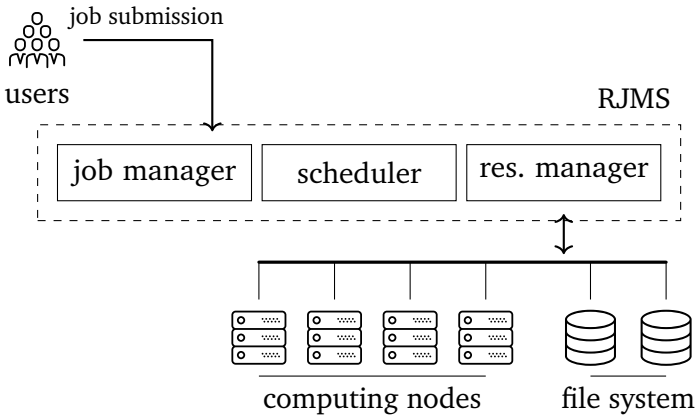


Figure 2.: Illustration of a Resource and Job Management System (RJMS) [Ble17]

rigid in the number of resources they require, it makes it difficult to improve the use of the idle CPU time. A lot of effort has been put on the research of better scheduling algorithms for the RJMS, meaning how to map computation to physical machines. However, they often require the users to estimate correctly the duration of their computation, which is notoriously difficult for humans to do correctly [CB01; MF01]. In practice, despite all this work, and because of the conservative behavior of the HPC community, the majority of HPC centers use the same, or a variation of, scheduling policy: EASY Backfilling. This scheduling technique is simple to implement, to understand, and with good enough performance, which eases its adoption in HPC centers.

The idle computing power can be harvested by executing smaller and interruptible jobs seen as second-class citizens. However, those jobs can introduce perturbations in the system and degrade the performances of the premium users jobs (*e.g.,* via an overload of the distributed file-system). As the jobs harvesting the idle resources are interruptible, it is tempting to kill them when the system is about to get overloaded. However, this killing also represent a lost of computing power as these jobs do not usually implement any check-pointing mechanism and will thus need to be restarted from the beginning later on.

In general, HPC systems are very unpredictable [Bha+13; SK05; Dor+14]. The performance of an HPC application can be affected by many exterior factors: network contention, file-system load, temperature in the rack, etc. All of these factors are near impossible to predict correctly, and computational costly to solve optimally, and thus impossible to incorporate into the decision process of the RJMS.

To deal with such runtime variations of computing systems, the field of Autonomic Computing proposes a feedback loop point-of-view to regulate the under control system and to steer it towards a desired behavior/state. The feedback loop of the Autonomic Computing can be interpreted and implemented in various fashions. One particularly interesting way is to use tools from the Control Theory field. Control Theory is usually applied to physical dynamical systems, but very rarely to computing systems. However, it provides mathematically proven properties on the controlled system, which makes it quite appealing compared to more black boxes such as Machine Learning approaches, or to simpler rule based strategies.

Experimenting on distributed systems, such as grid or cluster middlewares, is complex. To realistic evaluate a new solution, researchers need to deploy their own cluster, with a RJMS, a parallel file-system, etc. In practice, such deployments are rarely done, as they require a lot of machines, and because the software environment to deploy is deep, fragile, and complex. The fragility of the software environment

is a crucial reproducibility issue, as it means that an experiment done today could not be reproduced in the future. To still perform the experiment, one might use simulation techniques. But even state-of-the-art simulators fail to fully reproduce the behavior of experiment in real conditions.

This Thesis

We believe that the harvesting of idle resources represents a regulation problem which can be tackled via an Autonomic Computing approach. To provide a runtime behavior with guarantees, we claim that using tools and methods from Control Theory will help in terms of performance and adoption. However, controllers are usually tuned for a specific system, which might reduce their reusability on different systems.

When designing and implementing feedback loops, especially using Control Theory tools, the quality and robustness of the experimental setup is crucial. Reproducible experiments are a must to ensure the validity of the resulting controller. Moreover, distributed experiments involving computing cluster middlewares (*e.g.,* batch scheduler, Parallel File-Systems) are especially tricky. Faithful and realistic experiments with such middlewares would require deploying clusters at full scale, which cost is excessive. Simulation techniques can be used but require correct and proven models. Intermediate strategies to reduce the number of machines to deploy the experiments and keeping a full scale behavior of the system would allow to reduce the experimental costs while performing meaningful experiments.

From the above two observations, the work of this thesis will be organized in two parts: In Part I, we investigate an Autonomic Controller using Control Theory tools to regulate the harvesting of idle HPC cluster resources.

- In Chapter 1, we give some context to this work, and present the *CiGri* middleware which will be the central piece of this thesis.
- Chapter 2 studies the statistical characteristics of the *CiGri* jobs.
- Chapter 3 presents a controller regulating the *CiGri* injection speed based on the load of the distributed file-system to reduce the overhead for premium users jobs.
- We discuss the reusability of autonomic controllers using control theory tools from one system to another in Chapter 4.

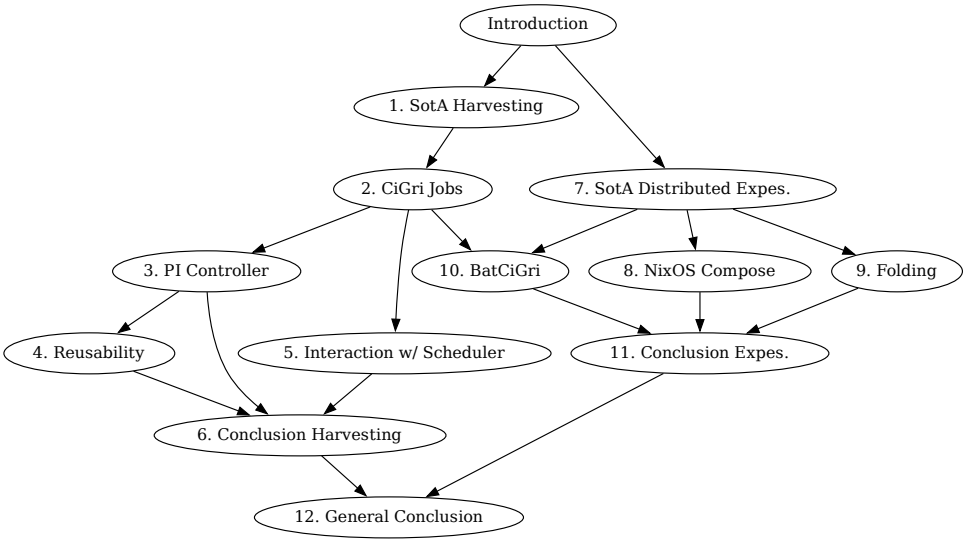


Figure 3.: Dependencies between the different chapters of this thesis.

- In Chapter 5, we present a controller reducing the wasted computing time of a cluster with *CiGri* by also considering the killed low priority jobs.
- We conclude Part I and give perspectives in Chapter 6.

In Part II, we discuss distributed experiments and their reproducibility.

- Chapter 7 gives context to the state of distributed experiments and their reproducibility.
- In Chapter 8, we present *NixOS Compose*, a new tool to create and deploy reproducible distributed software environments.
- Chapter 9 investigates the reduction of deployment of a distributed experiment involving a distributed file-system while keeping a realistic full scale behavior.
- Chapter 10 presents the first step towards a simulator of *CiGri* and our autonomic controllers of Part I using *Batsim*.
- We conclude Part II and give perspectives in Chapter 11.

Chapter 12 wraps up the entire work done in this thesis.

Some chapters can be read independently. Figure 3 depicts the dependency graph of the chapters of this thesis.

Work dissemination

The following are the communications resulting from this thesis.

International conferences

- Quentin Guilloteau, Jonathan Bleuzen, Millian Poquet, and Olivier Richard. “Painless Transposition of Reproducible Distributed Environments with NixOS Compose”. CLUSTER 2022 [Gui+22b]
- Quentin Guilloteau, Bogdan Robu, Cédric Join, Michel Flies and Eric Rutten. “Model-free control for resource harvesting in computing grids”. CCTA 2022 [Gui+22d]
- Quentin Guilloteau, Olivier Richard, Bogdan Robu, and Eric Rutten. “Controlling the Injection of Best-Effort Tasks to Harvest Idle Computing Grid Resources”. ICSTCC 2021 [Gui+21a]

National conferences

- Quentin Guilloteau, Adrien Faure, Millian Poquet, and Olivier Richard. “Comment rater la reproductibilité de ses expériences ?” COMPAS 2023 [Gui+23a].
- Quentin Guilloteau, Jonathan Bleuzen, Millian Poquet, and Olivier Richard. “Transposition d’environnements distribués reproductibles avec NixOS Compose”. COMPAS 2022 [Gui+22c]
- Quentin Guilloteau, Olivier Richard, and Éric Rutten. “Étude des applications Bag-of-Tasks du méso-centre Gricad”. COMPAS 2022 [GRR22a]
- Quentin Guilloteau, Olivier Richard, Eric Rutten, and Bogdan Robu. “Collecte de ressources libres dans une grille en préservant le système de fichiers : une approche autonomique”. COMPAS 2021 [Gui+21b]

Working papers

- Quentin Guilloteau, Olivier Richard, Raphaël Bleuse, and Eric Rutten. “Folding a Cluster containing a Distributed File-System”. 2023 [Gui+23d]
- Quentin Guilloteau. “Simulating a Multi-Layered Grid Middleware”. 2023 [Gui23d]

Tutorials

- Quentin Guilloteau, Jonathan Bleuzen, Millian Poquet, and Olivier Richard. Initiation to NixOS Compose. [Gui+a]
- Quentin Guilloteau, Sophie Cerf, Eric Rutten, Raphaël Bleuse, and Bogdan Robu. Introduction to Control Theory for Computer Scientists. [Gui+b]

Part I

Control-based Dynamic Harvesting of Idle HPC Resources

1 Discussion & State-of-the-Art

The first part of this Thesis is related to three areas: the harvesting of idle computing resources in High Performance Computing systems (Section 1.1), Autonomic Computing (Section 1.2), and Control Theory (Section 1.3). Section 1.4 presents *CiGri*, the case-study of this thesis.

1.1 Harvesting

The motivation for harvesting idle computing resources is two-fold. First, as computers are expensive and are powered on, it would be a waste of energy and money to leave them idle. Second, sparse idle machines represent a cheap way to expand the computing power available to users.

In this Section, we present various techniques from the literature to harvest idle resources of a set of machines in different contexts.

1.1.1 Personal Computers

Volunteer Computing [MC19] frameworks such as BOINC [And04] provide a way for anyone to lend their idle CPU times to community scientific projects. These scientific project range from extraterrestrial intelligence search with SETI@HOME [And+02], climate prediction [Sta+04], or simulating protein dynamics [Lar+09] which was helpful in the fight against Covid-19 [For+21]. The computation done by these projects are *embarrassingly parallel*, meaning that it is easy to separate the problem into smaller and independent sub-problems which can then be sent to the users. This characteristic make them a good candidate to exploit idle computing resources. Users install the framework on their machine and decide how many idle CPU time to donate to which project. When the machine is about to go idle, the volunteer computing framework pulls some work (*i.e.,* a piece of computation) from the server of the community project. The user machine then executes this work. If it finishes the previous task, and is still idle (*i.e.,* not used by the user), it requests a

new task. In the case where the computer is no more idle during the execution of a task, the task is killed and the CPU resources given back to the user.

A different approach is to push computation onto idle personal machines as with Condor [LLM87] for example. Users register their machines onto the Condor network, making their idle time available. Users can submit their computations to Condor, which will then find available machines on the network to execute these computations. The goal is to create a single computing cluster out of idle machines and make the machines available thought a scheduling mechanism. If a machine is no longer available while a Condor job is executing, the job is killed and put back in the Condor queue. The difference between Condor and Volunteer Computing is the nature of the workloads executed on the machines. Condor's workload is not necessarily embarrassingly parallel, but might be a big single job that is simply too resource demanding to be executed on a single machine.

1.1.2 In the Cloud

The cloud is also interested in using idle resources as every idle CPU tick is money lost. Amazon Web Services (AWS) introduced a 90% cheaper deal to use the idle resources of its EC2 cloud [Amaa; Amab]. Users submit their computations, and AWS will schedule them on appropriated available idle resources. Those computations are seen as second class citizens and will be interrupted when EC2 reclaims the resources. In [SRI16], the authors propose a pricing for idle cloud resources based on a probabilistic model of the potential revocation of the resources. The goal is to encourage the usage of the idle resources thanks to a fair pricing based on the offered guarantees. [Liu+20] approaches the problem with a game-theory strategy to dynamically define the price of cloud resources. The model the problem as a non-cooperative game between the cloud provider and the customers. The authors of [Wan+21] use online learning to predict the demand of users and compute the amount of resource that second class virtual machines can safely harvest.

1.1.3 On High-Performance Computing Clusters

HPC systems are also victims of idle resources. One solution to reduce the idle time would be to have finer, more precise, scheduling algorithm. But as smart as the scheduling algorithm could be, humans are still submitting the jobs. In particular, humans are giving the estimation of the duration of their computation (*i.e.,* the job walltime), and we are notoriously bad at it [CB01; MF01]. Work has been done to

improve the quality of the walltime estimations for the schedulers, but they forget other sources of variations (*e.g.,* network contention, file-system load, etc.). HPC systems are too dynamic and variable for scheduling to be the only solution. This section presents different approaches to harvest the idle resources HPC clusters.

OurGrid [Cir+06] aims to connect several geographically distributed clusters to a common grid interface. Users then have more available machines, and can make use during the day of currently idle machines in an opposing time-zone. This approach relies heavily on trust, as people from different laboratories can access any clusters, which raises some security questions. The harvesting of the idle resources is thus done manually by scientists that need more machines.

The work on the HPC/Big-Data convergence led to some solutions using Big-Data workloads to make use of the idle HPC resources.

In [Sou+19], the authors propose a co-scheduling approach based on the Slurm [YJG03] and Mesos resources managers [Hin+11]. The solution profiles the running application, and adapts the Linux control groups (`cgroup`) limits, *i.e.,* the amount of given resources at the operating system level, to fit the available resources to the actually needed resources. The remaining resources (from the cgroup point-of-view) are then available to the scheduler. This solution managed to reduce the overall resource usage and makespan, but does require a change of architecture for systems administrators, as well as setting up profiling.

Bebida [Mer+17] is another approach coupling HPC jobs and Big-Data workflows. The authors slightly modified the prologue and epilogue scripts of the *OAR* batch scheduler [Cap+05] to start and stop Hadoop [Shv+10] workers on the HPC resources when an HPC job finishes or starts. The idle HPC resources are then seen as a dynamic resource pool from the point-of-view of the Big-Data resource manager. Authors showed that Bebida improved the total usage of the machines, but degraded the mean waiting time of HPC jobs (due to the longer prologue and epilogue scripts).

Similarly to Bebida, the authors of [Prz+22] propose a solution to use the idle resource of a cluster using FaaS (Function as a Service), or Serverless, workloads. The jobs from such workloads last a few seconds, which make them good candidates to exploit the “holes” in the schedule. The solution couples the resources managers Slurm and OpenWhisk [Ope]. One limitation of the approach is the actual usage of FaaS for scientific workflows. The HPC community is conservative and the adoption of such complementary tools might be long [SMM18].

CiGri [GRC07] is the approach that we will focus on in this thesis. *CiGri* is a grid middleware that runs on top of several computing clusters managed by the *OAR* resources manager. It harvests the idle resources of the grid's clusters by periodically submitting jobs to the different resources managers with the lowest priority (*Best-Effort*). Those jobs come from Bag-of-Tasks applications, which are composed of numerous, small, independent, and similar jobs. Contrary to the previous solutions, the jobs submitted to *CiGri* are HPC jobs (just with different "shapes"), and there is a single scheduler decision at the cluster level. The limitation of *CiGri* is its submission algorithm which does not take into account the state of the cluster. This lack of feedback can lead to suboptimal resource usage and contention on shared resources of the clusters. A more detailed presentation of *CiGri* is done in Section 1.4.

Take away 1.1: Harvesting

HPC systems are victim to idle resources. Those resources are usually harvested by executing smaller, interruptible jobs. Solutions from the literature usually involve two levels of scheduling and jobs from two different natures.

1.2 Autonomic Computing

To deal with the increasing variability of computing systems and the error-prone manual management, IBM introduced in the early 2000s the notion of *Autonomic Computing* (AC) [KC03; HM08]. The main idea is to have systems that can self-adapt to the variations of their environment. This self-adaptation can take several forms based on the domain of the application.

- *self-configuration*: the system reconfigures itself automatically following high-level policies.
- *self-healing*: the system seeks opportunities to improve its performance.
- *self-optimization*: the system detects and repairs software or hardware problems.
- *self-protection*: the system defends itself against malicious attacks and cascading failures.

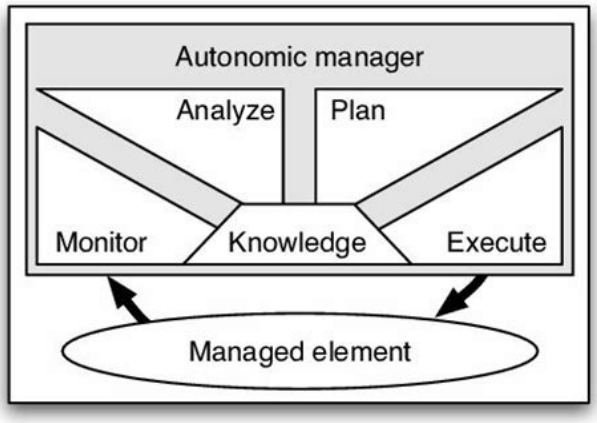


Figure 1.1.: Representation of the main tool of Autonomic Computing: the MAPE-K Loop [KC03].

To implement this vision onto a system, the latter needs to have a way to interact with the autonomic controller. Namely, the system must have a sensor and an actuator (or knob), which correspond to the interface between the autonomic controller and the managed element.

The main tool of AC is the *MAPE-K* loop, displayed in Figure 1.1. The acronym MAPE-K stands for the names of the different phases of the loop:

- **M**onitor: during this phase, the autonomic manager queries the sensor on the managed element.
- **A**nalyze: the value of the sensor is then analyzed by the manager to check if a change of direction is needed.
- **P**lan: if there is the need for a new trajectory, the "Plan" phase is responsible to define this trajectory.
- **E**xecute: finally, the new trajectory is translated into the action to execute on the knobs of the managed element.
- **K**nowledge: the knowledge is information shared among all the phases. It could be the feasible range of the knobs, previous sensor values, etc.

The MAPE-K loop is actually a framework to design feedback loops on computing systems, and it leaves room for interpretation, especially for the *Analyze* and *Plan* phases. Porter et al. [PRD20] surveyed the papers accepted to three autonomic and self-adaptive systems conferences (ICAC, SASO, and SEAMS) on how the MAPE-K loops present in the papers were implemented. They showed that over the 210

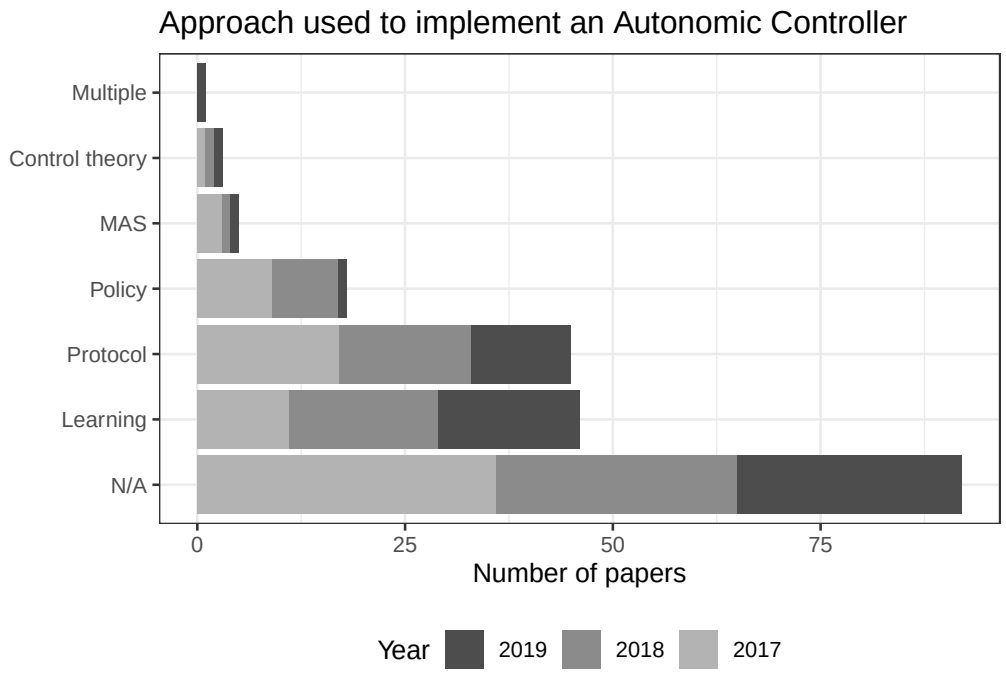


Figure 1.2.: Approaches used to implement an autonomic controller in 210 surveyed papers of three autonomic computing conferences. Regenerated from [PRD20].

surveyed papers, 63 were implementing their autonomic controller with rule-based strategies (Policy and Protocol categories in Figure 1.2), 46 were using Machine Learning strategies, 5 used multi-agents systems (MAS), and only 3 used Control Theory. The N/A category covers papers where no other approach fits.

The advantage of rule-based strategies is the simplicity of implementation and of understanding. It requires experts of the managed element to convert all their knowledge into rules or policies. The threshold controller depicted in Algorithm 1, is a type of such strategy.

Algorithm 1: Simple algorithm representing a rule-based autonomic controller. If the value of the sensor is greater than an upper bound, the action is decreased. If the sensor is lower than the lower bound, the action is increased. Otherwise, the current action satisfies the desired trajectory and is kept.

if *Sensor* $>$ *ThresholdMax* **then**
  $NextAction = PreviousAction - 1$
else
  if *Sensor* $<$ *ThresholdMin* **then**
    $NextAction = PreviousAction + 1$
  else
    $NextAction = PreviousAction$

Rule-based strategies do not give any guarantee on the behavior of the closed-loop system. The rules might also not cover all the accessible state-space, the system might oscillate, or maybe not even reach acceptable state [CDR14].

Using Machine Learning techniques is also very popular for implementing an autonomic controller. But it raises several difficult questions. How to be sure that the model has seen all the possible behaviors during its training to know how to react accordingly? How to actually be sure that the model will react accordingly? And let us not forget the computational cost of training the model, especially if the system is complex and has multiple inputs and outputs. The explainability of a solution is also a crucial factor for its adoption, and Machine Learning models lack in this aspect as they can be seen as total black boxes.

The proportion of Control Theory based autonomic controller in the Autonomic Computing literature can appear anecdotal, as seen on Figure 1.2, but they have a lot to offer. As opposed to rule-based and learning approaches, Control Theory is backed up by centuries of methods, theorems, proofs, protocols, etc., making it an interesting way to implement an autonomic controller. Control Theory has also been applied on distributed systems (HPC and Cloud) [CR23; Cer+21; MC20] with good results.

Take away 1.2: Autonomic Computing

Autonomic Computing deals with the runtime regulation of dynamical systems. The feedback loop can use different techniques (AI, rules, control theory, etc.). Control Theory represents a small fraction of the implemented solutions, but has a lot of potential.

1.3 Control Theory

This Section summaries the classical methodology of applying tools from Control Theory to computing systems [Fil+17; Lit+17; RMS17].

For the sake of clarity, we will use the following example. Imagine you have a brand-new CPU which is quite powerful, but can become very hot. High temperatures can damage the CPU, and you do not want to buy a new one (it was very expensive). In this example, we will go through the methodology to set up a controller on your CPU to avoid breaking it.

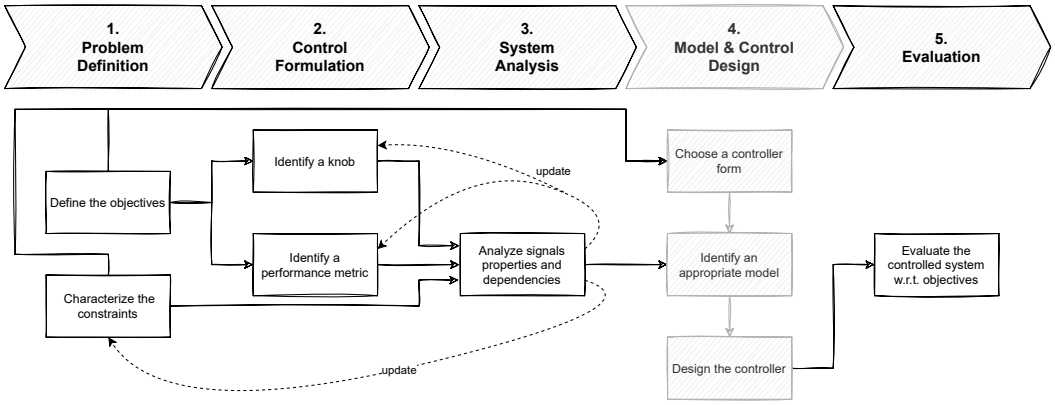


Figure 1.3.: Workflow of the Control Theory methodology [Cer+21].

Figure 1.3 presents the workflow for design a controller. Note that there can be cycles in the workflow where the objectives, knobs, sensors, etc., can be updated.

1.3.1 Identify the goals

The first step is to define what are the objectives of the closed-loop system. What are the metrics, available sensors on the system, states what we want to regulate, to control? What are the desired guarantees on the closed-loop system? Should the system be fast? Should it be allowed to oscillate, or to go over a certain limit? What about disturbances? What are the nature of the disturbances? Can they be modelled? Can they be anticipated? In Control Theory jargon, the desired state of the system is called *setpoint* or *reference value*.

In our simple example, the objective is to regulate the temperature. Lucky you, you have a thermometer on your board! So your sensor will be the value returned by this thermometer. There do not seem to be a lot of disturbances. You can imagine that your fan could break at some point and thus the temperature would increase.

1.3.2 Identify the knobs

Once the goals are described, one needs to identify the knobs of action, also called *actuators*. This means finding parts of the system whose variations impact the output of the system, positively or negatively. There might be several knobs of actions, or even some indirect knobs. Knowing the range of action of your knob is quite important in order not to ask for impossible inputs.

In our example, we can think of several knobs of action. The first one would be the amount of work given to the CPU. Another knob would be the frequency of the CPU itself. And this is much easier because you have a DVFS[1] mechanism on your CPU! Your knob will thus be the frequency of the CPU. The range of this knob is between the minimum and the maximum frequencies given on the CPU box.

1.3.3 Devise the model

This phase consists in identifying the relation between the change in the knobs and the next value of the sensors. In Control Theory terms, a *model* is a mathematical relation between the current state of the system (sensors) and the current input (knobs) to the next state of the system. If you already have knowledge of the system you can come up with such a relation. In physical systems, such knowledge could be physical laws (*e.g.,* Maxwell's equations, heat equation, etc.). Otherwise, *identification* experiments are required. During those experiments, the system is put under different types of inputs, and the output is observed. One kind of input is a *stair function* where the value of the knob is constant for some time, then changes to a new value (lesser or greater), stays constants for some time, etc. Then, with classical tools, such as linear regression, one is able to extract a model from those identification experiments.

For our CPU, we might be able to derive a model from thermodynamic equations, but we can also perform identification experiments. We can perform stair-shaped inputs, where the frequency would increase and then decrease and log the value of the temperature. What we might observe is that there might be a delay between the action taken on the system, inertia, and its visible reaction. This behavior can be taken into account in the model.

1.3.4 Design the controller

Once the model has been established, one can use the tools from Control Theory to design the controller adapted to the problem. There are plenty of controllers in the literature, all with their pros and cons. They can usually be derived from the model found previously, either by inserting the model into equations, or by using toolboxes like Matlab. During this phase, and based on the chosen controller, it might be possible to choose the closed-loop behavior of the system. The main properties, illustrated in Figure 1.4, are the following:

[1] Dynamic Voltage Frequency Scaling

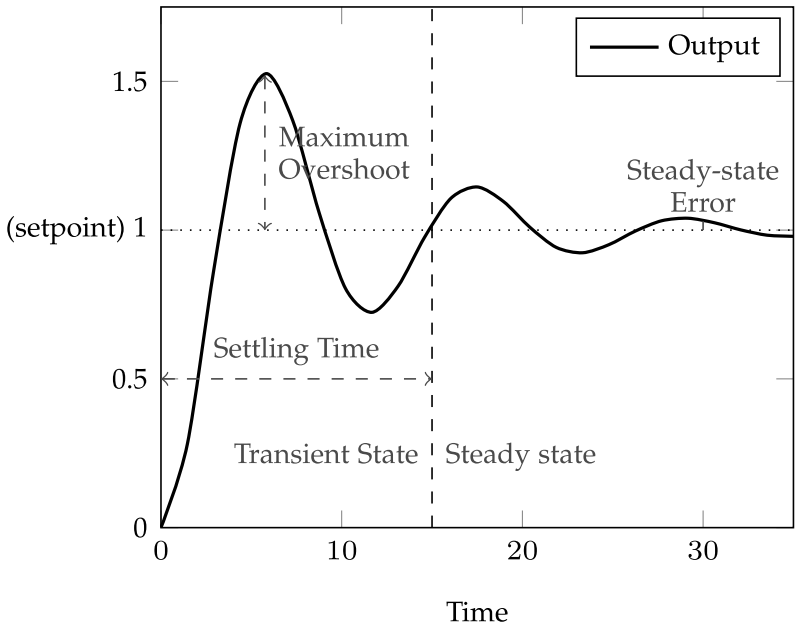


Figure 1.4.: Graphical explanation of the different properties of a controlled system (stability, settling time, overshoot) [She+17].

- *Stability*: the system reaches an equilibrium point regardless of the initial conditions.
- *Absence of overshooting*: the system does not exceed the desired state (setpoint/reference value).
- *Settling time*: the system reaches the equilibrium point in a guaranteed time.
- *Robustness*: the system converges to the setpoint despite the imprecision of the underlying model.

In the case of our CPU example, a "simple" Proportional-Integral controller seems more than good enough to be robust to potential disturbances and to converge to the desired state. Concerning the closed loop behavior, we really want to avoid overshooting as it could greatly damage the CPU. We would also like to have a decent settling time (maybe in the order of the second) and robustness.

In the following, we present the most common controllers of Control-Theory. Their description will be given in *discrete time* as we are in the context of computing systems.

Proportional Controller

The Proportional controller, or P controller, is the simplest controller of Control Theory. The idea of this controller is to react proportionally to the *current* error, *i.e.,* the different between the desired state (setpoint) and the current state (sensor). Note that the sign of the error is important as it helps the controller to steer the system towards the desired state. It is defined by a single gain K_p.

$$u(k+1) = K_p \times e(k) = K_p \times (y_{ref} - y(k)) \tag{1.1}$$

where u is the value to apply to the knob, y the value of the sensor, and y_{ref} the reference value.

The issue of the P controller is its imprecision. When designing a P controller, the experimenter must deal with some trade-off between all the desired closed-loop properties (stability, overshoot, settling time, precision, etc.). Increasing the precision might come at the cost to increase the maximum overshoot and increase settling time. In practice, P controllers are not often used due to these trade-offs.

Proportional-Integral Controller

One solution to improve the imprecision of the P controller is to add an Integral term on the *past* errors. The role of the integral term is to compensate the past errors and thus cancel out the steady state error. The integral part of the controller is parametrized by a gain noted K_i.

$$u(k+1) = K_p \times e(k) + K_i \times \sum_{j=0}^{k} e(j) \tag{1.2}$$

The PI controller is the most popular one as it remains simple but offers good guarantees.

Proportional-Integral-Derivative Controller

The P reacts on the current error, the PI also on the past error, and it is possible to add a component to react on the future. By adding a *derivative* term on the error to a PI, we get a PID controller. The idea of the derivative, is to detect changes

of trajectory. The derivative part of the controller is parametrized by a gain noted K_d.

$$u(k+1) = K_p \times e(k) + K_i \times \sum_{j=0}^{k} e(j) + K_d \times \frac{e(k) - e(k-1)}{\Delta t} \tag{1.3}$$

When a system is noisy, the computation of the derivative can lead to undesired behavior. A solution would be to add a filter after the sensor to smooth out the value and thus compute the derivative. But this defeats the purpose of the derivative term as the filter will slow down the reaction to the changes in trajectory. Thus, in practice, the derivative term is rarely used.

1.3.5 Trustable properties of the closed-loop system

After implementing the controller on the system, one should validate experimentally its closed-loop behavior compared to the expected ones of the design phase. This usually requires performing experiments where a set-point is given to the controller, and then observe its behavior.

Take away 1.3: Control Theory

Control Theory provides clear methodologies which yield controllers with proven guarantees for exact models, and trustable behavior for inexact ones. These controllers can also be robust to disturbances, and converge to the desired state within a desired time.

1.4 The *CiGri* Middleware & its need for regulation

This section gives more details about *CiGri*, the middleware of interest in this thesis.

1.4.1 The *Gricad* Computing Center

The *Gricad* computing center[2] provides computing and storing infrastructures to the researchers of the region of Grenoble, France. The center is composed of several

[2] `https://gricad.univ-grenoble-alpes.fr`

computing clusters, each with a focus (*e.g.,* `luke` for data processing, `froggy` for HPC, or `dahu` for HPCDA[3]). These clusters are linked together to form a computing grid. This grid is also impacted by the issue of unused resources explained above.

1.4.2 The *CiGri* middleware

CiGri [GRC07; oar23a] is a computing grid middleware set up in the *Gricad* computing center. It interacts with the *OAR* schedulers [Cap+05; oar23b] of each cluster. The goal of *CiGri* is to use the idle resources of the entire computing grid. Originally, *CiGri* was designed to reduce the stress on the different RJMSs of the grid when users had to execute large campaigns (tens of thousand of jobs or more).

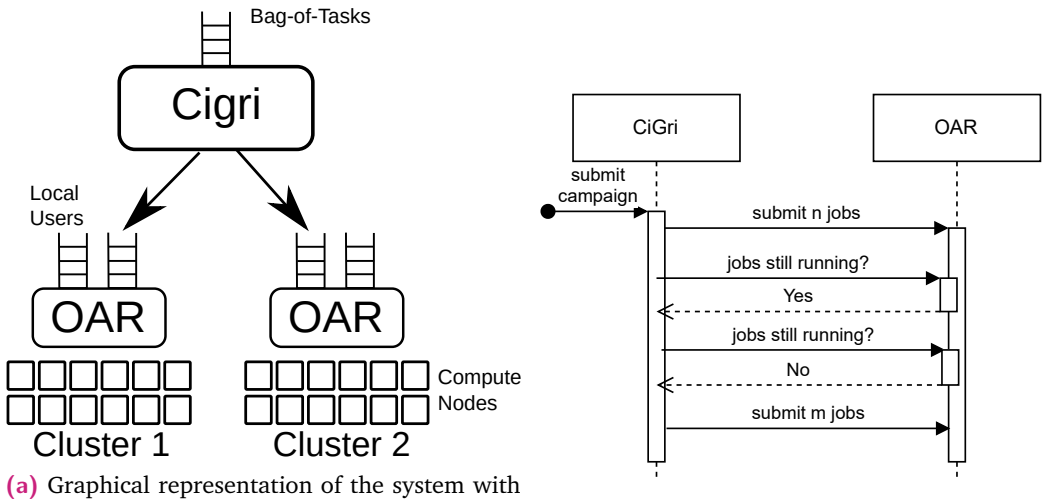
Users of *CiGri* submit *Bag-of-Tasks* applications to the middleware. These applications are composed of numerous small, identical, and independent tasks making them perfect candidates for the harvesting the idle resources. An example of such application is Monte-Carlo simulations, where the user will execute a large batch of random independent experiments to conclude on the aggregated results.

Once the application submitted to the middleware, *CiGri* will submit batches of jobs to the clusters of the grid. The jobs are submitted to the schedulers with the lowest priority (*Best-Effort*), which allows *OAR* to kill those *Best-Effort* job if a normal/premium user needs the resources. Figure 1.5a summarizes the interaction between *CiGri* and the different schedulers of the computing grid.

1.4.3 *CiGri* jobs

The hierarchy of *CiGri* jobs is as follows: a *job* belongs to a *campaign*, and a *campaign* belongs to a *project*. From the point-of-view of *CiGri*, a *campaign* is a *Bag-of-Tasks* application. One example of project is the processing of massive GPS data for the deformation of the Earth surface [Dép+18]. There are several geographic stations, with a lot of GPS data. The processing of each station can be grouped as a *campaign*, where the processing of a subregion is a *job*. Chapter 2 will present in more details the statistical characteristics *CiGri* jobs.

[3] *High Performance Computing and Data Analysis*



(a) Graphical representation of the system with *CiGri* and the different *OAR* schedulers of a computing grid.

(b) Graphical representation of the interaction between *CiGri* and a *OAR* scheduler.

Figure 1.5.: Interactions between *CiGri* and the different RJSMs of the grid.

1.4.4 The Need for Regulation

The current submission algorithm of *CiGri* works as a "tap". *CiGri* first submits a batch of *Best-Effort* jobs to the cluster scheduler, and then waits for all the submitted jobs to terminate before submitting again. Figure 1.5b represents the sequence diagram between *CiGri* and *OAR*. The size of the batch is defined by the simple ad-hoc Algorithm 2.

Algorithm 2: *CiGri* current job submission. It submits jobs like a tap: opens the tap and submits jobs to *OAR*. Then closes the tap and waits for all the jobs to be executed. Then it opens the tap again and submits jobs. The values of $rate$ and $increase\_factor$ have been empirically chosen by the administrators of *CiGri*.

Input : $rate$ (init. 3),
$increase\_factor$ (constant 1.5)
if *no running jobs* **then**
  $rate = \min(rate \times increase\_factor, 100)$;
  submit $rate$ jobs;
else
  submit 0 job;

One can think of scenarios where such a submission algorithm can lead to both the idling of resources or the unnecessary killing of *Best-Effort* jobs. For example, we can think of a situation where there are a lot of idle resources on the cluster, but a single job from the previous submission is still running and thus *CiGri* will not submit new jobs, as it must wait for this last job to terminate.

Take away 1.4

The harvesting capabilities of *CiGri* can be improved by integrating feedback mechanism in its decision process.

We believe that the utilization of the system can be improved by introducing feedback mechanism to the submission decision process. In particular, we think that using an Autonomic Computing [KC03] point-of-view coupled with tools from *Control-Theory* can yield an improvement in the total utilization of the platform. In this Thesis, we will focus on two regulation problems of *CiGri*.

Overload of the file-system

One regulation problem arises when considering the file-system of the cluster. As *CiGri* jobs are mostly short, and that no useful job does not either read or write to the file-system, then having too many *CiGri* jobs running can lead to an overload and a potential collapse of the distributed file-system. Such a problem is not taken into account in *OAR* for the scheduling of job, but could be with solutions such as [JPV23]. However, the performance of I/O-aware scheduling techniques are not yet satisfactory, and are not being implemented in production. They usually need either instrumentation with Darshan [Car+11; Car+09] for example, or guesses from the users. Previous work has been done in [Yab+19] by implementing a Model-Predictive controller, but the complexity of the solution, errors and scale of the experimental setup (10 nodes) yield a model that did not scale correctly. Hence, there is the need to keep exploring different control-based approaches.

Avoid loss of computing power from killing low priority jobs

In the case where we do not consider the distributed file-system of the cluster, one can think that using the all the resources of the cluster is easy and that *CiGri* "just" needs to keep enough jobs in the waiting queue of *OAR* to fill the entire cluster if available. Even if this represents a regulation problem of its own (addressed in [Sta+18]), it only considered idle resources to be wasted computing power. However, killed *CiGri* jobs also represent wasted computed power as *CiGri* jobs do not use check pointing techniques. The campaigns submitted by the users contain a given work to be executed, and will not be completed until all the work has been done. Thus, getting *CiGri* jobs killed is also counterproductive.

Take away 1.5: Limitations of *CiGri*

CiGri is able to harvest idle resources of a set of clusters, but does not take into account the state/load of the clusters. Applying Autonomic Computing with Control Theory tools could improve the resources' utilization.

1.5 Hypotheses of this thesis

In this work, we make the following hypotheses:

There is only one cluster in the grid This hypothesis allows us to focus on a single cluster. The generalization to several clusters can be done by considering one autonomic controller per cluster. An interesting path to consider would be the affinity between *CiGri* campaigns and the different clusters of the grid, with potentially similar techniques as in [Cas+00].

***CiGri* cannot kill one of its jobs** We suppose that once a job is submitted by *CiGri*, *CiGri* cannot kill it. The opposite case could lead to strange behaviors in the considered signal dynamics.

There are no other *Best-Effort* jobs in the system besides *CiGri*'s Regular users of the cluster can in practice also submit *Best-Effort* jobs. Having both *Best-Effort* jobs from *CiGri* and regular users could lead to some hidden competition for the idle resources, and introduce noise in the signals. As a first step, we consider that regular users cannot submit *Best-Effort* jobs.

The total number of resources in the cluster is constant We do not take into account the variation of the availability of the nodes. Meaning that no node are removed or added to the set of available nodes. Note that if we can have a dynamic sensor of the number of available resources in the cluster, the remaining of this thesis should be easily adaptable.

1.6 Conclusion & Research Questions

HPC systems are not making use of all their resources. This is due to the rigid nature of the jobs. The idle computing power can be harvested by executing smaller and interruptible jobs. However, those jobs can introduce perturbations in the system and degrade the performances of the premium users jobs. We believe that the harvesting of idle resources represents a regulation problem which can be tackled via an Autonomic Computing approach. To provide a runtime behavior with guarantees, we think that using tools and methods from Control Theory will help in terms of performance and adoption. The *CiGri* middleware represents a good framework to investigate these ideas and methods.

The remaining of this Part presents several autonomic feedback loops in the *CiGri* middleware to harvest the idle resources of an HPC cluster while considering the degradation of premium users' jobs performance. Chapters of this Part are articulated by the following *research questions* (RQ):

- **RQ1**: Can we characterize the *CiGri* jobs? How many jobs are in a campaign? How long is their execution time?
- **RQ2**: Can we design a controller with the methodology of Control Theory to harvest the idle computing resources of a cluster while regulating the load of a distributed file-system despite disturbances and unpredictability?
- **RQ3**: Controllers are usually tightly tuned for a specific system. To what extent can we plug a controller designed on a system A, on a system B?
- **RQ4**: Can a controller in *CiGri* reduce the computing power lost due to both idle resources and the killing of low priority jobs with some internal decision-making information from the scheduler?

A statistical study of the *CiGri* jobs from the last 10 years exploring **RQ1** is presented in Chapter 2. The **RQ2** will be addressed in Chapter 3 with the implementation of a Proportional-Integral controller following the methodology of Control Theory. Chapter 4 investigates **RQ3** by comparing the controller defined in Chapter 3 and two others types of controllers on their design cost and performance. We present a first step for answering **RQ4** in Chapter 5, and conclude Part I in Chapter 6.

2 Analyzing the characteristics of *CiGri* jobs

This chapter is based on [GRR22a] published at ComPAS 2022 with Olivier Richard and Eric Rutten. In this Chapter, we study the characteristics of the *Bag-of-Tasks* applications executed by *CiGri* on the Gricad computing center. The objective is to characterize such applications to be able to generated realistic synthetic *CiGri* workloads for the study of our controllers presented in the following chapters. This study relies on the *CiGri* jobs executed on the *Gricad* mesocenter between January 2013 and April 2023. The datasets as well as the analysis scripts are available on Zenodo [GRR22b].

2.1 Global Study of the *CiGri* Jobs

In this section, we are interested in the global characteristics of the *CiGri* jobs. Figure 2.1 shows the empirical cumulative distribution function (ecdf) for the execution times of the *CiGri* jobs, as well as the *Bag-of-Tasks* jobs from the DAS2 grid. *DAS2* [DAS] is the second generation of computing grids for Dutch universities. This is the only available workload containing explicit *Bag-of-Tasks* applications that we are aware of. Table 2.1 shows a side-to-side comparison of execution time metrics between the two considered computing grids. Globally, the execution times of the *Bag-of-Tasks* jobs executed on the *Gricad* center are longer than on *DAS2*. Indeed, half of the jobs on *DAS2* last less than 30 seconds, whereas half of the jobs on *Gricad* last less than a minute.
The longest job on *DAS2* runs for a few hours and the longest job for *Gricad* lasts several days. This difference could be explained by the usage of the grid by the users, the type of jobs, the number of available machines, etc.

Take away 2.1

Half of the *CiGri* jobs last less than a minute.

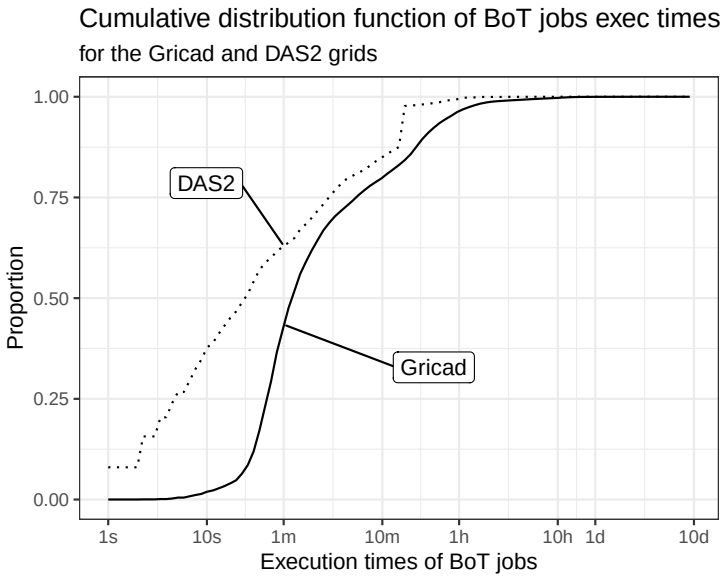


Figure 2.1.: Empirical Cumulative Distribution Function (ecdf) for the execution times of the *Bag-of-Tasks* jobs executed on the grids *Gricad* and *DAS2*. Globally, the *Bag-of-Tasks* jobs executed on the *Gricad* center are longer than on *DAS2*.

Metrics	Computing Grids	
	Gricad	*DAS2*
Number of BoT jobs	$\simeq 4.4 \times 10^7$	$\simeq 10^5$
Number of clusters	8	5
Minimum t_{exec}	1s	1s
Average t_{exec}	12m 43s	3m 51s
Maximum t_{exec}	9d 1h 27m	10h
Median t_{exec}	1m 13s	24s
Quantile 75 % t_{exec}	5m 25s	2m 51s
Quantile 95 % t_{exec}	48m 16s	15m 1s
Quantile 99 % t_{exec}	2h 49m 13s	43m 35s

Table 2.1.: Table summarazing the dataset. t_{exec} represents the execution times of the *Bag-of-Tasks* jobs from the two computing grids *Gricad* and *DAS2*.

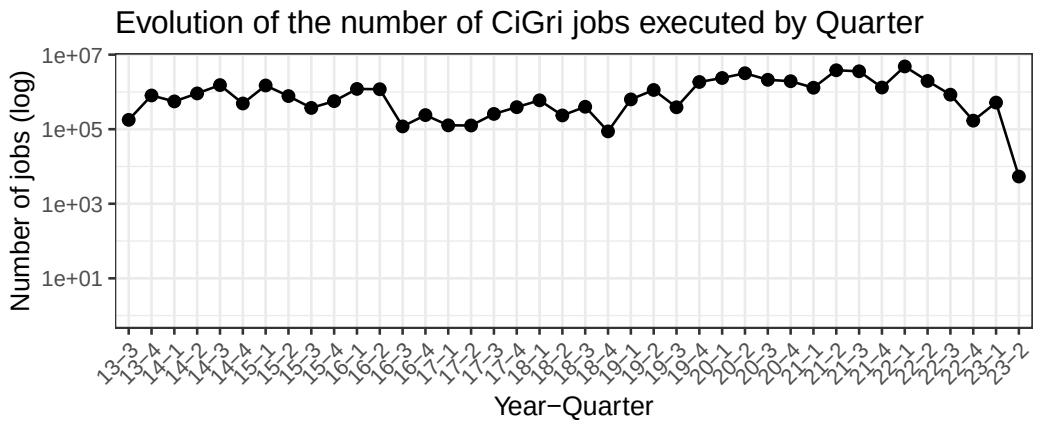


Figure 2.2.: Quarterly evolution of the number of *Bag-of-Tasks* jobs executed by *CiGri* on the *Gricad* mesocenter. Every quarter, there are at least 100000 jobs being executed, and in average around a million jobs.

Figure 2.1 shows the aggregated distributions during the last 10 years. The evolution during the years of the number of jobs executed by *CiGri* is depicted in Figure 2.2. We can see that there are at least 100000 jobs executed every quarter and often around a million *Bag-of-Tasks* jobs. We do not observe any trend either in increase or decrease of usage of *CiGri*.

Take away 2.2

In average, a million of *CiGri* jobs are being executed every quarter.

Figure 2.3 depicts the evolution of the mean and median execution times of the jobs executed per quarters of year. The median execution time is in the order of a few minutes, whereas the mean execution time is more in the order of dozen minutes or one hour. We observe variations in the two metrics, which hints for changes in usage of *CiGri*. But, as seen on Figure 2.2, the number of jobs does not vary much, which indicates that the execution time of the jobs executed varies.

To understand the reason of this change of behavior through the years, let us look at the projects executed. We remind the reader that a project contains campaigns which themselves contain jobs. Campaigns from the same project have similar behavior (number of jobs and execution times). Figure 2.4 shows the evolution of the distribution of the job execution times per quarter. We can see that there are often several modes of distributions. These modes correspond to the dominant usage of jobs from the same projects. For example, from the last quarter of 2019 (19-4) to the third quarter of 2022 (22-3) we can see that there is always a mode of distribution around one minute.

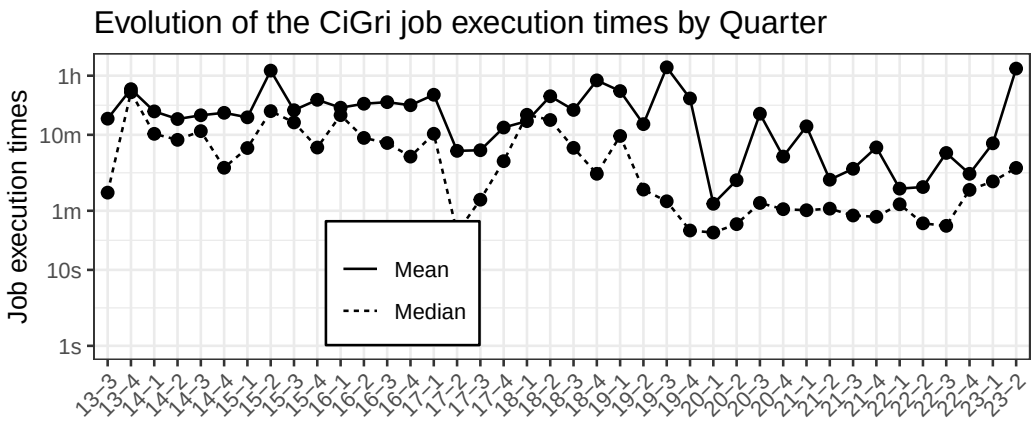


Figure 2.3.: Quarterly evolution of the mean and median execution times of the *Bag-of-Tasks* jobs executed by *CiGri* on the *Gricad* mesocenter. The median execution time is in the order of a few minutes, whereas the mean execution time is more in the order of dozen minutes or one hour.

Figure 2.5 shows the proportion of the *Bag-of-Tasks* jobs executed by quarter which belonging to given projects. It shows also the proportion of work (*i.e.,* total execution time times the number of machine used) for the *CiGri* projects. We can see that there is often a project submitting the majority of jobs executed for a given quarter. For example, during the years 2020 and 2021, the project `biggnss` was responsible for the majority of the *Bag-of-Tasks* jobs executed. However, it is noteworthy that the project having the majority of jobs executed during a period, is not necessary the one that consumed the most resources (work). For the same period, between 2020 and 2021, the `biggnss` project does not have the majority of the *CiGri* jobs executed during this period (quarter 4 of 2020 for example). Note that we can now indeed confirm that from 19-4 to 22-3, the same project (`biggnss` in this case) had the majority of jobs being executed.

Take away 2.3

The project with the most jobs executed is not always the one doing the most work. This hints that there are different "shapes" of *CiGri* campaigns: a lot of small jobs or fewer jobs but longer.

2.2 Study of the *CiGri* Projects

Let us now focus on the distribution of execution times among the projects. Figure 2.6 depicts the distribution of the execution times of the 10 *CiGri* projects with

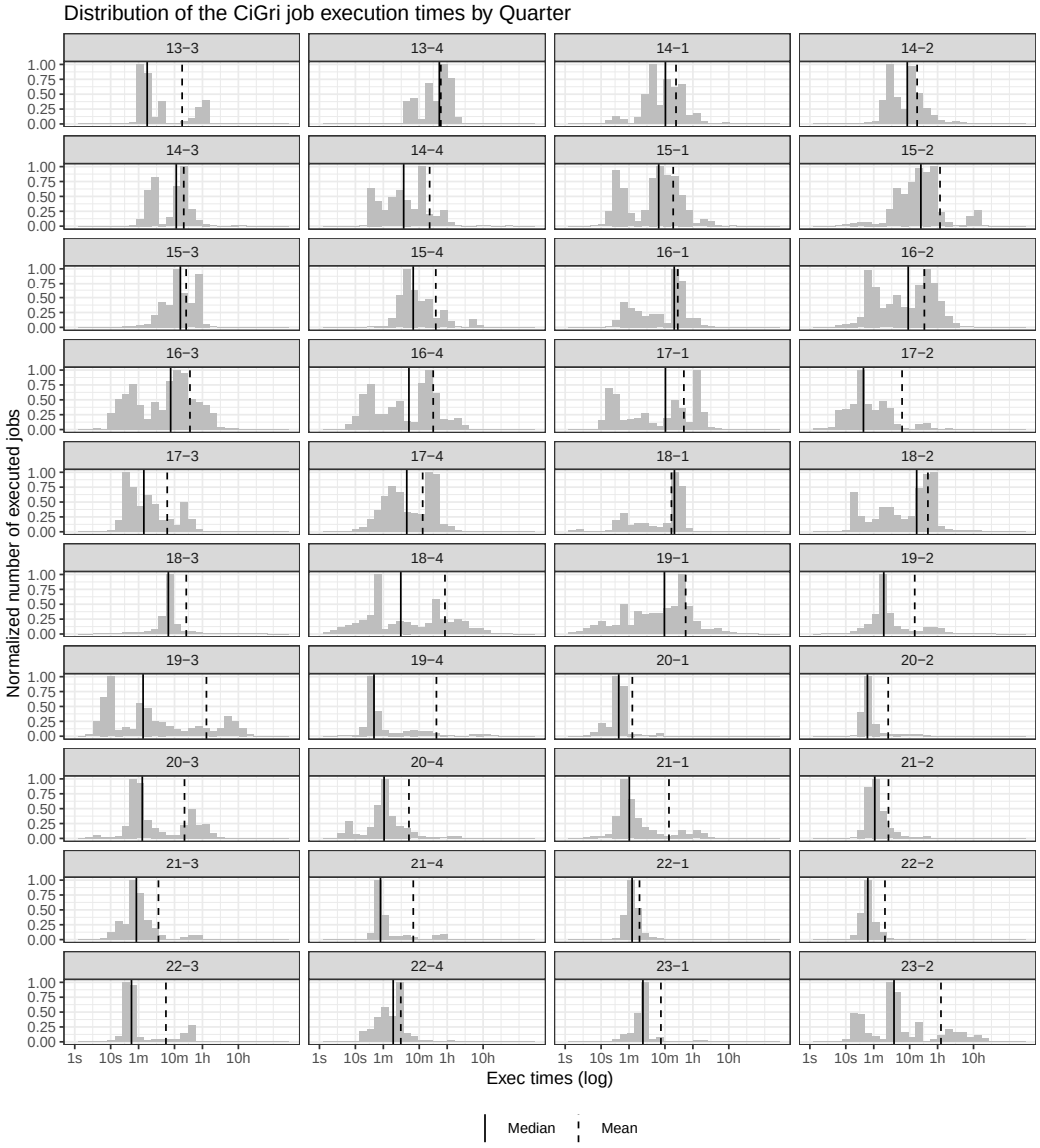


Figure 2.4.: Distribution of the execution times per quarter. We can see the dominant usage of *CiGri* by some project. From 19-4 to 22-3, most of the jobs last around 1 minutes, which hints that most of the jobs come from the same project.

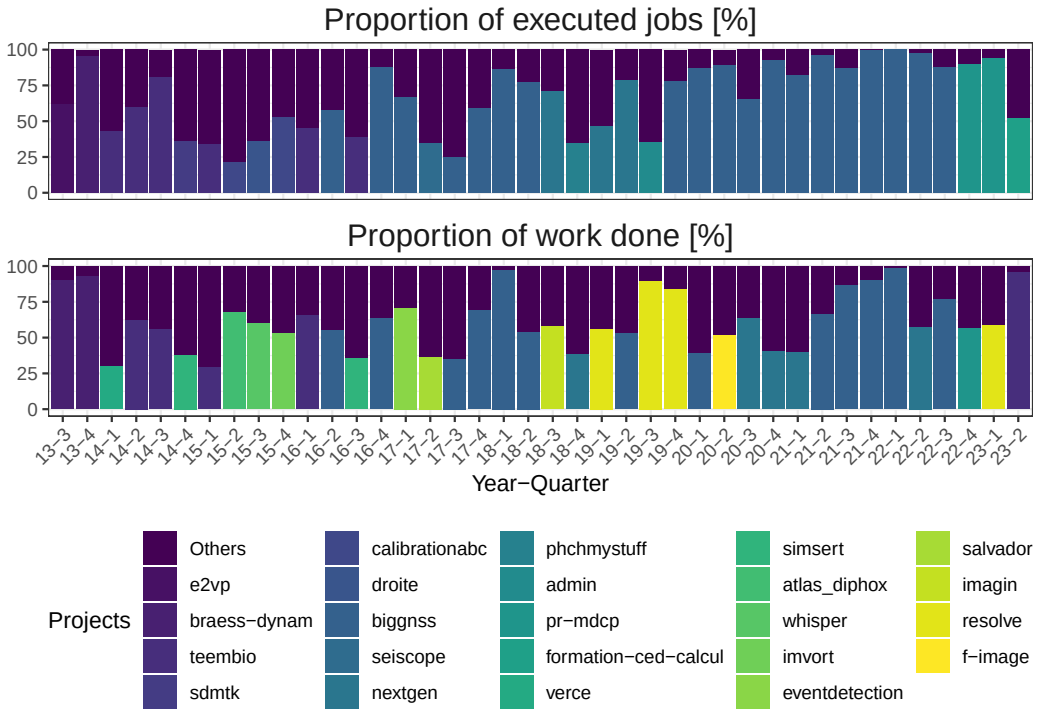


Figure 2.5.: Evolution of the proportion of *Bag-of-Tasks* jobs executed by *CiGri* on the *Gricad* computing center over the years, as well as the evolution of the work (execution time times the number of resources) of *CiGri* jobs. We observe that there is often one project which has the majority of the jobs executed during a quarter. However, this project does not necessarily perform the most work. For example, for the year 2020, the `biggnss` project has the majority of jobs executed, but not the majority of executed work.

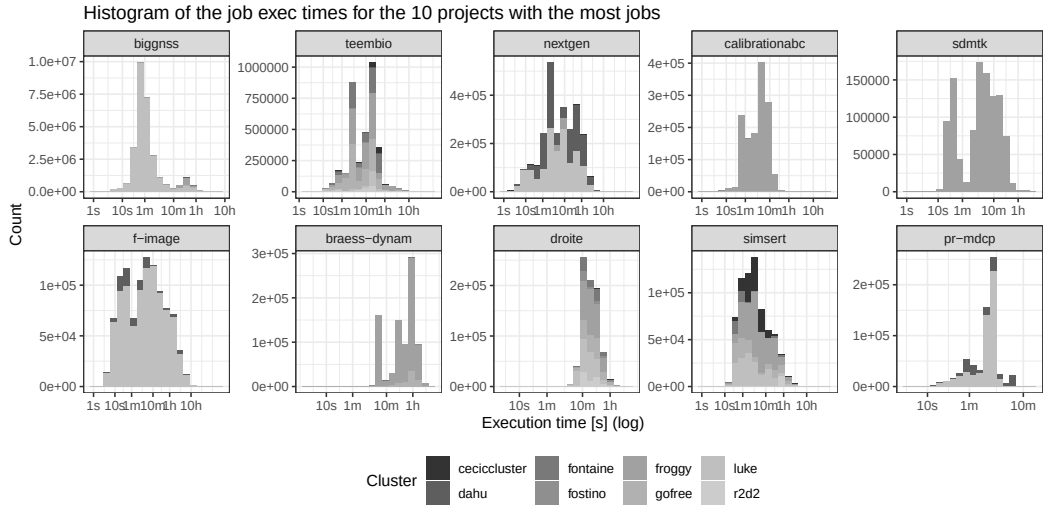


Figure 2.6.: Distribution of the execution times for the 10 *CiGri* projects with the most jobs. We observe that most projects have a clear unique mode of distribution. This mode can be very wide, like for `f-image` and `simsert`, or thin, like for `biggnss` and `pr-mdcp`.

the most jobs. We can see that for most of the project, there is one clear mode for the execution time as well as a heavy distribution tail. The projects like `teembio` and `sdmtk` have two modes for the execution times. In this case, there are several types of campaigns in the same project, one for each mode. The width of the mode is also different from project to project. For example, `biggnss` and `pr-mdcp` have a very thin mode of a few minutes, where projects like `f-image` and `simsert` have a range from a few seconds to a couple of hours. Note that as the jobs are executed on a computing grid composed of several clusters, we also plot the clusters on which the jobs have been executed.

To model the distributions showed in Figure 2.6, we perform a *goodness of fit* test with the following long-tail distributions: *Normale*, *Log-Normale*, *Frechet*, *Gamma* and *Weibull*. We consider only the 10 projects with the most jobs, and follow the methodology presented in [Jav+09; BNW03]. For each campaign of these projects, we randomly chose 50 jobs. As the behavior of the jobs can vary for different clusters of the computing grid, we will consider the distributions per pair (project, cluster). For each of these pairs, and for each of the selected long-tail distributions, we perform a Cramer-von Mises test [Dar57]. This test computes the distance between the empirical distribution function of the execution times of the selected jobs and the empirical distribution function of the selected distributions. The smaller the distance, the more likely the execution times distribution follows the tested distribution.
As the Cramer-von Mises test generates *randomly* the empirical distribution function

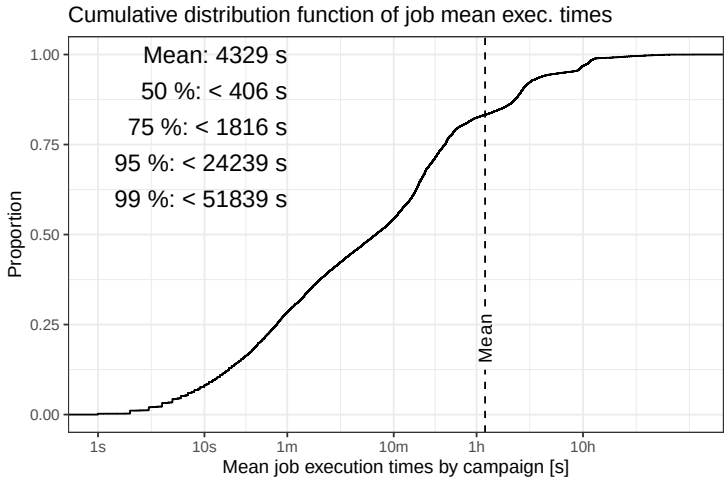


Figure 2.7.: Empirical cumulative distribution function of the mean execution time of a campaign. The average mean duration is around 1 hour, but the median mean duration is around 10 minutes.

of the selected distributions, we repeat this process 30 times and take the mean of the distances. Globally, the distributions which fit the best the execution times are the laws *Log-Normale*, *Frechet* and *Weibull*. The quality of the fitting for these laws is near identical. This concurs with the result of [IE10; Ios+08] for other computing grids.

Take away 2.4

Different projects have different distribution of execution times. Most project have a clearly defined unique mode and a long tail. The laws *Log-Normale*, *Frechet*, and *Weibull* fit the best the distributions of execution times.

2.3 Study of the *CiGri* Campaigns

Figures 2.7 and 2.8 show respectively the empirical cumulative distribution function for the mean job duration and the number of jobs in a campaign. Half of the campaigns have less than 50 jobs, and half of the campaigns have a mean execution time in the order of the dozen of minutes.

Take away 2.5

The average campaign has around 2500 jobs with a mean execution time of 1 hour and 15 minutes.

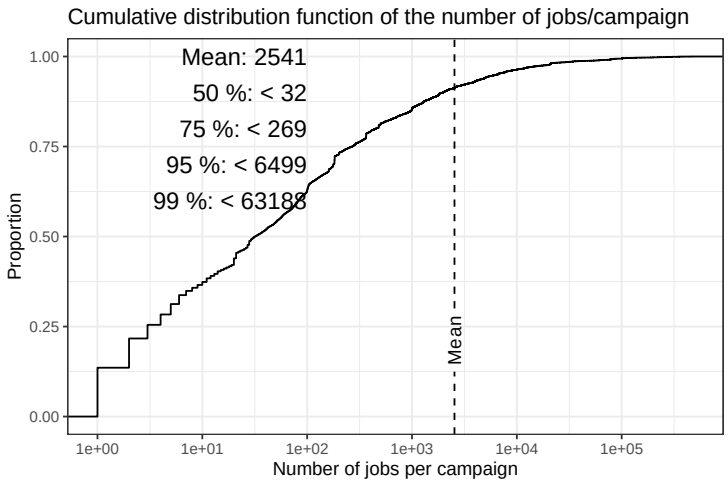


Figure 2.8.: Empirical cumulative distribution function of the number of jobs per campaign. Half of the campaigns have less than 50 jobs, but there are in average 2500 jobs per campaign.

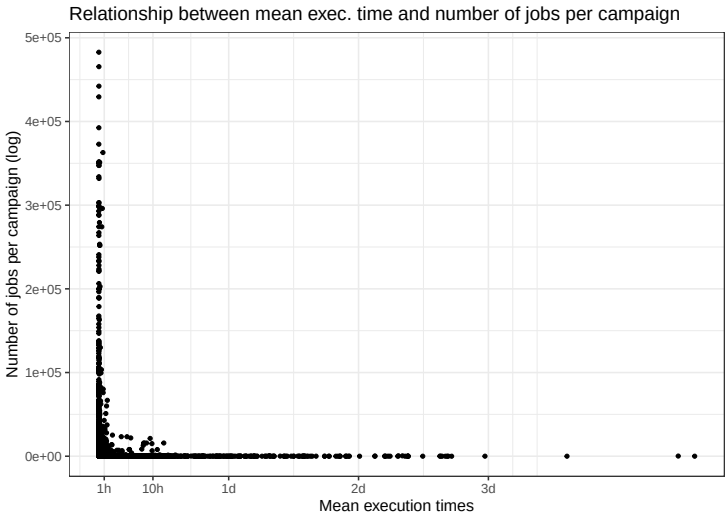


Figure 2.9.: Relation between the number of jobs in a campaign and the mean duration of its jobs. We observe that large campaigns have short jobs, and small campaigns have long-lasting jobs.

Figure 2.9 plots the relation between the mean execution times of the jobs in a campaign and the number of jobs in this campaign. We observe that large campaigns have short jobs, and small campaigns have long-lasting jobs.

Take away 2.6

Large campaigns have short jobs, and small campaigns have long-lasting jobs.

2.4 Job and Campaign model used in this Thesis

In this thesis, we will represent a campaign as a set of jobs having the exact same theoretical execution time. In practice, we implement a job using the `sleep` command with the desired execution time. Choosing this job representation, instead of executing real *Bag-of-Tasks* applications, allows us to better control the characteristics of the campaign and create all sorts of synthetic scenarios.

Some work remains to be done to be able to generate *realistic CiGri* campaigns. The *realistic* aspect might be tricky as there are several dimensions to a campaign (*e.g.,* number of jobs, job execution times, distribution of execution times), each with their variability. The work done in [CL22] for MPI applications might give a good example and starting point.

There are still some open questions. In particular, the origin of the long-tail of the execution time distribution is still unknown. We suspect that it can come from the overhead of the scheduler to start and stop the jobs and/or from the load of the cluster (*e.g.,* file-system, network) at the moment of the execution.

3 A Proportional-Integral Controller to harvest idle resources

In this Chapter we implement a Proportional-Integral (PI) controller in *CiGri* to regulate the load of a distributed file-system. As the target audience of this thesis is mainly computer scientists with limited knowledge of Control Theory, this Chapter aims to present the methodology in a pedagogical fashion. The reader can also go through a more hands-on introduction to Control Theory with our tutorial to introduce Control Theory to computer scientists: [Gui+b].

3.1 Goals, Control Objectives, and Sensors

In this Chapter, we are focusing on the research question *RQ2* seen in Section 1.6. We want to reduce the overhead on the *I/O* operations of the regular users of the cluster due to the *CiGri* jobs. In other words, we want to regulate the impact of the *CiGri* jobs on the shared file-system of a cluster.

3.1.1 Sensor on the File-System

The overhead on the *I/O* operations cannot directly be observed easily. One way would be to submit a specific job periodically and measure its performance to have an idea of the load of the file-system. But this would mean submitting an extra job, which might stay in the waiting queue and introduce some delay in the measurement. Moreover, such a job will also use nodes on the cluster. We instead will measure indirectly this overhead on *I/O* operations by using a sensor on the load of the file-server (*i.e.,* the load of the machine hosting the file-system). We decided to use the `loadavg` metric [FZ87], present on every UNIX system. This metric has some interesting properties. First, it is already well known by system administrators. This means that they know what values of this metric are acceptable on their system. It is also available on every Unix machine under `/proc/loadavg`. The value of this

sensor is updated every 5 seconds which is fast enough compared to the duration of most HPC jobs, see Chapter 2. This metric also carries some inertia due to its definition with an exponential filter:

$$Q_i = Q_{i-1} \times \left(1 - e^{-T}\right) + q_i \times e^{-T}, Q_0 = 0 \tag{3.1}$$

where, Q_i is the value of the `loadavg` at iteration i and q_i is the number of processes running or waiting for the disk at iteration i. There exists several variations of this metric with different factor of filtering (T). Those factors correspond to the period of consideration for the metric.

```
$  cat /proc/loadavg
0.62 0.63 0.58 1/1805 49884
```

The first three values returned correspond respectively to the load averaged over one minute, five minutes, and fifteen minutes.
The longer the period, the smoother the value of the `loadavg`, but also the slower the response of the sensor to a variation. In the following, we consider only the first value of `/proc/loadavg`, *i.e.,* the load averaged on the last minute.

This variation of the `loadavg` metric can however be noisy. Indeed, as this is a system-wide metric, it might capture behaviors that do not belong to the file-system. Moreover, this metric works nicely for a distributed file-system such as NFS, where there is a single machine hosting the file-system on the server side. In the case of parallel file-system, where there are several *I/O* nodes, meta-data nodes, etc., it might be more difficult to use this exact metric. One can average the different `loadavg` values of the different nodes belonging to the file-system, but the aggregated metric would probably lose most of its meaning. In our case, we are interested in small to medium computing centers, where using NFS as a shared file-system is a perfectly acceptable option. The implementation of this sensor is done by opening a `ssh` connection from the machine hosting *CiGri* to the file-system server and periodically reading the content of `/proc/loadavg`:

```
ssh fileserver 'while true; do L=$(cat /proc/loadavg);\
                D=$(date +%s);\
                echo -e $D $L;\
                sleep 5;\
                done' >> /tmp/loadavg_storage_server &
```

In the case of NFS, the configuration defines a number of *workers* that can manage *I/O* requests in parallel (8 by default). Each worker is in its own process, which makes it easy to observe with `loadavg`. Hence, the maximum acceptable load on the file-server is a value of `loadavg` equal to the number of NFS workers. Note that the metric can reach an even greater value, but it will not be because of the file-system load. The minimum load is zero, by definition of the metric.

3.1.2 Sensors on the Cluster

Only looking at the load of the file-system might lead to some degenerate cases for the closed-loop system. Imagine the situation where the *CiGri* jobs perform a very small quantity of *I/O*, and that even if all the resources are being used, the load of the file-server is below the reference value. Then the controller will perceive that it is possible to keep increasing the number of jobs submitted to reach the reference value. However, as the cluster is already full, the jobs will go in the waiting queue, and the only action of the controller is to fill faster and faster the waiting queue of *OAR*. Hence, we also need a sensor on the state of the cluster to avoid such cases. Fortunately, *OAR* offers an API where we can extract, directly or indirectly, the number of jobs in waiting queue or currently running.

Note that in order to be reactive to potential overload of the file-system, *CiGri* cannot afford to have a lot of jobs in the waiting queue of *OAR*. Indeed, as *OAR* has no notion of file-system load, if there are idle resources, and (fitting) waiting jobs, those jobs will be executed. The desired value for this sensor would be close to zero.

Take away 3.1: *CiGri* sensors

To sense the load of the distributed file-system, we use the `loadavg` metric of the machine hosting the file-system. For information about the state of the cluster, we use the *OAR* API.

3.2 Actuators

At first glance, the only actuator, or knob, at *CiGri*'s disposal to impact the load of the file-system is the amount of jobs it submits at every iteration (every 30 seconds). However, we can also consider which jobs are in the submission. Mixing jobs from different campaigns in a single submission could lead to a better control, where the

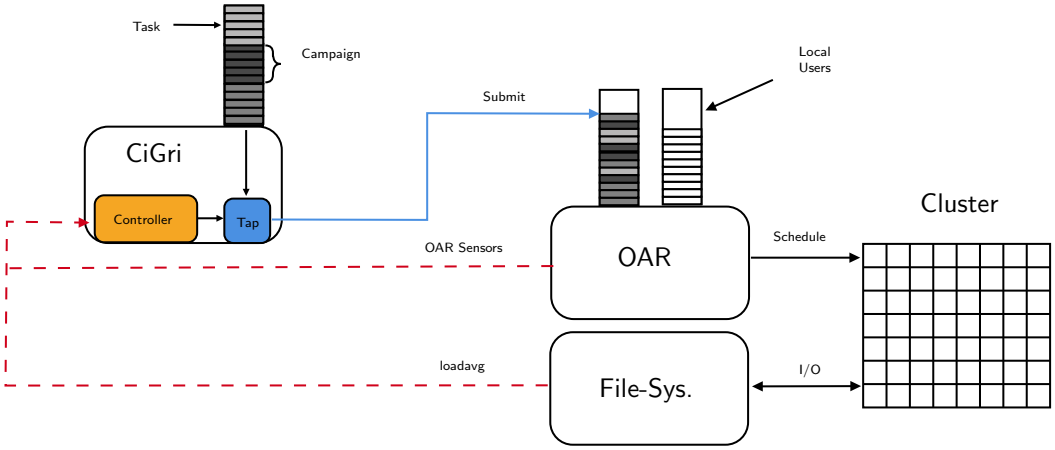


Figure 3.1.: Graphical representation of the feedback loop in *CiGri*.

additional knob is the proportion of jobs from each campaign. In this thesis, we consider only the number of jobs submitted by *CiGri* at each iteration. The work done in my Master Thesis [Gui20; Gui+21a] explores the proportion of jobs from two different campaigns to improve the control of the load of the file-server, and will be summarized in Section 3.7.2. Figure 3.1 summarizes the definition of the loop in *CiGri* with the sensors, and the actuator.

3.3 Model of the system

As we do not have any a priori knowledge of what would be a model of the system, we perform identification experiments. The identification aims at finding the relation governing our system. We intend to determine an expression of the following form:

$$y(k+1) = \sum_{i=0}^{k} a_i y(k-i) + \sum_{j=0}^{k} b_j u(k-j) \tag{3.2}$$

Where:

- $y(k)$: output of the system at step k. In our case, the load of the file-system
- $u(k)$: input to the system at step k. In our case, the number of jobs sent from *CiGri* to *OAR*
- $a_i, b_j \in \mathbb{R}$: coefficients of the model

Equation 3.2 represents a *linear model*. Such a model is simple, but does not necessary fit all the systems. In some cases, a change of variable is needed to fall back on a linear model. The model does not need to be *perfect* to have a good control. An approximated model which captures the behavior of the system might be enough in some cases.

3.3.1 Identification Experiments

To find the coefficients (a_i, b_j), we will study the system in open loop. This means without feedback from the system. We change the input and observe the variation in the output. We submit steps of number of jobs and see the impact on the load of the file-server. By step of the number of jobs we mean that at time $k, \forall k < k_{step}, u(k) = u_0$ and $\forall k \geq k_{step}, u(k) = u_{step}$, with $u_{step} \gg u_0$. We then look at the behavior of the file-server load during these steps to extract the coefficients of the model.

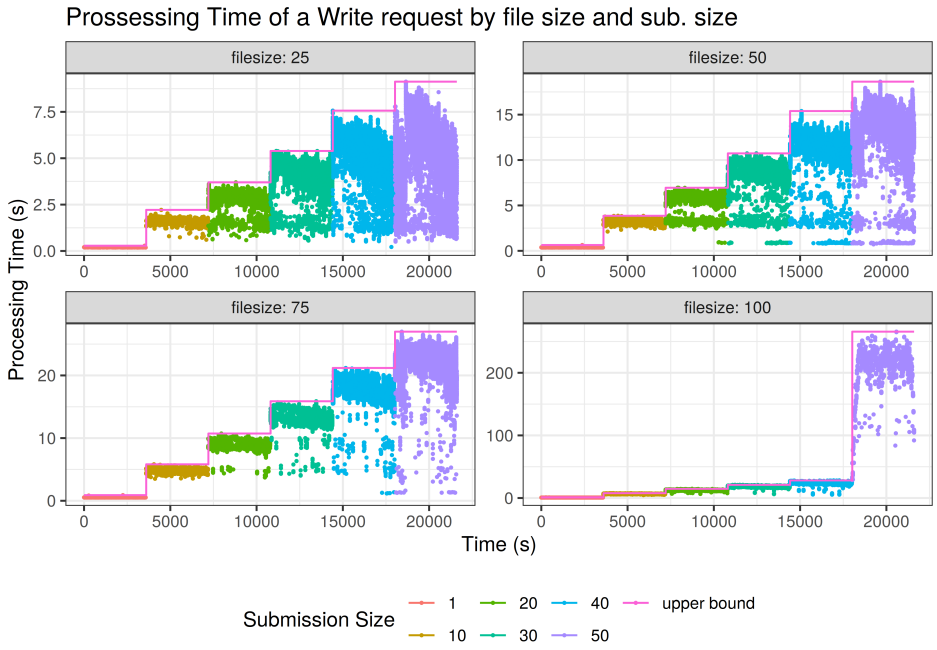
In the following, we consider 4 different sizes of files: 25, 50, 75 and 100 MBytes. For each size of file, we have 6 steps of values: 1, 10, 20, 30, 40 and 50 concurrent jobs. For each step, we execute 120 consecutive identical submissions of the step. Note that we will only consider the writing *I/O* operation as it is the most costly.

Figure 3.2 represents the results the identification phase. Figure 3.2a corresponds to the time to write the files, and Figure 3.2b depicts the file-server load, with the black dashed line representing its total number of NFS workers. We observe that increasing the number of simultaneous write requests increases the time to process these requests. The load of the file-server follows the same pattern.

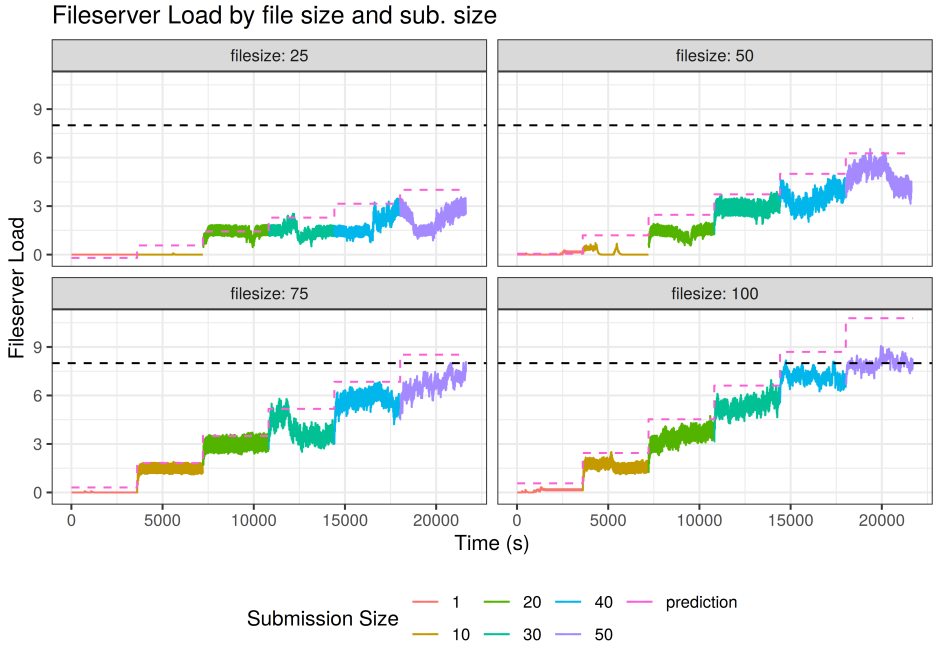
Take away 3.2

The `loadavg` metric seems to adequately represent the perturbations of the file-server and the overhead on the *I/O* operations.

Note that for a submission of 50 write requests of 100 MBytes, the processing time skyrockets, and the approximation of the system having a linear behavior is no longer valid. We see that for such a submission, the load of the file-server reached the dashed line, representing the number of workers in the system. In this situation, the file-system is overloaded and cannot deal with all the requests. **This motivates our goal to regulate the load in order not to reach this limit.** Thus, the `loadavg` metric also provides a way to detect such an overload.



(a) Time for the NFS file-system to process concurrent write requests (colors) of different sizes (facets).



(b) Load of the machine hosting the NFS file-system through the `loadavg` metric.

Figure 3.2.: Identification experiments. For the different file sizes, we write n concurrent files onto the NFS file-system and record the time to process each request (Figure 3.2a) and the load of the machine hosting the file-system (Figure 3.2b). The dashed line on Figure 3.2b represents the theoretical maximum load of that the NFS server can manage based on its number of workers. We can see that writing 50 concurrent 100Mbytes files overloads the file-system, the processing time explodes and the load is at the theoretical maximum.

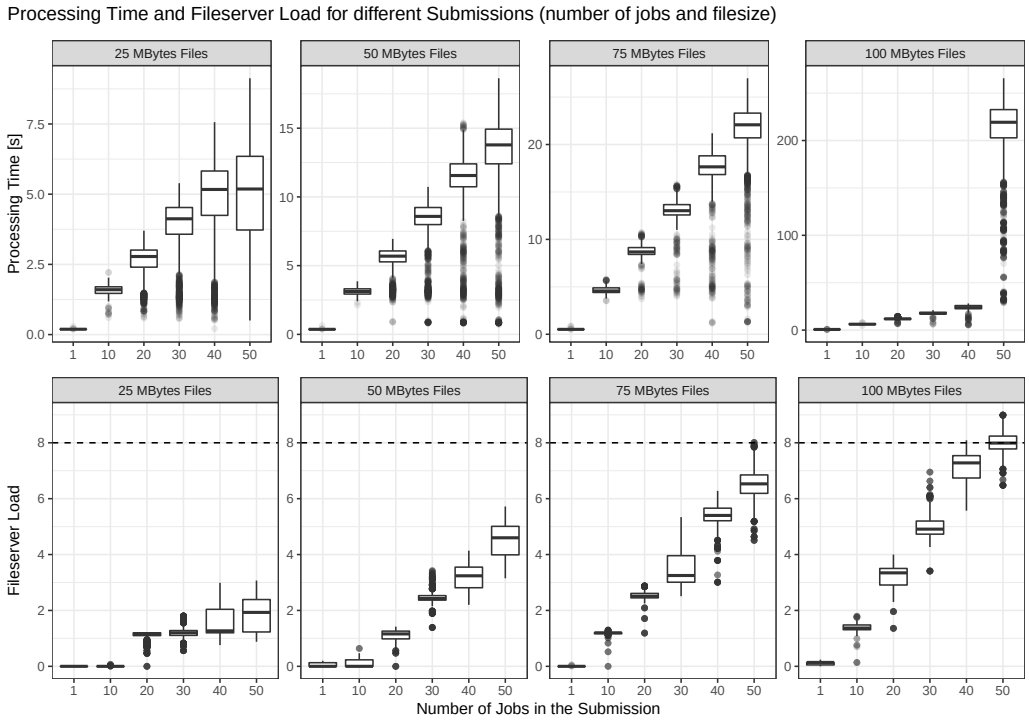


Figure 3.3.: Processing time (top) and the file-server load (bottom) for different submissions in number of jobs and *I/O* loads. It represents the identification phase. We vary the quantity of *I/O* (columns) and the number of simultaneous write requests/jobs in x-axis. We observe that the `loadavg` sensor captures the (over)load of the file-system.

Figure 3.3 is another representation of Figure 3.2 without the temporal dynamic of the system, which will be helpful for modelling the steady-state components of the model of our system.

3.3.2 Modelling

From the data gathered during the identification experiments in Figure 3.2, we model the relation between the value of the `loadavg` (y), the *I/O* load (f), and the number of jobs (u). We consider the upper bound of the `loadavg` in the modelling to be conservative. By fitting a linear regression on the open-loop data, we get the following relation:

$$y = \alpha + \beta_1 f + \beta_2 u + \gamma f \times u \tag{3.3}$$

To design the controller for our system, we need a relation as showed in Equation 3.2. As a first approach, we suppose that we are looking for a *first order system*. The order of the system corresponds to the degree of dependence of $y(k+1)$ to previous values of y (first order: $y(k)$, second order: $y(k)$ and $y(k-1)$, etc.). First order systems have a limited set of behaviors, which makes their study easier. Higher order systems have more than one degree of dependence, which increases their complexity but also their realism. But they can be approximated to a first order system with some hypothesis on their poles [Hel+04].

In our case, this means that $\forall i, j > 0, a_i = 0, b_j = 0$ in Equation 3.2. We are thus looking for the following relation between the input (u) and the output (y) at step k:

$$y(k+1) = ay(k) + bu(k) \tag{3.4}$$

By definition of the `loadavg` metric (Equation 3.1 and [FZ87]), we have $a = \exp\left(-\frac{5}{60}\right)$. The `loadavg` value updates itself every 5 seconds, faster than a period of *CiGri* ($\Delta t = 30s$). Thus,

$$a = \left(\exp\left(-\frac{5}{60}\right)\right)^{\frac{\Delta t}{5}} \tag{3.5}$$

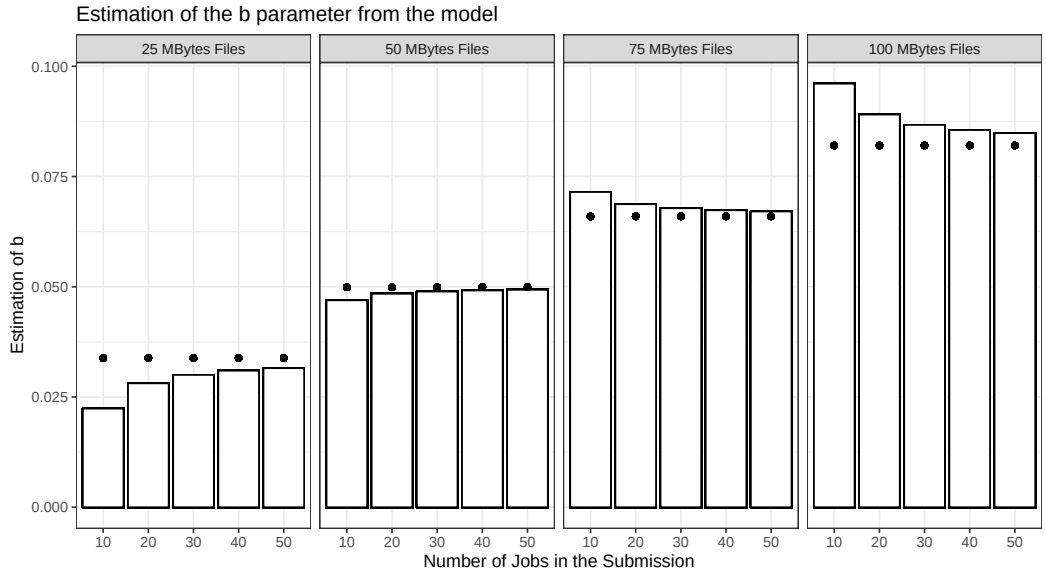


Figure 3.4.: Estimation of the parameter b for the model of the system. The bars correspond to the estimation for a number of jobs (u) and a *I/O* load (f). The points represent the value of the estimation when considering $u = +\infty$. We observe that the estimations converge towards these limits values. We will take these limits as values for b.

In order to find the value of b, we must look at the behavior of our system in steady state, meaning when the system has converged. In a steady state (ss), we have the following relation:

$$y_{ss} = a \times y_{ss} + b \times u_{ss} \implies b = \frac{y_{ss}\,(1-a)}{u_{ss}} \tag{3.6}$$

Figure 3.4 depicts the estimation of b for the values of u_{ss} and y_{ss} from the identification in Figure 3.2.

From Equation 3.6 and Equation 3.3 we get:

$$\begin{aligned} \lim_{u \to +\infty} b &= \lim_{u \to +\infty} \frac{(\alpha + \beta_1 f + \beta_2 u + \gamma f \times u)\,(1-a)}{u} \\ &= (\beta_2 + \gamma f) \times (1-a) \end{aligned} \tag{3.7}$$

This means that the value of b depends on the file size of the current jobs (f). We also plot this limit value of b on Figure 3.4 as points. We can see that the estimations of b converge to this limit value.

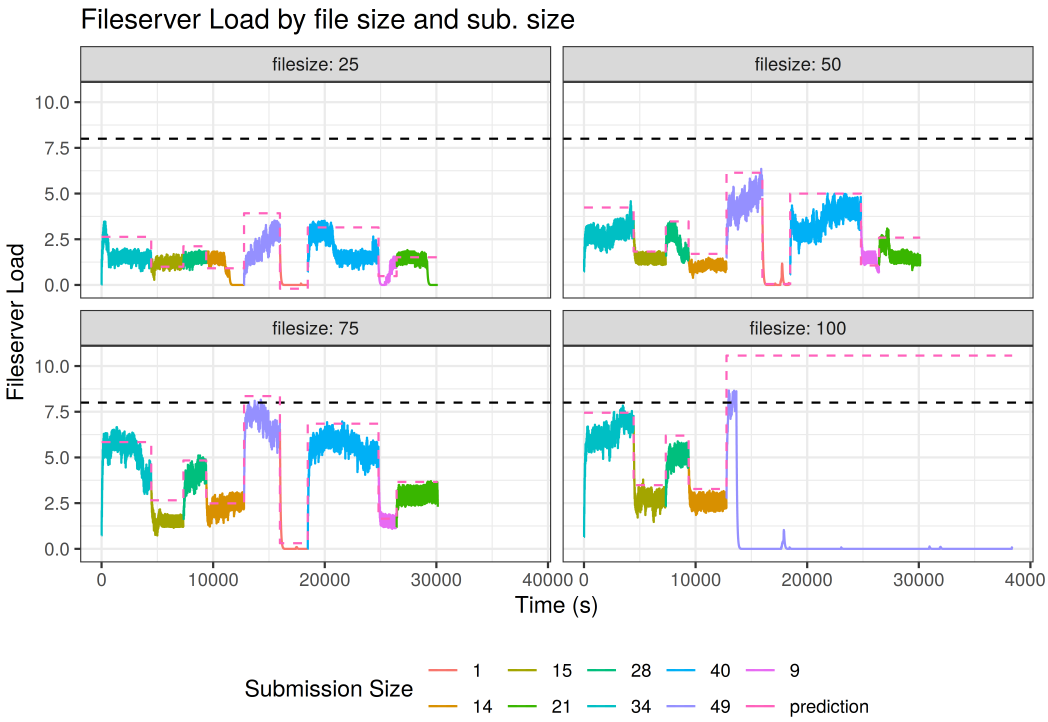


Figure 3.5.: Load of the NFS file-system for random steps in the number of concurrent writes. We observe that for 100Mbytes and 49 concurrent writes, the file-system collapses and fails to continue, which shows the importance of not overloading the distributed file-system of a cluster.

To sum up, we pick the model parameters to be:

$$\begin{cases} a = \left(\exp\left(-\frac{5}{60}\right)\right)^{\frac{\Delta t}{5}} \\ \quad = 0.6065307 \\ b = (\beta_2 + \gamma f) \times (1 - a) \\ \quad = 0.017761 + 6.4273488 \times 10^{-4} \times f \end{cases} \tag{3.8}$$

We evaluate our model by executing random steps of concurrent writes on the NFS file-system, and log the `loadavg` of the NFS server. Results can be seen on Figure 3.5. We observe that for 100Mbytes and 49 concurrent writes, the file-system collapses and fails to continue, which shows the importance of not overloading the distributed file-system of a cluster.

Take away 3.3

We performed identification experiments to find a linear model of the first order, linking the jobs submitted by *CiGri* to the load of the file-system (Equations 3.4 and 3.8).

3.4 Design of the Controller

We assumed in Section 3.3.2 that our system is a first order. We want to design a Proportional-Integral (PI) controller to be able to regulate the load of the file-system with *precision* and *robustness*. For a PI, there are two gains to set up:

- K_p is the proportional gain of the controller
- K_i is the integral gain of the controller

These gains are functions of the model of the system (Equations 3.4 and 3.8) and two parameters that allow to choose the closed loop behavior of the system:

- k_s which represents the maximum time for the closed-loop system to get to steady state, and which gives the rapidity of the system, *i.e.,* the time to react to a variation
- M_p which represents the maximum overshoot allowed

From these value given by the administrators of the system, we can derive the gains for the PI controller [Hel+04].

$$\begin{cases} K_p = \frac{a-r^2}{b} \\ K_i = \frac{1-2r\cos\theta+r^2}{b} \end{cases} \tag{3.9}$$

Where:

- $r = \exp\left(-\frac{4}{k_s}\right)$
- $\theta = \pi \frac{\log r}{\log M_p}$

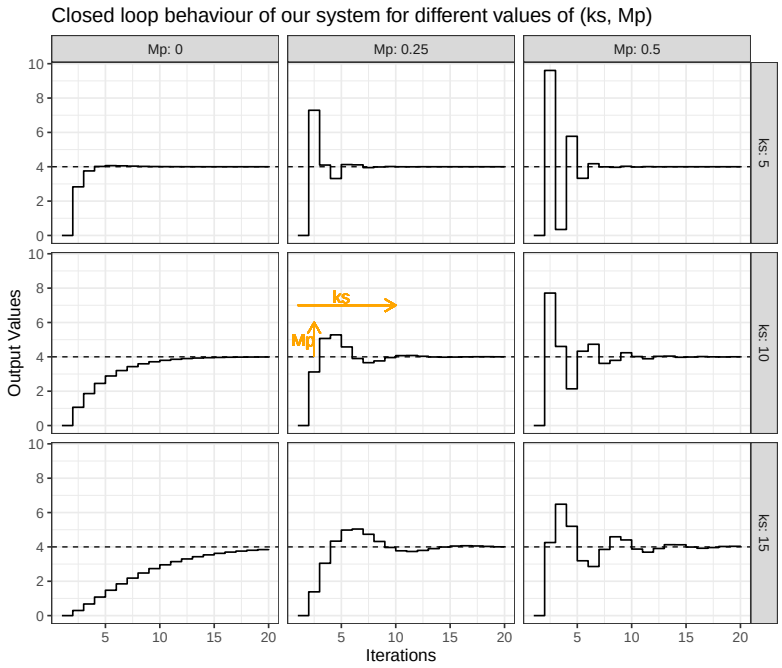


Figure 3.6.: Impact of the k_s and M_p parameters on the Closed Loop Behavior of the system. The smaller k_s, the fastest the closed-loop system will converge to the reference value. But too small values of k_s can lead to some overshooting. M_p controls the allowed overshoot. The greater M_p, the greater the allowed overshoot.

3.5 Choice of the Closed-Loop Behavior

Figure 3.6 shows the influence of the k_s and M_p parameters on the closed loop system. We now pick the values of k_s and M_p to meet with the desired behavior and extract the gains of the controller from Equation 3.9. In our case, we want to avoid any overshoot, and thus avoid overloading the file-system, but still have a fast response. Hence, we will take $k_s = 12$ and $M_p = 0$.

Take away 3.4

We designed a Proportional-Integral Controller from the model identified in the Section 3.3.2. Control Theory allows us to choose the closed-loop behavior of our system (Figure 3.6).

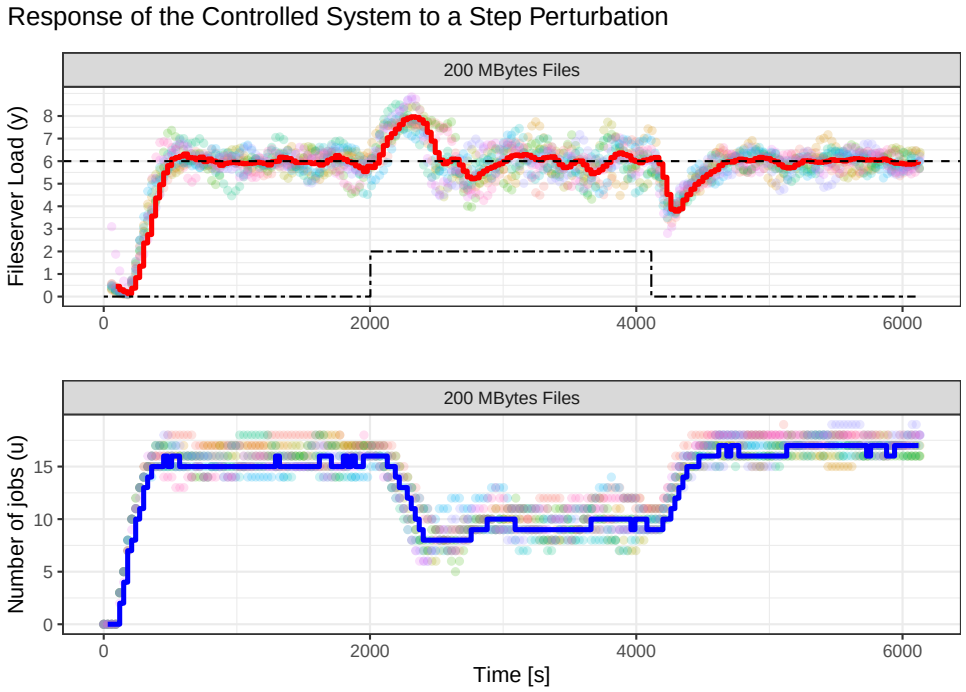


Figure 3.7.: Response of the closed-loop system to a step perturbation. The Proportional-Integral controller increases the number of jobs *CiGri* submits to *OAR* (bottom) to get the load to the reference value (top). At $t = 2000s$ we introduce a step-shape perturbation resulting in an increase of the load. The controller detects it and decreases the size of the submission to get the load back to the reference value.

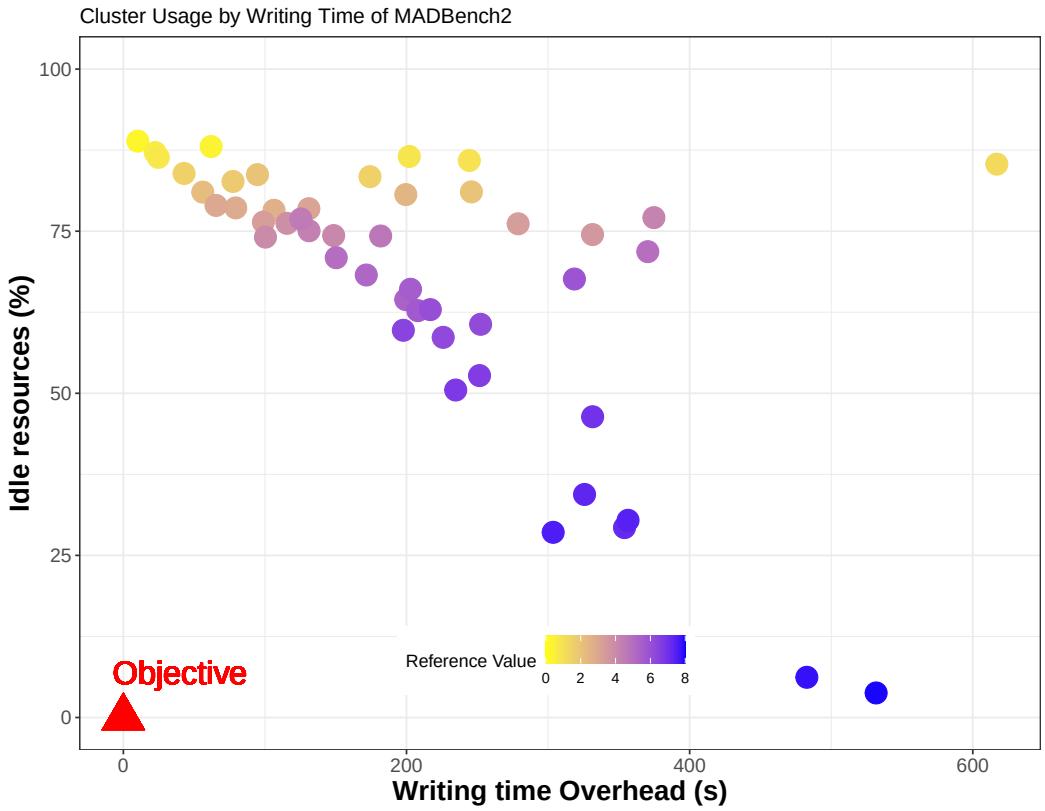


Figure 3.8.: Overhead on the MADBench2 *I/O* benchmark [Bor+07] based on the chosen reference value for our PI controller.

3.6 Example of Perturbation

Figure 3.7 represents the response of the closed-loop system to a step perturbation. We ask the controller to regulate the load of the fileserver around the value 6 (dashed line). At time $t = 2000s$, we simulate a premium user job that would produce a disturbance on the load (dotted and dashed line). We took a step-shaped disturbance to observe the reaction to an very abrupt change in load and then observe the convergence to a new stable state. If the disturbance was ramp-based, it will be easier for the controller to cope with it. We repeat this experiment several times. Each color on Figure 3.7 represents a single experiment. We also plot the aggregated mean behavior on these experiments with a continuous line.

We can see that the controller manages to get to the reference value. When the disturbances start, the controllers detects the rise in load due to the step. It then decreases the number of jobs sent to *OAR* to get the load back to the reference value.

In Figure 3.8, we run the MADBench2 *I/O* benchmark [Bor+07] as a priority job while running *CiGri* with the PI controller presented above. We replayed this scenario with different reference values for the controller. We confirm that the reference value is the key to the trade-off between the amount of resources harvested (y-axis) and the perturbation (x-axis) on the priority applications (MADBench2 here).

3.7 Limitations and Improvements

Proportional-Integral controllers have mathematically proven properties in particular for their convergence, robustness to disturbances, etc. But they do also have limits. Indeed, they rely on the system model, defined in Equation 3.4. In our case, the model depends on the jobs sent by *CiGri*. Hence, when a new campaign starts, the model needs to change. In practice, we do not know the quantity of *I/O* per job and thus the model. There exists ways to deal with this issue. First, we could measure this quantity of *I/O*. But all HPC systems do not provide such metrics as they require metrics collectors such as Colmet [oar23c] or DCDB [Net+19]. As *CiGri* campaigns gather lots of jobs and as they have similar behaviors, we could submit a subset of these jobs and compute an estimation of their *I/O*s (f in Equation 3.8). Then we would adapt the model with this estimation and start the controller with the newly computed gains. Or, we could perform an online estimation of the model parameters, in particular b.

The remaining of this chapter presents some modifications of the feedback loop to improve the behavior of the closed loop system.

3.7.1 Dynamic Reference Value

In this chapter, we showed that a PI controller was able to track correctly a reference value for the load of the file-system. As we can see at t = 2500s on Figure 3.7, the response time could lead to an overload of the file-system if the reference value is too high. The choice of the reference value is thus crucial. A too low reference value would lead to a poor harvesting of idle resources, and a high reference value would increase the overhead on priority jobs while increasing the chances of a collapse of the file-system due to an overload.

One approach would be to dynamically change the reference value based on the number of premium jobs currently running on the cluster. If there are no premium jobs, then the reference value is high to allow *CiGri* to harvest. If there are some

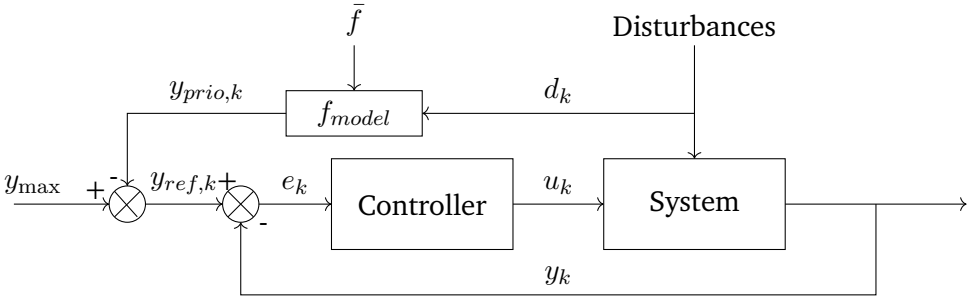


Figure 3.9.: Feedback loop representing the control scheme with the dynamic reference value. The current number of premium jobs (d_k) is fed into a model returning the maximum load that the premium jobs could put on the file-system. This load is then subtracted to the maximum load for the file-system, and then is defined as the reference value for the *CiGri* controller. The value of $\bar{f}$ represents a representative file size for the cluster and is chosen by the administrators of the cluster. This information can be retrieved with Darshan [Car+11] for example. It could be the mean or median file size, or the 95% percentile to be more conservative.

premium jobs, then the reference value is decreased to leave more room for the controller to react. The amount to reduce the reference value depends on how many nodes are used by priority jobs, and prior global knowledge of the *I/O* load of premium jobs.

Figure 3.9 gives an idea of the control loop. The quantity d_k represents the number of resources used by premium jobs at iteration k. f_{model} is the estimated maximum load that can be produced by all the premium jobs simultaneously. It can be found by performing an identification experiment where, instead of having steps inputs as previously, we have Diracs.

Figure 3.10 shows the results of this new identification experiment. For different file sizes, we write concurrently to the distributed file-system and observe the load of the machine hosting the file-system. We then wait for the load to reach (near) zero and write again. From this experiment we extract a model of the maximum load (f_{model}) based on the file size and the number of concurrent writes using a linear regression. Using this model, we can define the reference value for the load of the file-system:

$$y_{ref,k} = y_{max} - y_{prio_k} = y_{max} - \max(0, y_{max} - f_{model}(d_k, \bar{f})) \tag{3.10}$$

The value of $\bar{f}$ represents a representative file size for the cluster. This information can be retrieved with Darshan [Car+11] for example. It could be the mean or

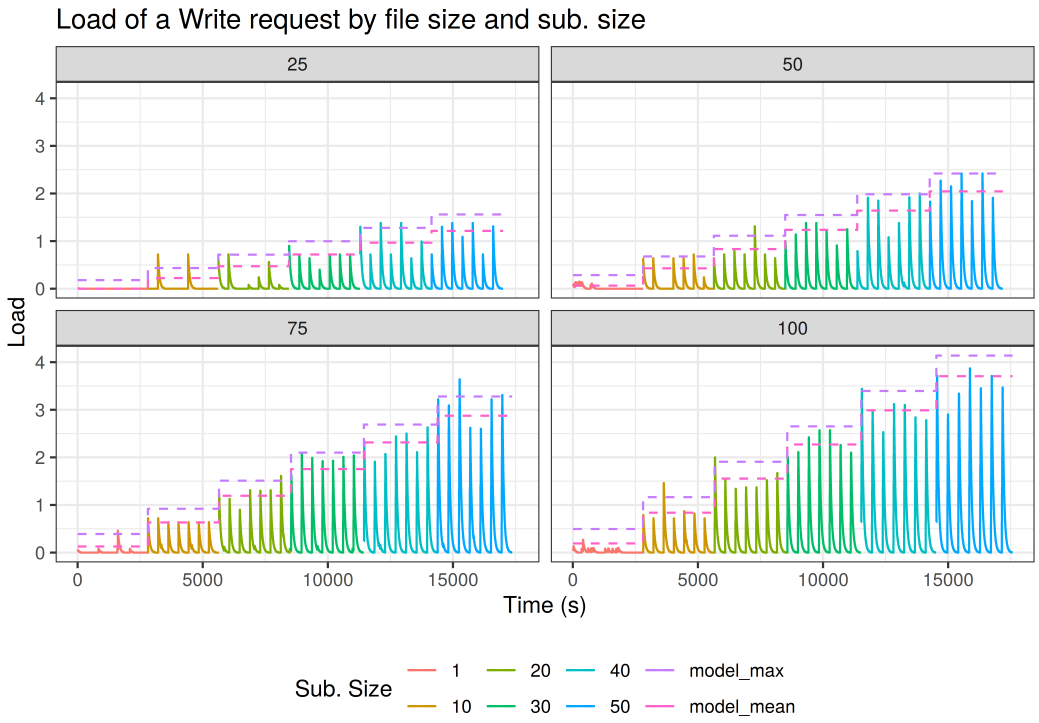


Figure 3.10.: Identification experiment to model the maximum load (y-axis) that N concurrent writes (color) of a given size (facets) on the distributed file-system. After performing concurrent write requests, we wait for the load of the machine hosting the file-system to reach a near zero value before executing the next write requests.

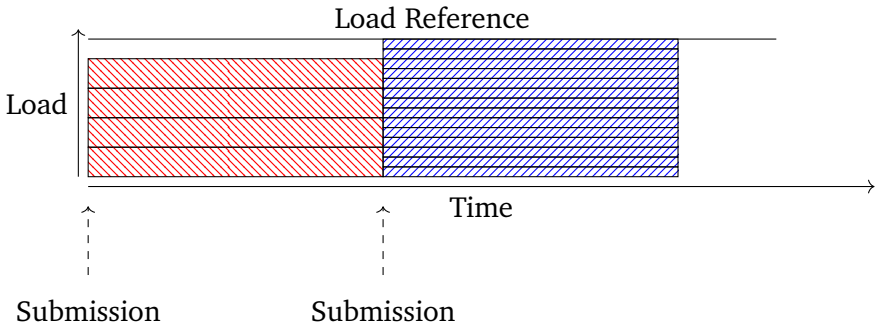


Figure 3.11.: Submissions with jobs from single campaign

median file size, or the 95% percentile to be more conservative. The administrators of the cluster would set the value of $\bar{f}$.

Take away 3.5

With extra information from the premium jobs, we can adapt the reference value dynamically to reduce the risks of a collapse of the file-system.

3.7.2 Tagging *I/O* intensive campaigns

This section summarizes the work done during my master thesis [Gui20] and published in [Gui+21a]. Originally, *CiGri* submits batches of jobs from the same campaign to *OAR*. If the quantity of *I/O* done in each job of the same campaign is similar, it differs between campaigns of different projects. Suppose that there are two campaigns submitted to *CiGri*, one with high *I/O* load jobs and the other with light *I/O* load jobs.

Figure 3.11 represents a situation using submissions composed of jobs from the same campaign. If we suppose that we are able to regulate perfectly the number of jobs submitted for the red campaign to keep the load of the cluster under the reference load, there will be a "gap" between the actual load of the cluster and the reference load. We could exploit this "gap" by submitting jobs that have a smaller impact on the load (blue jobs). Figure 3.12 shows a situation where we submit a set of jobs coming from two different campaigns to improve the cluster usage, the number of resources used while keeping the load under the reference value.

A *CiGri* campaign is represented by a JSON file. We extended the definition of this JSON file to include a boolean field to the *I/O* "heaviness" of the jobs of the submitted campaign. We suppose that at any moment, there are at least one

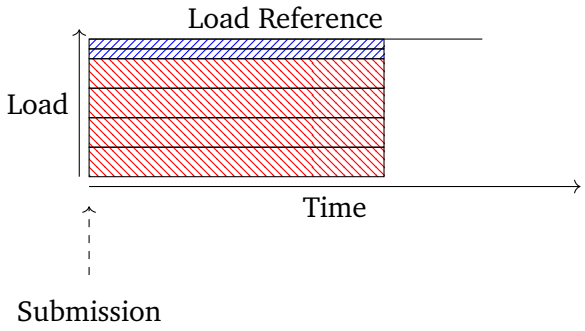


Figure 3.12.: Submission with jobs from several campaigns

I/O heavy campaign and one *I/O* light campaign. This consideration gives us one additional knob of action on the system: the proportion of jobs from each type of campaign in the batch of jobs to submit to *OAR*.

Changing the number of jobs in the submission has a bigger impact than changing the percentage of *I/O* heavy jobs in the submission. Thus, to regulate the load of the file-system precisely, we should do these actions in the following order:

1. Change the number of jobs submitted to *OAR*
2. Change the percentage of *I/O* heavy jobs in the submission

The controller will have two modes running in exclusion. The idea could be summed up as "big step, small step". The controller will firstly regulate the number of jobs sent to *OAR* (big step). Then, when the load of the file-system is "close" to the reference value, the controller will regulate the proportion of *I/O* heavy jobs in the submission (small step). Figure 3.13 gives a graphical representation of the controller.

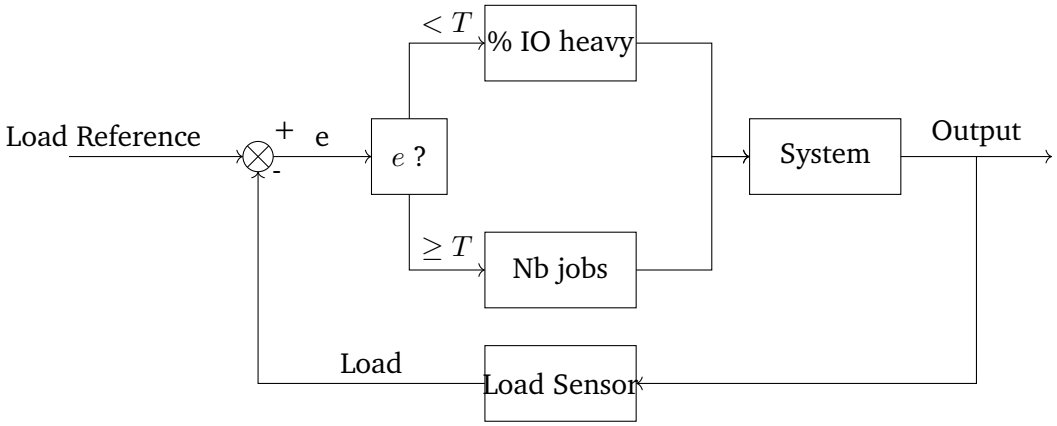
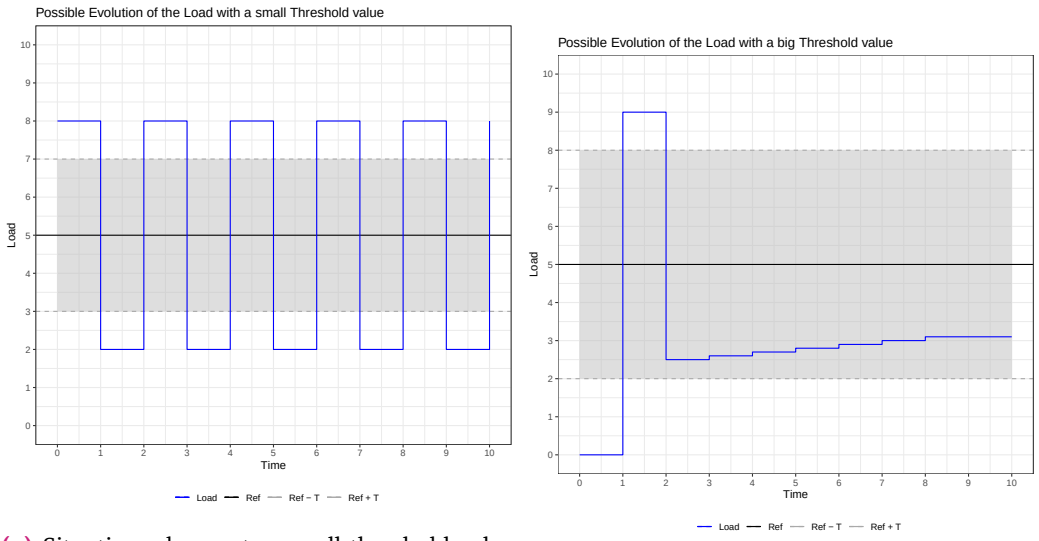


Figure 3.13.: Representation of the feedback loop for a submission with different campaigns with different *I/O* loads.



(a) Situation where a too small threshold value could lead to oscillations in the system and an imprecision.

(b) Situation where a too large threshold value could lead to an unavoidable static error.

Figure 3.14.: Graphical representation of the potential behavior of the load if the threshold is too small (Figure 3.14a) or too big (Figure 3.14b).

The choice of the threshold value (T in Figure 3.13) between the two modes is not easy, but we give some guidelines below.

If T is too small, then it will be more difficult to get into the mode regulating the percentage of *I/O* heavy jobs, and there might be an unavoidable static error plus some oscillations. Figure 3.14a gives a visual representation of this situation.

If T is too large, the controller might get "stuck" in the mode regulating the percentage of *I/O* heavy jobs. This could lead to reaching a non-optimal stationary state. Figure 3.14b gives a visual representation of this situation, where the percentage of *I/O* heavy jobs reached 100% and the controller cannot get closer to the reference value. In [Gui+21a], we choose a threshold value of 1 ($T = 1$) as experiments have shown not to be too small nor too big.

Figure 3.15 depicts and example of regulation of the load of the NFS file-system by considering *I/O* heavy jobs and *I/O* light jobs. The dashed lines on the bottom plot represent the threshold between the two modes. The solid line is the reference value. We can see that the controller first enters the threshold zone around 700 seconds, and then starts regulating the percentage of *I/O* heavy jobs in the submission (top plot).

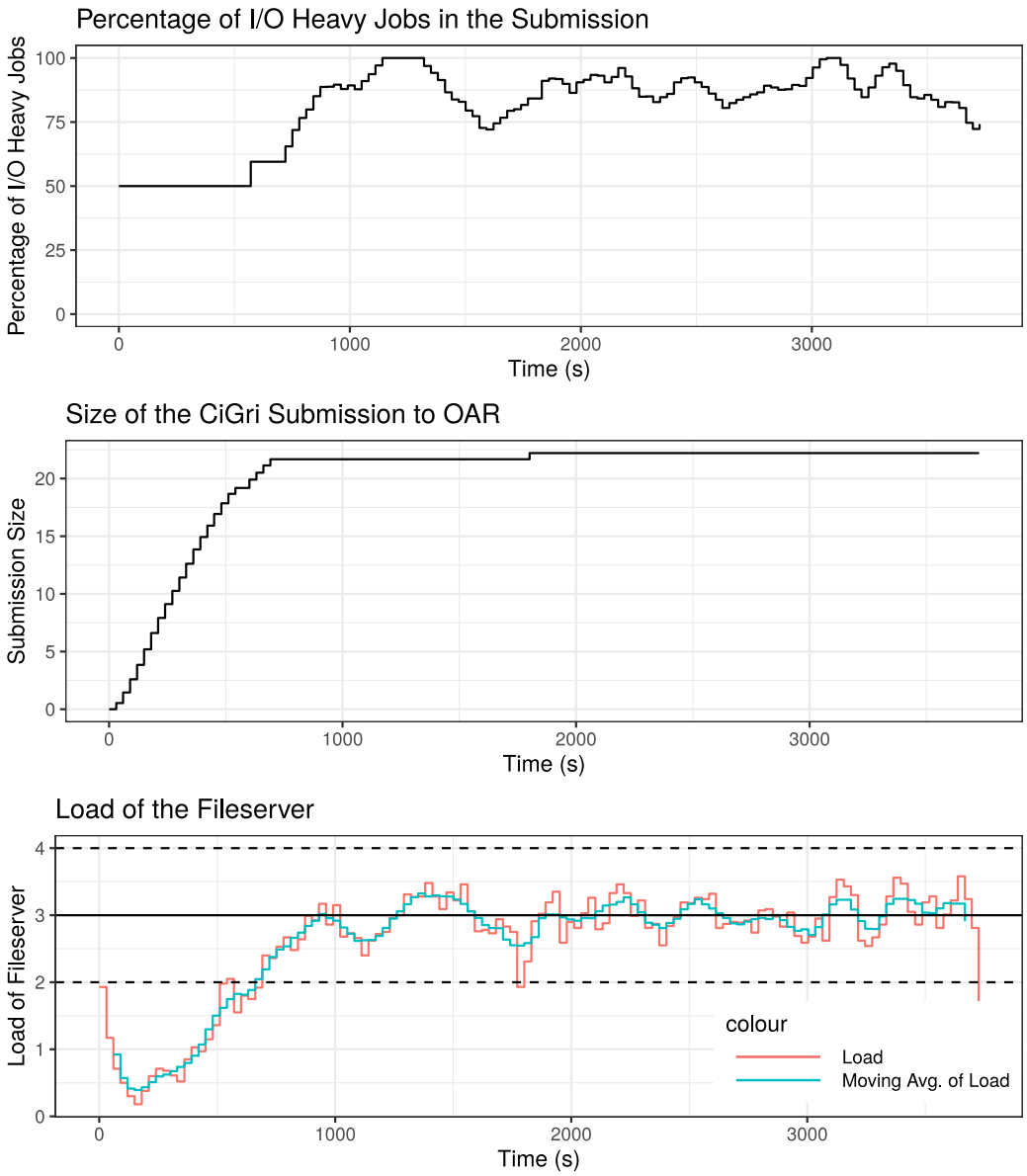


Figure 3.15.: Example of regulation of the load of the NFS file-system by considering *I/O* heavy jobs and *I/O* light jobs. The dashed lines on the bottom plot represent the threshold between the two modes. The solid line is the reference value. We can see that the controller first enters the threshold zone around 700 seconds, and then starts regulating the percentage of *I/O* heavy jobs in the submission (top plot).

Experiments showed that considering *I/O* heavy and *I/O* light campaigns allows to improve the usage of the cluster, while having slightly better control on the load of the file-system [Gui+21a].

Take away 3.6

By considering jobs from campaigns with different *I/O* loads, it is possible to improve the usage of the machines.

4 Reusability of Autonomic Controllers for HPC Systems

This chapter is based on work submitted in ACM TAAS with Raphaël Bleuse, Sophie Cerf, Bogdan Robu, Rosa Pagano, and Eric Rutten.

The controller defined in the previous chapter is tightly coupled with its underlying system. This is due to the identification phase that creates a model of the system. Implementing such a controller requires a control-theory background. However, HPC system administrators are not control theory experts. And they cannot develop *on their own* a new controller for every new cluster, file-systems, disks, network, in their platform. To help with the adoption of control-theory based solutions, and with software engineering concerns in mind, we explore in this Chapter, the reusability of autonomic controllers using control-theory techniques with the example of *CiGri* as a study case. In the following sections, we present the implementation of controllers of different nature (PI, adaptive PI, and Model-Free Control) to regulate the same system presented in Chapter 3, and propose a comparative framework to conclude on their *reusability*. We further discuss their interdependence and known trade-offs, from a theoretical point-of-view.

4.1 Criteria for comparison

The controllers are compared based on five criteria, each answering a specific design or evaluation problematic:

- **Nominal Performance**: To what extent, and how, does the controlled system meet its objectives, *in the experimental conditions in which the controller has been designed*?
- **Portability**: How will the controller behave *if the system varies from the nominal version considered for design*? What set of systems can the controller manage?
- **Setup complexity**: To what extent is the controller "plug and play"? How easy is it to set up the controller?

- **Support Guarantees**: Is the controller design backed by a methodology bringing behavioral guarantees? Under which hypothesis do the guarantees hold?

- **Competence required**: How knowledgeable one needs to be to soundly design the controller?

Take away 4.1

We propose 5 criteria to compare controllers: their *nominal performance*, *portability*, *setup complexity*, as well as their *guarantees* and the *competence* they require.

The next sections define in details each criterion, discuss their relevance, and propose metrics to characterize them.

4.1.1 Nominal Performance

Definition Nominal performance characterizes the ability of the system to perform its adaptation task. Data- and model-based controllers are inherently dependent on the configuration and experimental conditions in which the design was done. Usually, performance of an autonomous system is evaluated in similar conditions, what we call *nominal* system. Nominal performance covers both aspects of transient, dynamical behavior, and converged, stabilized behavior.

Metrics Classical control theory provide three metrics to characterize the dynamical behavior of a controlled system. Note that stability is eluded here, as we suppose all controllers are functional. **Precision** reflects the ability to reach the desired target. It is measured using the *static error* metric, giving the distance between the measured performance and its reference value, once the system converged. 0 is the best value: the smaller, the better. **Rapidity** characterizes the speed of convergence of the controlled system. It is measured using the *response time* metric, computed as the time needed from the performance signal to reach 95% of its final value. The smaller, the better. **Quality** of the signal is also considered in cases there are oscillations in the performance signal. It can be measured via the *overshoot* metric, the relative value of the highest peak of performance signal. 0 is the best value: the smaller, the better.

4.1.2 Portability

Definition When used in real conditions, the controller may have to manage a system that does not correspond to the nominal situation. Portability captures the ability of a controller to be applied, without or with limited modification, to a system different from the nominal design one, and to keep adequate performance. Two cases can be distinguished for the portability conditions. First, the variation occurs at runtime, *i.e.,* the system deviates from the nominal version, possibly due to some disturbances or external events. The ability of a controller to keep good performance in case of runtime variations of the system is to be linked with the well studied property in Control Theory of robustness. Robust controllers are able to reject disturbances, *i.e.,* ensuring that the effect of the disturbance on the system remains limited and mastered. Second, the controller may be applied to a system with different configurations, *i.e.,* variations occur offline, on different runs. Such software or hardware configuration changes can be significant, for instance when different clusters are considered. This property is rarely considered in Control Theory for physical system, where the notion of different *runs* does not exist: controllers continuously monitor the same physical system.

Metrics Performance metrics of precision, rapidity, and quality can be used to analyze the controlled system. Different systems can be tested, by varying the configurations and injecting disturbances, when possible. The distance between the current system and the nominal system can be characterized, to create a set of systems that the controller can handle. Portability is then the performance achieved by a controller for several configurations and disturbances.

4.1.3 Setup complexity

Definition Controllers are implemented and used by system experts, that are not a priori knowledgeable in Control Theory. A controller with low design complexity is more likely to be understood, correctly implemented, and tuned. If performances are acceptable, the simplest controller may be the best option. Note that design complexity is different from the mathematical complexity of the controller algorithm, whereas rather addresses the user experience aspect.

Metrics There is no commonly agreed and general metric for design complexity. We propose to use as a metric the number of parameters, or initial conditions, that have to be tuned in the controller algorithm. While this metric only covers part of the difficulties a user can experience in developing a controller, we believe that it gives a meaningful quantitative metric that can be applied to any algorithm.

4.1.4 Support Guarantees

Definition One advantage of Control Theory-based adaptation is regarding the mathematical guarantees one can have that the controlled system will perform as desired. Guarantees are not a binary property: there are nuances beyond having or not having guarantees. First, a methodology is needed for the controller tuning. Then, the performance properties that are ensured come with some conditions, depending on the system's variations or on appropriate system excitation. There exist a gradation of guarantees: from safety-critical system requiring strong guarantees, system subject to SLAs, to best-effort system without the need of guarantees.

Metrics Measuring guarantees level to compare controllers is not an easy task, there is no agreed-upon metric to characterize this criterion. We opt out for a qualitative evaluation of guarantees: the availability of a mathematically-backed methodology for controller tuning, and the hypothesis and conditions that need to be fulfilled to have those guarantees.

4.1.5 Competence required

Definition Designing and tuning a controller is not straightforward. PID controllers are known for being fairly accessible, resulting in their predominance in the industrial applications. However, the proper design of a PID controller may require control expertise: as it is the case to perform sound identification experiments and tune accordingly the PID parameters. Advanced techniques such as optimal control, model predictive control, or robust control require even more domain-specific knowledge. The need for a control expert leads to additional costs and delays, that may orient systems' experts toward simpler solutions.

Metrics Evaluating the competence required for designing and using a controller will also be done qualitatively. It can be by evaluating the level of education/knowledge required to design the controller, or the amount of available bibliography on the controller. Such metric reflects the complexity of the design process, where the need for complementary actions such as feedforward, anti-windup, preventing bursting, etc., have to be investigated. The metric also covers the domain-specific knowledge needed for fine-tuning and to properly understand the achieved guarantees.

4.1.6 Theoretical trade-offs between criteria

No controller is expected to outperform the others on all criteria. Control Theory highlights the inherent discrepancies between nominal performances and reusability. Design complexity and guarantees are also hardly compatible. Only a trade-off can be achieved. Then, strategies for concealing multi-criteria have to be applied, such as setting priorities, weighting the criteria, or finding Pareto-optimal sets. Such choice of priorities or weights is not straightforward, and can largely differ depending on the application context and objectives. This work focuses on this challenge by evaluating various controllers on complementary criteria. The objective is to highlight which controller to chose, given one's priorities.

In the following, we discuss two important trade-offs between pairs of criteria.

Trading nominal performance for reusability Conciliating optimality and robustness is a primary concern in the control field. Having highly performing controllers on the nominal system may come at the cost of limited performances, or even instability, in case the actual system differs significantly. Relaxing constraints on precision, rapidity, and quality can allow for the reusability of the controller on a large class of systems.

Trading support guarantees for low setup complexity and limited control competences Strong guarantees on the behavior of the controlled system over a wide variety of conditions can be achieved, however it often requires using advanced control techniques. Properly designing such controllers assumed significant control engineering knowledge, and its implementation and usage are consequently more complex. This discussion on the simplicity of the controller usage, even at the cost of loss of performance guarantees, is very common in the control field. However, it reflects the reality of systems' experts when using controllers in practice.

4.2 Considered Controllers

This Section presents the design of the controllers considered in this study of reusability: a PI controller (Section 4.2.1), an adaptive PI (Section 4.2.2), and a Model-Free controller (Section 4.2.3). Sections 4.2.2 and 4.2.3 summarize, respectively, the work done during the Master 2 internship of Rosa Pagano [Pag23], and [Gui+22d] published at CCTA 2022 with Bogdan Robu, Cédric Join, Michel Fliess, Eric Rutten, and Olivier Richard.

4.2.1 Proportional-Integral Controller

For the PI controller, we reuse the controller designed in Chapter 3. To summarize, we have the following model of our system $y(k+1) = ay(k) + bu(k)$

Tuning We have two parameters to choose the closed-loop behavior of our system: k_s and M_p. From those parameters and the model (a and b), we find the gains K_p and K_i of the PI controller.

4.2.2 Adaptive PI

The PI controller defined above is coupled to one kind of jobs (execution time and *I/O* load). An adaptive controller, however, would be able to *adapt* its configuration at runtime to fit the jobs running even if they are different from the one used for the design of the controller. In our case, the parameter b of our model depends on the *I/O* load of the jobs (f), see Figure 3.3 and Equation 3.8, which thus motivates the use of adaptive control.

We build adaptation on top of the previously defined PI controller, and the adaptive algorithm adjusts its parameters to account for the unmodeled dynamics beyond the scope of the linear model [She+17].

There are two primary approaches for the adaptation: updating the parameters of the controller (*direct*), or updating the parameters of the model (*indirect*). In our case, the indirect approach appears more suitable because of the dependence of the previous PI controller on the b parameter of the model. We will thus, at each iteration of the control loop, estimate the value of $b(k)$ of our model, noted $\hat{b}(k)$, and recompute the value of the gains (K_p and K_i) accordingly.

Controller design

Our objective is to estimate $\hat{b}(k)$ using an online algorithm. We define the prediction error ε representing how far the estimated model is from the reality measured:

$$\varepsilon(k) = y(k) - ay(k-1) - \hat{b}(k-1)u(k-1), \tag{4.1}$$

Due to the presence of significant noise in the measured data, see for instance Figure 3.3, the algorithm should be capable to mitigate noise. Therefore, we use a modified Recursive Least Square (RLS) [ÅW08] algorithm to improve the robustness against noise. That is, we substitute ε with its filtered version $\Phi(\varepsilon(k))$, given by:

$$\Phi(\varepsilon(k)) = \frac{\varepsilon(k)}{1 + \phi|\varepsilon(k)|} \tag{4.2}$$

where $\phi \geq 0$ represents a smoothing parameter to be chosen. By doing so, we preserve a linear structure when the error is small, while introducing a slower-than-linear behavior for larger errors. Eventually, we define our robust estimator as:

$$\hat{b}(k+1) = \hat{b}(k) + V(k+1) \times u(k) \times \Phi(\varepsilon(k+1)) \tag{4.3}$$

With $V(k+1)$:

$$V(k+1) = \frac{V(k)}{\mu} - \frac{V(k)u(k)^2V(k)}{1 + u(k)V(k)u(k)} \cdot \frac{1}{\mu^2} \tag{4.4}$$

where μ represents the forgetting factor. The presence of changing parameters over time, such as b varying with different campaigns, requires the use of forgetting effect of $\mu \in (0, 1]$. However, the μ parameter introduces a significant drawback known as the "bursting phenomena" [And85; FKY81]: when the system lacks sufficient excitation, the algorithm may approach a singularity, resulting in large spikes in the estimated parameter.

Tuning

The algorithm relies on different parameters: the forgetting factor μ, the smoothing parameter ϕ, and the initial conditions $\hat{b}(0)$ and $V(0)$. Note also that the adaptive PI is built on top of the PI, which requires the computation of the model parameter a and the definition of k_s and M_p, two quantities that allow choosing for the close-loop behavior (see fig. 3.6).

Forgetting factor μ A small μ results in a fast, but noise-sensitive, response. Using a Parallel Estimation [ÅW08; Kow92] allows us to try various values of μ and pick the one returning the less error.

Smoothing parameter ϕ Strong smoothing leads to slow estimation, but also reduces overshoots and oscillations. There are no methodology to define ϕ, and we chose a satisfactory value via an experimental approach.

Initial conditions There are two distinct initial conditions: $\hat{b}(0)$ and $V(0)$. For $\hat{b}(0)$, a value of 0.5 was chosen. Small values, *i.e.,* underestimations, of $\hat{b}(0)$ might lead to overshooting due the relation between u and b. The significance of $V(0)$ lies in its impact on the algorithm's initial speed. Both initial conditions were chosen from experiments results.

4.2.3 Model-Free Controller

The third type of controller we will compare is one that does not rely on a model. Many choices exist such as Active Disturbance Rejection Control [Han09], Model-Free Control [FJ13], or the ones using learning techniques (*e.g.,* [Pon+18]), each of them with several variants and improvements.

Model-Free control (MFC) is an approach that has the advantage of not necessitating the possibly tedious phase of modeling. The main ideas and formulations are summarized in the following of this section.

The MFC formulation relies on an *ultra-local model*, but does not require building a reliable global model. In that sense, it justifies its name of Model-Free Control. The considered ultra-local model is of the form:

$$\dot{y}(k) = F(k) + \alpha u(k) \tag{4.5}$$

with $\dot{y}(k)$ the derivative of $y(k)$, $\alpha \in \mathbb{R}$ a constant, and $F(k)$ a term to be estimated (see eq. (4.7)), reflecting the unknown structure of the system and its disturbances.

The Model-Free approach allows plugging PID controllers, defining the notion of *intelligent* PIDs. For an intelligent Proportional (*iP*), which is linked to a classical Proportional-Integral controller, the control action is computed as:

$$u(k) = -\frac{\hat{F}(k) - \dot{y}^{\star}(k) + K_p \times e(k)}{\alpha} \tag{4.6}$$

where $\hat{F}(k)$ is the estimated value of $F(k)$, and $K_p \in \mathbb{R}$ the proportional gain. [FJ13] present different techniques to compute $\hat{F}$. Yet, as a first approach, we use past values of y and u to estimate it from (4.5):

$$\hat{F}(k) = \dot{y}(k) - \alpha u(k-1) \tag{4.7}$$

Tuning There are two parameters to tune: α and K_p. In [FJ13], the authors recommend taking α such that $\dot{y}$ and $\alpha \times u$ have the same order of magnitude. Thus, as in [Gui+22d], we rely on experiments to set the parameter α. The value of the gain K_p accounts for the closed loop behavior of the system. Small values of K_p yield conservative and slow controllers, whereas greater values yield more aggressive controllers prone to overshooting and oscillations. Therefore, the K_p value should be chosen by trial and error methods or by realizing a Pareto type analysis.

4.3 Evaluation and Comparison

4.3.1 Experimental setup

The experiments were carried out on the nodes from the Grisou Cluster of Grid'5000 [Bal+13] which is a shared French testbed for experimental research in distributed and parallel computing. Each node of this cluster has two Intel Xeon E5-2630 v3 CPU with eight cores per CPU and $128\,\mathrm{GiB}$ of memory. Each server of our system is being deployed onto a single Grid'5000 node from a Kameleon system image [Rui+15]. We used four nodes for the deployment setup: one for the *CiGri* server, one for

the RJMS (*OAR* version 3 [Cap+05; oar23b]), one for the fileserver (implemented with NFSv3), and one node emulating a cluster of 100 *OAR* resources. The job model used in this work (job = `sleep` + *I/O*) allows us to efficiently and realistically emulate several computing resources on a single physical node, and thus reduce the number of machines required to perform experiments at full scale. The impact on the realism of such a *folded* deployment is studied in [Gui+23d] and presented in Chapter 9.

4.3.2 Controller configurations

The PI is tuned based on the model of Equation 3.3. Based on identification from experimental data (Figure 3.3), we have the following values: $\alpha_1 = -0.5071484$, $\beta_1 = 0.0086335$, $\beta_2 = 0.0451394$, and $\gamma = 0.001633$. The gains of the PI controller are then defined using eq. (3.9), with the design parameters set to $k_s = 12$ and $M_p = 0$ in order to avoid any overshoot, but still have a fast response. The adaptive PI relies on the PI tuning, with additional parameters set as follows: $\mu = [0.5, 0.6, 0.7, 0.8, 0.9, 1]$, $\phi = 2$, $\hat{b}(0) = 0.5$, and $V(0) = 10^4$. For the MFC, refer to [Gui+22d] for the numerical details of its configuration.

4.3.3 Experimental protocol

For the three controllers considered in this study, we will evaluate their performance by varying the characteristics of the *CiGri* jobs from the identified system. Controllers were designed for jobs lasting 30 seconds and writing 100 MBytes to the fileserver. This configuration represents our nominal system. For testing, we consider variations on two dimensions: the amount of *I/O* written by the jobs, and the execution time of the jobs. For the *I/O*, we consider the following file sizes: 50, 100, 200, and 400MBytes and for the execution times, we consider: 10, 30, 60, and 120 seconds.

For each triplet (*I/O* load, execution time, controller), we repeat 5 times the following scenario:

1. we start with an empty cluster
2. we define the reference value for the controller at 3
3. we submit a campaign of jobs to *CiGri* with the desired characteristics (*I/O* load, execution time)

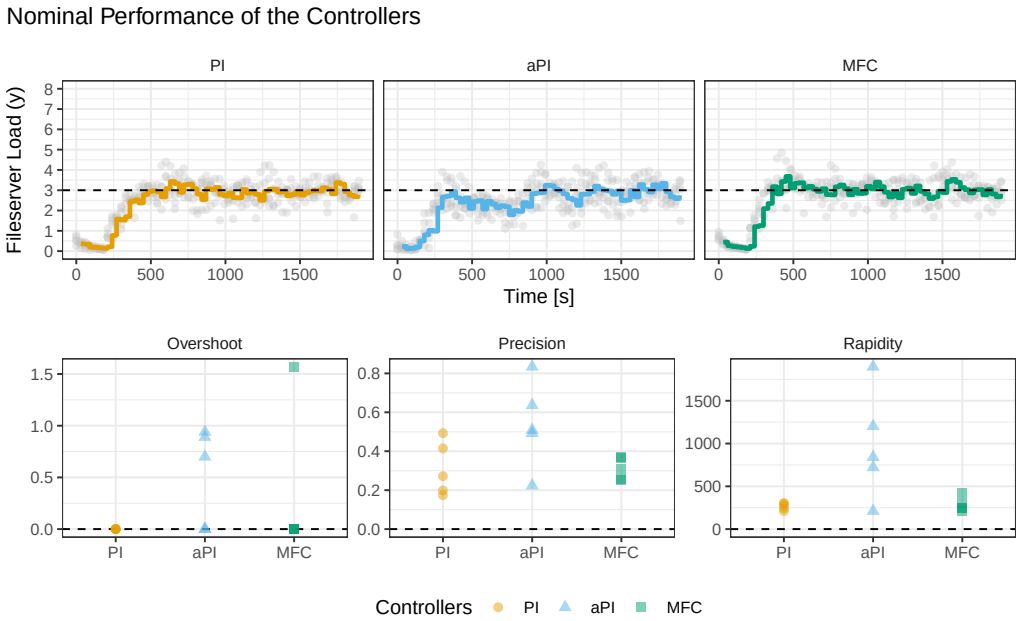


Figure 4.1.: Nominal performance of the different controllers through time. The solid line represents the aggregated behavior from 5 different experiments (light dots). The bottom plots compare the controllers on the performance metrics defined in Section 4.1.

4. we start the controller

5. at $t = 2000\,\mathrm{s}$, we introduce a disturbance in I/O which represents a step of 2 for the load of the fileserver

6. at $t = 4100\,\mathrm{s}$, we stop the disturbance.

4.3.4 Performance-related comparison

Nominal performance

We first compare the performance of controllers on the nominal system, *i.e.,* the system's configuration that was used for design and tuning. The three controllers are executed, and the measure of the fileserver load through time is reported in Figure 4.1. The top plots represent the first $2000\,\mathrm{s}$ of the scenario (before the introduction of the disturbance) for each controller. The solid lines aggregate the behavior of 5 repetitions of each experiment (light dots), and the dashed line represents the reference value that the controllers follow.

All controllers are tracking well the reference imposed, however with varying performances. The PI and the MFC have similar behaviors. The MFC is slightly faster which can lead to greater overshoot, but the PI seems more stable and precise. The nominal performance of the adaptive PI (aPI) appears quite poor, as the response is slow and imprecise.

In order to compare quantitatively the controllers, we use the performance metrics defined in Section 4.1: overshoot, response time, and precision. We evaluate the controllers on the first 2000 seconds, *i.e.,* before the disturbance, to allow for fair comparison. The results are presented in the bottom plots of Figure 4.1. The PI has no overshoot, as defined during the design phase. Similarly, the MFC mostly never overshoots, but a single experiment lead to an overshoot for the MFC. Despite the slow response of the adaptive PI, some experiments lead to small overshoots. The precision of the three controllers is similar, with the one of the adaptive PI being more volatile. The slow response time of the adaptive PI can be observed on the *Rapidity* plot of Figure 4.1.

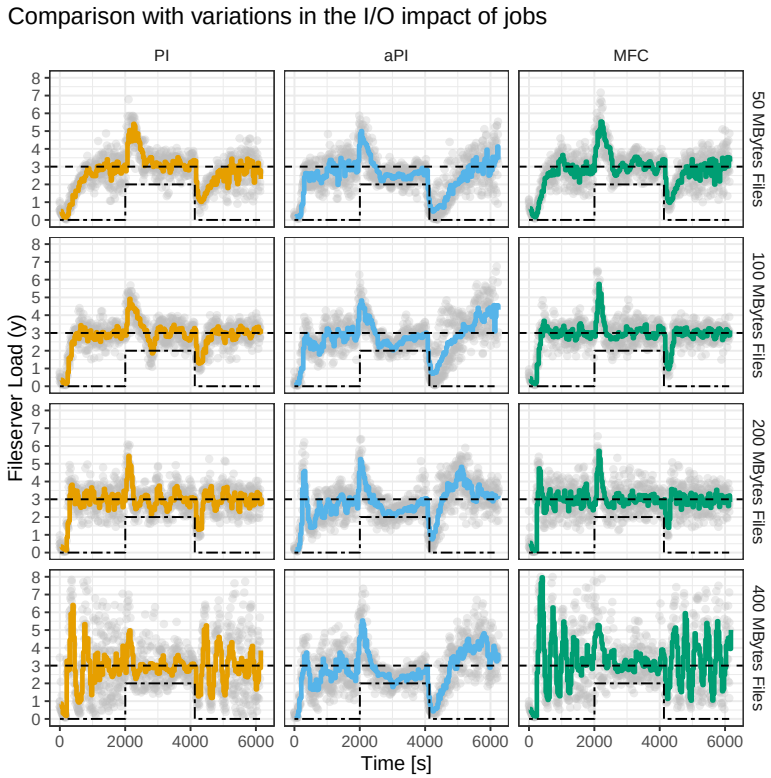
Take away 4.2

All controllers successfully track the reference value, with overall similar performances. The adaptive PI is slightly worst in terms of rapidity. The MFC is fast and precise, but can lead to an overshoot. The PI is fairly performant on all criteria.

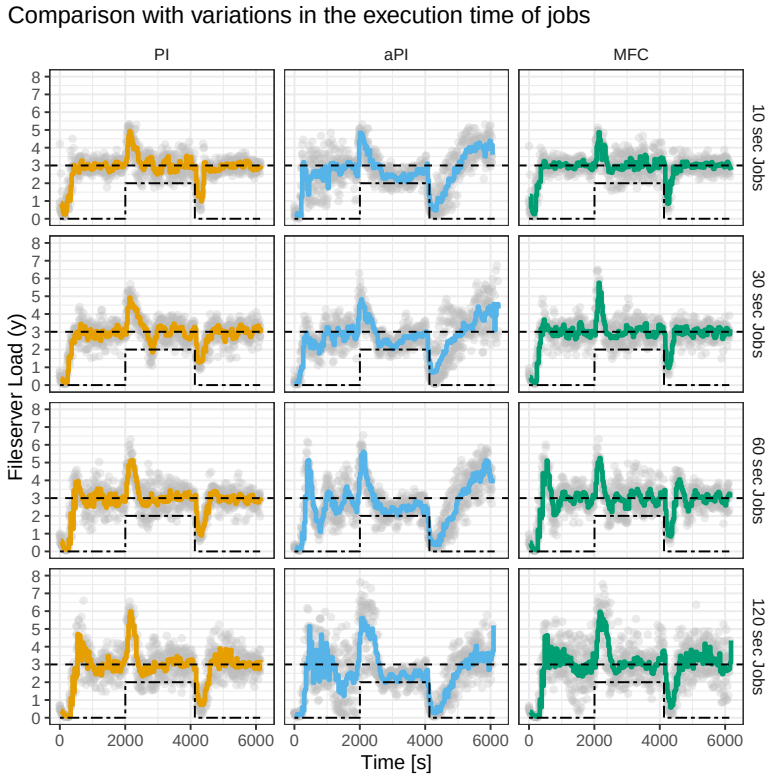
Portability

To evaluate the portability of controllers, we look at their performance when used on a different system than the nominal one. Figure 4.2 depicts the dynamic behavior of the different controllers for the different scenarios: variations in *I/O* and in execution times (Figures 4.2a and 4.2b respectively). Note that the nominal performance, *i.e.,* Figure 4.1, are repeated (second line of the plots) to allow for comparison. Similarly to the previous analysis, aggregated metrics of performance on the first 2000 seconds are computed in Figure 4.3.

Variations of *I/O* Load All controllers manage to track the reference value, however some configurations lead to large oscillations in the fileserver load. The disturbance does not seem to have a detrimental impact, it even reduces the oscillations in some cases.

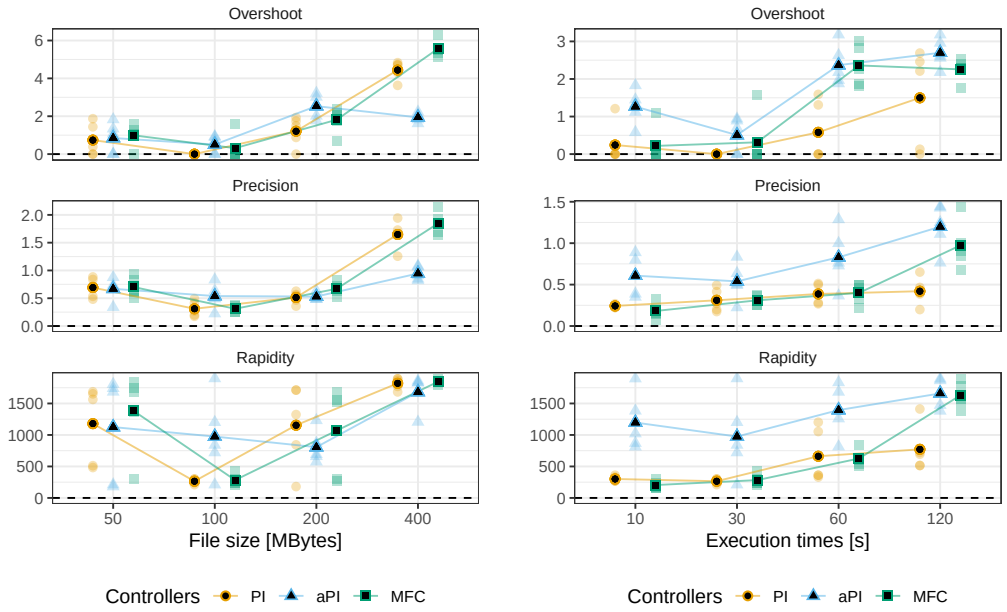


(a) Performance with varying I/O loads of the jobs.



(b) Performance with varying jobs execution time.

Figure 4.2.: Global comparison of the controllers' performance for variations in *I/O* loads (fig. 4.2a) and jobs execution times (fig. 4.2b). The solid lines represent the aggregated behavior from 5 different experiments (light dots).



(a) Performance with varying I/O of the jobs. **(b)** Performance with varying jobs execution time.

Figure 4.3.: Comparison of controllers' portability. The plots represent the performance metrics computed on various systems between 2000 seconds. The solid lines link the means over varying configurations for each controller.

For the MFC, the trend is similar than in nominal conditions: fast, but can lead to large overshoots. The PI is slower but has fewer oscillations when the system is highly different from the nominal one. The adaptive PI seems to have worse performance on the nominal system (100MBytes) than the other two controllers, but then it does not seem to be much impacted by the variations in *I/O* load.

The quantitative analysis of Figure 4.3a clarifies those analyses:

1. overshoot in the first 2000 seconds is larger for the adaptive PI, except with 400 MBytes (when we are far away from the nominal system), where both PI and MFC present huge values

2. precision is similar for all controllers, except with 400 MBytes where the adaptive PI is significantly better rapidity of the adaptive PI is better than for the other controllers on non-nominal systems

3. rapidity of the adaptive PI is better than for the other controllers on non-nominal systems (note that for the experiments with large oscillatory behavior, the rapidity criteria should be mitigated as convergence is not reached)

Take away 4.3

The MFC and the PI perform very well close to the nominal system, but have difficulties not overshooting and oscillating when the system is too far from the nominal system, *i.e.,* 400 MBytes in our scenario. On the other hand, the adaptive PI has a similar behavior whatever the system configuration.

Variations of Jobs Execution Time When variation in the jobs' execution time is introduced, controllers successfully track the reference, with a better behavior than in the case of *I/O* variations, see Figures 4.2b and 4.3b.

For the PI and MFC controllers, reducing the job duration to 10 seconds even improves the performances by reducing the overshoot. When increasing the duration, the PI seems to better control the system, with few oscillations and small overshoot.

The relative poor performance of the adaptive PI for job duration variations can be theoretically explained. The increase of jobs duration leads to the addition of a delay in the system, *i.e.,* it takes more time to see the effect of an action on the system. The adaptive PI was built to adapt the model parameter b of the model, not to identify a delay. Thus, the adaptive PI is not designed to deal with such type of system's variation, and its adaptation can be detrimental in some cases.

Take away 4.4

Overall, concerning the variations in execution times, the controllers appear less sensible than with *I/O* variations. There is overshoot, especially with the MFC and the adaptive PI, but not as many oscillations.

4.3.5 Methodology and Implementation-related comparison

This section complements the comparison of controllers on their performance, by focusing on aspects linked to their design, tuning, theoretic soundness, and implementation. Such considerations are not classic for the point-of-view of control-theory oriented evaluation, that focuses mainly on performance and robustness to unpredicted external events. On the contrary, from the practical engineering point-of-view, simplicity in design, and usage can be a valuable asset. Some performance can even be traded to allow lower complexity, so that system administrators can master solutions without the help from an external control expert, or then only in a limited

and temporary fashion. In the following, we evaluate the three controllers on the methodology- and implementation-related criteria of Section 4.1 set-up complexity, guarantees, and control-theoretical competence required.

Set-up complexity

In a first attempt to evaluate the setup complexity, we count the number of parameters to be tuned for each controller.

The MFC requires two parameters: K_p and α. α is a non-physical constant parameter. It is chosen by the practitioner such that $\dot{y}$ and $\alpha \times u$ are of the same magnitude. It should be therefore clear that its numerical value, which is obtained by trials and errors, is not a priori precisely defined [FJ13]. K_p is chosen through experiments.

The PI also requires two parameters: K_p and K_i. They are computed as in Equation (3.9) based on the model of the system (two parameters in the case of the linear first order model: a and b), that can be derived from the identification experiments. Moreover, the tuning allows translating the problem of finding K_p and K_i into the choice of the design quantities k_s and M_p. Thus, two parameters are needed for setting up the PI.

Finally, the adaptive PI is based on the PI: thus it requires the same two parameters. It additionally requires parameters specific to the adaptive part: μ, ϕ, $\hat{b}(0)$, $V(0)$, summing up to a total of six parameters.

Besides the raw count of parameters, we can point out that some parameters are more sensitive than others in terms of precision needed. We therefore see three cases:

1. the very sensitive parameters for which an ill-tuned value can even make the system unstable (*e.g.,* K_p)

2. the parameters for which an ill-tuned value makes the system slower (*e.g.,* ϕ)

3. the parameters which are continuously updated and therefore an approximate initial value has little impact (*e.g.,* $V(0)$)

Guarantees

The controllers studied in this chapter have different guarantees from the point-of-view of their behavior.

As seen in Chapter 3, the design of a PI controller requires following a well-defined methodology. This has the benefits of allowing users to choose the closed-loop behavior (choice of k_s and M_p as illustrated in Figure 3.6), and to have guarantees that the controller will behave as intended.

Designing and using a Model-Free controller requires less knowledge of Control Theory than a classical PI. It removes the need for complex identification and modeling of the system. It also has an online estimation of the underlying dynamics of the system. Yet, this simplicity comes at a cost: the MFC does not have a fixed methodology to choose all the parameters. It hence does not provide the same guarantees as the classical PI with respect to convergence or robustness.

The adaptive PI provides additional guarantees to the PI, by broadening the conditions in which the controller is operational. That is to say that the guarantees that one has on the nominal system, also extend to systems close to the nominal one, as seen in Section 4.3.4. However, knowing in advance if a system falls into the region of stability of a controller is not straightforward for a computing system, especially without performing a preliminary modeling identification phase. Additionally, the adaptive PI has the drawback of requiring specific input signals – known as persistent excitation – to avoid its divergence, namely the bursting phenomena.

Competence in Control required

On most computer science curriculums, there are no classes on control-theory. Most cloud/grid platforms usually do not hire control scientists or engineers in their system administration team. Therefore, the choice of a control method should be weighted by its difficulty to be mastered by the system administrators.

In this perspective, MFC has the advantage of very low requirements, as it does not require specific engineering education. On the other hand, it is a novel, emerging method, with only little literature and experience from which to draw practical guidelines.

In the case of the classical PI, there is a need to perform open-loop identification experiments. It has a practical cost, but there is only relative low mathematical difficulty in the equations at play. Therefore, it is accessible and taught to undergraduate

students in control engineering curriculums. As it is very widely used in industry (around $90\,\%$ of the industrial processes use a form of PI/PID controller) there is a huge amount of available literature [ÅH06; LAC06; DB21].

The adaptive PI builds up on PI: it hence requires a greater set of skills to use [Lan+11]. Moreover, over the years, many upgrading elements and variants have been proposed in order to correct and improve certain aspects in its design (*e.g.,* [ÅW13; IS96; Mid+88]). In terms of skills, this usually is taught in Master curriculums dedicated to control engineering or even at the PhD level for the more advanced variants.

4.4 Conclusion

This chapter addressed the need for the reusability of autonomic controllers in the context of HPC with the study case of *CiGri*. Controllers from control-theory can have a bound too tight to the nominal system, which can make them lose performance dramatically when they are used in different contexts, with variations in time or in space. We proposed a comparison framework to evaluate the reusability of autonomic controllers, defined criteria of interest, and exposed tradeoffs between criteria. Using the study case of *CiGri*, we implemented three controllers of different nature (PI, adaptive PI, and Model-Free control), and compared them with respect to our criteria.

Criteria	PI	Adaptive PI	MFC
Nominal Performance	+	+	+
Portability	-	+	-
Design Complexity	+	-	+
Guarantees	+	+	-
Competence required	+	-	+

Table 4.1.: Comparison of controllers on all criteria. "+" indicates a positive evaluation of the criteria – *e.g.,* high portability, low complexity. "-" that the criteria is poorly fulfilled – *e.g.,* few guarantees, high competence required.

Table 4.1 summarizes the comparison of the different controllers on the criteria of the framework of Section 4.1. More general comparative results are still an open question due to the great variety of cases according to the control problems. As expected, there is no clear "best controller" from a computer engineer point-of-view, but the table exhibits several tradeoffs.

The recommendations and guidelines that can be given to system administrators depend on different contexts: whether a control engineer is available or not, whether guarantees on the behavior are crucial or not, etc. For example, if the system on which the controller will be deployed only varies "close" to the nominal system, then both the PI and the MFC are good solutions from the reusability point-of-view. However, if the system can vary a lot, then an adaptive PI appears to be a more judicious choice concerning portability. Note that it will be adaptive only with respect to the variables for which it has been designed.

In the case where one wants to give away some design complexity and required competence, but still want acceptable nominal performance, the MFC appears as an excellent candidate as it has a low setup complexity.

In the case where a well documented and explainable solution is required, then the PI controller has the advantage of being a very well-known control algorithm.

5 A Control Theory Approach to Reduce Wasted Computing Power in HPC

5.1 Introduction

Researchers have been studying the problem of harvesting idle HPC resources [Mer+17; TTL05; Prz+22] by submitting jobs that are more flexible (smaller, interruptible). These jobs are viewed as second class citizens by the scheduler and can be killed if needed. However, these jobs are still being executed on the compute nodes and impact the shared resources of the cluster (*e.g.,* file-system, communication network), thus inevitably disturbing the jobs of normal users. Meaning, there is a trade-off to exploit between the amount of harvesting and the maximum perturbation that these premium users can accept. Unfortunately, these solutions introduce a new source of computing power waste by killing jobs. Indeed, most of these small jobs do not implement a check-pointing mechanism, thus, all the computations done before the jobs are killed will be lost and will need to be started again.

In this chapter, we tackle the problem of harvesting the idle resources of a cluster while reducing the total amount of wasted computing time (from either idle resources or killed jobs) in the context of *CiGri*.

5.2 Towards a better usage of the resources

5.2.1 Problem Definition

In this work, we want to improve the usage of a set of computing clusters. We identify two ways of misusing the computing resources. The first one is to leave computing nodes idle. The second one is to submit a *Best-Effort* job while risking that it will be killed in a near future.

Table 5.1.: Summary of the notations used.

Notation	Definition
u_k	Number of *Best-Effort* resources submitted by *CiGri* at iteration k
w_k	Number of *Best-Effort* resources in the *OAR* waiting queue at iteration k
r_k	Number of used resources used by *Best-Effort* jobs in the cluster at iteration k
y_k	Output of the sensor at iteration k. ($y_k = r_k + w_k$)
$r_{\max}$	Total number of resources in the cluster (supposed constant)
$\bar{p_j}$	Mean processing time of jobs in a *CiGri* campaign
y_{ref}	*Reference value*, or desired state of the system to maintain. ($y_{ref} = (1 \pm \varepsilon) \times r_{\max}$ with $\varepsilon \geq 0$)
Δt	Time between two *CiGri* submissions (constant, chosen by the system administrator). 30 seconds in this paper.
e_k	Control error. The difference between the desired state and the current state ($e_k = y_{ref} - y_k$)
K_p, K_i	Proportional and Integral gains of a PI Controller
h	Horizon value, amount of time in the future to look at the predictive Gantt of *OAR*.

One challenge is to keep enough *Best-Effort* jobs in the waiting queue of *OAR* so they can be started immediately when some resources are freed, but not too many to not overload the waiting queue. An overload of the waiting queue can lead to a longer response time for the controller, as *OAR* will schedule the jobs no matter if they are destined to be killed or not.

5.2.2 Control Formulation

Our knob of action is the number of *Best-Effort* jobs submitted by *CiGri* at each of its iterations. To sense the current state of the clusters, we can query the API of the *OAR* scheduler to get the number of resources currently used, the number of *Best-Effort* jobs waiting.

For the sensor, we will consider resources and not jobs as we want 100% of usage of the nodes. Considering jobs instead of resources could lead to some degenerate situations. One example would be *Best-Effort* jobs using 2 resources each and having a single idle resource on the cluster. By considering jobs instead of resources, the controller could continue the injection of *Bag-of-Tasks* jobs thinking that there is always some idle resources, even though it cannot be used.

We note:

- u_k: number of *Best-Effort* resources submitted by *CiGri* to *OAR* at iteration k.

- r_k: number of used resources by *Best-Effort* jobs on the cluster at iteration k.
- w_k: number of resources from *Best-Effort* jobs in the *OAR* waiting queue at iteration k.
- $r_{\max}$: the total number of resources in the cluster.

We thus want to regulate the quantity "number of currently used resources + number of *Best-Effort* resources waiting" around the total number of resources available in the cluster. Meaning that we want to regulate $r_k + w_k$ around $r_{\max}$ by varying the value of u_k.

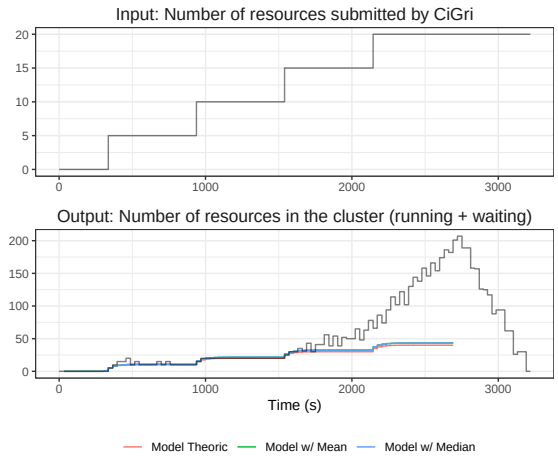
We note $y_k = r_k + w_k$. We also note the *reference value* $y_{ref} = r_{\max}$, which is the control theory term for the desired state of the system. The reference value could also be $(1 \pm \varepsilon) \times r_{\max}$, with $\varepsilon \geq 0$, based on the cluster administration preferences. A reference value greater than $r_{\max}$ could lead to longer response time for the controller and more killing of *Best-Effort* jobs, whereas a reference value smaller than $r_{\max}$ will leave some resources idle but kill less jobs.

5.2.3 System Analysis

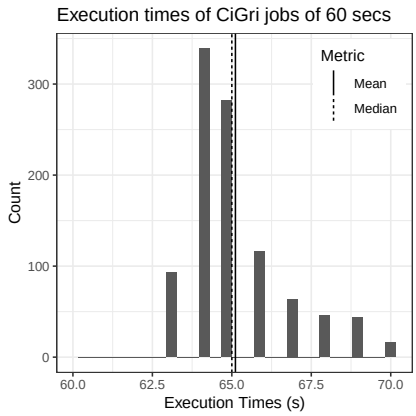
In this section, we investigate the relationship between u_k, the number of *Best-Effort* jobs submitted by *CiGri* at every iteration, and y_k, the number of *Best-Effort* resources in the system either waiting or running. We perform the identification phase of the *Control-Theory* methodology.

Let us consider a campaign of jobs with 60 seconds execution time. We modified *CiGri* to submit the jobs of the campaign as steps. Meaning that for the first 20 *CiGri* iterations, jobs are submitted by batches of 5. Then, for the next 20 iterations, by batches of 10 jobs, then 15 jobs and finally 20 jobs. We record the output of our sensor during the entire experiment and plot the results in Figure 5.1a.

We observe that around 1500 seconds, *CiGri* starts to overflow the waiting queue. And as a result, the value of the sensor does not stabilize but keeps increasing, until 2600 seconds where there are no more submission from *CiGri*. This can be explained by the fact that *CiGri* is submitting more jobs than the cluster can process in one iteration. Let p_j be the execution time of the jobs of the campaign. Note that we supposed that the jobs have the same execution times. Then, the number of resources freed at each iteration is:



(a) Identification experiment. We vary the size of the batch that *CiGri* send to *OAR* as steps. *CiGri* first submits batches of 5 jobs for 20 iterations, then batches of 10 jobs for another 20 iterations, then 15 jobs and finally 20 jobs.



(b) Histogram of the execution times for a *CiGri* campaign during the identification experiment. The campaign is composed of synthetic jobs of theoretical duration 60 seconds.

Figure 5.1.: Results of the identification of the system. Figure 5.1a depicts the link between the input and the output of our system. Figure 5.1b shows that the distribution of execution times is impacted by the commission and decommission of the nodes by *OAR*, and thus must be taken into account in the modelling.

$$\frac{\Delta t}{p_j} \times r_{\max} \tag{5.1}$$

In our experiment, Δt would be 30 seconds, p_j 60 seconds, and $r_{\max}$ 32 resources. Meaning that the cluster should be able to free up to 16 resources per *CiGri* iteration.

However, when we look at the actual execution times of the jobs, we remark that the execution times are not constant. Figure 5.1b shows the histogram of the execution times of the *CiGri* campaign used for the identification experiment. As we can see, no job actually lasted the theoretical 60 seconds. This is due to the node setup and node cleaning mechanisms of *OAR*. To improve the precision of the processing rate, we can replace p_j in Equation 5.1 by the mean of the execution times ($\bar{p_j}$). This modification leads to a processing rate of about 14.8 jobs per iterations. This corrected rate thus explains why the value of the sensor starts growing when *CiGri* starts to submit batches of 15 jobs.

We want to avoid the overflowing the waiting queue. Indeed, if the queue is not empty, *OAR* will try to schedule the *Best-Effort* jobs on the machines no matter if they are going to be killed soon or not. By regulating the number of jobs in the waiting

queue, we reduce the undesired killing of *Best-Effort* jobs. But the objective is to have a 100% usage of the cluster.

5.2.4 Model & Control Design

From the discussion in the previous section, we can model the system as:

$$r_{k+1} + w_{k+1} = \left(1 - \frac{\Delta t}{\bar{p_j}}\right) \times r_k + w_k + u_k \tag{5.2}$$

Equation 5.2 describes perfectly the behavior of the system. However, we cannot transform this equation into a *linear* model easily exploitable by control theory tools. More complex models might be possible, but as a first step we considered linear models. Linear models have the following form:

$$y_{k+1} = \sum_i a_i \times y_{k-i} + \sum_i b_i \times u_{k-i} \tag{5.3}$$

with $a_i, b_i \in \mathbb{R}$. However, as in our case $y_k = r_k + w_k$, this required formulation is impossible. As we are designing the controller to regulate y_k around the value $r_{\max}$, we can suppose that all the resources are being used and that the cluster is able to process $\frac{\Delta t}{\bar{p_j}} r_{\max}$ jobs per *CiGri* iteration. In this case, we can rewrite Equation 5.2 as:

$$\begin{aligned} r_{k+1} + w_{k+1} &\simeq r_k + w_k - \frac{\Delta t}{\bar{p_j}} r_{\max} + u_k \\ \Leftrightarrow y_{k+1} &= y_k + (u_k - \frac{\Delta t}{\bar{p_j}} r_{\max}) \end{aligned} \tag{5.4}$$

$\frac{\Delta t}{\bar{p_j}} r_{\max}$ is called the operating region of the system.

Knowing the model of the open-loop system, we can design a controller to regulate the closed-loop system. Similarly to the work in Chapter 3, we decide to use a Proportional-Integral controller for its precision and robustness. In the following, we use $k_s = 10$ and $M_p = 0$.

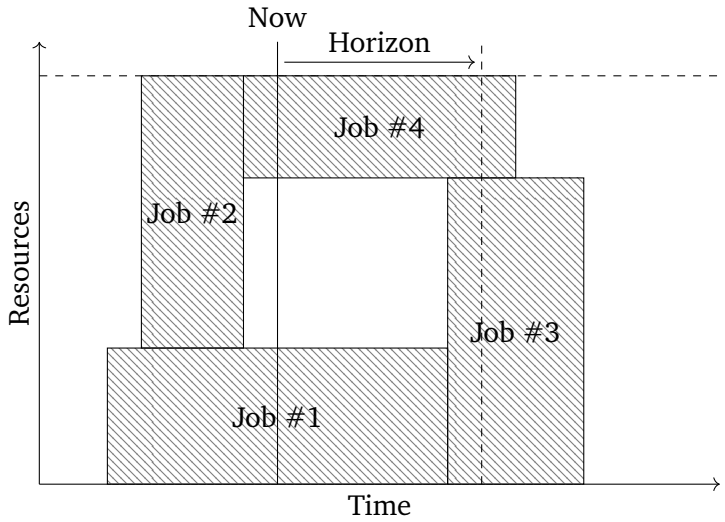


Figure 5.2.: Graphical explanation of the prediction Gantt sensor. The value returned by the sensor is the number of resources that will be used by normal jobs in *horizon* seconds.

5.2.5 Taking the future into account with help from the scheduler

For now, our controller only reacts to the instant changes in the system (arrival or departure of normal jobs). To reduce the wasted computing time (idle and killed), we need a way to predict those changes on the system and take them into account in the controller. Doing this would allow the controller to proactively increase or decrease the number of *CiGri* jobs to submit to *OAR* to avoid the killing of *Best-Effort* jobs or the idling of some resources. We thus need a way to query the provisional schedule of *OAR* to extract information.

In the current state of *OAR*, the available information through the API are not enough to know the provisional schedule. We thus slightly modified *OAR* (about 30 lines of code) to implement a new software sensor by introducing a new route in its API that returns the number of *normal* jobs that are predicted to be running at a given time in the future. We call this time the horizon (h), and its value is decided in *CiGri*. Figure 5.2 depicts the idea of this sensor.

There are several ways to inject the information returned by this new sensor into the controller. We took inspiration from the *feedforward* techniques of Control Theory, and decided to change dynamically the reference value. If we note d_k^h the number of resources that will be used by normal jobs in h seconds (the horizon), then we can redefine the reference value for the controller as $y_{ref,k} = r_{\max} - d_k^h$. Roughly, the reference value is the number of available resources for the *CiGri* jobs in h seconds. We also adapt the operating point in Equation 5.4.

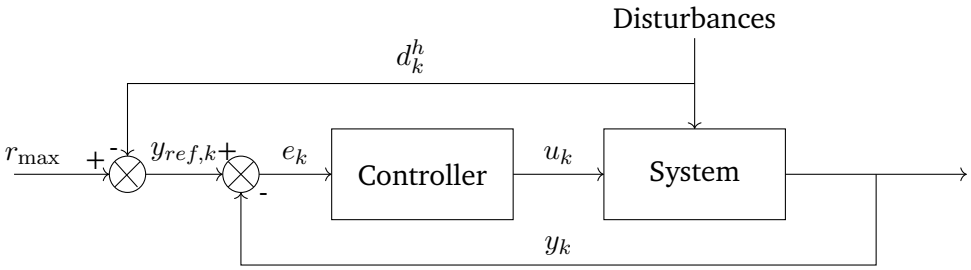


Figure 5.3.: Feedback loop representing the control scheme. The reference value (y_{ref}) is proactively changed to take into account the future availability of the resources (d_k^h).

The value of the horizon is a parameter of the controller, which carries some trade off. A small horizon value makes the sensor highly sensitive to miss evaluation of the walltimes of the normal jobs, and requires a fast response from the controller to meet the desired state. This would lead to an increase of the killing of *Best-Effort* jobs. On the other hand, a large horizon value might be too conservative and lead to more idle machines.

Figure 5.3 summarizes the control scheme of our system with this new sensor.

Take away 5.1

By slightly modifying *OAR* we can extract information about the provisional schedule. From this information, we adapt the reference value proactively to avoid the killing of best-effort jobs. The introduction of this new sensor constitutes a more advanced coordination between the controller and the scheduler.

5.3 Evaluation

5.3.1 Experimental Setup

The experiments were carried on the `dahu` cluster of *Grid'5000* [Bal+13] where the nodes have 2 Intel Xeon Gold 613 with 16 cores per CPU and 192 GiB of memory. The reproducibility of the deployed environments is ensured by *NixOS Compose* [Gui+22b]. The environments are available at [fee23]. For each experiment, we deploy 3 nodes: one for the *OAR* server, one for *CiGri*, and one for a *OAR* cluster of 32 resources. Note that we do not deploy 32 nodes for the cluster, but instead deploy a single node and define 32 *OAR* resources. This choice is made with

energy concerns in mind, as deploying a full size cluster to perform the following experiments would be an aberration. We do deploy the real software stack (*OAR*, *CiGri*), but no real job is executed, only `sleeps` (see discussion about the job model in Chapter 2). This representation of the jobs allows us to emulate several *OAR* resources on a single physical machine without introducing noise to the sharing of computing resources for computation, communication, storage, etc.

5.3.2 Experimental Protocol

We want to evaluate the controller proposed in Section 5.2. We consider the following workload from the normal users of the cluster: at time 1000, a new job arrives and takes 8 resources, then at time 1005, a job requesting 25 resources is submitted. The second job has to wait for the first one to finish in order to start as there are not enough available resources for it to start. With this scenario, we want to observe (*i*) the behavior of the controller when there is an abrupt change in available resources (arrival of the first job), (*ii*) the reaction to the killing of *Best-Effort* jobs, and (*iii*) how the controller will proactively decrease and increase the number of jobs to submit to meet the start and end of the second job.

We will evaluate the controller on this scenario with different parameters: different execution times of the *CiGri* jobs, and different horizon values for the sensor described in Section 5.2.5. We consider 3 execution times (p_j) for the *CiGri* campaigns: 30s, 60s, and 240s. Those execution times correspond respectively to the first quartile, median, and third quartile of the execution times of the *CiGri* jobs on the *Gricad* platform (see Figure 2.1). For the horizon, we consider 9 different duration: no horizon (or 0 second), 30s, 60s, 90s, 120s, 150s, 180s, 210s, 240s. The horizon values are multiples of $\Delta t = 30s$ as smaller values would miss some behaviors. The maximum horizon that we consider is 4 minutes which is the third quartile of the execution times. Each experiment will be repeated 10 times to reduce the noise due to the time *OAR* needs to set up and clean the resources before and after a job.

5.3.3 Constant Injection taking into account the future

As a first comparison point, we also implemented a constant submission algorithm. At each *CiGri* iteration, we submit $\frac{\Delta t}{\bar{p_j}} \times r_{\max}$. The value of $\bar{p_j}$ is updated during the execution of the campaigns jobs. This constant injection is supposed to fill exactly the cluster if there are no normal jobs.

We also made a variation of this submission algorithm by using the sensor defined in Section 5.2.5. In this variation we change the constant submission by taking into account the number of resources that will be available. The value of r_{avail} represents the number of resources available for *CiGri*. In brief: $r_{avail,k} = r_{\max} - d_k^h$. Thus, at each *CiGri* iteration, we submit $\frac{\Delta t}{\bar{p}_j} \times r_{avail,k}$.

Figure 5.4a depicts the distribution of the percentage of computing time lost due to idle resources (left column) and due to killed jobs (right column). We can see that considering the horizon does not have a noticeable impact on the percentage of lost time due to idle resources for $p_j = 30s$ and $240s$. In the case of $p_j = 60s$, we see that longer horizons lead to more idle resources, which is expected. Note that for $p_j = 240s$, we see a decrease in the percentage of killed jobs with the increase of the horizon. This decrease is not noticeable for the other processing times considered. In our scenario, a constant submission will fill the waiting queue of *Best-Effort* jobs of *OAR*, and the jobs accumulated in the waiting queue will be scheduled by *OAR* and get killed when the second job starts.

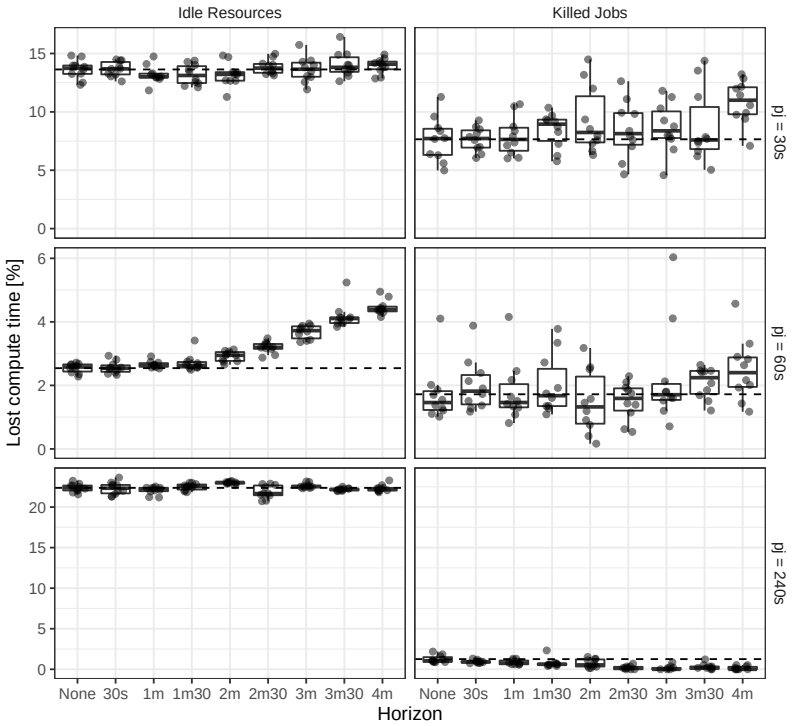
Take away 5.2

Taking the future into account in a step-based submission strategy does not yield any noticeable improvement of the usage of the resources.

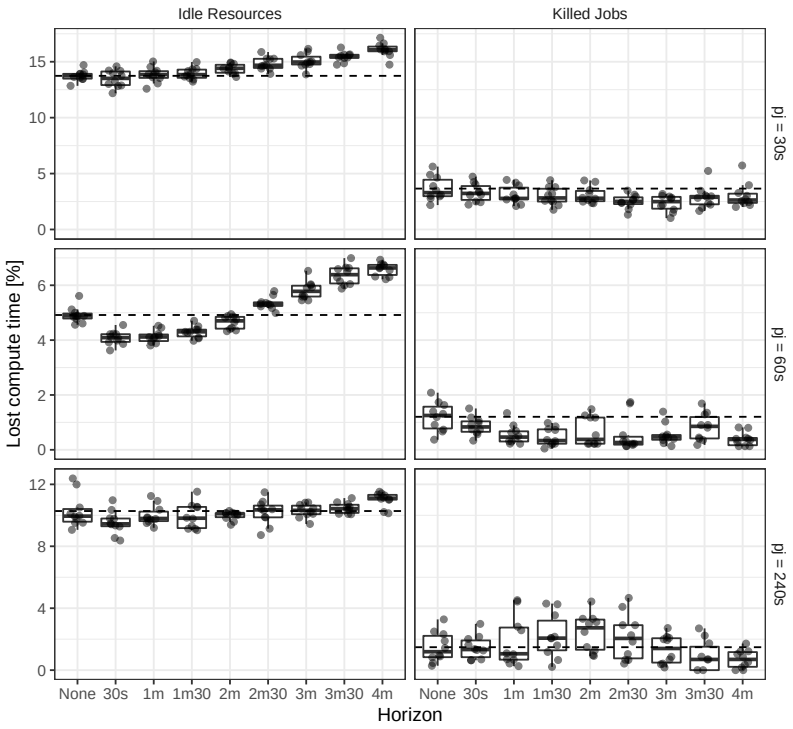
5.3.4 PI Controller taking into account the future

Figure 5.4b depicts the distribution of the lost compute time due to idle resources and killed *Best-Effort* jobs for a PI Controller with various horizon lengths. We can see that for small horizons (30 or 60 seconds), having this mechanism allows to reduce the idle time of the system. However, the longer the horizon, the more the idle time, but the less killing of the jobs. **There is thus a trade-off between killing less and harvesting more**.

Compared to the constant injection depicted in Figure 5.4a, we can see that the percentages of lost computing powers are slightly different. A PI controller without horizon will lose around 5% of computing power due to idle resources, while losing one percent due to killed jobs. On the other hand, the constant controller will lose less concerning idle resources (around 2.5%), and lose a lot more by killing jobs (about 2%). This is because of the accumulation of jobs in the waiting queue, which are scheduled by *OAR* even if they are destined to be killed. This highlights again the added value of the feedback regulation of the system.



(a) Distribution of the lost compute times for the constant submission with horizon.



(b) Distribution of the lost compute times for the PI Controller with horizon.

Figure 5.4.: Distribution of the lost compute times due to idle resources (left column) and because of killing *Best-Effort* jobs (right column). The x-axis represents the horizon of the sensor described in Section 5.2.5. Figure 5.4a presents the results for the constant submission with horizon, and Figure 5.4b for the PI Controller with horizon. The dashed line is the mean lost time for the solution without horizon.

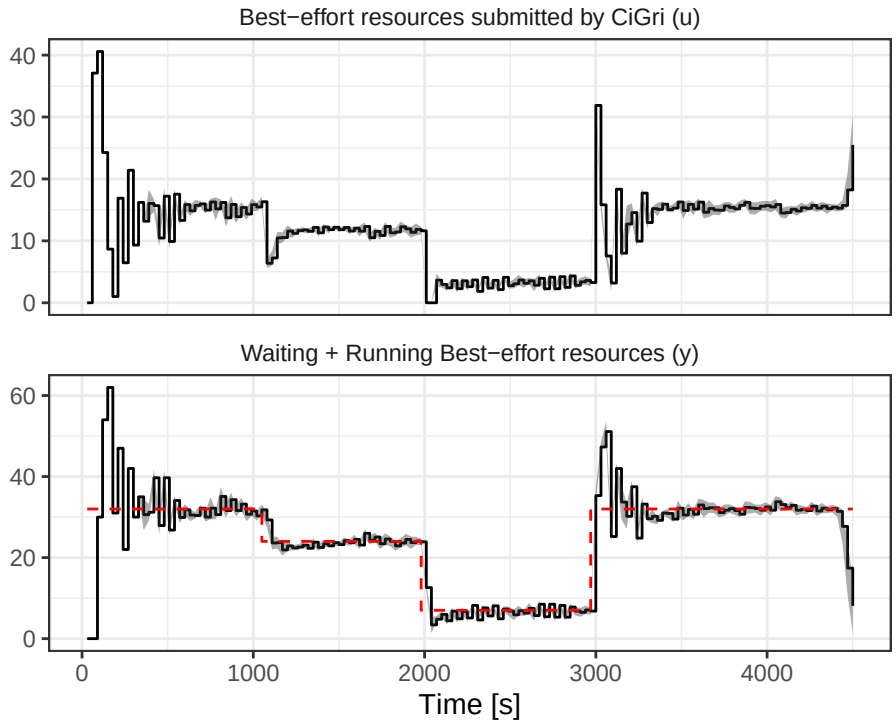


Figure 5.5.: Control signals for a scenario with $p_j = 60s$ and a horizon of 60s. The top plot represents that number of resources submitted by *CiGri* through time. The bottom plot depicts the value of our sensor, as well as the number of available resources to *CiGri* in dashed red.

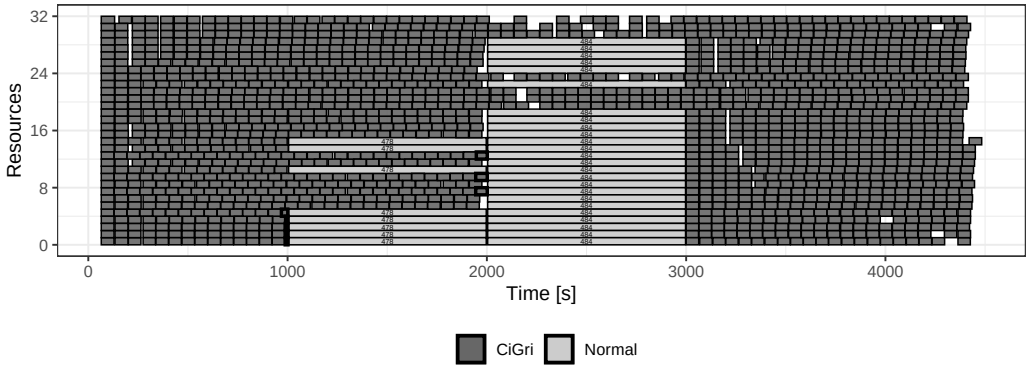


Figure 5.6.: Gantt chart for a scenario with $p_j = 60s$ and a horizon of 60s. The killed jobs are depicted with a thicker contour.

Figure 5.5 shows the temporal evolution of the control signals for a scenario with a campaign of one-minute long jobs and a horizon value of 60 seconds. The number of available resources through time is depicted in dashed red line. We can see that the controller is able to adapt the number of jobs it submits to *OAR* (top graph) to meet the reference value (bottom graph). We do notice some overshooting around 100s and 3000s. Those are due to a large variation in the reference value, but the controller manages to stabilize the system in a couple of minutes. The overshooting could be tamed by changing the gains of the controller, and especially the M_p parameter presented in Section 5.2.4. This issue might also come from the imprecision of our model due to the estimation presented in Section 5.2.3. Even if we designed the controller to reach the reference value within 10 *CiGri* iterations, it seems able to reach it less for small variations of reference value.

The Gantt chart of the previous scenario is presented in Figure 5.6. The killed jobs are depicted with a thicker border. We can see that there are some *CiGri* jobs killed at $t = 1000s$, which is unavoidable for this scenario as the priority job starts immediately and the sensor controller cannot anticipate it. Only three jobs are killed when the normal job starts at $t = 2000s$. We observe that there are some idle time right after the start of the second normal job. This is due to the reaction of the controller, and it also can be observed on the bottom plot of Figure 5.5 around 2000 seconds, where the output signal is under the reference value (red dashed line).

Take away 5.3

When using the additional information about the future schedule with a PI controller, we are able to improve both the number of idle resources and the number of killed jobs. For large values of horizon, the controller anticipates too much and it leads to an increase of the idle time.

5.3.5 Global Comparison

From the point-of-view of energy consumption, idle resources and killed jobs do not consume the same amount of energy. Indeed, in practice, the power used by a machine when a job is running is about twice more than when the machine is idle [Hei+17]. In this work, we did not measure the energy consumption of the jobs during the experiments as we are using `sleeps` to represent the CPU time. To estimate the gain in energy consumption of the cluster for each strategy of submission presented in this paper, we will simply multiply by two the weight of the computing time lost due to the killing of jobs.

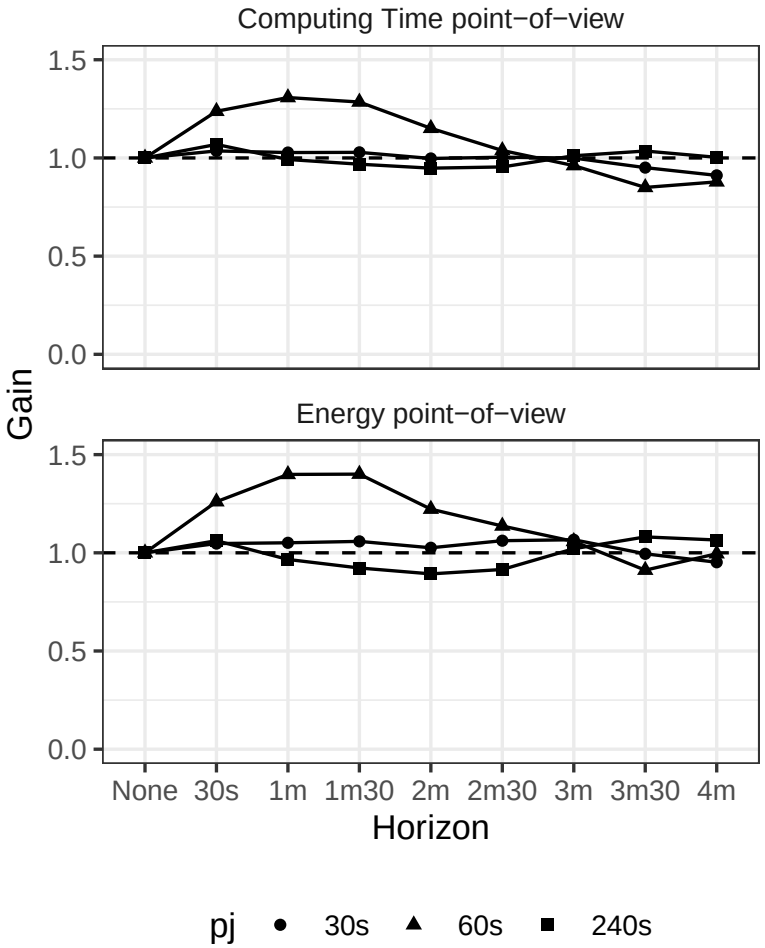


Figure 5.7.: Gain of using the PI controller compared to the constant submission algorithm. The top plot represents the gain from the point-of-view of the total energy consumption loss. The bottom plot considers the computing time lost. Values above one indicate that the PI out-performs the constant submission algorithm.

Figure 5.7 shows the gain of considering the prediction sensor in the PI Controller. We look at two gains: the one in lost computing time (top plot) and the energy loss (bottom plot). We can see that for all the considered processing times (p_j) just having a horizon of 30 seconds improves the usage of the cluster. The best improvement is reached for a processing time of 60 seconds and a horizon of 60 seconds with about 25% gains in computing time and 40% in energy.

We think that it is possible to get the same behavior of the best horizon by changing the sampling time of *CiGri* (Δt) to match the characteristics of the *CiGri* campaign:

$$\Delta t \simeq \frac{\bar{p_j}}{2} \text{ and } h \simeq \bar{p_j} \tag{5.5}$$

This choice of Δt could be explained by the Nyquist frequency which stays that a process should be sampled twice faster than its dynamics. Concerning the choice of the horizon value, the intuition is that values smaller than $\bar{p_j}$ could lead to some killing, and values larger than $\bar{p_j}$ could lead to the idling of machines.

Take away 5.4

Using a PI with information about the future improves the global usage of resources of 25%. If we consider the energy point-of-view, with a rough approximation that an idle node consumes twice less than a working node, then our controller can improve the energy usage by 40%.

5.4 Conclusion and Future Work

This chapter has presented an approach to the harvesting of idle computing resources in an HPC system using a feedback loop approach coupled with *Control-Theory* tools. Contrary to the harvesting solutions of the literature, we also considered the killed jobs as a source of wasted computing time. We have shown that our solution, a Proportional-Integral Controller, can reduce the total amount of wasted computing time compared to an ad-hoc solution. We show that by also taking into account the predictive Gantt chart of the scheduler, we are able to reduce even more the wasted computing time. The choice of the horizon value has an impact on the total usage of the cluster. A horizon value of 30 seconds yields an improvement for all the considered campaigns. It is however possible to reach better performance by adapting the sampling time of *CiGri* to the running *Bag-of-Tasks* campaign (see Equation 5.5).

Our work shows promises, but has some limitations. We supposed that the normal user know exactly the duration of their jobs and set the walltime accordingly, but in practice such behavior rarely happens [CB01; MF01]. We think that the controller presented in this chapter should adapt relatively well to imprecise walltimes due to the robust nature of the Proportional-Integral controller. However, we believe that the robustness to these imprecisions of walltimes could be improved by considering a probabilistic approach to the durations of the jobs. Control Theory also proposes *Feedforward controllers* which bare the same idea as the controller above, but are more complex and powerful because they also model the disturbances. Future work include to implement such a controller with the help of Control Theory experts.

6 Conclusion and Perspectives

The *CiGri* middleware exhibits several regulations problems. In this thesis, we focused on two aspects: regulation based on the load of the distributed file-system (Chapter 3), and regulation to reduce the amount of lost computing power (idle machines and killed jobs) (Chapter 5). We showed that such problems can be addressed with tools from the fields of the Autonomic Computing and Control Theory. The main limitation of these works is the need for a "strict" model of the target system. In Chapter 4, we investigated the impact in performance of reusing a controller designed on a system on another system. We defined comparison criteria and gave guidelines for choosing a controller based on tradeoffs involving performance, complexity, guarantees, and competence required.

Even if we tackled the regulations problems independently, we can merge the controller defined in Chapter 3 and Chapter 5 into a single Autonomic Controller that can satisfy the different objectives. Figure 6.1 depicts what could be the single feedback loop for the different controllers and objectives.

6.1 Perspectives on *CiGri*

Perspectives on *CiGri* include:

Considering several clusters In this document we consider a single cluster, but *CiGri* actually can submit to several clusters. The immediate solution would be to have a single controller per cluster and apply the work in this thesis. This approach could miss the opportunity to exploit potential affinity between campaigns and clusters. Determining this affinity could be done by a combinatorial bandit approach [CL12; SG19] with a process of *exploration* where *CiGri* would submit jobs from a campaign to different clusters, and observe the performance of the jobs or the impact on the shared resources, and then *exploit* a chosen coupling between campaign and cluster until the next phase of exploration.

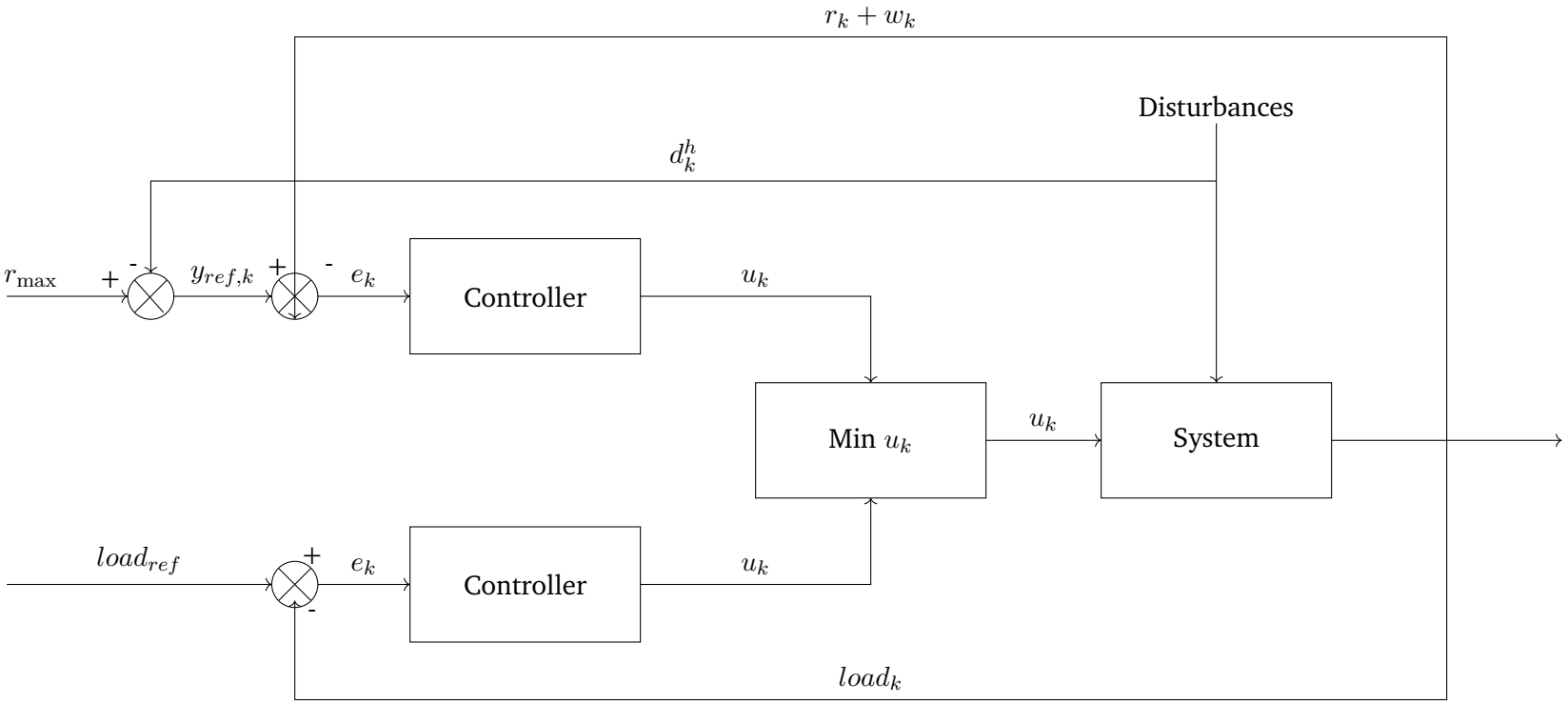


Figure 6.1.: Feedback loop gathering the controllers and objectives from Chapter 3 (bottom) and Chapter 5 (top). At every iteration, we ran both controllers and take the minimum number of jobs in the submission (u_k) to satisfy the objectives. In the case of a PI controller, we would only add the error to the integral part only if the chosen submission size comes from this controller. It would be also possible to add the feedback loop presented in Figure 3.13.

Sampling time of *CiGri* By default, the *CiGri*'s loop is executed every 30 seconds. This value is fixed and ad-hoc, but does bear some importance. Indeed, this *sampling time* must be small enough to capture the behavior of the system, but not too small to react on noise. In Chapter 3 we take the *CiGri* jobs to have an execution time of 30 seconds. This helps the controller as longer lasting jobs would add some *delay* in the system, which cannot be capture by a first order model, as used in this thesis. One immediate solution would be to adapt the sampling time based on the current campaign being executed. In Chapter 4, we showed that the controllers were quite resistant to variation of the execution times. But in Chapter 5, we concluded that adapting the sampling time of *CiGri* to half of the execution time of the jobs of the campaigns could lead to significant improvement of the global usage of the machines.

***I/O* characteristics of *CiGri* jobs** In Chapter 3, we consider known the amount of *I/O* done by a *CiGri* job. In practice this information might be tricky to obtain. Users could give an estimation of the *I/O* load of their job, but there will be no guarantee on the quality of this estimation. A heavy instrumentation and monitoring of the jobs and/or the system would make this information available and more precise than given by users. Tools such as Darshan [Car+11] or Colmet [Eme+14], could help to get this information respectively by instrumenting the jobs or the platform. Using Adaptive controllers can help to reduce the need of such a characterization, but require more Control Theory knowledge, as seen in Chapter 4.

Sensors for Parallel File-Systems The work presented in this thesis supposes the use of a centralized distributed file-system like NFS. This assumption makes it easier to implement a sensor on a file-system. However, in large HPC centers, shared file-systems are usually parallel with Lustre, BeeGFS, GlusterFS, etc.
Such Parallel File-Systems (PFS) are composed of several nodes to balance the load. Implementing a sensor on a PFS might be tricky. A simple solution would be to keep the `loadavg` and average it over all the nodes of the PFS. But this solution might lose the meaning of the `loadavg`, and weird behaviors might happen. Another solution might be to look at the bandwidth of the PFS. But such metrics could be greatly varying through time, and would require some sort of filtering to be usable by a controller.

It might also be interesting to investigate even lower level sensors with the help of the eBPF technology. The development of such sensors would enable to have very specific, and less noisy information on the system.

Try using models of greater orders In this thesis, we only considered models, in the Control Theory sense, of the first order. Such models have the advantage of being simple, but might also lack realism. For instance, in some cases, the *CiGri* system might be impacted by delay, which should be considered in the model. This delay requires to use a greater model order.

Phase detection HPC applications are usually iterative and thus have repeating phases where they do computation, then perform *I/O* operations, then computation again, etc. Detecting such *I/O* phases (*e.g.,* [Sto+21; Tar+23]) would be a great additional information for the *CiGri* controller. This new sensor can then be coupled with a feedforward control strategy to proactively adapt the submission of *CiGri* to avoid *I/O* overload of the file-system.

Deployment onto the Gricad computing grid The end goal is of course to deploy our controllers onto the real platform. From the software management point-of-view, it should be quite easy as we maintain a fork of *CiGri* with our modifications [cig23]. It will require the additional work of setting up the controller, which is done with a single JSON file.

6.2 Potential Regulation Problems of Interest

This Section presents some potential interesting systems which suffer from regulation problems that could be addressed with tools from Autonomic Computing and Control Theory.

Bebida ([Mer+17]) In Bebida, a partition of the cluster is allocated for Big-Data jobs only, and the remaining of the platform is available for both HPC jobs and Big-Data jobs. The authors showed that their solution improves the total usage of the resources at the cost of a higher mean waiting times for the HPC jobs.

As a first approach, the size of the partitions is fixed. An interesting question would be to try regulating Quality-of-Service metrics for the HPC jobs based on the size of the partitions. The actuator would be the size of the partition only of Big-Data jobs, and the sensor could be the waiting times of jobs, or the bounded slowdown of the HPC jobs.

Such a regulation could provide a finer Quality-of-Service to the users, while potentially also reducing the number of Big-Data jobs killed and rescheduled.

Turning on and off machines (Chapter 7 of [Poq17]) When cluster nodes are not being used, one easy way to save energy is to turn them off. However, when users need the turned off machines to run their jobs, the switching on process takes time and energy, which would degrade the Quality-of-Service for the users. A regulation problem thus arises. The objective would be to decide how many nodes to keep turned on based on metrics on the jobs (waiting time, bounded slowdown, etc.) Additional sensors on the platforms like the provisional schedule, or historical usage metrics could also be used to increase precision. The work from [Cer19] can give inspiration.

Best-effort time-sharing with accepted degradation of performance In the spirit of harvesting idle resources, the time-sharing of HPC applications on the same node could improve the usage of machines at a finer grain than *CiGri* for example. A best-effort application could run in a `cgroup` which resource quotas are dynamically regulated to guarantee an accepted degradation of performance by the priority application running on the same node. Similarly to [Cer+21], the priority application would need to be iterative and send a "heartbeat" [Ram+19] to the controller at the end of each iteration. The heartbeat frequency would be the signal to control, and the actuator the resources quotas of the `cgroup` hosting the best-effort application.

Part II

Improving the Reproducibility and Cost of Distributed Experiments

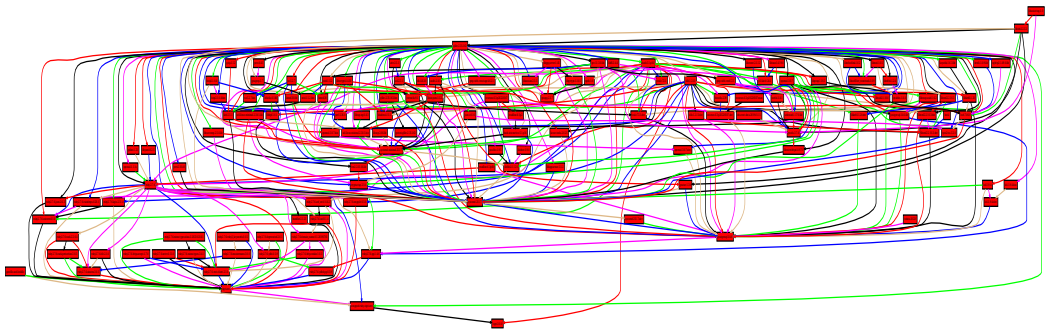
7 Discussion and State-of-the-Art

To prove the quality of a proposed solution, experiments must be performed. However, different research communities of Computer Science have different visions of what is an experiment. More theoretical fields tend to implement their own small simulator to prove the quality of their novel algorithm. Those simulators are then run on synthetic workloads that match the requirements of the algorithm to evaluate (*e.g.,* jobs execution times follows distribution X with parameters Y and Z), even if those requirements are partially met in the real life. One advantage of such simulators is that they can run on a personal laptop in reasonable time. However, only using such simulation techniques also does not take care to the hidden technical problematics of implementing in practice the proposed solution, which might in some cases require a modification of the solution. The realism of simulation, in the context of distributed systems, can be improved by using theoretically and experimentally evaluated and validated simulators such as SimGrid [Cas+14].

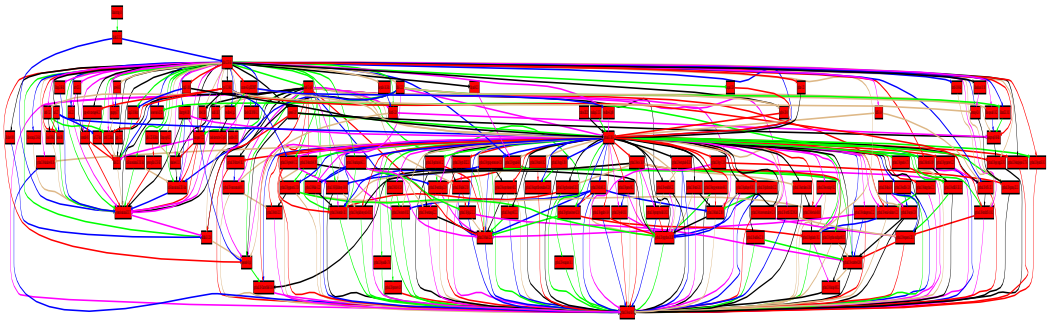
On distributed systems, the delay of communication, the I/O, between actors must be taken into account. Setting up an experiment on distributed systems is so complex, that researchers often make experiments directly on computing centers (*e.g.,* evaluating the parallel file-system of the computing center, instead of deploying one for the experiment).

Experimenting in research topics requiring specific hardware (*e.g.,* GPU, NVRAM, FPGA, Watt-meters, special architectures, etc.) is more tricky. Usually those pieces of hardware are not present on regular laptops, and experiments must thus be done on test-beds such as *Grid'5000* providing access to various hardware. Experimenters can then reserve a node of a cluster with the desired component to experiment.

In the previous part of this thesis, every experiment deploys the actual *CiGri* software, an actual *OAR* cluster, and set up a NFS file-system. Each piece of the deployment requires its own software environment and configuration which can be quite complex to set up correctly, and especially on the first try. Figures 7.1a and 7.1b illustrate the complexity of a software environment for a grid or cluster middleware, *CiGri* and *OAR* in this case. The usual way to set up experiments on a cluster middleware



(a) Graph of the software dependencies of *CiGri*



(b) Graph of the software dependencies of *OAR*

Figure 7.1.: Software dependencies of *CiGri* (Figure 7.1a) and *OAR* (Figure 7.1b).

is to create software environments and then deploy them onto physical machines. Both creating the environment and deploying it is time-consuming (around tens of minutes for each). Moreover, experimenters always need more than one iteration before reaching the desired state of the image. A forgotten package, a typo in a configuration file, a closed port, forgetting to copy an ssh key, such simple mistakes require to edit the recipe of the image and rebuild again (Figure 7.2). Such iteration times do not encourage experimenters to set up their environment properly, and they will often call it "good enough" even if there are still some "dirty hacks" remaining, which inevitably deteriorates the reproducibility of the experiments, and thus the quality of the scientific results.

7.1 Reproducibility

This section is based on [Gui+23a] published at ComPAS 2023 with Adrien Faure, Millian Poquet and Olivier Richard.

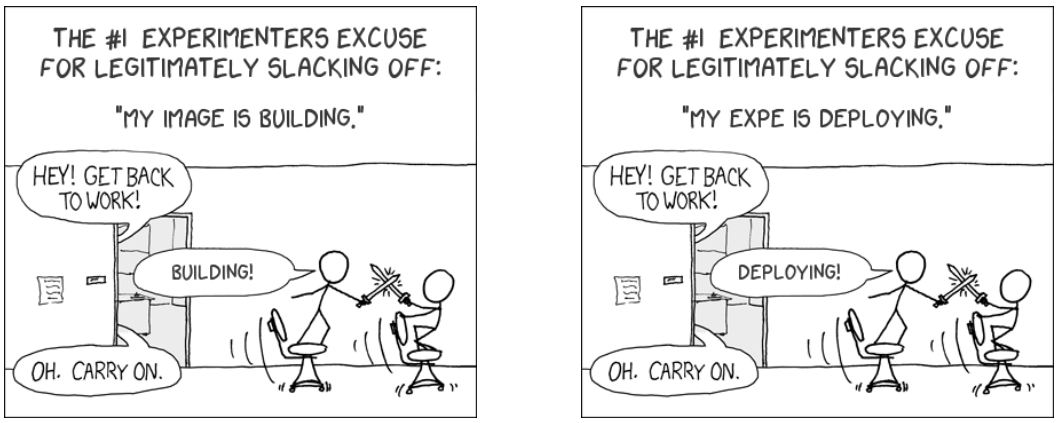


Figure 7.2.: Illustration of the process of creating a software environment for a distributed experiment. Adapted from [XKC].

The scientific community as a whole has been traversing a reproducibility crisis for the last decade. Computer science does not make an exception [RW18; Bak16].

The reproducibility of the research work is essential to build robust knowledge, and it increases the reliability of results while limiting the number of methodology and analysis bias. In 2015, Collberg et al. [CPW15] studied the reproducibility of 402 experimental papers published in *system* conferences and journals. Each studied paper linked the source code used to perform their experiments. On those 402 papers, 46% were not reproducible. The main causes were: *(i)* the source code was actually not available, *(ii)* the code did not compile or did not run, *(iii)* the experiments required specific hardware

To highlight the reproducible research works, several publishers (like ACM or Springer) set up an artifact evaluation of a submission. This peer review process of the experimental artifact can yield one or several badgers to the authors based on the level of reproducibility of their artifacts.

The term reproducibility is often used in a broad sense and gathers several concepts. The definitions that we will use in the rest of this thesis are the ones defined by ACM for the validation of the submitted artifacts [ACM]. It is composed of three levels of reproducibility:

1. *Repeatable*: the measures can be obtained again by the people at the origin of the work.

2. *Reproducible*: the measures can be obtained again by people who do not belong to the original work and with the original artifact of the authors.

3. *Replicable*: the measures can be obtained again by people who do not belong to the original work without the original artifact.

The evaluation of artifact is a crucial point which allows to guarantee the reproducibility of the experiments and the results. However, this reproducibility is not sufficient. Even if being able to reproduce an experiment is proof a scientific validation, the experiment and its environment are often too rigid to be extended by a third party, or even by the authors themselves. We believe that the notion that should be pushed by the community is the *reproducibility with variation*. By "variation" we mean that a third party is able to easily modify the environment of the experience to continue the research. This means that the hardware and software environments as well as the experimental scripts must be correctly defined and can be modified easily.

This section focuses on the software environment. For a global vision of the reproducibility problems, the readers might be interested in [IT18].

Section 7.1.1 gives the context and the motivation of reproducibility for distributed systems. Several frequently seen traps that might break the reproducibility of the software environment of an experiment are presented in Section 7.1.2. Finally, Section 7.1.3 motivates the use of Functional Package Managers as a solution to most of the reproducibility problems of the software environments.

7.1.1 Context & Motivation

Imagine that in your childhood your grandma cooked a delicious chocolate cake, and that you now want to know how to do it yourself. You even think that you can improve on it, and make it tastier! You can try to reproduce the cake based on your far memories and culinary intuition, but the result might be disappointing... Maybe your parents know the recipe! But you might just get some fuzzy instructions. Maybe your grandma just improvised the recipe! Who knows?

You decide to go through the old stuff of your grandma's house. By chance, you find an old recipe book. You open it and from it falls a piece of paper with what looks like a cake recipe. The handwriting matches the one from your grandma!

The recipe is clear, well detailed, and contains all the quantities for all the ingredients, the order of the different steps, the cooking time, etc. You decide to follow the recipe literally, but the final result is not what you remembered... Maybe you did not use

the correct type of eggs, or that your oven is too different from your grandma's. How to know?

An experiment without the environment in which it was executed makes it much more difficult to reproduce. Indeed, side effects from the environment can happen and change the results of the experiment. It is easy to forget to include in the software environment an element which impacts the performance of the experiment. The performances, but also the results of a simple C application can depend on the compilation options [SK18] or also from the quantity of UNIX environment variables [Myt+09].

Most of the current solutions in terms of "reproducibility" fall under the storage of artifacts (system images, containers, virtual machines) and replay of experiments [Ros+20; Bra+11; Bri+19]. Even if this is an important part of the reproducibility spectrum, nothing guarantees that the software environment can be re-built in the future, and thus nothing guarantees that the experiments can be re-run if the artifacts disappear.

The step of artifact evaluation for the conferences is done soon after their initial construction. It is thus very probable that the construction of the artifacts will be executed in a similar state of the packages mirrors (`apt`, `rpm`, etc.). However, what will happen when someone will try to rebuild the environment in 1 year? 5 years? 10 years? The objective of science is to base itself on robust works to continue to go forward (*Stand on the shoulders of giants*). This vision of "*short term reproducibility*" is a major obstacle to scientific progress and is in complete opposition to the science philosophy.

We think that the notion that should be highlighted is the concept of **variation** [MFR18; Fei15]. This means allowing a third party to use the environment defined for an experiment in order to investigate another research idea. An example of variation would be to change the MPI implementation used in an experiment (*e.g.,* MPICH instead of OpenMPI). Being able to introduce such a variation is only possible if the initial environment is correctly defined.

7.1.2 Frequent Traps of the reproducibility of software environments

Sharing the Environment An obvious way to fail the reproducibility of its experiments is not to share the used environments, or to share them in a perennial place. Platforms such as Zenodo [zen] or Software-Heritage [Her] allow users to store artifacts (scripts, data, environments, etc.) permanently.

Knowledge of the environment In the case where the software environment contains only Python packages, freezing the dependencies with `pip` (`pip freeze`) is not enough. `pip` only describes the Python environment, and ignores the system dependencies that numerous packages have. For example, freezing an environment containing the `zmq` Python package will not freeze the ZeroMQ system package installed on the system.
Even if re-creating a Python environment from a `requirements.txt` is simple, installing a list of system packages with specific version is on the other hand much more complex. Moreover, listing manually all the system packages by hand is error-prone, and the best way to forget a package.

Tools such as Spack [Gam+15] have a similar approach as `pip` but also for all the system packages and their dependencies. It is possible to export the environment as a text file and to rebuild it on another machine. However, the produced environment might not be completely identical. Indeed, Spack uses applications that are already present on the machine to build the packages from the sources. Especially, Spack assumes the presence of a C compiler on the system, and will use this C compiler to build the dependencies of the environment. Hence, if two different machines have two different C compiler then the resulting environment could differ from the desired environment. One clear advantage of Spack is the ease to introduce a variation in an environment through the command line.

Solutions such as Spack, `pip`, `conda` only focus on the software stack above the operating system. However, results from experiments might depend on the version of the kernel, some drivers, etc. Thus, it is important to capture *entirely* the software stack.

Pitfall 1: Partially capturing the software environment of the experiment

Experiments do not only depend on Python packages, but they can also depend on system packages, or even the version of the Linux kernel. The software environment must be capture *entirely*.

Usually, the capture of the entire software stack goes through it encapsulation in a system image. This image can then be deployed on machines to execute the experiments. A way to generate a system image is to start from a base image, deploy this image, execute the commands required to set up the desired environment, and finally compress the image. Platforms such as *Grid'5000* [Bal+13] and Chamelon [Kea+20] propose to their users such tools (`tgz-g5k` [Gri] and `cc-snapshot` [Clo] respectively). In the context of repeatability and replicability, if the image stays available then this way to produce system images is adequate at

best. But, concerning the traceability of the build, one cannot verify the commands that have been used to generate the image, and thus relies completely on the documentation from the experimenter. Moreover, such images are not adapted to be versioned with tools like `git` as they are in a binary format. In the situation where the image is no longer available, re-building the exact image is complex and the precise introduction of variation is utopian.

Depending on a uncontrollable state One way to deploy a complete environment could be to rebuild it at each experiment. EnOSlib [Che+22] is a Python library to manage distributed experiments. It integrates a mechanism to install system packages in a programmatic fashion. Users of EnOSlib can then, by executing their Python script, base their environment on a default image that they can modify as they need for their experiment. This strategy of starting from scratch at every deployment has the advantage of making it harder to forget a dependency in the environment, as its absence would be detected at every execution. However, it is still possible to make mistakes. The reproducibility of the experiments using such solutions strongly depends on the base environment on which they are based. These base environments are managed by administrators of the platforms and also have a finite lifetime, which raises the question of their permanence.
Suppose that the administrators update the version of the base environment, what happens to the reproducibility of an experiment?

Pitfall 2: Forgetting to capture the software environment of the experiment/workflow manager

Capturing the software environment in which the experiment/workflow manager is executed is as important and the environment of the experiment.

A better approach to generate image is via *recipes*. Those recipes, like `Dockerfiles` for Docker containers or Kameleon [Rui+15] recipes for system images, are a sequence of commands to execute on a base image to generate the desired environment. The text format of recipes make then much more suitable to version, share, and reconstruct them. These base images have often several versions which are identified by labels called *tags*. In the case of Docker, the tag of the latest version is often called `latest`. Basing an environment on this tag breaks the traceability, and thus the reconstruction of the image itself. Indeed, if a newer version is available at the time of a future rebuild of the environment, then the image will be based on this newer version and not the original version. Another important question is to know if the base image and all the version are themselves reconstructive, and if it is

not the case, what is the permanence of the platforms hosting those images? For instance, the lifetime of `nvidia/cuda` Docker image is 6 months, after 6 months, the administrators delete the images.

Pitfall 3: Basing environments on non-reproducible images

Be cautious with the longevity of the images on which you base your software environment. By transitivity, if these images are not reproducible, so are yours.

Another frequent problem is that the recipe performs an update of the mirror (*e.g.,* `apt get update`) before installing the required packages for the environment. This has the bad property of breaking the reproducibility of the image. Indeed, the image depends on the state of an external entity which is not controllable. In order to be sure to use the exact same packages during a rebuild, a solution could be to use a *snapshot* of the mirror[1]. EnOSlib allows users to use this snapshot for the construction of the environment. Concerning the introduction of variation, to base an image on a *snapshot* is quite constraining and can make impossible the installation of specific version of packages, or can create conflicts with already installed packages.

When the recipe must download an object from the outside world, it is crucial to verify the content of the fetched object and compare it to the expected one. In the case where the object is not checked, the environment depends on the state of the source of the object at the time of construction. Hence, if the recipe calls `curl` to get a configuration file or a snapshot of a mirror for example, the recipe must also check the content of the files. The usual way to do it is to compare the cryptographic hash of the downloaded object and the one of the expected object.

A similar problem arises when the recipe downloads a package via `git` and build it from source. In this case, it is paramount to correctly set the commit used in the recipe of the image. Indeed, not knowing the commit used in the original image leads to having a dependence to the latest commit of the repository on the main branch. Setting the commit used allows to know exactly the sources used, and simplifies the *controlled* introduction of variation in the environment (by changing commit for example).

[1] Example for `debian`: `http://snapshot.debian.org/`

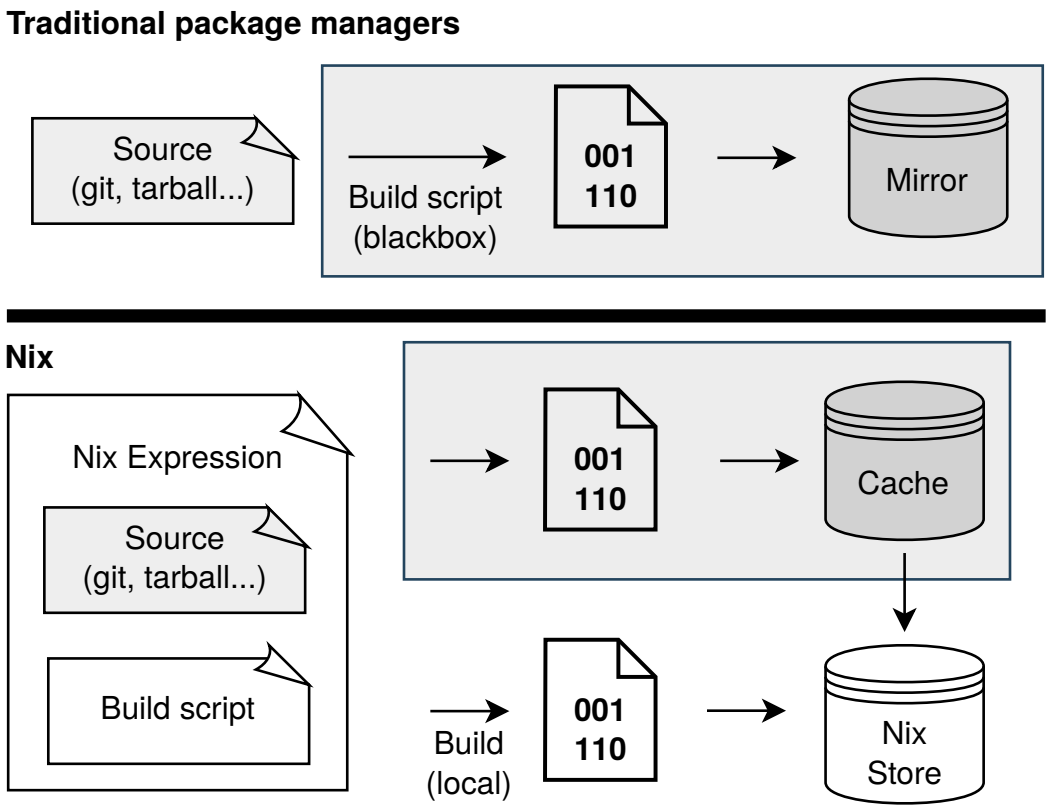


Figure 7.3.: Comparison between traditional package managers and *Nix*. Traditional package managers fetch a built version of the package from a mirror, but information on how they have been built is unknown. In the case of *Nix*, the package is described as a *Nix* function that takes as input the source code and how to build it. If the package with these inputs has already been built and is available in the *Nix* caches (equivalents of mirrors) it is simply downloaded to the *Nix Store*. Otherwise, it is built locally and added to the *Nix Store*.

Pitfall 4: No checking the content of downloaded objects

Every object coming from the outside of the environment must be examined to be sure that it contains the expected content. It is more important that the image fails to build if the content differs from the expected one, rather than the image silently builds with a different content.

7.1.3 Functional Package Managers

Tools such as Nix [DJV04] or Guix [Cou13] fix most of the problems described in the previous section. Nix and Guix share the similar concepts, in the following we will focus on Nix.

Nix is a pure functional package manager for the reproducibility of the packages. A Nix package is defined as a function where the dependencies of the packages are the inputs of the function, the body of the function contains the instructions to build the package. The building of the packages is done in a *sandbox* which guarantees the build in a strict and controlled environment. First, the sources are fetched, and the content verified by Nix. If the hash of the sources differs from the expected hash, Nix stops the building of the package and yields an error. Nix fetches also the dependencies and recursively. The build commands are then executed in the sandbox with the environment defined by the user. At this stage, no network access or access to the file system is possible.

Nix can generate environments that can be assimilated as multi-languages `virtualenvs`. But it can also create containers images (Docker, Singularity, LXC, etc.), virtual machines, or full system images. The process of building an image with classical tools (`Dockerfile`, Kameleon recipe, etc.) is often iterative and arduous. Defining an image with Nix is done in a *declarative* fashion. This has the advantage of making the building of the image faster when modifying an already built recipe [Gui+22b]. It also avoids the annoying optimization of the order of operations, frequent when building from a `Dockerfile` [Doc]. As Nix packages are functions, introducing a variation means changing an argument when the function is called.

Systems like `debian` store all the packages in the `/usr/bin` and `/usr/lib` directories. This ordering can lead to conflicts between different versions of the same library, and it thus limits the introduction of variation in the environment without breaking the system. On the other hand, Nix creates one directory per package. Each directory name is prefixed by the hash of its sources. Hence, if a user wants to install a different version of an already installed package, the sources will be different, thus the hash will be different, and Nix will then create a new directory to store the new package. Those individual directories are stored in the *Nix Store* located at `/nix/store`, in a read-only file-system. Figure 7.3 summarizes the differences between traditional package manager and *Nix*. The advantage of this fine-grained isolation method, is the *precise* definition of the `$PATH` environment variable to manage software environments.

The definition of packages through function also eases their sharing and distribution. There is a large base of package definition done by the community, called `nixpkgs` [Nix23]. Users can easily base their new packages, or environment on those definitions. It is also possible for independent teams and research groups to have their own base of packages. Guix-HPC [Gui23e], NUR-Kapack [OAR23], or

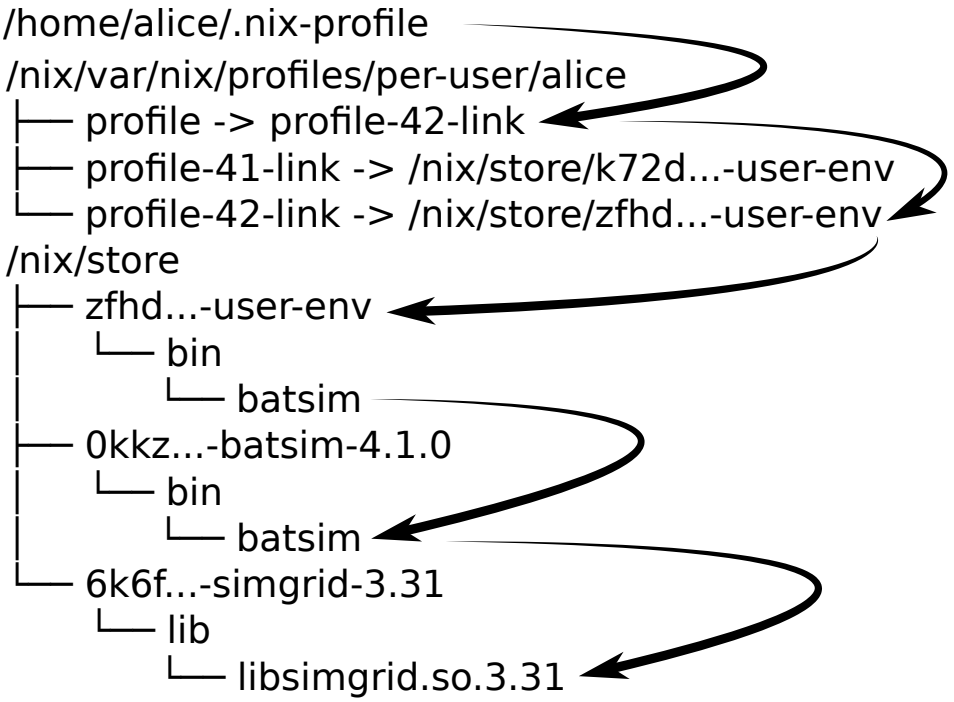


Figure 7.4.: Figuration of the *Nix Store* content when the `alice` user has installed a Batsim [Dut+16] binary in her profile. As Batsim requires the SimGrid [Cas+14] library at runtime, SimGrid must also be in the store. Packages are stored in their own subdirectory, but common dependencies are not duplicated as symbolic links and shared libraries are used.

Ciment-channel [Gri23] are examples of independent packages base for HPC and distributed systems.

A ***Nix* system profile** defines the configuration of the system (packages, `initrd`, etc.). Among many features, a profile can define filesystems such as NFS and mount them automatically at boot time. Figure 7.4 depicts an example of user profile containing the Batsim application [Dut+16], which requires the SimGrid [Cas+14] library at runtime. *NixOS* extend the ideas of *Nix* to the entire operating system. A *NixOS* image can contain several profiles and *Nix* can switch between them by modifying symbolic links and restarting services via `systemd`.

7.1.4 Limits of Functional Package Managers

Even though tools like Nix and Guix greatly improve the state of reproducibility for software environments, it is still possible to go wrong and make a package impure or to depend on an exterior state. Nix is currently addressing this issue with the experimental feature *Flake* [Twe].

To ensure the reproducibility and traceability of an environment, Nix requires that all the packages and their dependencies have their source code open and that the packages are packaged with Nix. This could seem limiting in the case of proprietary software where the source code is unavailable (Intel compilers for example). It is still possible to use such proprietary packages with the `impure` mode of Nix, but it breaks the traceability and thus the reproducibility of the software environment.

The construction of the packages in a sandbox goes through an isolation mechanism of the file-system using `chroot`. Historically, this feature is only available to users with `root` privileges. But in the case of computing clusters, this kind of permissions greatly limits the adoption of Nix or Guix. However, the *unprivileged user namespace* feature of the Linux Kernel allows users to bypass this need of specific rights in most of the cases.

As Nix needs to recompile from source the packages are not available in its binary cache, it is possible that a future rebuild is impossible if the host of the source code disappear [bli]. However, as Software Heritage now performs frequent archives of the open source repositories, it should be possible to find the sources of interest if needed.

These tools also require a change of point-of-view in the way of managing a software environment, which might make the learning curve intimidating.

7.1.5 Conclusion

The computer science community starts to get interested in the problems of reproducibility of experiments. However, the problems of reproducibility at the software level are not truly understood. Setting up reproducible experiments is *extremely* complex. The management of the software environment illustrate one facet of this complexity. The usual tools (`pip`, `spack`, `docker`, etc.) do not answer the reproducibility problems without a huge effort by the experimenters, and only allow a *short-term reproducibility*. The *graal* of reproducibility is the precise introduction of variation in a third party defined environment. This need for variation allows scientists to use solid contributions to continue research. Even if there are no perfect solution yet, the need for a change of practice concerning reproducibility is needed.

7.2 Research Questions

The experiments conducted in this thesis on the *CiGri* system are relatively long-lasting. They should be long enough to observe the impact of the controllers, which can take a few hours. Then, as the system is relatively complex and noisy, every experiment must be repeated several times in order to compute an average behavior. They have the advantage of running the *actual* middlewares (*CiGri*, *OAR*, File-systems).

The experiments of this thesis raised several questions from the experimental point-of-view.

- **RQ1**: How to perform reproducible *distributed* experiments with a complex software environment such as *CiGri*'s?
- **RQ2**: At which scale should experiment on cluster and grid middlewares be performed?
- **RQ3**: Can State-of-the-Art simulators help to reduce the experimental cost to set up an autonomic controller for systems such as *CiGri*?

The **RQ1** will be addressed in Chapter 8 with the presentation of *NixOS Compose*, a tool based on *NixOS* aiming at reducing the development time to create a reproducible distributed environment. Chapter 9 investigates **RQ2** by considering a technique we call *folding* of defining several computing resources on a single physical node. We will focus on the performance impact of the distributed file-system of the *folded* cluster. Chapter 10 presents the first step to answer **RQ3** by implementing a simplified version of *CiGri* in *Batsim*, and comparing to a real experiment the different behavior from the point-of-view of *Control-Theory*.

The following chapters aim to explore the different stages of experimentation while reducing the high experimental cost of distributed experiments on cluster or grid applications such as *CiGri*:

1. perform preliminary experiments in simulation to reduce exploration space at very low cost (**RQ3**),
2. deploy experiments at partial scale with lower cost and with acceptable loss of realism (**RQ2**),
3. while keeping a reproducible experimental environment (**RQ1**).

8 Reproducible Distributed Environments

This Chapter is based on [Gui+22b] published at CLUSTER 2022 with Jonathan Bleuzen, Millian Poquet, and Olivier Richard.

8.1 Introduction

In this chapter, we tackle **RQ1**. We aim to make the **entire** software stack involved in a experiment of distributed systems reproducible. This implies making the compilation and the deployment of this stack reproducible, allowing one to rerun the experiment on an identical environment in one week or ten years.

To reach this goal, we exploit the declarative approach for system configuration provided by the *NixOS* [DL08] Linux distribution. *NixOS* makes use of a configuration that describes the environment of the entire system, from user-space to the kernel. The distribution is itself based on the purely functional *Nix* package manager [DJV04]. The definition of packages (or system configuration in the case of *NixOS*) are functions without side effects, which enables *Nix* and *NixOS* to reproduce the exact same software when the same inputs are given.

We extend this notion of system configuration for distributed systems in a new tool named *NixOS Compose*. *NixOS Compose* enables to define distributed environments and to deploy them on various platforms that can either be physical (*e.g.,*on the *Grid'5000* testbed [Bal+13]) or virtualized (Docker or QEMU [Bel05]). *NixOS Compose* exposes the **exact same user interface** to define and deploy environments regardless of the targeted platform. We think that this functionality paired with fast rebuild time of environments improves user experience, and we hope that it will help the adoption of experimental practices that foster reproducibility.

Projects such as *NixOps* [Nix22], *deploy-rs* [ser23] or *Disnix(OS)* [Bur22a; Bur22b] enable to activate new *NixOS* configuration on target machines, but they are limited to machines that already run *NixOS*. These technologies only change the running

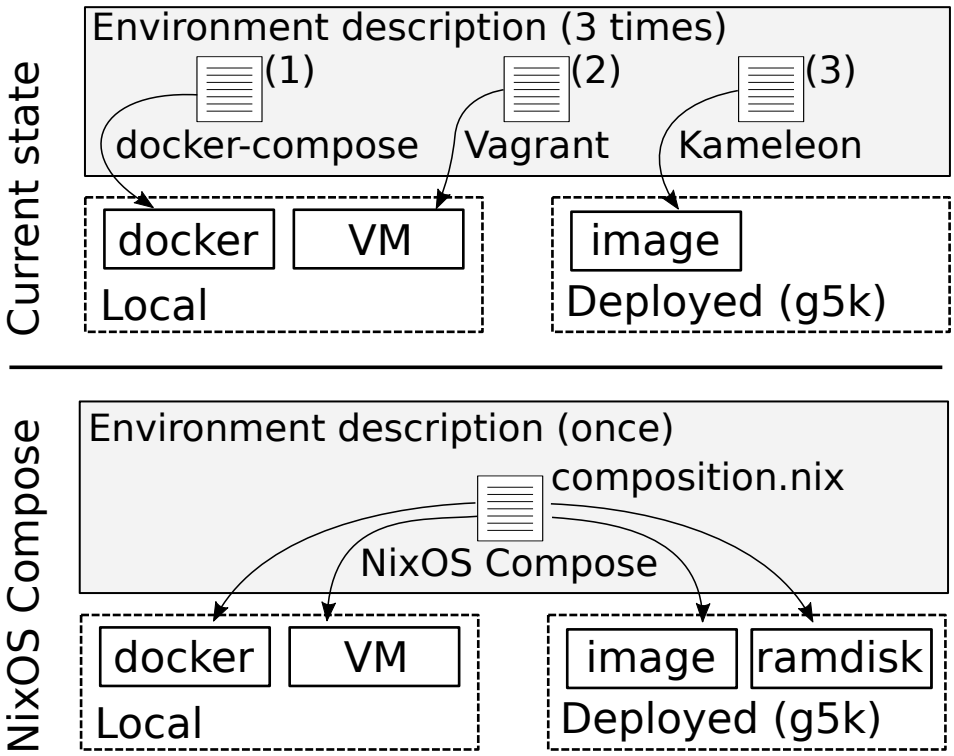


Figure 8.1.: Motivation of *NixOS Compose*. Currently, to produce a reproducible environment for each platform, users must maintain a description file for each target platform. We want *NixOS Compose* to only use a **single description** file (called a composition) that can build reproducible distributed environments and deploy them to **several platforms**.

configuration on the machines without rebooting them, which keeps a state on the machines and is detrimental for reproducibility.

Take away 8.1

State-of-the-art solutions to deploy software environements are either not focused on their reproducibility, or require long development cycles, which encourages bad reproducibility practices.

We believe that there is a necessity to propose a solution to **deploy reproducible environments** for a distributed system with **fast development cycles**. Figure 8.1 summarizes the motivation that has led us to create *NixOS Compose*. To generate reproducible environments with the current solutions, users most often need a configuration file for every platform they target. *NixOS Compose* aims at having a **single description** of the distributed environment that can be deployed to **several platforms**. *NixOS Compose* relies on *Nix* and *NixOS* and thus inherits their properties to make the environment **completely reproducible**. We fully embraced

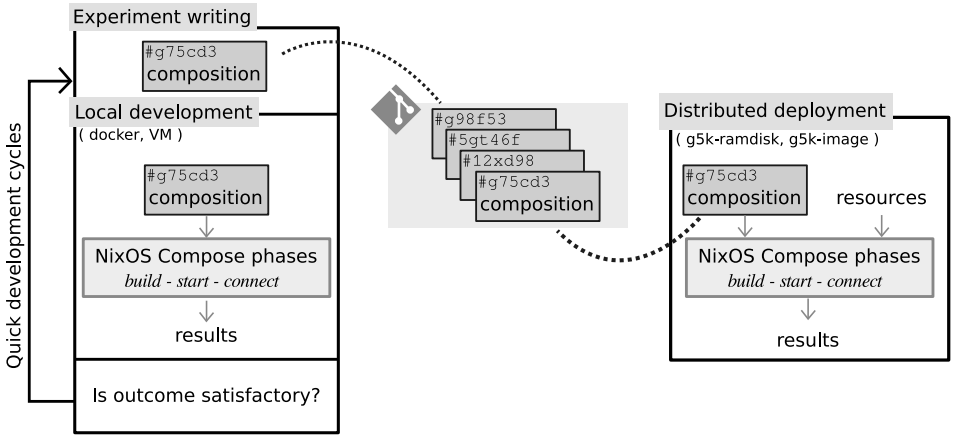


Figure 8.2.: Workflow of *NixOS Compose*. Local development of the environment is done using light and fast development with containers and virtual machines. Once the description of the environment (composition) has been tested, deployments on a distributed platform can be done with the **exact same interface**.

the *descriptive* approach for reproducibilityreasons and decided to do most of the configuration at the image build time. Some part of the configuration must however still be done at runtime via a provisioning phase, typically to synchronize services between different machines.

The chapter is structured as follows. Section 8.2 presents *NixOS Compose*, its main concepts, the external concepts it relies on, and the users' utilization workflow for setting up a complete reproducible system and software stack. Section 8.3 gives technical details on how *NixOS Compose* works. Section 8.4 presents how *NixOS Compose* can be used on a complex example that combines several distributed middlewares. Section 8.5 offers experimental performance results of *NixOS Compose* against standard state-of-the-art tools using multiple metrics. Finally, Section 8.6 concludes the section with final remarks and perspectives on reproducible works and environments.

8.2 Presentation of *NixOS Compose*

This section gives the main concepts and terminology of *NixOS Compose*. A **software environment** is a set of applications, libraries, configurations and services. We define the **notion of *Transposition*** as the capacity to deploy a *uniquely defined environment* on several platforms of different natures. For example one may want to deploy an environment on local virtual machines to test and develop, and then want to deploy onto real machines on a distributed platform with the *exact same*

description. As depicted on Figure 8.1, it enables users to have a **single definition** of their environment and to **deploy it to different platforms**. For the sake of clarity, *NixOS Compose* concepts will be illustrated on an example environment that contains *k3s* [K3s], a lightweight version of Kubernetes for the orchestration of containers.

```
{ pkgs, ... }:
let k3sToken = "..."; in {
  roles = {
    server = { pkgs, ... }: {
      environment.systemPackages = with pkgs; [
        k3s gzip
      ];
      networking.firewall.allowedTCPPorts = [
        6443
      ];
      services.k3s = {
        enable = true;
        role = "server";
        package = pkgs.k3s;
        extraFlags = "--agent-token ${k3sToken}";
      };
    };
    agent = { pkgs, ... }: {
      environment.systemPackages = with pkgs; [
        k3s gzip
      ];
      services.k3s = {
        enable = true;
        role = "agent";
        serverAddr = "https://server:6443";
        token = k3sToken;
      };
    };
  };
}
```

Listing 8.1: *NixOS Compose* composition file example for *k3s* [K3s].

Role A **role** is a type of configuration associated with the mission of a node. *k3s* is a client-server application where clients are named *agents*. There would therefore be 2 roles: one for the `server` and another for the `agents`. As all the agents have the same configuration they can use the same role. Note that in cases where there is one node per role, the notion of role and the notion of node overlap.

Composition A **composition** is a *Nix* expression describing the *NixOS* configuration of every role in the environment. Listing 8.1 shows an example of composition for *k3s*. The composition in Listing 8.1 defines the open port of the `server` (line

10-12), the available packages (lines 7-9 and 23-25, packages are `k3s` and `gzip`), as well as the `systemd` services (lines 13-19 and 26-31). The `k3s` service is not defined explicitly in this example, as we reuse an existing definition that is available in the collection of *Nix* expressions `nixpkgs`. For personal applications users might have to define their own `systemd` services, which can be done declaratively via a *Nix* expression. *Nix* variables can be used to avoid information duplication, as seen for the *k3s* token (used to manage authentication) defined on line 2 and used to configure both the `server` (line 18) and the `agents` (line 30).

Deployment A **deployment** assigns a role to every node. *NixOS Compose* proposes two ways to define deployment as YAML files, as illustrated on Listings 8.2 and 8.3. The first way is to define the number of nodes that should be used for each role (Listing 8.2). This is convenient when working on homogeneous nodes for simple deployments. For the sake of reproducibility, *NixOS Compose* generates a deterministic assignment – if the same nodes are reserved and the same deployment is used, the role of each node remains the same. The second way to define a deployment is to directly define the role that each node should take (Listing 8.3). This is more suited for complex scenarios, as it enables users to generate their deployment file depending on their needs.

```
# Users can define the number of nodes per role
server: 1
agent: 3
```

Listing 8.2: Deployment file example for the *k3s* example with 1 server and 3 agents where the users defined the quantity of nodes per role. The hostnames are generated by *NixOS Compose*. In this example there would be 3 agents with the hostnames `agent1`, `agent2` and `agent3`.

```
# Users can also define the hostnames per role
server: 1
agent:
    - agent1
    - agent2
    - agent3
```

Listing 8.3: Deployment file example for the *k3s* example with 1 server and 3 agents where the users defined the hostnames of every node per role.

Flavours A **flavour** is a target for the deployment of the environment. This notion includes the (virtual or physical) platform onto which the deployment should be

done, and also the *deployment method* that should be used (*e.g.,* full system image or ramdisk). As we write these lines *NixOS Compose* supports the following flavours:

- `docker` for `docker-compose` [Doc22] configurations.
- `vm-ramdisk` for in-memory QEMU virtual machines.
- `g5k-image` for full system tarball images that can be deployed on *Grid'5000* [Bal+13] via *Kadeploy* [Geo+06].
- `g5k-ramdisk` for `initrds` that can be quickly deployed in memory without the need to reboot the host machine on *Grid'5000*(via the `kexec` syscall).

During the development phase of the environment, users can deploy *locally, lightly and quickly* with the `docker` and `vm-ramdisk` flavours. At a later stage, users can test their environment on real nodes from the *Grid'5000* testbed with the `g5k-ramdisk`, which is convenient for trial-and-error operations thanks to its fast boot time. Finally, the environment can be deployed at real scale on *Grid'5000* with the `g5k-image` flavour. Please note that some flavours have reproducibilitylimitations due to the underlying technologies. For example, controlling the version of the Linux kernel is impossible when using the `docker` flavour.

Take away 8.2

NixOS Compose introduces the notions of *composition* which reprents the configurations of the different *roles* in the deployement. This *composition* can be *transposed* to different target systems, called *flavours*.

8.2.1 Workflow

This section presents the workflow of *NixOS Compose* and how it enables users to simply transpose their environment from one platform to another.

Local Testing When developing an environment, users can work with the `docker` and `vm-ramdisk` flavours with the following workflow:

1. Building the image: `nxc build -f docker` or `nxc build -f vm-ramdisk`
2. Deploy the environment: `nxc start`. By default, *NixOS Compose* takes the last composition built.

3. Connect to the nodes: `nxc connect [node name]`. This opens a connection to the desired node. If name is omitted, a terminal multiplexer[1] opens with one pane per node, which is convenient to run commands on the nodes.

Distributed Deployment Once the environment has been tested with local flavours, it can be tested in a distributed system.

1. Building the image: `nxc build -f g5k-ramdisk` or `nxc build -f g5k-image`
2. Reservation of the nodes to use for the deployment: depends on your platform resource manager. For example `salloc` for *Slurm* [YJG03] or `oarsub` for *OAR* [Cap+05].
3. Deploy the environment: `nxc start`.
4. Connect to the nodes: `nxc connect [node name]`.

Figure 8.2 summarizes the *NixOS Compose* workflow. *NixOS Compose* aims at making the transition between platforms as seamless as possible. Thus, the workflow in a distributed setting (Section 8.2.1) **is identical** to the workflow in a local one (Section 8.2.1). The only difference is that in a distributed setting, users need to first reserve the resources before deploying.

8.3 Technical Details

Nix can generate a *NixOS* configuration from a *Nix* expression, including the `boot` and `init` phases required to start the kernel. *Nix* stores those phases in the *Nix Store*, which enables *NixOS Compose* to call them later on. *NixOS Compose* works in two steps: *Building* and *Deploying*. The building phase is done using *Nix* tools wrapped with Python for the command-line interface. The deployment is fully done with Python and mostly consists in the interaction with the different deployment tools (*Kadeploy*, *docker-compose*, *QEMU*). The *Nix* part is around 2000 lines of code, and the Python part around 4000.

The following section (8.3.1) details how *NixOS Compose* manages `g5k-ramdisk`, the flavour that enables quick in-memory deployment without the need to reboot host machines on *Grid'5000*. Details are omitted for the other flavours, but please

[1] `tmux: https://github.com/tmux/tmux`

Table 8.1.: Table summarizing the different flavours with their building and deployment phases.

Flavour	Phase		Comments
	Building	Deployment	
`docker`	Generate a `docker-compose` configuration and `docker` containers.	Call the `docker-compose` application with the right arguments.	Fastest and light but limited in application due to virtualization.
`vm-ramdisk`	Generate the kernel and `initrd` for the roles of the composition.	Create a virtual network with Virtual Distributed Ethernet (VDE) and starts the Virtual Machines with QEMU.	Fast but takes a lot of memory. Limited to a couple of VMs on a laptop.
`g5k-ramdisk`	Generate the kernel and `initrd` for the roles of the composition.	Use `kexec` to quicky start the new kernel without rebooting. Send the deployment information through the kernel parameters.	Long to build but fast to deploy. `kexec` has reproducibility limitations and consumes a lot of memory which can be limiting for large images.
`g5k-image`	Generate a tarball of the image of the composition.	Use *Kadeploy* to deploy the image to the nodes. Send the deployment information through the kernel parameters.	Longer to build and deploy, but it has the best reproducibility properties.

refer to Table 8.1 for a summary of the difference in the building and deployment phases for all supported flavours.

8.3.1 Details on the `g5k-ramdisk` Flavour

Construction *NixOS Compose* uses *Nix* to evaluate the configuration of every role in the composition. *Nix* then generates the kernel and the `initrd` of the profiles.

Deployment *NixOS Compose* relies on the `kexec` Linux system call for this flavour. `kexec` enables to boot a new kernel from the currently running one. This skips the initialization of the hardware usually done by the BIOS, which avoids an entire reboot of the machines and greatly reduces the time to boot the new kernel. The `kexec` commands takes as input the desired kernel, the kernel parameters and the `initrd`. *NixOS Compose* passes the kernel and `initrd` generated in the construction phase to `kexec`.

As *NixOS Compose* produces a single image containing the profiles of all the roles, *NixOS Compose* needs at deployment time to tell each node the role it should take. To achieve this, we pass this information using the kernel parameters to set up environment variables based on the role. *NixOS Compose* also uses the kernel parameters to pass ssh keys and information about the other hosts (*e.g.,* the `/etc/hosts`). There is however a size limit of 4096 bytes on the kernel parameters, which prevents us to use this method to send the deployment information to nodes when users want to deploy a lot of nodes. To deal with this, *NixOS Compose* starts a light HTTP server on the cluster frontend for the duration of the deployment. We pass the URL of this server using the kernel parameters. Then, the nodes query this server with `wget` to retrieve the information associated with their roles. Note that the deployed images do not include a HTTP server but only the `wget` application to fetch the data. Figure 8.3 represents how the nodes get the deployment information based on the quantity of nodes involved.

8.4 A Complex Example: Melissa

This section shows how complex distributed environments can be developed with *NixOS Compose* by taking the *Melissa* [Ter+17] framework as an example.

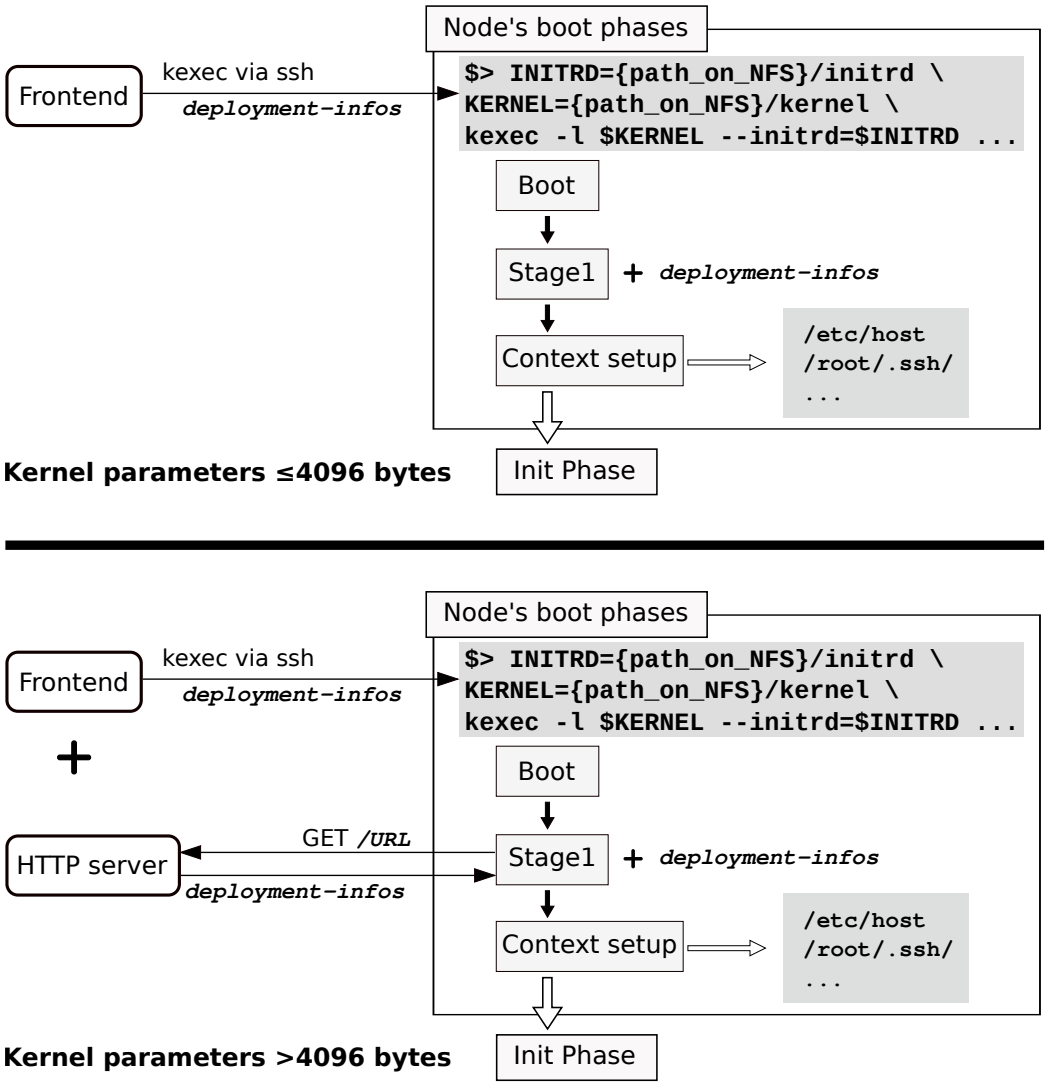


Figure 8.3.: Mechanism for the nodes to get the deployment information. For a few nodes, the information is passed via the kernel parameters. For a higher number of nodes, this is not possible due to the size limit on the kernel parameters (4096 bytes). In this case *NixOS Compose* starts a light HTTP server on the frontend and passes its URL to the nodes via the kernel parameters. The nodes then query this server to retrieve the deployment information.

8.4.1 Presentation of *Melissa*

Melissa is a framework to run large-scale sensitivity analyses. Its main specificity is the online processing of data to limit usage of intermediate file storage, contrary to postmortem approaches. It can be used in several HPC environments as it is compatible with two resource managers (RM): Slurm [YJG03] and OAR [Cap+05]. *Melissa* implements a client/server model where the clients are simulations that generate and send data to a server that runs the statistics algorithms.

NixOS Compose enables to deploy a resource manager (including all the components it requires, *e.g.,* a database), *Melissa*, and all the components required by *Melissa* at runtime (*e.g.,* a distributed file system).

In the following example we deploy *Melissa* with the Slurm resource manager. Four roles are needed to define the environment.

- `server`: RM server and file system server
- `dbd`: MariaDB database (accounting for the RM)
- `computeNode`: worker node
- `frontend`: node from where initial jobs are submitted

```
server = { pkgs, ... }: {
  imports = [ slurmconfig nfsConfigs.server ];
  services.slurm.server.enable = true;
  systemd.services.slurmctld.serviceConfig = {
    Restart = "on-failure";
    RestartSec = 3;
  };
};
computeNode = { pkgs, ... }: {
  imports = [ slurmconfig nfsConfigs.client ];
  environment.systemPackages = [
    melissa melissa-heat-pde
  ];
  services.slurm.client.enable = true;
  systemd.services.slurmd.serviceConfig = {
    Restart = "on-failure";
    RestartSec = 3;
  };
};
```

Listing 8.4: Configuration of the `server` and `computeNode` roles.

Some roles share parts of their configuration, like `server` and `computeNode`. They both use Slurm but their configuration differs in terms of services, as they respectively enable the `slurm.server` and `slurm.client` services.

Melissa itself also needs to be part of the environment. Unlike the *k3s* example shown on Listing 8.1, *Melissa* is not available in *nixpkgs* and thus needs to be packaged. Listing 8.5 is a snippet of *Melissa*'s package definition that notably defines which source code should be used (lines 5-9) and which build dependencies should be used (lines 10-13). The build commands (line 14) are omitted for the sake of readability.

```
1  { pkgs, ... }:
   pkgs.stdenv.mkDerivation rec {
3    pname = "melissa-${version}";
     version = "0.7.1";
5    src = pkgs.fetchgit {
       url="https://gitlab.inria.fr/melissa/melissa";
7      rev="e6d09...";
       sha256="sha256-IiJad...";
9    };
     buildInputs = with pkgs; [
11     cmake gfortran python3 openmpi
       zeromq pkg-config libsodium
13   ];
     # Build phases are omitted
15 }
```

Listing 8.5: Snippet of the package definition for *Melissa*

Similarly, the `melissa-heat-pde` simulation application must also be in the environment (line 12 of Listing 8.4). Finally, as a distributed filesystem is necessary in this environment, our composition imports a NFS module for the roles that need it.

8.4.2 Key difficulties

This section emphasizes the advantages of using *NixOS Compose* to deploy the *Melissa* distributed environment.

NFS Server Setting up a NFS server with tools like *Kameleon* or *EnOSlib* is cumbersome. The users would first need to install the `nfs` tools on the nodes and define the NFS server. Then, the users would have to mount the newly defined server on every client node. These steps can be automated with scripts, but that would be fragile.

```
# Configuration of the NFS server
nfsServer = {
  services.nfs.server.enable = true;
  services.nfs.server.exports =
    "/srv/shared *(rw,no_subtree_check,fsid=0,no_root_squash)";
  services.nfs.server.createMountPoints = true;
};
```

Listing 8.6: Definition of the NFS server. It exposes the `/srv/shared` folder.

```
# Mouting of the NFS server
nfsClient = {
  fileSystems = {
    "/data" = {
      device = "server:/";
      fsType = "nfs";
    };
  };
};
```

Listing 8.7: Mounting of the NFS server for the compute nodes. The local mounting point is `/data`.

The declarative definition of NFS with *NixOS* is based on `systemd` services (see Listings 8.6 and 8.7). This makes them easier to define and more robust as they can be restarted until they perform the mount successfully in the case of the NFS server starting after the clients.

Resource Manager *Melissa* runs on production clusters managed by resource managers such as Slurm and OAR. However, experimenting on *Melissa*'s behavior in different scenarios requires controlling the resource manager part of the environment. The installation of such systems is far from trivial as they include several distributed services that must interact, and each one of these services require configuration. A composition can be made modular so that users can descriptively change the resource manager. The same benefit is achievable for comparison studies of versions of the *Melissa* framework, as the *Nix* function that defines *Melissa* can be written in such a way that it takes *Melissa*'s source code as a function input.

We have deployed the *Melissa* composition we have written with *NixOS Compose*on 13 nodes with the experimental setup described in Section 8.5.1. The deployment took approximately 2 minutes with the `g5k-ramdisk` flavour.

8.4.3 *Melissa* Images Content Comparison

NixOS Compose aims to provide the same environment on different target platforms. This section analyses the content of the *Nix Store* in the *Melissa* images generated for every flavour, as the *Nix Store* content represents the software environment available on the node. The `docker` flavour is omitted here as the containers do not have a well-defined specific *Nix Store* but mount the *Nix Store* of the host machine instead.

Figure 8.4 presents the content of the *Nix Store* of the *Melissa* image for the different flavours. The smaller packages are gathered under the `others-*` name. We can see that the vast majority of the software stack is shared by several flavours. There is a common 2 GiB set of packages common to every flavour containing the *NixOS* definition and the dependencies of *Melissa*. The two flavours targeting the *Grid'5000* platform need more packages, for example the firmware to use the nodes' hardware. Then for each of the flavour, there is about 5 % of the total *Nix Store* size for packages specific to the flavour. For example, deploying a `g5k-image` image requires a complete reboot of the host and to go through the boot loader, hence the presence of `grub` in this image.

Take away 8.3

The resulting environments for the different flavours differ because of the particularities of the target platform.

8.5 Evaluation

NixOS Compose brings reproducibility guarantees to distributed environments contrary to state-of-the-art solutions. The overall goal of this evaluation section is to determine whether this is done with a significant overhead or not.

8.5.1 Experimental Setup

The following experiments have been carried out on the `dahu` cluster of the *Grid'5000* testbed. This cluster has machines with 2 Intel Xeon Gold 6130 CPUs with 16 cores per CPU and 192 Gib of memory. The nodes of this cluster have SSD SATA Samsung MZ7KM240HMHQ0D3 disks with a capacity of 240 GB formatted in ext4.

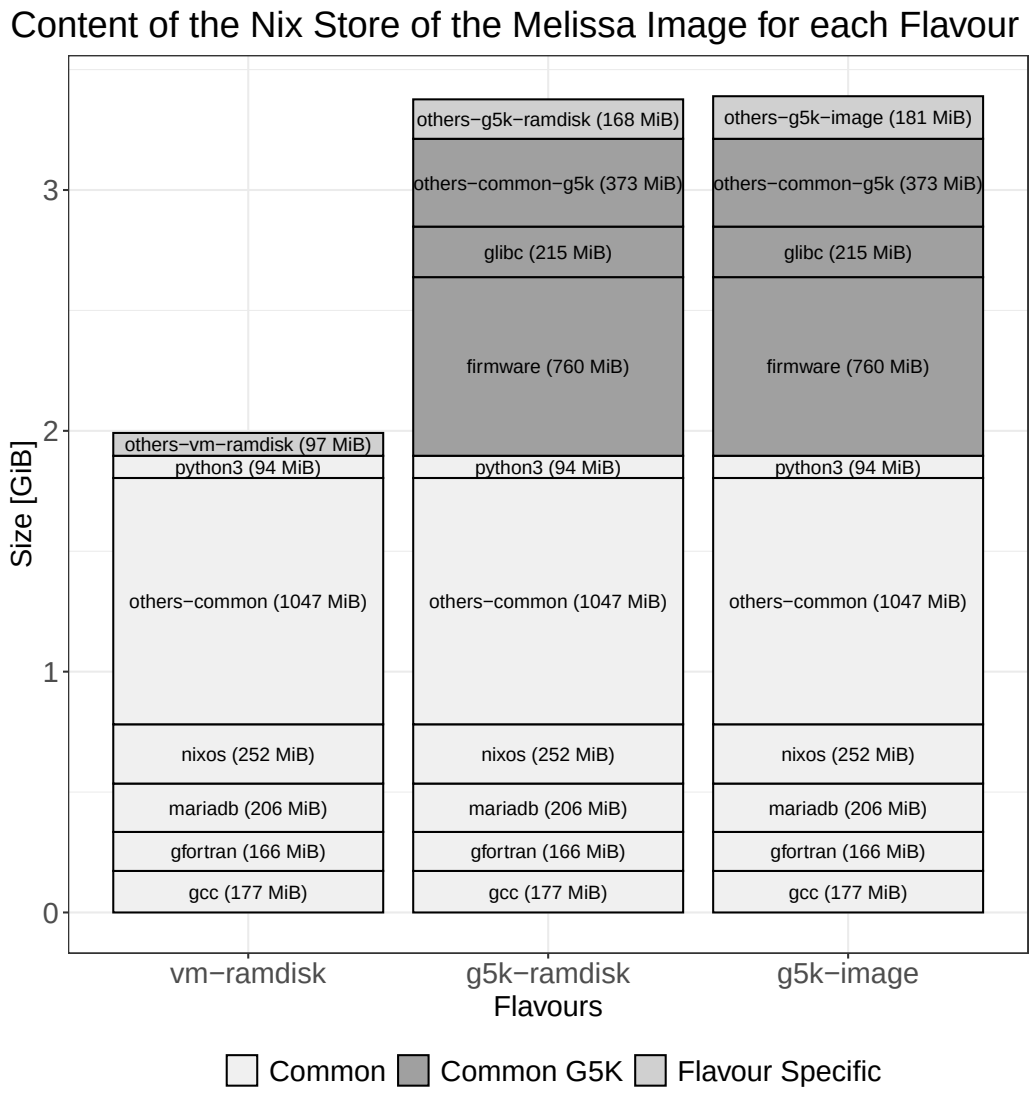


Figure 8.4.: Packages present in the *Nix Store* of the *Melissa* image for the different flavours. The colors represent the packages common to the flavours. The smaller packages are gathered under the `others-*` name. The `docker` flavour is omitted as it mounts the *Nix Store* of the host machine.

The experiments conducted in this article are repeatable with variation. Data and analysis scripts are available on Zenodo[2] with the link to the experiments' repository.

8.5.2 Comparison to *Kameleon*

Grid'5000 provides base images for several Linux distributions and versions. Users need to build their own images if they want to use more complete images. This is usually done with *Kameleon* on *Grid'5000* — in fact all the images provided by *Grid'5000* are generated by *Kameleon* recipes.

In this study, we want to compare the performance of *NixOS Compose* and *Kameleon* to build images. We will focus on the image build time, as well as the size of the generated images. We also want to evaluate whether caching the *Nix Store* enables an interesting build time speedup.

Protocol The following steps are executed in this order.

1. Construction. Build an image from a recipe.
2. Modification. Change the recipe slightly.
3. Reconstruction. Build an image from the new recipe.

We first build a `base` image with *NixOS Compose* and *Kameleon*, measuring its build time and the size of the generated images. `base` contains the basic software needed to conduct a distributed experiment: `grid5000/debian11-x64-nfs` for *Kameleon* as this is the most convenient and common image for distributed experiments on *Grid'5000*, and all the packages required by the flavour for *NixOS Compose*. Then we add the `hello` package to the recipes and build a new image (`base + hello`) while measuring the same metrics.

This experiment has been executed using *Grid'5000*'s NFS (mounted on `/home`) or without it (using local disks on machines mounted on `/tmp`), in order to compare the performance of the tools depending on the filesystem setup used.

We clear the *Nix Store* before building the `base` image, but not before building `base + hello`, in order to evaluate the impact of cached builds on *NixOS Compose*. *Kameleon* has an indirect caching mechanism via the HTTP proxy Polipo [Chr22].

[2] `https://zenodo.org/record/6568218`

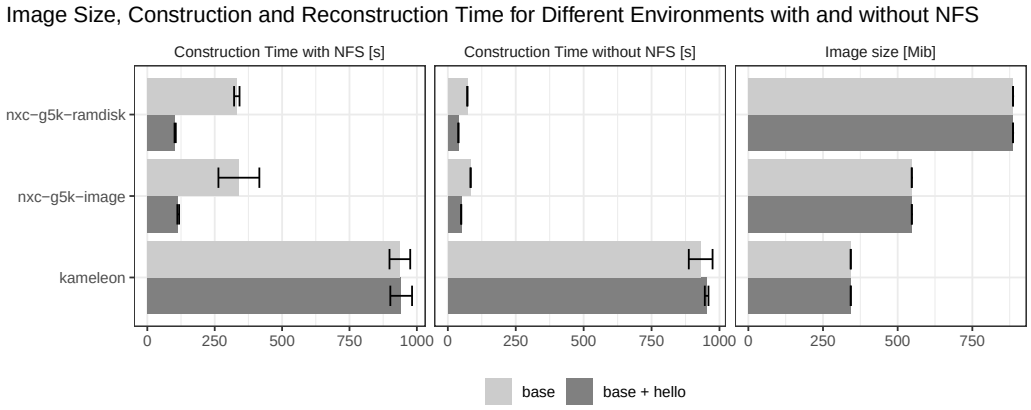


Figure 8.5.: Performance comparison between *Kameleon* and *NixOS Compose* (`nxc`). A `base` image is first built, then a new image (`base + hello`) that contains the additional `hello` package is built. As *Grid'5000* is the targeted platform of this experiment, images are built with the `g5k-ramdisk` and `g5k-image` flavours. Shown values are averages over 5 repetitions. Error bars represent 99 % confidence intervals.

However, we did not manage to make it work with *Kameleon*, and Polipo is no longer maintained as its utility has become arguable – most of today's traffic is encrypted, including fetching packages from mirrors.

Results and Comments As seen on Figure 8.5, *NixOS Compose* substantially outperforms *Kameleon* in terms of image build time. When building from an empty cache on local disks, *NixOS Compose* is 11x faster than *Kameleon*. Moreover, *NixOS Compose* uses its local cache efficiently, which enables it to build the image variation 1.7x faster than the initial image build time when the filesystem is used efficiently (local disks).

Figure 8.5 also shows that *NixOS Compose* produces bigger images than *Kameleon*. This is mainly because we have not optimized the content size of the images as we write these lines — *e.g.,* many firmwares are kept in the images instead of only the ones needed on *Grid'5000* (see Figure 8.4). Another reason for this image size comes from our design choice to prioritize compression speed over compression quality. This is important for *NixOS Compose* as image variations should be built as fast as possible to improve user experience. For information, we have measured that *NixOS Compose* takes about 25 s to compress each image, which is a non-negligible portion of a variation build time (30 - 35 %).

Finally, Figure 8.5 shows how the filesystem setup impacts the build time. Here, using an efficient filesystem setup (local disks) greatly benefits to *NixOS Compose* as

it makes it 4x faster. This is caused by the many smalls writes done by *Nix* in the *Nix Store*. Filesystem setup has little impact on *Kameleon* as it uses local disks by default to generate the whole image, that is later on copied to the NFS.

Take away 8.4

NixOS Compose builds system images faster than *Kameleon*, but really shines when introducing a variation in the environment. *Nix* is victim of the NFS slowness, and build times can be even more improved when not building on NFS.

8.5.3 Comparison to *EnOSlib*

EnOSlib is a state-of-the-art solution to conduct distributed experiments. *EnOSlib* does not claim to be reproducible, but it inherits the reproducibility of its underlying components. In this section, our goal is to compare the performance of *NixOS Compose* and *EnOSlib* to set up *fully reproducible* distributed environments. In particular, we want to know how much time is taken for each phase of a deployment.

Case Studies We chose to study the two following distributed applications that are already implemented in *EnOSlib*:

- *k3s* [K3s] is a lightweight version of Kubernetes.
- *flent* [toh22] is a network benchmarking tool.

Our *k3s* environment consists in two nodes: one *k3s* server and one *k3s* agent. The agent deploys a `nginx` web server to the agents. The *run* part of the experiment simply consists in querying the webserver to retrieve the `nginx` web page.

Our *flent* environment consists in 2 nodes: one server and one client. The *run* part of the experiment simply runs the *flent* benchmark.

For both case studies, we made sure that *NixOS Compose* and *EnOSlib* set up similar environments, and that they execute the same *run* part of the experiment after the deployment and provisioning phases have been done.

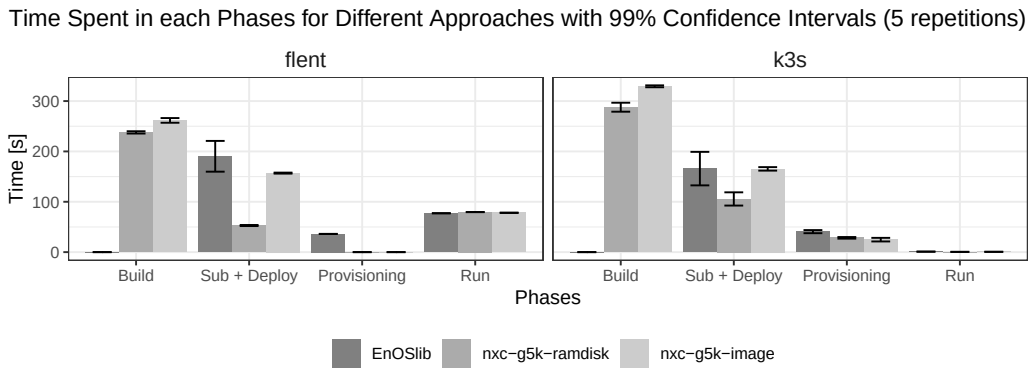


Figure 8.6.: Time spent in the different phases of the deployment of a distributed experiment (build, submission + deploy, provisioning, run). We compare *EnOSlib* and *NixOS Compose* (with the flavours `g5k-ramdisk` and `g5k-image`) on two examples: a network benchmarking tool (*flent*) and a containers' orchestrator (*k3s*). The errors bars represent the confidence intervals at 99 %.

Protocol *EnOSlib*'s approach is mostly based on the configuring/provisioning phase, which executes commands to set up a desired environment (software, services...) on a machine that is already available. As this limits reproducibility on the kernel side, we decided to deploy traceable images on the nodes. We used the `grid5000/debian11-x64-nfs` image that is pre-built by *Grid'5000*. *EnOSlib* deploys the image with *Kadeploy*, then executes the provisioning phase, and then execute the *run* part of the experiment once the environment is ready.

In *NixOS Compose* most of the configuration is done inside the composition and thus in the image. This enables us to completely skip the provisioning phase for *flent*. For *k3s*, our provisioning phase simply consists in waiting that the web server becomes available.

For both tools we measure the time to build the image, to deploy it, to execute the provisioning and to run the experiment script. We use an empty *Nix Store* for a fair build time.

Results and Comments As seen on Figure 8.6, *NixOS Compose*'s `g5k-ramdisk` flavour is faster to deploy than a full image as it uses `kexec` and does not require a full reboot. As expected, we also can see that the solutions with *NixOS Compose* have a smaller provisioning time than with *EnOSlib*. This is because *NixOS Compose* includes this as part of this provisioning in the build and deployment phase.

Please note that *EnOSlib* does not directly provide the time taken by the deployment phase, but only provides the time between the submission and the first command of

the provisioning. That is why the submission and deployment time (Sub + Deploy) are shown together on Figure 8.6 both for *EnOSlib* and *NixOS Compose*, for the sake of fairness.

From Figure 8.6, it seems that packing part of the provisioning in the image improves the provisioning time without deteriorating the deployment time. The only drawbacks are the non-negligible build times. However, those times can be improved by utilizing the *Nix Store* as a local cache (see Section 8.5.2). Note that *NixOS Compose* proposes several flavours that can be executed locally to develop and test the environment. The cost of the construction of these images is thus amortized by the numerous quick and light local deployments. Finally, please also note that the build time of *EnOSlib* is null on Figure 8.6 as a pre-built image is used. However, depending on their scenario users may need to actually build an image via another technology (*e.g.,Kameleon* or *NixOS Compose*).

Take away 8.5

Using *NixOS Compose* instead of *EnOSlib* can reduce the time spent after deployment to configure services (*provisioning*) to the build time thanks to the declarative nature of *NixOS* services.

8.6 Conclusion

This Chapter has presented *NixOS Compose*, a free tool, under MIT license [Gui+22a], that enables the generation of reproducible distributed environments. We have showed that *NixOS Compose* deploys the exact same software stack on various platforms of different natures, without requiring specific work from users. The software stack is reconstructible by design, as *NixOS Compose* inherits its reproducibility properties from *Nix* and *NixOS*. Our experiments showed that *NixOS Compose*'s reproducibility and platform versatility properties are achieved without deployment performance overhead in comparison to the existing solutions *Kameleon* and *EnOSlib*.

NixOS Compose enables to build and deploy reproducible distributed environments. This is crucial for conducting reproducible distributed experiments, but this is only a part of the bigger picture. We plan to explore how *NixOS Compose* can be coupled to other tools that solve other parts of this problem. *EnOSlib* is for example well-suited to control the dynamic part of complex distributed experiments but lacks

reproducibility properties, which makes us think that a well-designed coupling may be beneficial for practitioners.

The experiments conducted in this article showed that build caches greatly improves *NixOS Compose*'s build times, and that properly using the file system is important for its performance. From a cluster administration perspective, providing a shared *Nix Store* between users would be very interesting to avoid data duplication and to prevent different *NixOS Compose* users to build the same packages over and over. There are many ways to implement a distributed shared *Nix Store* and we think that exploring their trade-offs would provide valuable insights, as reproducibility improvements should not be done at the cost of a higher resource waste on clusters.

User experience is a crucial factor that must be considered for reproducible experimental practices becoming the standard. With this in mind, we think that the notion of *Transposition* we have defined in this article and implemented in *NixOS Compose* is very beneficial. *Transposition* reduces the development time of distributed environments, as it enables users to do most of the trial-and-error parts of this iterative process with fast cycles, without any reproducibility penalty on real-scale deployments. However, practitioners that adopt *NixOS Compose* are likely to experience a paradigm shift if they are not already accustomed to *Nix*'s approach. We strongly believe that the reproducibility and serenity gains it brings are worth it. To help with the adoption of *NixOS Compose* a tutorial has been created and presented at two occasions [Gui+a].

NixOS Compose currently only provides first-class support for *Grid'5000*. We would like to support bare-metal and virtualized deployments on other experimental testbeds such as CloudLab [Dup+19] and Chameleon [Kea+20]. Moreover, the hand off from *Grid'5000* to SLICES(-FR) might require to also support deployments technologies such as OpenStack [Ope13].

9 Reducing the Cost of Experimenting with Distributed File-System

This chapter is based on [Gui+23d] presented at the 15th JLESC workshop with Olivier Richard, Raphaël Bleuse, and Eric Rutten.

9.1 Introduction

The increase of computing demands from scientists from all fields led to the development of computing cluster and grid architectures. And with these new architectures came new challenges. Due to their high prices, clusters are often shared among several research laboratories and teams. This sharing motivates the development of middlewares such as batch schedulers that are responsible to assign the user jobs to physical machines, manage the state of the resources, deal with reservation, etc. Such cluster, or grid, middlewares are complex applications of great research interest, and they must be tested before reaching production. However, they are usually destined to operate in an environment of hundreds or thousands of machines. Deploying a full scale environment is too costly to perform simple tests and very specific evaluations.

We thus want to answer the following question: **Can we reduce the number of physical machines needed to perform a full scale experiment while keeping a similar behavior as the full scale system?**

One solution to reduce the number of machines used for experiments would be to use simulation techniques. The system, the applications, and the middlewares are modeled and can then be executed on a single node. Beside reducing the number of machines used, simulators also reduce the execution time of the experiments. In the context of distributed systems and applications, projects such as Simgrid [Cas+14] and Batsim [Dut+16] are leading the way. One drawback of simulation is that the real middleware is not being executed, or not fully executed, but instead, a partial

or modeled version is executed in a modeled environment. In the case of cluster and grid middlewares, the applications are often far too complex to model them fully correctly.

Another approach is to *fold* the experiment by deploying more "virtual" resources on physical machines. In the case of a Resources and Jobs Management Systems (RJMS), like Slurm [YJG03] or OAR [Cap+05], it is possible to define several resources, from the point of view of the RJMS, on a single machine, which would be completely transparent for the users. For example, one can deploy a 1000-node virtual cluster on 10 physical nodes by defining 100 virtual resources on each node. The advantage is that the experiment takes place on the real system (CPU, network, disk, etc.) and with the real middleware code. The drawback is that this folding can introduce noise in the experiment and degrade performance.

In this chapter, we evaluate the performance of a distributed I/O benchmark when we fold a computing cluster onto itself, and quantify the impact of folding a computing cluster containing a file system. We then give a *rule of thumb* for choosing the amount of folding for distributed experiences containing a distributed file-system.

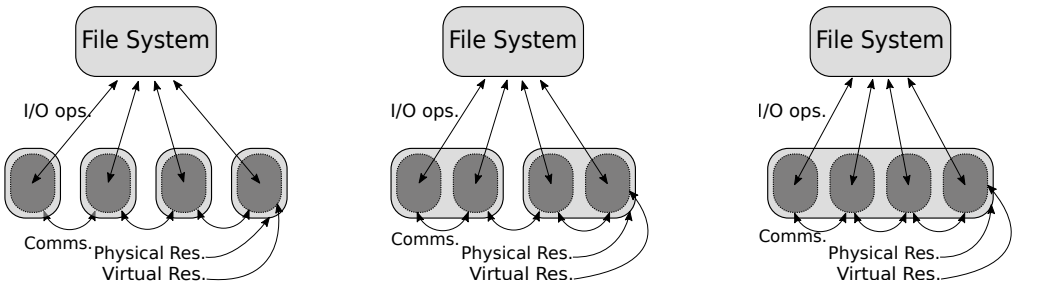
Section 9.2 defines notions and concepts. We present the experimental protocol in Section 9.3 and perform the evaluation in Section 9.4.

This problematic arises in the context of experiments with *CiGri* where we must deploy and perform experimentation evaluation of a modified version of *CiGri* on a realistic environment. It is unreasonable to deploy on the *entire Gricad* meso-center, or using simulation due to the complex software stack (*CiGri* + *OAR* + users jobs). We are thus interested in folding strategies to perform the evaluation of our *CiGri* modifications.

The statistical description of the execution times of the *CiGri* jobs presented in Chapter 2 allows us to use a `sleep` model to represent the execution times. As `sleep` calls are extremely lightweight for a CPU to deal with, we are able to fold several virtual resources onto one physical resource. However, no realistic job only does computation without reading and/or writing data. We can extend the previous job model by adding I/O operations before or after the `sleep` with the `dd` command. This addition makes it less obvious how much we can afford to fold.

The questions that we want to answer are thus the following:

1. **What is the minimum number of machines we need to deploy to emulate a full scale cluster with this job model while keeping the same performance in I/O?**



(a) Folding with a factor of 1 (4 physical resources and 4 virtual resources).

(b) Folding with a factor of 2 (2 physical resources and 4 virtual resources).

(c) Folding with a factor of 4 (1 physical resource and 4 virtual resources).

Figure 9.1.: Example of folding a deployment for a system with 4 resources. Figure 9.1a depicts the system deployed at full scale. Figure 9.1c represent the system completely folded. And Figure 9.1b shows an intermediate folded deployment.

2. **Is there a trade-off between the number of physical resources used and the overhead of performances due to the folding?**

9.2 Definitions & Concepts

We define the **folding** of a deployment as the action of defining several "virtual resources" on a physical resource. A physical resource is a node from a cluster, and a virtual resource represents a resource on the full scale system from the point of view of the RJMS.

We also define the **folding factor** (f_{fold}) as the division of the number of virtual resources in the deployment divided by the number of physical resources. Intuitively, it represents the number of virtual resources for each physical resource:

$$f_{fold} = \frac{\#resource_{virtual}}{\#resource_{physical}} \in [1, +\infty[\tag{9.1}$$

Figure 9.1 depicts an example of folding a deployment.

9.3 Methodology

We aim at evaluating the variations in performance of a distributed application using a distributed file-system for different value of folding factors (f_{fold}). As explained previously, the job model that we are using is a `sleep` time to represent the CPU

bound phase, and a `dd` operation to represent the I/O phase. The sleep operation allows us to fold the CPU bound phase on the same machine easily without noise. We thus focus on the performance of the I/O operations in a folded deployment.

9.3.1 Experimental Setup

The following experiments were carried on the `gros` cluster, located in Nancy, of the *Grid'5000* [Bal+13] French test bed. The machines of this cluster have an Intel Xeon Gold 5220 CPU with 18 cores, 96 GiB of memory, a 2 x 25 Gbps (SR-IOV) network and a 480 GB SSD SATA Micron MTFDDAK480TDN disk. The reproducibility of the deployed environment was ensured by *NixOS Compose* [Gui+22b]. The environment definitions are available at [hpc23a; hpc23b], the analysis scripts at [Gui+23b], and the data on Zenodo [Gui+23c].

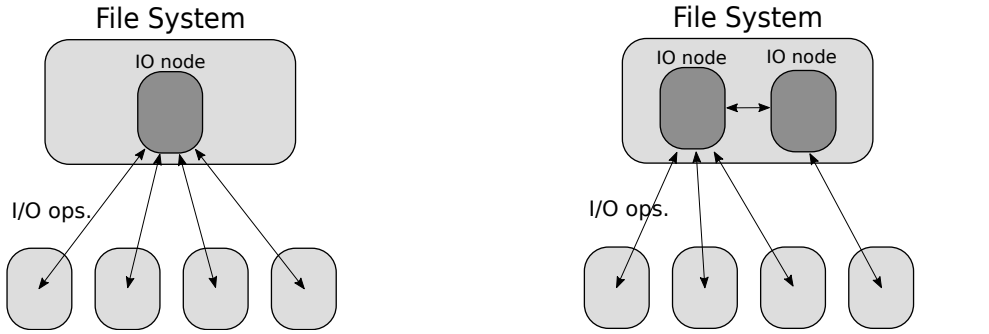
9.3.2 Benchmark application

To evaluate the performance of the cluster file system, we chose to use the IOR [Cal23] benchmark. IOR is a MPI application to benchmark I/O performances. It is the most popular benchmark among research on I/O in HPC [Boi+18], and is also used in the context of the IO500 list [IO5].

IOR has a multitude of parameters, but give the main ones here. We used the POSIX protocol (`api=POSIX`), a transfer size (`transferSize`) of 1Mbytes and a segment count of 1 (`segmentCount`). We assigned a single file per IOR process (`filePerProc`), and checked the correct writing and reading afterwards (`checkWrite` and `checkRead`). The number of tasks (`numTasks`), which represent the number of IOR processes, and the block size (`blockSize`), which is the total size of the file to read/write in this case, are parameters of the following experiments. We used OpenMPI with the TCP backend.

9.3.3 Distributed File-Systems

For the chosen distributed file-systems, we used NFS and OrangeFS.



(a) Distributed File system like NFS. Multiple clients query a single I/O node that process all the requests.

(b) Parallel File system like OrangeFS. Multiple clients query in parallel all the I/O nodes.

Figure 9.2.: Architectures for a distributed file system (Figure 9.2a), and parallel file system (Figure 9.2b).

NFS (v4) NFS [Paw+00] is a popular **distributed** file-system for small clusters. There is only one server. Clients mount the file-system and can perform POSIX operations. Figure 9.2a depicts the simplified architecture of a distributed file system like NFS. All the clients query the same I/O node for their files. NFS servers do have several workers that can manage the requests concurrently. The NFS export options used are: `*(rw,no_subtree_check,fsid=0,no_root_squash)`. The NFS server runs under the default configuration (8 workers).

OrangeFS OrangeFS [Bon+11] (or PVFS2) is a **parallel** file-system. This means that there are several servers (also called I/O nodes) to manage the requests of the clients. Figure 9.2b depicts the simplified architecture of a parallel file-system like OrangeFS. The clients query a I/O node of the file system. If one stripe asked by the client is not present on the query I/O nodes, the file system indicates on which I/O node to find it. We used the default configuration recommended by the OrangeFS installer (the I/O nodes host both the metadata and the storage).

Number of I/O nodes In the case of PFS, like OrangeFS, we did not find any methodology nor "rule of thumb" to define the number of I/O nodes for a computing cluster. In the following, we will consider OrangeFS file-systems with 1, 2, or 4 I/O nodes. Note that in the case of NFS there is only one I/O node.

I/O load We consider 5 different sizes of I/O operations to perform, both in writing and reading: 1Mbytes, 10Mbytes, 100Mbytes, 500Mbytes, and 1Gbytes. These file sizes will be the values of the IOR `blockSize` option.

Number of CPU nodes As a first step, and because we aim at emulating clusters from regional meso-centers, we will thus consider small clusters of 8, 16, 24, and 32 nodes. These number of nodes will be the values of the IOR `numTasks` option: one tasks per node of the cluster.

Protocol Let us consider a system with N CPU nodes. We first deploy the system at full scale, *i.e.,* N machines and the I/O nodes of the file system. As IOR is an MPI application, we compute the `hostfile` with one `slots` per compute node. We then start the IOR benchmark, that we repeat 5 times (the IOR `repetitions` option), and gather the performance reports. We remove one compute node from the `hostfile` and recompute the `slots` for the remaining nodes to keep the number of processes (`numTasks`) constant. This protocol is then repeated for all the different variations: number of CPU, size of I/O operations, type of file system, number of I/O nodes.

Figure 9.3 shows a visual representation of the experimental protocol for an experiment with 4 CPU nodes.

9.4 Evaluation

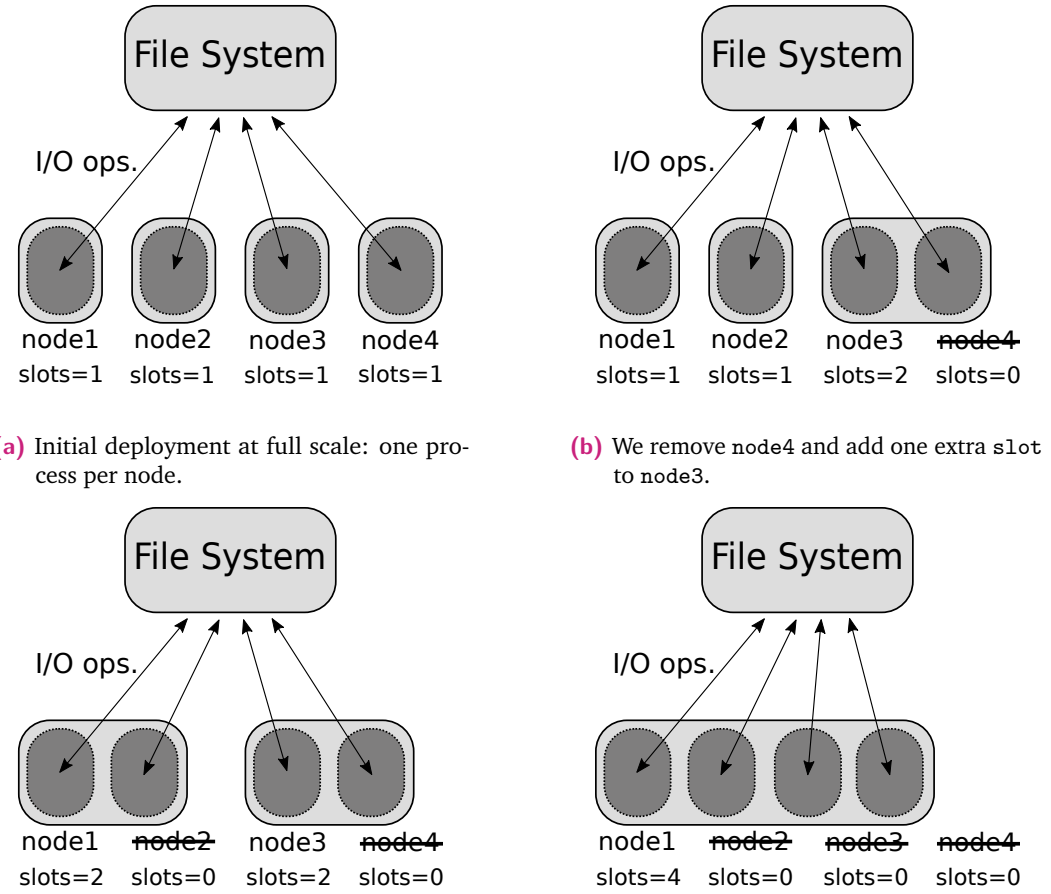
In this Section, we present the results of the experiments presented in the previous section. We first consider the file-system of the cluster to be NFS in Section 9.4.1, and then OrangeFS in Section 9.4.2.

9.4.1 Evaluation of NFS

Figure 9.4 shows the results of the experiments when using NFS. We also plot the 95% confidence intervals. Note that we **did not** remove the outliers. We notice that the writing performances (bottom row) does not seem to be affected by the folding of the deployment as it remains flat. However, for the reading performances (top row), we can see that the reading times increasing for higher folding factor. This means that the more we fold, the more we degrade the reading performances.

Take away 9.1

Write operations on NFS do not seem to be affected by the folding of the deployment. On the other hand, read operation performances degrade the more the cluster is folded with a quadratic behavior.



(a) Initial deployment at full scale: one process per node.

(b) We remove `node4` and add one extra `slot` to `node3`.

(c) We remove `node2` and add one extra `slot` to `node1`.

(d) We remove `node3` and reach a fully folded deployment.

Figure 9.3.: Graphical representation of the experimental protocol presented in Section 9.3. We start with a full scale deployment, *i.e.,* one MPI process (viewed as a virtual resource) per physical node (Figure 9.3a), and remove from the `hostfile` the physical nodes one by one while keeping the number of MPI processes (*i.e.,* the number of virtual resources) constant. We recompute the number of `slots` per node to balance the processes (Figures 9.3b and 9.3c). The experiment stops when there is no more node to remove (*i.e.,* after Figure 9.3d).

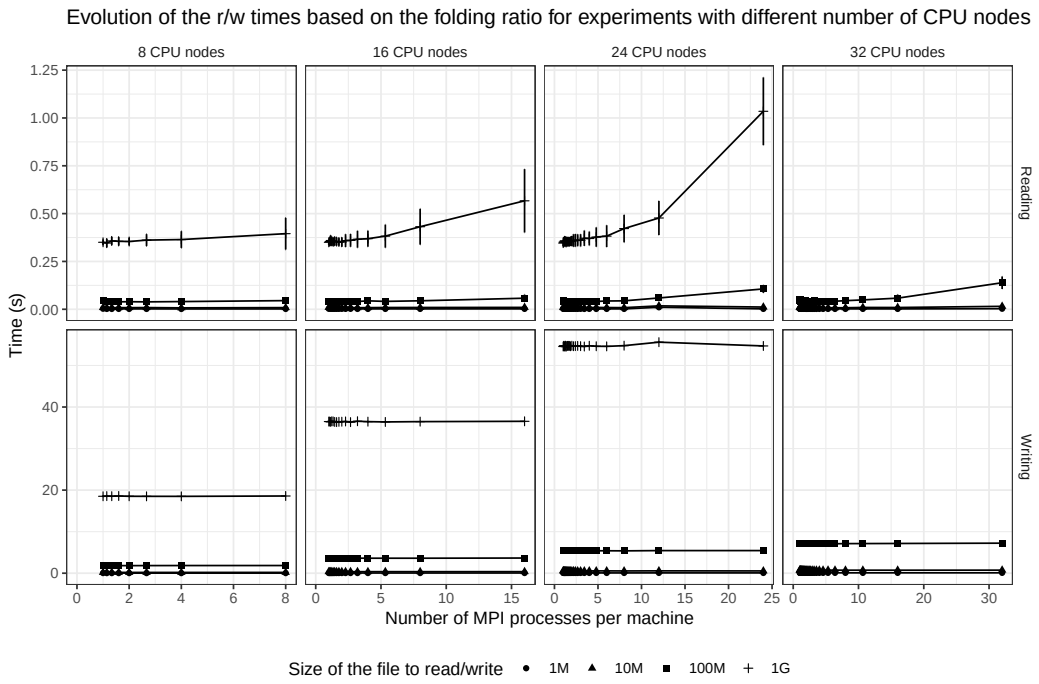


Figure 9.4.: Evolution of the reading (top row) and writing (bottom row) times based on the folding factor (x-axis) for experiments with different cluster size (*i.e.,* number of CPU nodes) and different sizes of file to read and write (point shape). We observe that the writing performances are not affected by the folding, but that the reading ones are, and that the degradation has quadratic growth with respect to the folding factor.

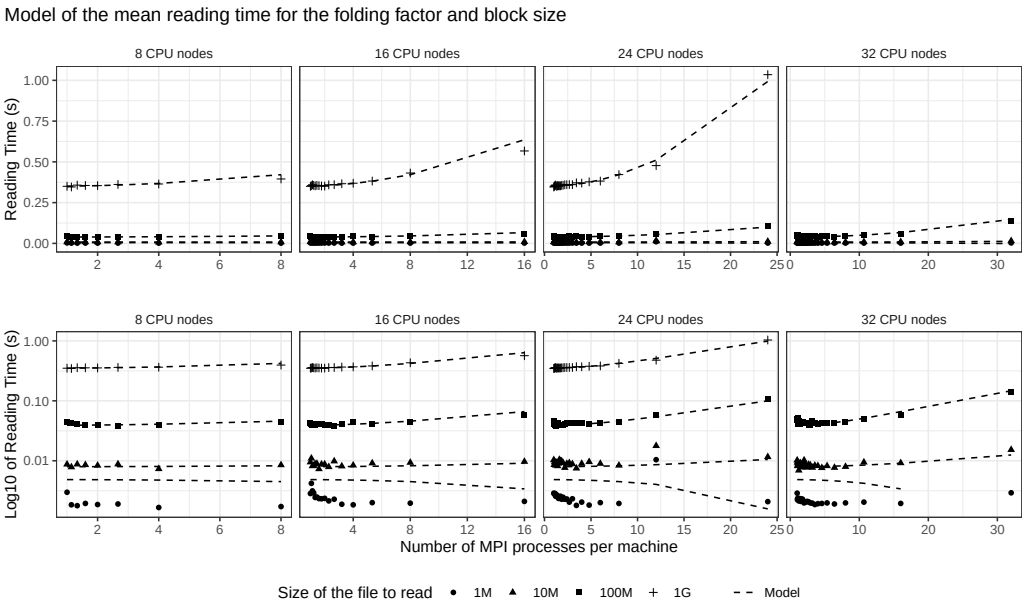
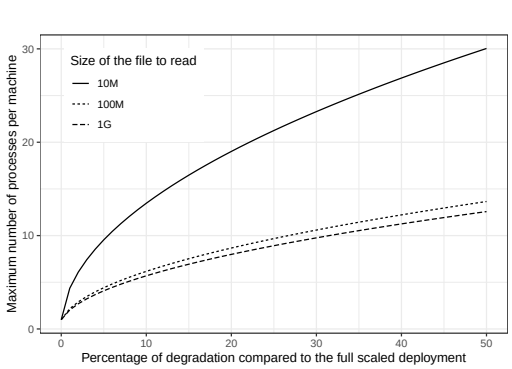
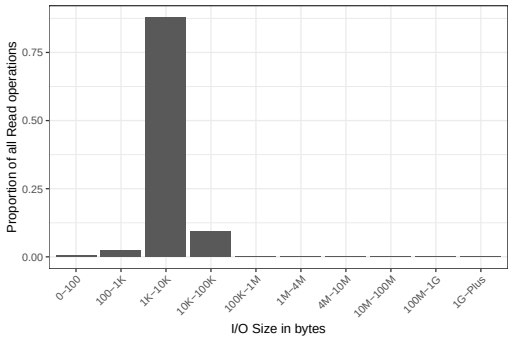


Figure 9.5.: Linear regression modeling the reading times (y-axis) and the folding factor (x-axis), file size (point shape). The top row shows the fitting of the model on the data, and the bottom row the same data but in *log* scale. We can see that the model fits correctly the data for file sizes greater than 10M. 1M files does not seem to be affected by the folding, and their variation in performance seem to be due to noise.



(a) Maximum folding factor (f_{fold}) based on the accepted degradation of the reading time compared to the full scale deployment and to the size of the file to read.



(b) Proportion of the size of the read operations on ANL-Theta between 2019 and 2022. The majority is smaller than 100K. These values are extracted from Darshan [Car+11; Car+09] logs.

Figure 9.6.: Figure 9.6a shows the maximum folding factor to use to have a desired overhead on the reading times on NFS base on the file size. Figure 9.6b shows the distribution of the number of read requests per size on ANL-Theta.

We modeled the reading performances based on the size of the file to read and the number of CPU nodes involved. Figure 9.5 shows the results of a linear regression between the reading time and the folding ratio, file size and number of CPU nodes involved. We fitted a model with the following form:

$$t_{read}(f_{fold}, f_{size}) \simeq \alpha + \beta_1 {f_{fold}}^2 + \beta_2 f_{size} + \gamma {f_{fold}}^2 f_{size} \tag{9.2}$$

with $\alpha, \beta_1, \beta_2, \gamma \in \mathbb{R}$ the coefficients of the model. The R^2 of the fitting is 0.9982.

Figure 9.5 shows the fitting of the model on the NFS data. The bottom row represents the same information as the top one, but in log scale. We can see that the model fits all the file sizes but for 1M. We believe that the variations in performance for the 1M files are due to noise.

We are interested in knowing the maximum folding factor (f_{fold}) possible for a desired file size (f_{size}). Let $p > 1$ be the percentage of increased reading time compared to the full scale deployment containing nb_{cpu} CPU nodes. By using the definition of the model for t_{read} in Equation 9.2, we get:

$$\frac{t_{read}(f, f_{size})}{t_{read}(nb_{cpu}, f_{size})} < p \implies f < \sqrt{\frac{p \times t_{read}(nb_{cpu}, f_{size}) - (\alpha + \beta_2 f_{size})}{\beta_1 + \gamma f_{size}}} \tag{9.3}$$

Take away 9.2

For files of size 10M, folding 10 resources onto a single physical resource leads to a degradation of 5%. To reach the same degradation for file size of 100M or 1G, the maximum folding factor would be 5. (Figure 9.6a)

From the model defined previously, Figure 9.6a, and the Darshan [Car+11; Car+09] logs from ANL-Theta between 2019 and 2022 (Figure 9.6b)[1], we can have an estimation of the overhead if we decided to rerun these Darshan logs (on NFS and with the job model considered in Chapter 2) with different folding factors. For example, a folding factor of 10 would lead to an overhead of 64 hours over 4 years (*i.e.,* an increase of 0.2%), while requiring 10 times fewer machines.

9.4.2 Evaluation of OrangeFS

In the case of OrangeFS, there is an extra dimension to explore: the number of I/O nodes in the file-system.

Figures 9.7 and 9.8 show respectively the evolution of the performance in reading and writing time of the IOR benchmark for different number of CPU nodes in the cluster and I/O nodes in the file-system, as well as different sizes of file to read/write.

We can see on Figure 9.7 that contrary to NFS, there is a significant loss of performance for the write operations when increasing the folding factor. This loss of performance appears more significant when there are more I/O nodes in the file-system.

Concerning the reading performances (Figure 9.8), we observe the same behavior as for NFS. High folding factors lead to an increase of reading time. The increase appears greater when there are more I/O nodes in the file-system. The start of this increase seems to depend on the number of CPU nodes. The more CPU nodes, the later the increase starts. It is interesting to note that the variation when measuring the reading time during the experiments is smaller (thus more stable) than for NFS.

[1]This data was generated from resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

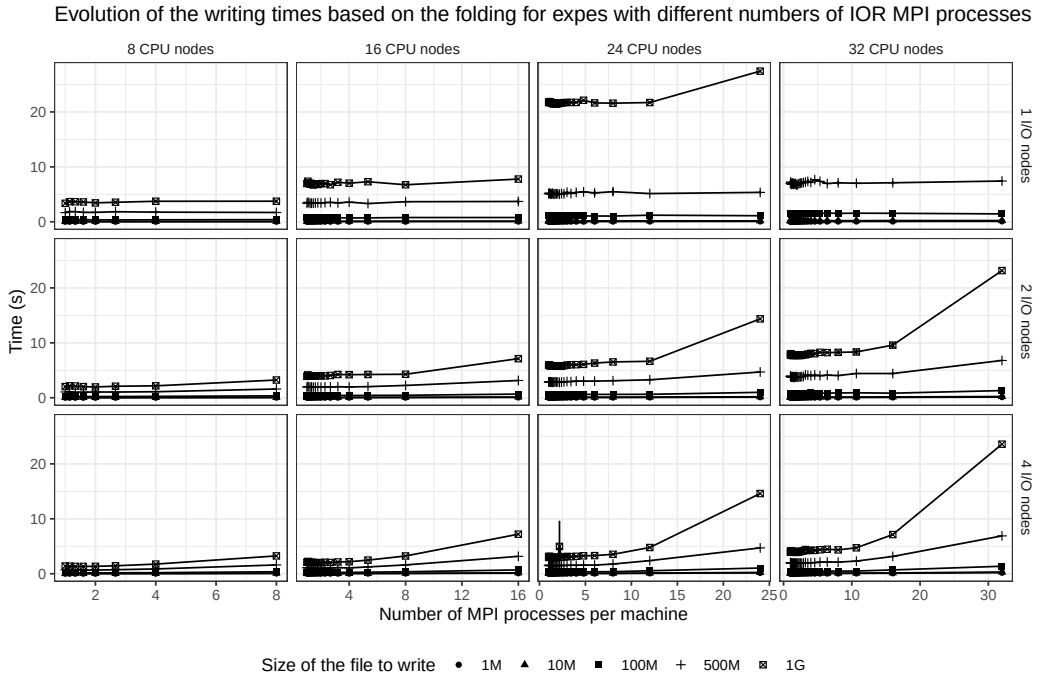


Figure 9.7.: Evolution of the writing times with OrangeFS based on the folding factor (f_{fold}) for experiments with different number of CPU and I/O nodes and different sizes of file to write.

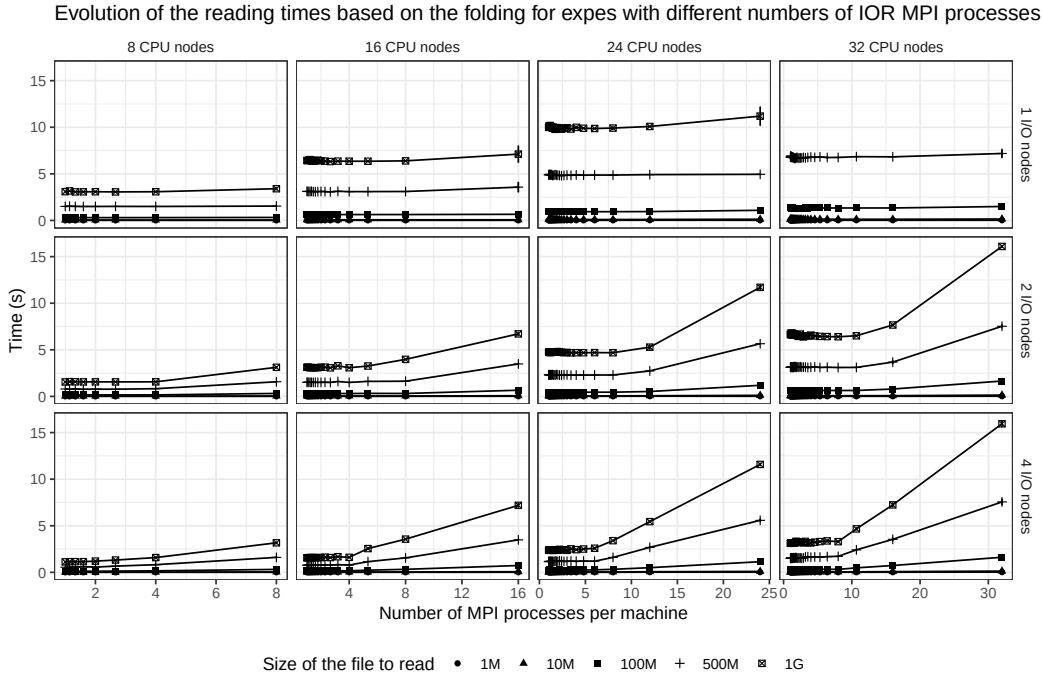


Figure 9.8.: Evolution of the reading times with OrangeFS based on the folding factor (f_{fold}) for experiments with different number of CPU and IO nodes and different sizes of file to read.

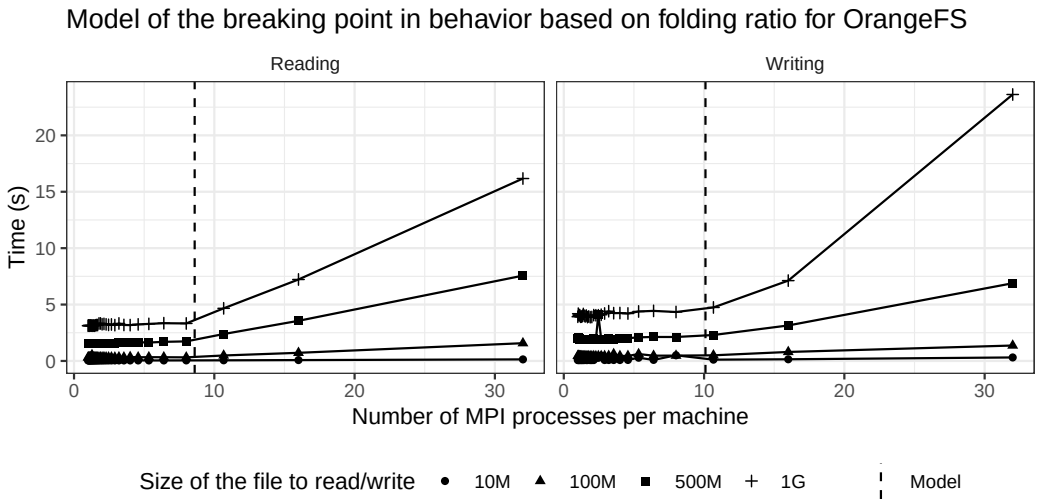


Figure 9.9.: Model of the breaking point in behavior of performance in reading (left) and writing (right) for 32 CPU nodes (nb_{cpu}) and 4 I/O nodes (nb_{io}). The model (dashed line) comes from Equation 9.4.

Take away 9.3

The reading performance of a fully folded deployment **does not depend** on the number of I/O nodes in OrangeFS (Figure 9.8).

For both reading and writing performances there seem to be two behaviors. First a phase where the folding does not affect the performances, and then, from a given folding factor, the reading/writing times grow linearly. As we want to know the maximum folding factor until which the folded system behaves like a full scale system, we are now interested in finding this breakpoint where the behavior changes. Using *segmented regression* techniques [Mug03], we found a model that we simplified to make it a *rule of thumb*:

$$\begin{aligned} f_{break,r} &\simeq 1 + 0.3 \times nb_{cpu} - 0.5 \times nb_{io} \\ f_{break,w} &\simeq 2 + 0.3 \times nb_{cpu} - 0.5 \times nb_{io} \end{aligned} \tag{9.4}$$

where, $f_{break,r}$ and $f_{break,w}$ are respectively the folding factor of the breakpoint for the reading and writing performances, nb_{cpu} and nb_{io} are the number of CPU and I/O nodes in the system.

Take away 9.4

For writing and reading times on OrangeFS, there is a breakpoint in performance before which there is no degradation. We modeled *rules of thumb* to them in Equation 9.4. These *rules of thumb* are depicted in Figure 9.9.

As $f_{break,w}$ (Equation 9.4) will always be greater than $f_{break,r}$, **the overall breaking point in performance for OrangeFS is** $f_{break,r}$.

9.5 Conclusion

In this Chapter, we investigated the use of *folding* techniques to reduce the number of deployed machines for the large scale evaluation of applications such as grid and cluster middlewares like *CiGri*. We have seen that *folding* requires a change in the job model We focused on the impact on the performances of the file-system. To do so, we took a distributed I/O benchmark, IOR, and ran it for several cluster sizes, I/O loads and distributed file-systems. We analyzed the results of the benchmark and reached to the following conclusions:

- Write operations on NFS are not subject to folding (Take away 9.1).
- The performance of read operations on NFS can be modeled with a quadratic relation, and this model can be used to determine the maximum folding for an accepted degradation (Take away 9.2).
- The performance of read operations in a fully folded cluster with OrangeFS do not depend on the number of I/O nodes (Take away 9.3).
- There are breaking points in reading and writing performance for OrangeFS when folding the cluster. Equation 9.4 gives *rules of thumb* to estimate these breakpoints (Take away 9.4).

This study presents some limitations. The studied file-systems are not among the most popular in large HPC centers. It would be interesting to consider file-systems such as Lustre [Lus], BeeGFS [Bee], GlusterFS [BBP12], and Ceph [Wei+06]. During his Master internship, Alexandre Lithaud helped setting up distributed environments in *NixOS Compose* containing such PFS, which will allow to extend this study [Lit23]. The reasons of this loss of performance due to folding are still unclear. The main suspect is the network. We did observe a speed similar (23 Gbps) to the one advertised on the NIC (25 Gbps). We think that the overhead can be due to the

TCP protocol. Investigating different network protocol like InfiniBand or OmniPath would be interesting.

Recent discussions with Francieli Boito[2] highlighted potential imperfections of our protocol concerning the potential impact of caching on read requests on NFS.

[2] https://www.labri.fr/perso/fzanonboito/

10 Towards Simulating *CiGri* and its Control Loop

10.1 Introduction

Distributed experiments are complex and often require several machines for several hours or days. Such experiments are time and resource consuming, but are nevertheless mandatory to validate research work. Deploying and running long-lasting experiments during the exploring phases of research is an obstacle to careful and sane work and must be addressed.

Simulation techniques are an adequate solution as they allow users to execute in reasonable time and on a single laptop, experiments that would have taken hours on a production platform. In the context of High-Performance Computing (HPC), most of the effort in terms of simulators is focused on tools to evaluate scheduling algorithms [Dut+16; GKN18; KSS20]. These solutions reduce considerably the time and computing power required to replay long scientific workloads with a new scheduling strategy instead of deploying a modified batch scheduler and re-executing the jobs of the workload, but have limitations in terms of realism due the underlying models.

Experiments on systems such a grid or cluster middlewares, such as *CiGri*, are also victim of high experimental costs and could benefit from simulation techniques. However, due to this additional layer, the simulators cited above are not directly equipped to simulate such systems.

For our experiments with *CiGri*, having access to information about the state of the machines, the state of the scheduler, etc. would be ideal to implement sensors and feedback loops. However, simulators usually expose a strict interface and set of events to the users, without the possibility for the users to obtain more internal information inside the simulator core.

In this chapter, we present and evaluate *BatCiGri*, a simulator of the *CiGri* grid middleware within *Batsim*. In Section 10.2 we present the desired properties of the simulations, the design of the simulation of the middleware as well as its calibration

to better match reality. The evaluation of the simulation is performed in Section 10.3.

To test our version of *CiGri* with our controllers, we deploy a modified *CiGri* as well as an instance of *OAR* and compute nodes. In order to perform faithful evaluations, it is unreasonable to deploy on the *entire Gricad* meso-center, and replay long workloads. We are thus interested in simulation techniques to reduce the experimental costs.

However, one potential limitation of using simulation techniques in our case, is the inability to obtain the same signals, or for the signals to have a different dynamic or properties.

Take away 10.1

Simulators from the state-of-the-art are not directly equipped for simulating middlewares such as *CiGri* with the need for internal sensors on the system.

10.2 *BatCiGri*

In this Section, we present a solution based on *Batsim* [Dut+16] to simulate *CiGri*: *BatCiGri*.

10.2.1 Expected Properties of the Simulation

For the *CiGri* simulations to be useful from the point-of-view of Control Theory, they must have the following properties: (*i*) jobs must have realistic execution times, (*ii*) best-effort jobs must be killed and release resources for the normal jobs, (*iii*) the killing and releasing of the resources must happen in a realistic time, and (*iv*) information about the usage of the platform and about the inner state of the scheduler must be accessible.

10.2.2 Hypotheses

We work under the following hypotheses: (*i*) there is only one cluster in the grid, and (*ii*) the only best-effort jobs come from *CiGri*. Regular users of the cluster cannot submit best-effort jobs.

10.2.3 *Batsim* in a Nutshell

Batsim is a batch scheduler simulator which allows users to test their scheduling algorithms, *i.e.,* how the jobs are mapped to the resources. *Batsim* relies on *Simgrid* [Cas+14] for sound simulation models. The remaining of this section presents two important concepts of *Batsim*: *platforms* and *workloads*.

Platforms *Batsim* platforms, similarly to *Simgrid* platforms, contain information about the underlying platform of the simulation. It contains the number of hosts, CPU information, the network topology, the speed of the links, the capacity of the disks, etc.

Workloads Workloads contain information about the jobs that will participate in the simulation. There are two main components: `jobs` and `profiles`. Profiles define the behavior of the jobs, *i.e.,* the underlying simulation to use (delay, parallel tasks, SMPI, etc.), execution times, SMPI trace to replay, etc. In a *Batsim* workload, a job refers to a profile. Each job must have an identifier, a submission time and a requested number of resources. Listing 10.1 shows a simple example of *Batsim* workload.

The study of the *CiGri* jobs running on the *Gricad* platform presented in Chapter 2 gives a statistical description of the execution times of those jobs. This allows us to use a delay model to represent the execution times.

10.2.4 Two Schedulers

The computing grid, in the context of *CiGri*, requires two levels of scheduling. The first level is from *CiGri* to *OAR* for best-effort jobs, and then from *OAR* to the nodes for normal users. Our simulation needs to capture these two levels.

To do so we will have two *Batsim* schedulers: one for the *CiGri* jobs and one for the priority jobs. Each scheduler will manage their own workload but will schedule on the same platform.

As best-effort jobs need to have less priority on the normal jobs, we need a way to kill them. The *CiGri* scheduler will thus only see the free resources of the cluster to perform its schedule of best-effort jobs. On the other hand, the priority scheduler does not see the resources taken by *CiGri* jobs as occupied, and can decide to

```
{
    "jobs": [
        {
            "id": 1,
            "profile": "cigri",
            "res": 1,
            "subtime": 0
        },
        {
            "id": 2,
            "profile": "cigri",
            "res": 1,
            "subtime": 0
        },
        {
            "id": 3,
            "profile": "cigri",
            "res": 1,
            "subtime": 0
        }
    ],
    "nb_res": 32,
    "profiles": {
        "cigri": {
            "delay": 235.0,
            "type": "delay"
        }
    }
}
```

Listing 10.1: Example of *Batsim* workload with 3 jobs belonging to the `cigri` profile. Each job requests one resource and are submitted at the start of the simulation.

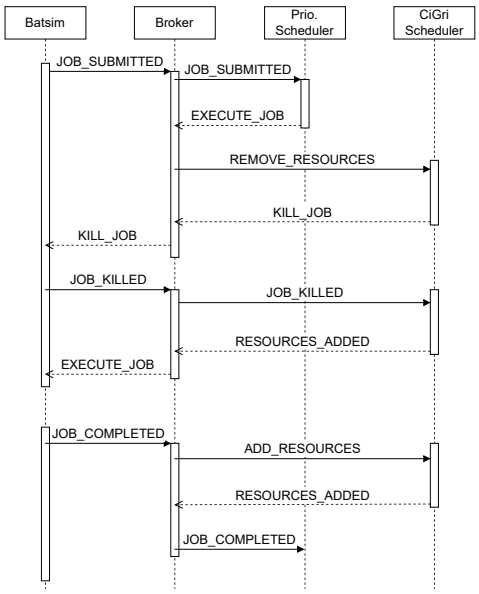


Figure 10.1.: Sequence Diagram representing the killing of best-effort jobs when a new priority job is submitted, as well as when a priority job finishes making its resources idle and thus exploitable by *CiGri*.

schedule jobs on those resources. In this case, the *CiGri* scheduler must manage the killing of its jobs.

To be as close to reality, we used the same scheduling algorithms as the real system: conservative backfilling for the priority jobs, and First-Come-First-Served (FCFS) for the *CiGri* jobs. The *CiGri* scheduler is implemented with the Python interface to Batsim [bat19].

10.2.5 Broker

Batsim can only communicate with a single scheduler. However, as seen in the previous section, we have two different schedulers. To deal with this limitation, we used the work done in [Mer19] which implements a message broker between *Batsim* and the schedulers [Mer18; Gui23a].

The two schedulers connect to the broker and the broker connects to *Batsim*. It filters and redirect the message between the different actors. The main of the work is to manage adaptation of the available resources for the *CiGri* scheduler. When a priority job is submitted, *Batsim* sends a `JOB_SUBMITTED` message to the broker. The broker will then forward this message to the priority job scheduler. If the allocation of resources returned by the scheduler contains best-effort jobs, the broker will inform the *CiGri* scheduler by sending a `REMOVE_RESOURCES` message.
In this case, the *CiGri* scheduler must take care of the killing of the concerned jobs and their resubmission in its queue. When a priority job terminates, its resources become free and thus available to the *CiGri* scheduler. Then, the broker will send a `ADD_RESOURCES` message to *CiGri* to indicate the availability of new resources. Figure 10.1 depicts the sequence diagram of a killing of a best-effort job due to a submission of a normal job.

Take away 10.2

To represent *CiGri* jobs and regular jobs, we have two different schedulers. As *Batsim* can only communicate with a single scheduler, we use a broker to redirect the messages between *Batsim* and the correct scheduler.

10.2.6 The *CiGri* Submission Loop

By taking advantage of the `CALL_ME_LATER` event of *Batsim*, we are able to simulate the cyclic behavior of *CiGri*. At every cycle, the *CiGri* scheduler will read the value

of the sensors, compute the control error, compute the number of jobs to submit and submit them.

In our case, the sensor is the number of best-effort resources in waiting queue and the number of resources used on the platform. The length of the waiting queue is internal information for the scheduler, whereas the number of resources used is computed indirectly. Remember that the *CiGri* scheduler only sees the resources that are not used by the priority scheduler. Thus, the number of resources currently used on the cluster is the total number of resources minus the number of resources visible by *CiGri* and plus the number of resources used by *CiGri* jobs.

The remaining of the *CiGri* cycle is relatively straightforward and is shown in Listing 10.2. All the *CiGri* jobs are available at the start of the simulation. This means that in the *Batsim* workload, they are submitted at time 0.

Take away 10.3

The `CALL_ME_LATER` event of *Batsim* allows use to implement a loop. But the available sensors are limited to information about the platform.

10.2.7 Workload Adjustments

The synchronization between the real experiments and the simulation is complex, and thus the simulation workload needs to be adjusted to match the real workload.

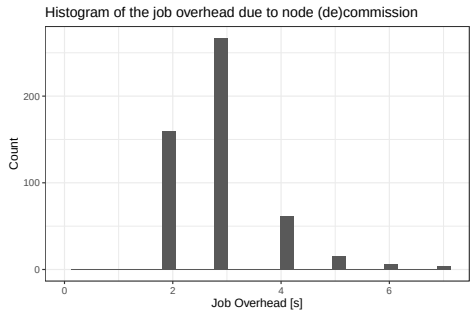
Starting Delay of *OAR* Performed experiments showed that *OAR* needs about 1 minutes and 30 seconds to start the first jobs after the first submission. This delay should be taken into account in the simulation. From the point of view of the *CiGri* scheduler, this delay can be approximated by not starting the jobs submitting from the first 3 *CiGri* cycles.

```
def onRequestedCall(self):
  # Controller Part ------------------------------------------------------------------
  occupied_resources = self.nb_total_resources - len(self.free_resources)
  sensor = len(self.waiting_queue) + occupied_resources

  self.controller.update_error(sensor)
  self.controller.update_input()
  nb_resources_to_submit = self.controller.get_input()
  # ----------------------------------------------------------------------------------

  # Submission Part ------------------------------------------------------------------
  self.add_to_waiting_queue(nb_resources_to_submit)
  to_schedule_jobs = self.to_schedule_jobs()
  # ----------------------------------------------------------------------------------

  if len(to_schedule_jobs) > 0:
      # Ask Batsim to notify for the next cycle
      self.bs.wake_me_up_at(self.bs.time() + self.cigri_period)
  else:
      self.bs.notify_registration_finished()
```

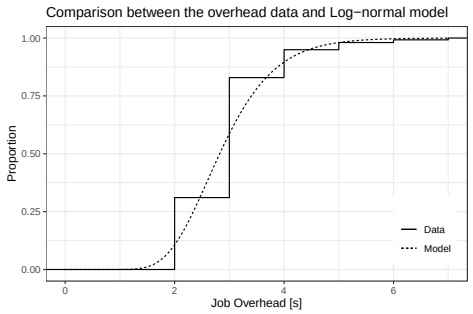
Listing 10.2: Implementation of the *CiGri* submission loop in *Batsim*. It is triggered by the `CALL_ME_LATER` event. At the end of each loop, we ask *Batsim* to notify us for the next loop (line 18).

Commission and Decommission Times Another source of divergence between simulation and real execution, is the commission and decommission of the resources by *OAR*. This (de)commission time is required to set up the computing nodes for the starting jobs, and to clean the nodes after the termination of the jobs. This delay is not present in *Batsim* and must be considered for realism. We evaluated the (de)commission overhead by submitting jobs which perform an identical and precise amount of work, and compare it to the execution time given by *OAR* (*i.e.,* termination time minus starting time). Figure 10.2a shows the distribution of overheads in seconds. This distribution shows that the overheads are mostly about 2 or 3 seconds and that the distributions has a long tail.
We performed a fitting of a Log-Normal law on the overheads' data to retrieve a statistical model. The fitting yielded that the overheads follow a distribution $Lognormal(1.04, 0.27)$. Figure 10.2b shows the cumulative distribution functions of the overhead (solid line) and the model (dashed line). This model allows us to generate *Batsim* workloads containing this overhead in the execution time of the jobs.

Killing of Best-Effort Jobs In *Batsim*, when priority jobs are submitted, and they can be scheduled by killing best-effort jobs, the best-effort jobs are immediately



(a) Histogram of the distribution of job overhead due to the commission and decommission of resources by *OAR*. Most of the overhead is around 2 and 3 seconds.



(b) Comparison between the empirical cumulative distribution function (CDF) of the overhead (solid) and the CDF of the Log-normal model identified (dashed).

Figure 10.2.: Distribution of the job overheads due to *OAR* commissioning and decommissioning the nodes of the cluster. Figure 10.2b shows the comparison between the data and the identified model.

stopped, and the priority jobs started instantaneously. In practice, the priority jobs spend some time in the waiting queue while the best-effort jobs are being killed and the nodes cleaned and set up. This delay can be taken into account in the description of the priority jobs. The execution time in *Batsim* must also contain this delay.

Take away 10.4

OAR introduces variations in the execution times of the jobs due to several factors. To improve the realism of the simulation, we model this delay and inject it during the generation of the *CiGri* workload.

10.3 Evaluation

In this Section, we evaluate the quality of the simulation.

Experimental Protocol For both the real system and the simulated one we will conduct the same scenario. There are 500 *CiGri* jobs with an execution time of 235 seconds. The submission loop of *CiGri* is called every 30 seconds in order to see how the system respond to delay in the control input. After 2000 seconds, a priority job is submitted and takes half of the resources of the cluster for 1800 seconds. The controller of *CiGri* aims to regulate the quantity $w_k + r_k$ around the value 64 (which is the double of the number of resources in the cluster).

Experimental Setup The real experiments were carried on the `dahu` cluster of *Grid'5000* [Bal+13] where the nodes have 2 Intel Xeon Gold 613 with 16 cores per CPU and 192 GiB of memory. The reproducibility of the deployed environment is ensured by *NixOS Compose* [Gui+22b]. The environment is available at [Gui23b], and the data on Zenodo [Gui23c].

We deploy 3 nodes: one for the *OAR* server, one for *CiGri*, and one for the *OAR* cluster. We do not deploy 32 nodes for the cluster, but instead deploy a single node and define 32 OAR resources.

Execution time One of the motivation of this study if the cost in time in resources of experiments. Real experiments require deploying 3 resources (around 10 minutes), and then to execute the scenario (around 1h20 minutes). In total, a single execution of the scenario consumes around 9 CPU hours.

In comparison, a simulation requires a single CPU, and needs 2 seconds to complete, thus consuming approximately 5.5×10^{-4} CPU hours

Signals Comparison For the simulation of *CiGri* to be useful, we need the signals of interest to have the same properties and behave the same in both simulation and real experiments. The signals of interest are:

- the number of best-effort resources in the waiting queue
- the number of currently used resources on the cluster
- the dynamic of a *CiGri* submission (*i.e.,* the time it takes to see the impact of a submission)

Figure 10.3 shows the comparison of the signals of interest between experiments of the same scenario executed in simulation (red) and deployed on real machines (blue). The signals appear to be in sync. The amplitude does differ, as can be observed around 500 seconds. The real system is obviously more sensitive to noise. This noise can be noticed when looking at the used resources (top left graph on Figure 10.3). The cluster in the simulation is always full, whereas the cluster during real experiments is not (*e.g.,* at 1000, 2000, 4500 seconds).

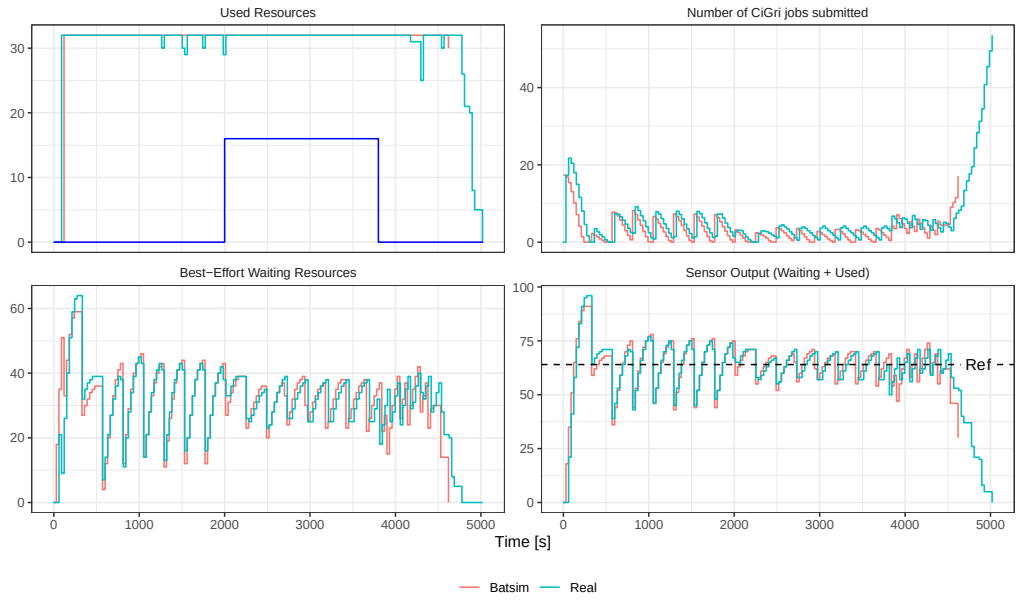


Figure 10.3.: Comparison of the signals of interest for the same experiment executed in simulation with *Batsim* (red) and deploy (blue). Signals appear to be in sync, but some amplitudes might differ.

Gantt Charts Comparison Figure 10.4 compares the resulting Gantt charts of the experiment for the simulation (top) and real execution (bottom). We notice that there are "gaps" in the real schedule (*e.g.,* at time 1500 seconds on resource 24). These gaps create a lag in the schedule which also impact the signals.

This lag comes from *OAR* scheduling algorithm. Once the *OAR* decided to start to compute a scheduler, if any job arrives during the execution of the scheduler, those jobs will not be taken into account until the next schedule call. Taking into account this lag in the simulation is complex, as *Batsim* is responsible for the management of the simulation time, and because the time "stops" during the computation of the schedule.

10.4 Conclusion

Distributed experiments are complex and costly. Simulation techniques can help reduce the cost of such experiments. However, simulators rely on models that can lose information compared to the real system. In this chapter, we implemented the essential behavior of *CiGri* in *Batsim*. Real experiments with *CiGri* requires 3 compute nodes for several hours, while simulation last a few seconds on a laptop.

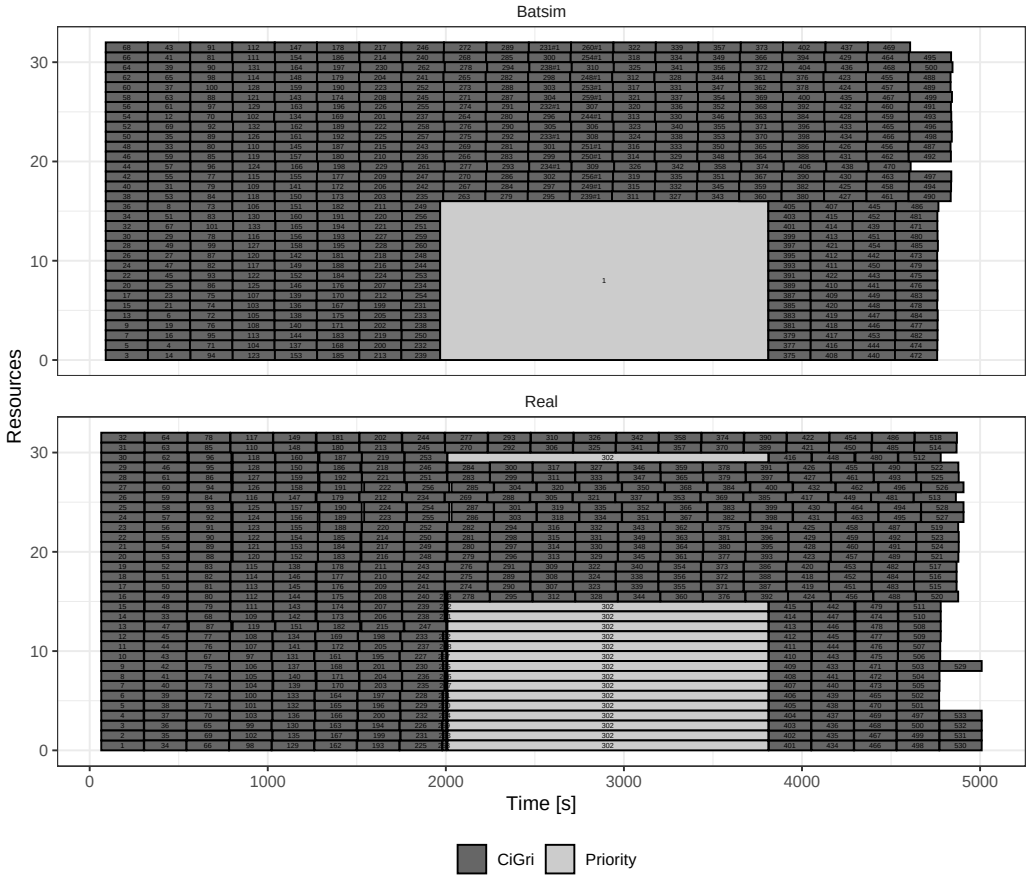


Figure 10.4.: Comparison of the Gantt charts for the simulation (top) and real experiment (bottom) of the same scenario. We observe a small lag, which is due to *OAR*, but both schedules are similar.

We compared the behavior and similarities of signals of interest of our system in simulation and real experiments. We had to modify the workload of the simulation to match the different overheads induced by the real system. This work is still at the very early stages, but results showed satisfying quality of signals in simulation, and this simulator would thus be usable in the early stages of a controller design to reduce the experimental costs. However, the current limitation of *Batsim* is the lack of probe, *i.e.,* a mechanism allowing users to define callbacks on *Batsim* or Simgrid events, to sense internal information and states. Without such probes, it is difficult to implement more complex sensors, actuators, and feedback loops such as the one presented in Chapter 5. Such probes might however ask a fine-grained level of realism that simulators cannot provide, which would limit the range of systems being able to be satisfactory simulated.

11 Conclusion and Perspectives

Distributed experiments are complex. In this Thesis, we explored several research questions linked to distributed experiments. In Chapter 8 we presented *NixOS Compose*, a tool to set up and deploy reproducible distributed environment. With *NixOS Compose* we claim to have a tool to enable researchers to develop quickly and efficiently reproducible distributed experiments. This tool has already been used in various research and educational contexts (European projects, PEPRs, teaching of distributed systems, internships, etc.). Chapter 9 exhibits a method to reduce the number of machines to deploy when performing experiments involving a distributed file-system. By folding a cluster, we can reduce the cost of a distributed experiment while keeping similar performance/behavior as the full scale system. Finally, in Chapter 10, we show a proof of concept to reduce even more the experimental cost of experimenting with *CiGri* by using simulation techniques. These chapters aim to give an overview of the experimental spectrum: from full scale experiments, to reduced deployment, to simulation on a single machine. Each of these techniques trade some realism for reduced experimental cost (either speed or number of machines required).

11.1 Perspectives and open questions

Simulation of distributed file-systems One of the limitations of the simulation of *CiGri* in *Batsim* presented in Chapter 10 is the lack of support for the distributed file-system required to reduce the experimental costs of studies like in Chapters 3 and 4. The simulation of PFS is a complex task that some work have tried to tackle. PFSSim [Liu+11] was the most promising project by developing a framework to evaluate *I/O* scheduling algorithms, but the project has been inactive for 10 years. StorAlloc [Mon+22] focuses on the simulation of intermediate storage, such as burst buffers, between the compute nodes and the file-system. Recent work on *I/O* model validation in Simgrid [Leb+15] could lead to a validated framework to evaluating *I/O* scheduling. For now, Simgrid is able to simulate remote disks, files, and MPIIO calls. But PFS are much more complex: metadata servers, data servers, *I/O*, etc.

Being able to simulate the behavior of classical PFS within Simgrid would lead to a decrease of experimental costs. Integration with *Batsim* would be a plus. Moreover, having a framework to test designs of PFS would enable system administrators to find the best PFS configuration for their system.

Integration between *NixOS Compose* and *EnOSlib* As for now, *NixOS Compose* does not provide its own engine to conduct experiments, and this is not our desire. We developed an integration of *NixOS Compose* into the Execo engine [Imb+13]. Discussions have been started to support deployments with *NixOS Compose* in *EnOSlib*. This integration would benefit *NixOS Compose* with the large collection of tools from *EnOSlib*, and would also benefit *EnOSlib* with the integration of fully reproducible environments.

Filesytems and Functional Package Managers on HPC centers The architecture of store-based package managers (*Nix*, *Guix*, or Spack) creates new kinds of stress on the underlying filesystems designed for classical FHS[1] filesystems. We recently added support in *NixOS Compose* to deploy *NixOS* images with `kexec` which `/nix/store` are located on the shared NFS of *Grid'5000*. This reduces the size of the ramdisk image as it does not contain the store and thus increases deployment speed. But it also raised interesting new questions. As the systems are booting up, the different `systemd` services are starting, and as the `/nix/store` is located on the NFS, a **lot** of syscalls are being made to resolve the dynamic libraries of the services. This problem has been called the *stat storm* (because of the `stat` system call). The blog post [Cou] gives a very nice presentation of the issue. The hierarchy of the `/nix/store` catalyzes this problem by the explosion of the combination exploration required to resolve the dependencies. Variations of this problem have starting to be addressed in the literature [Zak+22; ST17; Fri+13], but are mostly focused on the loading of user code and often require to modify it. During the Master 1 internship of Samuel Brun [Bru23], we started investigating a solution for *NixOS Compose* based on FUSE [Sze10], where the nodes loading the services will cache the result of the `openat` and `stat` syscalls from the NFS and share the results by multicast to the other nodes of the deployment.

Mixed deployments When deploying for *CiGri* experiments for example, we do not deploy a 1-to-1 scale cluster, but instead, we define several computing resources (from the point-of-view of *OAR*) on a single machine. This allows us to reduce the

[1] Filesystem Hierarchy Standard

experimental cost while keeping the same production software stack. To increase the realism of this deployment, *NixOS Compose* could deploy several virtual machines containing the environment of a computing node onto one or several physical nodes. This would allow for isolation between computing resources while reducing the number of physical nodes to perform an experiment.

Study of the lifetime of artifacts The introduction of the artefacts' evaluation in the reviewing process in 2011 for conferences is a good step towards ensuring reproducibility of the submitted work. However, as the artifacts are evaluated right after their creation, they pass the reviewing process. In practice, a 6-month-old artifact that passed the reviewing process has a high probability to not build, or build to a different state than the original one.

A study of the *lifetime* of the artifacts, meaning the duration during which the artifacts rebuild from source the correct/intended one, of the top conferences implementing an artifact reviewing process would most likely reveal the imperfection of the usual tools, as seen in Chapter 7.

Reproducibility metric and partial reproducibility The current badge systems during artifact reviews do not capture all the potential loss of reproducibility presented in Section 7.1.2. A system indicating the pitfalls of the work, on several dimensions, might be more insightful for the readers, *e.g.,* "machine dependent", "using a non-free library". Such criteria be evaluated by artifact reviewers with the help of a "checklist", without having to replay any experiments, only by inspecting the source code. The filled checklist would then be attached to the paper after publication. The creation of such a checklist that would cover most reproducibility cases would be a great asset for reviewer to ease the reviewing process, readers to know what to expect from the presented results, and authors to know what to improve.

The potential issue with such an approach would be the backlash from the community. Today, authors gain badges for doing more in terms of reproducibility, and this is a way for the publisher to gently invite authors to improve the quality of their submission. However, as the artifact review is not always mandatory, and that failing the reproducibility review does not stop the publication of the paper. A system as presented above takes the problem from the other side and would explicitly point to the shortfalls of the work, which could be seen as "public shaming" by authors.

The current artifact reviewing system is far from perfect, and must evolve towards something more strict and precise. Considering that it took 10 years for the artifact

reviews to be integrated into most conferences, changing the existing system might be long.

Environmental cost of reproducibility in computer science Reproducing research works also has costs. First, humans need to evaluate the reproducibility of the papers, which is time-consuming and often require a lot of efforts [Bel]. But also an environmental cost. Replaying experiments requires using additional computing resources, and in the case of large scale and long-lasting experiments this could represent a significant energetic cost.

Similar, but at smaller scales, problematics arise from the use of functional package managers. When replaying an experiment that was packaged with a FPM, the first step is to set up the software environment. This step can be very long as it will need to download a lot of packages and their dependencies. Moreover, if the work is not recent, or is introducing variation deep into the software stack, the packages required will not be present in the binary caches, and will thus need to be recompiled from scratch. Storage is also an issue of FPMs as the stores have the tendency to grow very large with every variation of packages.

Such questions should be addressed by the community. Is a long-lasting experiment requiring hundreds of machines really reproducible? Should reviewers even try to reproduce it? Verifying reproducibility has a cost, but also ensures research quality. Is there a tradeoff between the two? Is the peer-review scheme reaching its limits?

12 General Conclusion

This thesis presented solutions to regulation problems in the harvesting of idle computing resources in HPC cluster. We used tools from Autonomic Computing coupled with Control Theory to design and implement controllers with proven guarantees in the *CiGri* middleware (Part I). The experimental setup of *CiGri* led us to investigate more experiment-oriented research questions (Part II), *i.e.,* how to reduce the number of machines required to deploy an experiment, and in a reproducible fashion.

We gave some detailed perspectives of this thesis Chapters 6 and 11. In this Chapter we give an higher-level conclusion.

Regulation problems are frequent in Computer Science, but especially in fields where there are runtime components or Service-Level Agreements. Scheduling or Machine learning-based strategies are very popular but have limitations. Scheduling usually make assumption on the system, information given by the users, etc. This information is then processed by complex (cognitively and computationally) algorithms that return a schedule of the jobs. It is possible to prove performance bounds of the scheduling algorithm. Machine learning strategies are black boxes. The cost of training the model is high, and the resulting model does not have any proven guarantees other than the precision resulting of the training. Control Theory is a powerful, yet underused, technique for the runtime management of computing systems. It really excels at regulating trade-offs, degradation levels, SLAs, QoS, etc. Moreover, using Control Theory tools on computing systems can foster interaction between research fields, as it requires a knowledge and expertise very rarely seen among computer scientists and system administrators. We tried to tackle this issue at our modest scale by creating and presenting a tutorial to introduce Control Theory to computer scientists [Gui+b].

The experiments required to design controllers need to be robust to be able to design a correct controller. The system on which we worked on during this thesis is complex, with deep software stacks (*CiGri*, *OAR*, file-system, jobs, etc.). Deploying this complete stack in a reproducible fashion is tricky. Usual tools to deploy software environments are not focused on reproducibility. Nix and Guix are, as of today, the

best tools to manage reproducible software environments. We hope that *NixOS Compose* will help researchers in distributed systems to transition from non-reproducible tools. In general, there is the need to educate the communities (not only computer scientists) to these reproducibility questions (*e.g.,* [Gui+a; FPG; LHP; Leg+]), and develop reproducibility not as a afterthought, but as a fundamental skill. The cost of experiments in distributed systems, and especially when evaluation grid or cluster middlewares, is significant. Having realistic experiments implies to perform large scale deployment, which is expensive. Simulations are a good alternative, but are limited by the fact that the real software is not executed. Moreover, the state of simulation of parallel file-system is not mature enough to perform the *CiGri* experiments in simulation. We believe that an interesting alternative is the use of intermediate deployment techniques such as the one presented in Chapter 9.

Three concerns of scientific research are: the *explainability* and the *energy efficiency* of the solutions, and the *reproducibility* of research. This thesis attempted to take into account these concerns, and to raise awareness about them.

Bibliography

[ACM] ACM. *Artefact review badging*. https://www.acm.org/publications/policies/artifact-review-badging. Accessed: 2023-04-04. cit. on p. 107

[ÅH06] Karl Johan Åström and Tore Hägglund. *Advanced PID Control*. English. ISA - The Instrumentation, Systems and Automation Society, 2006. cit. on p. 78

[Amaa] Amazon. *Amazon EC2 Spot Instances*. URL: https://aws.amazon.com/ec2/. cit. on p. 12

[Amab] Amazon. *Amazon EC2 Spot Instances*. URL: https://aws.amazon.com/ec2/spot/?cards.sort-by=item.additionalFields.startDateTime&cards.sort-order=asc. cit. on p. 12

[And+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. "SETI@home: an experiment in public-resource computing". en. In: *Communications of the ACM* 45.11 (Nov. 2002), pp. 56–61. DOI: 10.1145/581571.581573. URL: http://portal.acm.org/citation.cfm?doid=581571.581573 (visited on June 11, 2020). cit. on p. 11

[And04] D.P. Anderson. "BOINC: A System for Public-Resource Computing and Storage". en. In: *Fifth IEEE/ACM International Workshop on Grid Computing*. Pittsburgh, PA, USA: IEEE, 2004, pp. 4–10. DOI: 10.1109/GRID.2004.14. URL: http://ieeexplore.ieee.org/document/1382809/ (visited on June 11, 2020). cit. on p. 11

[And85] Brian D. O. Anderson. "Adaptive Systems, Lack of Persistency of Excitation and Bursting Phenomena". In: *Automatica* 21.3 (May 1985), pp. 247–258. DOI: 10.1016/0005-1098(85)90058-5. cit. on p. 67

[ÅW08] Karl Johan Åström and Björn Wittenmark. *Adaptive Control*. 2nd ed. Dover Publications, 2008. cit. on pp. 67, 68

[ÅW13] Karl J Åström and Björn Wittenmark. *Adaptive control*. Courier Corporation, 2013. cit. on p. 78

[Bak16] Monya Baker. "1,500 scientists lift the lid on reproducibility". In: *Nature* 533.7604 (May 2016), pp. 452–454. DOI: 10.1038/533452a. URL: http://www.nature.com/doifinder/10.1038/533452a (visited on May 3, 2019). cit. on p. 107

[Bal+13] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, et al. “Adding Virtualization Capabilities to the Grid’5000 Testbed”. In: *Cloud Computing and Services Science*. Ed. by Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20. DOI: 10.1007/978-3-319-04519-1_1. cit. on pp. 69, 87, 110, 119, 124, 144, 165

[bat19] [SW] batsim, 2019. URL: https://gitlab.inria.fr/batsim/pybatsim, SWHID: ⟨swh:1:dir:116fa04d25d22b26f6dcf110f976b681ffa9f027;origin=https://gitlab.inria.fr/batsim/pybatsim⟩. cit. on p. 161

[BBP12] Eric B Boyer, Matthew C Broomfield, and Terrell A Perrotti. *Glusterfs one storage server to rule them all*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2012. cit. on p. 154

[Bee] BeeGFS. *BeeGFS Filesystem*. Accessed: 2023-09-29. URL: https://www.beegfs.io/c/. cit. on p. 154

[Bel] Moritz Beller. *I will never review artefact again*. https://inventitech.com/blog/why-i-will-never-review-artifacts-again/. Accessed: 2023-01-19. cit. on p. 172

[Bel05] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. 46. Califor-nia, USA. 2005, pp. 10–5555. cit. on p. 119

[Bha+13] Abhinav Bhatele, Kathryn Mohror, Steven H Langer, and Katherine E Isaacs. “There goes the neighborhood: performance degradation due to nearby jobs”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–12. cit. on p. 3

[Ble17] Raphaël Bleuse. “Apprehending heterogeneity at (very) large scale”. Theses. Université Grenoble Alpes, Oct. 2017. URL: https://hal.science/tel-01722991. cit. on p. 2

[bli] blinry. *Building 15-year-old software with Nix*. https://blinry.org/nix-time-travel/. Accessed: 2023-04-16. cit. on p. 116

[BNW03] John Brevik, Daniel Nurmi, and Rich Wolski. *Quantifying machine availability in networked and desktop grid systems*. Tech. rep. Technical Report CS2003-37, Dept. of Computer Science and Engineering . . ., 2003. cit. on p. 35

[Boi+18] Francieli Zanon Boito, Eduardo C Inacio, Jean Luca Bez, et al. “A checkpoint of research on parallel i/o for high-performance computing”. In: *ACM Computing Surveys (CSUR)* 51.2 (2018), pp. 1–35. cit. on p. 144

[Bon+11] Michael Moore David Bonnie, Becky Ligon, Mike Marshall, et al. “OrangeFS: Advancing PVFS”. In: *USENIX Conference on File and Storage Technologies (FAST)*. 2011. cit. on p. 145

[Bor+07] Julian Borrill, Leonid Oliker, John Shalf, and Hongzhang Shan. “Investigation of leading HPC I/O performance using a scientific-application derived benchmark”. In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. 2007, pp. 1–12. cit. on pp. 52, 53

[Bra+11] Grant R Brammer, Ralph W Crosby, Suzanne J Matthews, and Tiffani L Williams. “Paper mâché: Creating dynamic reproducible science”. In: *Procedia Computer Science* 4 (2011), pp. 658–667. cit. on p. 109

[Bri+19] Adam Brinckman, Kyle Chard, Niall Gaffney, et al. “Computing environments for reproducibility: Capturing the “Whole Tale””. In: *Future Generation Computer Systems* 94 (2019), pp. 854–867. cit. on p. 109

[Bru23] Samuel Brun. “Étude du phénomène Stat-Storm - Limitation des appels systèmes pour les systèmes de fichiers distribués de type store”. MA thesis. Université Grenoble Alpes, Sept. 2023. URL: https://inria.hal.science/hal-04197724. cit. on p. 170

[Bur22a] [SW] Sander van der Burg, *Disnix* Apr. 2022. LIC: LGPL-2.1. URL: https://github.com/svanderburg/disnix, SWHID: ⟨swh:1:dir:c4f91c498853331fc9a1c4607d0e6d7c01aa7d6c;origin=https://github.com/svanderburg/disnix⟩. cit. on p. 119

[Bur22b] [SW] Sander van der Burg, *DisnixOS* 2022. LIC: LGPL-2.1. URL: https://github.com/svanderburg/disnixos, SWHID: ⟨swh:1:dir:13bcd4e967b9abf53066ebd558b30d8d82920f77;origin=https://github.com/svanderburg/disnixos⟩. cit. on p. 119

[Cal23] [SW] University of California, *IOR Benchmark* 2023. URL: https://github.com/hpc/ior, SWHID: ⟨swh:1:dir:006907fd99642d89025fa685ae72d9055c065a3e;origin=https://github.com/hpc/ior;⟩. cit. on p. 144

[Cap+05] N. Capit, G. Da Costa, Y. Georgiou, et al. “A batch scheduler with high level components”. en. In: *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.* Cardiff, Wales, UK: IEEE, 2005, 776–783 Vol. 2. DOI: 10.1109/CCGRID.2005.1558641. URL: http://ieeexplore.ieee.org/document/1558641/ (visited on May 25, 2020). cit. on pp. 13, 23, 70, 125, 129, 142

[Car+09] Philip Carns, Robert Latham, Robert Ross, et al. “24/7 characterization of petascale I/O workloads”. In: *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE. 2009, pp. 1–10. cit. on pp. 25, 150, 151

[Car+11] Philip Carns, Kevin Harms, William Allcock, et al. “Understanding and improving computational science storage access through continuous characterization”. In: *ACM Transactions on Storage (TOS)* 7.3 (2011), pp. 1–26. cit. on pp. 25, 54, 99, 150, 151

[Cas+00] Henri Casanova, Arnaud Legrand, Dmitrii Zagorodnov, and Francine Berman. “Heuristics for scheduling parameter sweep applications in grid environments”. In: *Proceedings 9th Heterogeneous Computing Workshop (HCW 2000)(Cat. No. PR00556)*. IEEE. 2000, pp. 349–363. cit. on p. 26

[Cas+14] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. “Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms”. In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917. URL: http://hal.inria.fr/hal-01017319. cit. on pp. 105, 115, 141, 159

[CB01] Walfredo Cirne and Francine Berman. “A comprehensive model of the supercomputer workload”. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*. IEEE. 2001, pp. 140–148. cit. on pp. 3, 12, 95

[CDR14] Julio Cano, Gwenaël Delaval, and Eric Rutten. “Coordination of ECA rules by verification and control”. In: *Coordination Models and Languages: 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings 16*. Springer. 2014, pp. 33–48. cit. on p. 17

[Cer+21] Sophie Cerf, Raphaël Bleuse, Valentin Reis, Swann Perarnau, and Eric Rutten. “Sustaining performance while reducing energy consumption: a control theory approach”. In: *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings 27*. Springer. 2021, pp. 334–349. cit. on pp. 17, 18, 101

[Cer19] Sophie Cerf. “Control Theory for Computing Systems: Application to big-data cloud services & location privacy protection”. Theses. UNIVERSITÉ GRENOBLE ALPES, May 2019. URL: https://hal.science/tel-02272258. cit. on p. 101

[Che+22] Ronan-Alexandre Cherrueau, Marie Delavergne, Alexandre van Kempen, et al. “EnosLib: A Library for Experiment-Driven Research in Distributed Computing”. en. In: *IEEE Transactions on Parallel and Distributed Systems* 33.6 (June 2022), pp. 1464–1477. DOI: 10.1109/TPDS.2021.3111159. URL: https://ieeexplore.ieee.org/document/9534688/ (visited on Nov. 22, 2021). cit. on p. 111

[Chr22] [SW] Juliusz Chroboczek, *Polipo — a caching web proxy* 2022. URL: https://www.irif.fr/~jch/software/polipo/, SWHID: ⟨swh:1:dir:ce183542e1abba5aa847fed92a700d23cf591f9f;origin=https://github.com/jech/polipo;⟩. cit. on p. 134

[cig23] [SW] cigri-feedforward-approach, 2023. URL: https://gitlab.inria.fr/cigri-ctrl/feedforward-approach/cigri-src, SWHID: ⟨swh:1:dir:5fe2f4d3ccf8da04730c8501db73dbaaca37aeb0;origin=https://gitlab.inria.fr/cigri-ctrl/feedforward-approach/cigri-src⟩. cit. on p. 100

[Cir+06] Walfredo Cirne, Francisco Brasileiro, Nazareno Andrade, et al. “Labs of the world, unite!!!” In: *Journal of Grid Computing* 4 (2006), pp. 225–246. cit. on p. 13

[CL12] Nicolo Cesa-Bianchi and Gábor Lugosi. “Combinatorial bandits”. In: *Journal of Computer and System Sciences* 78.5 (2012), pp. 1404–1422. cit. on p. 97

[CL22] Tom Cornebize and Arnaud Legrand. “Simulation-based optimization and sensibility analysis of MPI applications: Variability matters”. In: *Journal of Parallel and Distributed Computing* 166 (2022), pp. 111–125. cit. on p. 38

[Clo] Chameleon Cloud. *The cc-snapshot utility*. `https://chameleoncloud.readthedocs.io/en/latest/technical/images.html#the-cc-snapshot-utility`. Accessed: 2023-04-03. cit. on p. 110

[Cor21] Tom Cornebize. “High performance computing: Towards better performance predictions and experiments”. PhD thesis. Université Grenoble Alpes [2020-....], 2021. cit. on p. 2

[Cou] Ludovic Courtès. *Taming the ‘stat’ storm with a loader cache*. URL: `https://guix.gnu.org/blog/2021/taming-the-stat-storm-with-a-loader-cache/`. cit. on p. 170

[Cou13] Ludovic Courtès. “Functional Package Management with Guix”. en. In: *arXiv:1305.4584 [cs]* (May 2013). URL: `http://arxiv.org/abs/1305.4584` (visited on June 13, 2020). cit. on p. 113

[CPW15] Christian Collberg, Todd Proebsting, and Alex M Warren. “Repeatability and Benefaction in Computer Systems Research - A Study and a Modest Proposal”. en. In: (2015), p. 68. cit. on p. 107

[CR23] Sophie Cerf and Eric Rutten. “Combining neural networks and control: potentialities, patterns and perspectives”. In: *22nd World Congress of the International Federation of Automatic Control*. 2023. cit. on p. 17

[Dar57] Donald A Darling. “The kolmogorov-smirnov, cramer-von mises tests”. In: *The Annals of Mathematical Statistics* 28.4 (1957), pp. 823–838. cit. on p. 35

[DAS] DAS. *DAS2*. URL: `https://www.cs.vu.nl/das5/home.shtml` (visited on Apr. 25, 2023). cit. on p. 29

[DB21] R.C. Dorf and R.H. Bishop. *Modern Control Systems*. Pearson, 14th edition, 2021. cit. on p. 78

[Dép+18] Aline Déprez, Anne Socquet, Nathalie Cotte, and Andrea Walpersdorf. “Toward the generation of EPOS-GNSS products”. In: *19th General Assembly of WEGENER: on Earth deformation & the study of earthquakes using geodesy and geodynamics*. 2018. cit. on p. 23

[DJV04] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. “Nix: A Safe and Policy-Free System for Software Deployment”. en. In: (2004), p. 14. cit. on pp. 113, 119

[DL08] Eelco Dolstra and Andres Löh. “NixOS: A Purely Functional Linux Distribution”. In: *SIGPLAN Not.* 43.9 (Sept. 2008), pp. 367–378. DOI: `10.1145/1411203.1411255`. URL: `https://doi.org/10.1145/1411203.1411255`. cit. on p. 119

[Doc] Docker. *Optimizing builds with cache management*. `https://docs.docker.com/build/cache/`. Accessed: 2023-06-20. cit. on p. 114

[Doc22] [SW] Docker, *Docker Compose* 2022. URL: https://docs.docker.com/compose/, SWHID: ⟨swh:1:dir:09c1c47b2ec22e82d5218e3c1d7193b397a3224d;origin=https://github.com/docker/compose;⟩. cit. on p. 124

[Dor+14] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination". In: *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE. 2014, pp. 155–164. cit. on p. 3

[Dup+19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, et al. "The Design and Operation of CloudLab". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14. URL: https://www.flux.utah.edu/paper/duplyakin-atc19. cit. on p. 139

[Dut+16] Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard. "Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator". In: *20th Workshop on Job Scheduling Strategies for Parallel Processing*. Chicago, United States, May 2016. URL: https://hal.archives-ouvertes.fr/hal-01333471. cit. on pp. 115, 141, 157, 158

[Eme+14] Joseph Emeras, Cristian Ruiz, Jean-Marc Vincent, and Olivier Richard. "Analysis of the jobs resource utilization on a production system". In: *Job Scheduling Strategies for Parallel Processing: 17th International Workshop, JSSPP 2013, Boston, MA, USA, May 24, 2013 Revised Selected Papers 17*. Springer. 2014, pp. 1–21. cit. on p. 99

[fee23] [SW] feedforward-approach, 2023. URL: https://gitlab.inria.fr/cigri-ctrl/feedforward-approach/cigri-feedforward, SWHID: ⟨swh:1:dir:5a540eb9a13d6be14e18f47f997bc58d394cb3db;origin=https://gitlab.inria.fr/cigri-ctrl/feedforward-approach/cigri-feedforward⟩. cit. on p. 87

[Fei15] Dror G. Feitelson. "From Repeatability to Reproducibility and Corroboration". en. In: *ACM SIGOPS Operating Systems Review* 49.1 (Jan. 2015), pp. 3–11. DOI: 10.1145/2723872.2723875. URL: https://dl.acm.org/doi/10.1145/2723872.2723875 (visited on May 21, 2020). cit. on p. 109

[Fil+17] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, et al. "Control strategies for self-adaptive software systems". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 11.4 (2017), pp. 1–31. cit. on p. 17

[FJ13] Michel Fliess and Cédric Join. "Model-free control". In: *International Journal of Control* 86.12 (2013), pp. 2228–2252. cit. on pp. 68, 69, 76

[FKY81] T. R. Fortescue, Lester S. Kershenbaum, and B. Erik Ydstie. "Implementation of Self-tuning Regulators with Variable F orgetting Factors". In: *Automatica.* 17.6 (Nov. 1981), pp. 831–835. DOI: 10.1016/0005-1098(81)90070-4. cit. on p. 67

[For+21] Alessandra Forti, Ivan Glushkov, Lukas Heinrich, et al. "The fight against COVID-19: Running Folding@ Home simulations on ATLAS resources". In: *EPJ Web of Conferences*. Vol. 251. EDP Sciences. 2021, p. 02003. cit. on p. 11

[FPG] Adrien Faure, Millian Poquet, and Quentin Guilloteau. *Nix tutorial*. URL: https://nix-tutorial.gitlabpages.inria.fr/nix-tutorial/index.html. cit. on p. A1

[Fri+13] Wolfgang Frings, Dong H Ahn, Matthew LeGendre, et al. "Massively parallel loading". In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 2013, pp. 389–398. cit. on p. 170

[FZ87] Domenico Ferrari and Songnian Zhou. *An empirical investigation of load indices for load balancing applications*. Computer Science Division, University of California, 1987. cit. on pp. 39, 46

[Gam+15] Todd Gamblin, Matthew LeGendre, Michael R. Collette, et al. "The Spack package manager: bringing order to HPC software chaos". en. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Austin Texas: ACM, Nov. 2015, pp. 1–12. DOI: 10.1145/2807591.2807623. URL: https://dl.acm.org/doi/10.1145/2807591.2807623 (visited on Nov. 22, 2021). cit. on p. 110

[Geo+06] Yiannis Georgiou, Julien Leduc, Brice Videau, Johann Peyrard, and Olivier Richard. "A tool for environment deployment in clusters and light grids". In: *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2006, 8–pp. cit. on p. 124

[GKN18] Cristian Galleguillos, Zeynep Kiziltan, and Alessio Netti. "Accasim: an HPC simulator for workload management". In: *High Performance Computing: 4th Latin American Conference, CARLA 2017, Buenos Aires, Argentina, and Colonia del Sacramento, Uruguay, September 20-22, 2017, Revised Selected Papers 4*. Springer. 2018, pp. 169–184. cit. on p. 157

[GRC07] Yiannis Georgiou, Olivier Richard, and Nicolas Capit. "Evaluations of the lightweight grid cigri upon the grid5000 platform". In: *Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*. IEEE. 2007, pp. 279–286. cit. on pp. 14, 23

[Gri] Grid'5000. *Creating an environment images using tgz-g5k*. https://grid5000.fr/w/Environment_creation#Creating_an_environment_images_using_tgz-g5k. Accessed: 2023-04-03. cit. on p. 110

[Gri23] [SW] Gricad, *Nix Ciment Channel* 2023. URL: https://github.com/Gricad/nix-ciment-channel, SWHID: ⟨swh:1:dir:dec8c22b23ba51650f65352fa2fc2640f1532bea;origin=https://github.com/Gricad/nix-ciment-channel⟩. cit. on p. 115

[GRR22a] Quentin Guilloteau, Olivier Richard, and Éric Rutten. "Étude des applications Bag-of-Tasks du méso-centre Gricad". In: *COMPAS 2022-Conférence d'informatique en Parallélisme, Architecture et Système*. 2022. cit. on pp. 6, 29

[GRR22b] Quentin Guilloteau, Olivier Richard, and Eric Rutten. *Étude des applications Bag-of-Tasks du méso-centre Gricad*. Zenodo, July 2022. DOI: 10.5281/zenodo.8410346. URL: https://doi.org/10.5281/zenodo.8410346. cit. on p. 29

[Gui+a] Quentin Guilloteau, Jonathan Bleuzen, Millian Poquet, and Olivier Richard. *Initiation to NixOS Compose*. URL: https://nixos-compose.gitlabpages.inria.fr/tuto-nxc/. cit. on pp. 7, 139, A1

[Gui+b] Quentin Guilloteau, Sophie Cerf, Eric Rutten, Raphaël Bleuse, and Bogdan Robu. *Introduction to Control Theory for Computer Scientists*. URL: https://control-for-computing.gitlabpages.inria.fr/tutorial/intro.html. cit. on pp. 7, 39, 173

[Gui+21a] Quentin Guilloteau, Olivier Richard, Bogdan Robu, and Eric Rutten. "Controlling the Injection of Best-Effort Tasks to Harvest Idle Computing Grid Resources". In: *ICSTCC 2021 - 25th International Conference on System Theory, Control and Computing*. Iași, Romania, Oct. 2021, pp. 1–6. DOI: 10.1109/ICSTCC52150.2021.9607292. URL: https://hal.inria.fr/hal-03363709. cit. on pp. 6, 42, 56, 58, 60

[Gui+21b] Quentin Guilloteau, Olivier Richard, Eric Rutten, and Bogdan Robu. "Collecte de ressources libres dans une grille en préservant le système de fichiers : une approche autonomique". In: *COMPAS 2021 - Conférence d'informatique en Parallélisme, Architecture et Système*. Lyon, France, July 2021, pp. 1–11. URL: https://hal.inria.fr/hal-03282727. cit. on p. 6

[Gui+22a] [SW] Quentin Guilloteau, Jonathan Bleuzen, Millian Poquet, and Olivier Richard, version 1.0, 2022. URL: https://gitlab.inria.fr/nixos-compose/nixos-compose, SWHID: ⟨swh:1:dir:6308ca57ea23fbdcd4ea84006149583a2db8f881;origin=https://gitlab.inria.fr/nixos-compose/nixos-compose⟩. cit. on p. 138

[Gui+22b] Quentin Guilloteau, Jonathan Bleuzen, Millian Poquet, and Olivier Richard. "Painless Transposition of Reproducible Distributed Environments with NixOS Compose". In: *CLUSTER 2022 - IEEE International Conference on Cluster Computing*. Vol. CLUSTER 2022 - IEEE International Conference on Cluster Computing. Heidelberg, Germany, Sept. 2022, pp. 1–12. URL: https://hal.science/hal-03723771. cit. on pp. 6, 87, 114, 119, 144, 165

[Gui+22c] Quentin Guilloteau, Jonathan Bleuzen, Millian Poquet, and Olivier Richard. "Transposition d'environnements distribués reproductibles avec NixOS Compose". In: COMPAS 2022 (July 2022), pp. 1–9. URL: https://hal.science/hal-03696485. cit. on p. 6

[Gui+22d] Quentin Guilloteau, Bogdan Robu, Cédric Join, et al. "Model-free control for resource harvesting in computing grids". In: *Conference on Control Technology and Applications, CCTA 2022*. Trieste, Italy: IEEE, Aug. 2022. URL: https://hal.archives-ouvertes.fr/hal-03663273. cit. on pp. 6, 66, 69, 70

[Gui+23a] Quentin Guilloteau, Adrien Faure, Millian Poquet, and Olivier Richard. "Comment rater la reproductibilité de ses expériences ?" In: *Conférence francophone en informatique (ComPAS 2023)*. Annecy, France, July 2023, à paraître. URL: https://hal.science/hal-04132438. cit. on pp. 6, 106

[Gui+23b] [SW] Quentin Guilloteau, Olivier Richard, Raphaël Bleuse, and Eric Rutten, 2023. URL: https://gitlab.inria.fr/nixos-compose/hpc-io/articles/folding, SWHID: ⟨swh:1:dir:7b37ae5308065c18081acae7fac97d5028492948;origin=https://gitlab.inria.fr/nixos-compose/hpc-io/articles/folding⟩. cit. on p. 144

[Gui+23c] Quentin Guilloteau, Olivier Richard, Raphaël Bleuse, and Eric Rutten. *Data for the paper: "Folding a Cluster containing a Distributed File-System"*. Zenodo, Oct. 2023. DOI: 10.5281/zenodo.10005463. URL: https://doi.org/10.5281/zenodo.10005463. cit. on p. 144

[Gui+23d] Quentin Guilloteau, Olivier Richard, Raphaël Bleuse, and Eric Rutten. "Folding a Cluster containing a Distributed File-System". working paper or preprint. 2023. URL: https://hal.science/hal-04038000. cit. on pp. 6, 70, 141

[Gui20] Quentin Guilloteau. "Minimizing Cluster Under-use using a Control-Based Approach". Internship report. Grenoble INP Ensimag ; Université Grenoble Alpes, June 2020. URL: https://hal.inria.fr/hal-03167242. cit. on pp. 42, 56

[Gui23a] [SW] Quentin Guilloteau, 2023. URL: https://gitlab.inria.fr/qguillot/batbroker, SWHID: ⟨swh:1:dir:261eeaa67ea32a2c6fa382ecf1ee72dd6a746173;origin=https://gitlab.inria.fr/qguillot/batbroker⟩. cit. on p. 161

[Gui23b] [SW] Quentin Guilloteau, 2023. URL: https://gitlab.inria.fr/cigri-ctrl/batcigri, SWHID: ⟨swh:1:dir:c2dbf035e87b3d9d032fc902ba32d5ea135f75e7;origin=https://gitlab.inria.fr/cigri-ctrl/batcigri⟩. cit. on p. 165

[Gui23c] Quentin Guilloteau. *Data for the paper: "Simulating a Multi-Layered Grid Middleware"*. Zenodo, Oct. 2023. DOI: 10.5281/zenodo.10005440. URL: https://doi.org/10.5281/zenodo.10005440. cit. on p. 165

[Gui23d] Quentin Guilloteau. "Simulating a Multi-Layered Grid Middleware". working paper or preprint. May 2023. URL: https://hal.science/hal-04101015. cit. on p. 6

[Gui23e] [SW] Guix-HPC, *Guix-HPC* 2023. URL: https://gitlab.inria.fr/guix-hpc/guix-hpc, SWHID: ⟨swh:1:dir:aabd69b98944499630729faf0184f4d68ff13fa;origin=https://gitlab.inria.fr/guix-hpc/guix-hpc⟩. cit. on p. 114

[Han09] Jingqing Han. "From PID to Active Disturbance Rejection Control". In: *IEEE Trans. Ind. Electron.* 56.3 (Mar. 2009), pp. 900–906. DOI: 10.1109/TIE.2008.2011621. cit. on p. 68

[Hei+17] Franz Christian Heinrich, Tom Cornebize, Augustin Degomme, et al. "Predicting the energy-consumption of mpi applications at scale using only a single node". In: *2017 IEEE international conference on cluster computing (CLUSTER)*. IEEE. 2017, pp. 92–102. cit. on pp. 2, 92

[Hel+04] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. *Feedback control of computing systems*. John Wiley & Sons, 2004. cit. on pp. 46, 49

[Her] Software Heritage. *Software Heritage*. Accessed: 2023-03-30. URL: https://www.softwareheritage.org/. cit. on p. 109

[Hin+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, et al. "Mesos: A platform for {Fine-Grained} resource sharing in the data center". In: *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. 2011. cit. on p. 13

[HM08] Markus C Huebscher and Julie A McCann. "A survey of autonomic computing—degrees, models, and applications". In: *ACM Computing Surveys (CSUR)* 40.3 (2008), pp. 1–28. cit. on p. 14

[hpc23a] [SW] hpc-io, 2023. URL: https://gitlab.inria.fr/nixos-compose/hpc-io/nfs, SWHID: ⟨swh:1:dir:882f068849839146a5fd0eb0c56493b64aaa91ec;origin=https://gitlab.inria.fr/nixos-compose/hpc-io/nfs⟩. cit. on p. 144

[hpc23b] [SW] hpc-io, 2023. URL: https://gitlab.inria.fr/nixos-compose/hpc-io/orangefs, SWHID: ⟨swh:1:dir:8f05ae8165fc653aeab06b96039aef7972f4bece;origin=https://gitlab.inria.fr/nixos-compose/hpc-io/orangefs⟩. cit. on p. 144

[IE10] Alexandru Iosup and Dick Epema. "Grid computing workloads". In: *IEEE Internet Computing* 15.2 (2010), pp. 19–26. cit. on p. 36

[Imb+13] Matthieu Imbert, Laurent Pouilloux, Jonathan Rouzaud-Cornabas, Adrien Lebre, and Takahiro Hirofuchi. "Using the EXECO toolbox to perform automatic and reproducible cloud experiments". In: *1st International Workshop on UsiNg and building ClOud Testbeds (UNICO, collocated with IEEE CloudCom 2013*. Bristol, United Kingdom: IEEE, Dec. 2013. DOI: 10.1109/CloudCom.2013.119. URL: https://hal.inria.fr/hal-00861886. cit. on p. 170

[IO5] IO500. *IO500 Ranking*. Accessed: 2023-01-19. URL: https://io500.org/. cit. on p. 144

[Ios+08] Alexandru Iosup, Hui Li, Mathieu Jan, et al. "The grid workloads archive". In: *Future Generation Computer Systems* 24.7 (2008), pp. 672–686. cit. on p. 36

[IS96] Petros A Ioannou and Jing Sun. *Robust adaptive control*. Vol. 1. PTR Prentice-Hall Upper Saddle River, NJ, 1996. cit. on p. 78

[IT18] Peter Ivie and Douglas Thain. "Reproducibility in scientific computing". In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–36. cit. on p. 108

[Jav+09] Bahman Javadi, Derrick Kondo, Jean-Marc Vincent, and David P Anderson. "Mining for statistical models of availability in large-scale distributed systems: An empirical study of seti@ home". In: *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*. IEEE. 2009, pp. 1–10. cit. on p. 35

[JPV23] Emmanuel Jeannot, Guillaume Pallez, and Nicolas Vidal. "IO-aware Job-Scheduling: Exploiting the Impacts of Workload Characterizations to select the Mapping Strategy". In: *The International Journal of High Performance Computing Applications* (2023), p. 10943420231175854. cit. on p. 25

[K3s] [SW] K3s-io, *K3s: Lightweight Kubernetes*. URL: https://k3s.io/(visited on Apr. 28, 2022), SWHID: ⟨swh:1:dir:c8498fe26be8ef891d37b7d89b71dde0579a35ed;origin=https://github.com/k3s-io/k3s.git⟩. cit. on pp. 122, 136

[KC03] Jeffrey O Kephart and David M Chess. "The vision of autonomic computing". In: *Computer* 36.1 (2003), pp. 41–50. cit. on pp. 14, 15, 25

[Kea+20] Kate Keahey, Jason Anderson, Zhuo Zhen, et al. "Lessons Learned from the Chameleon Testbed". In: *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020. cit. on pp. 110, 139

[Kow92] Zdzisław Kowalczuk. "Competitive Identification for Self-tuning Control: R obust Estimation Design and Simulation Experiments". In: *Automatica* 28.1 (Jan. 1992), pp. 193–201. DOI: 10.1016/0005-1098(92)90021-7. cit. on p. 68

[KSS20] Dalibor Klusáček, Mehmet Soysal, and Fréd éric Suter. "Alea–complex job scheduling simulator". In: *Parallel Processing and Applied Mathematics: 13th International Conference, PPAM 2019, Bialystok, Poland, September 8–11, 2019, Revised Selected Papers, Part II 13*. Springer. 2020, pp. 217–229. cit. on p. 157

[LAC06] Yun Li, Kiam Heong Ang, and G.C.Y. Chong. "Patents, software, and hardware for PID control: an overview and analysis of the current art". In: *IEEE Control Systems Magazine* 26.1 (2006), pp. 42–54. DOI: 10.1109/MCS.2006.1580153. cit. on p. 78

[Lan+11] Ioan Doré Landau, Rogelio Lozano, Mohammed M'Saad, and Alireza Karimi. *Adaptive Control. Algorithms, Analysis and Applications*. 2nd ed. Communications and Control Engineering. Springer, 2011. DOI: 10.1007/978-0-85729-664-1. cit. on p. 78

[Lar+09] Stefan M Larson, Christopher D Snow, Michael Shirts, and Vijay S Pande. "Folding@ Home and Genome@ Home: Using distributed computing to tackle previously intractable problems in computational biology". In: *arXiv preprint arXiv:0901.0866* (2009). cit. on p. 11

[Leb+15] Adrien Lebre, Arnaud Legrand, Frédéric Suter, and Pierre Veyre. "Adding storage simulation capacities to the simgrid toolkit: Concepts, models, and api". In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2015, pp. 251–260. cit. on p. 169

[Leg+] Arnaud Legrand, Konrad Hinsen, Christophe Pouzat, et al. *Mooc Reproducible research 2*. URL: https://learninglab.gitlabpages.inria.fr/mooc-rr/mooc-rr2-ressources/. cit. on p. A1

[Len21] [SW] Dan Lenski, 2021. URL: https://github.com/dlenski/top500, SWHID: ⟨swh:1:dir:535da402e4285b7b26cdd294db27fee5abbf39ad;origin=https://github.com/dlenski/top500⟩. cit. on p. 2

[LHP] Arnaud Legrand, Konrad Hinsen, and Christophe Pouzat. *Mooc Reproducible research*. URL: https://lms.fun-mooc.fr/courses/course-v1:inria+41023+session01/info. cit. on p. A1

[Lit+17] Marin Litoiu, Mary Shaw, Gabriel Tamura, et al. "What can control theory teach us about assurances in self-adaptive software systems?" In: *Software Engineering for Self-Adaptive Systems III. Assurances: International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*. Springer. 2017, pp. 90–134. cit. on p. 17

[Lit23] Alexandre Lithaud. "Contribution au projet NixOS Compose". MA thesis. Université Grenoble Alpes, Sept. 2023. URL: https://inria.hal.science/hal-04197720. cit. on p. 154

[Liu+11] Yonggang Liu, Renato Figueiredo, Dulcardo Clavijo, Yiqi Xu, and Ming Zhao. "Towards simulation of parallel file system scheduling algorithms with PFSsim". In: *Proceedings of the 7th IEEE International Workshop on Storage Network Architectures and Parallel I/O (May 2011)*. 2011. cit. on p. 169

[Liu+20] Gang Liu, Zheng Xiao, GuangHua Tan, Kenli Li, and Anthony Theodore Chronopoulos. "Game theory-based optimization of distributed idle computing resources in cloud environments". In: *Theoretical Computer Science* 806 (2020), pp. 468–488. cit. on p. 12

[LLM87] Michel J Litzkow, Miron Livny, and Matt W Mutka. *Condor-a hunter of idle workstations*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 1987. cit. on p. 12

[Lus] Lustre. *Lustre Filesystem*. Accessed: 2023-09-29. URL: https://www.lustre.org/. cit. on p. 154

[MC19] Tessema M Mengistu and Dunren Che. "Survey and taxonomy of volunteer computing". In: *ACM Computing Surveys (CSUR)* 52.3 (2019), pp. 1–35. cit. on p. 11

[MC20] Ali Mohammed and Florina M Ciorba. "SimAS: A simulation-assisted approach for the scheduling algorithm selection under perturbations". In: *Concurrency and computation: practice and experience* 32.15 (2020), e5648. cit. on p. 17

[Mer+17] Michael Mercier, David Glesser, Yiannis Georgiou, and Olivier Richard. "Big data and HPC collocation: Using HPC idle resources for Big Data analytics". In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 347–352. cit. on pp. 13, 81, 100

[Mer18] [SW] Michael Mercier, 2018. URL: https://gitlab.inria.fr/batsim/batbroker, SWHID: ⟨swh:1:dir:428e9cbe73f2da728c1166bdf8f6290af37d8be1;origin=https://gitlab.inria.fr/batsim/batbroker⟩. cit. on p. 161

[Mer19] Michael Mercier. “Contribution to High Performance Computing and Big Data Infrastructure Convergence”. en. PhD Thesis. Universite Grenoble Alpes, 2019. cit. on p. 161

[MF01] Ahuva W. Mu’alem and Dror G. Feitelson. “Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling”. In: *IEEE transactions on parallel and distributed systems* 12.6 (2001), pp. 529–543. cit. on pp. 3, 12, 95

[MFR18] Michael Mercier, Adrien Faure, and Olivier Richard. “Considering the Development Workflow to Achieve Reproducibility with Variation”. In: *SC 2018-Workshop: ResCuE-HPC*. 2018, pp. 1–5. cit. on p. 109

[Mid+88] Richard H Middleton, Graham C Goodwin, David J Hill, and David Q Mayne. “Design issues in adaptive control”. In: *IEEE transactions on automatic control* 33.1 (1988), pp. 50–58. cit. on p. 78

[Mon+22] Julien Monniot, François Tessier, Matthieu Robert, and Gabriel Antoniu. “StorAlloc: A Simulator for Job Scheduling on Heterogeneous Storage Resources”. In: *European Conference on Parallel Processing*. Springer. 2022, pp. 211–222. cit. on p. 169

[Mug03] Vito MR Muggeo. “Estimating regression models with unknown break-points”. In: *Statistics in medicine* 22.19 (2003), pp. 3055–3071. cit. on p. 153

[Myt+09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. “Producing wrong data without doing anything obviously wrong!” In: *ACM Sigplan Notices* 44.3 (2009), pp. 265–276. cit. on p. 109

[Net+19] Alessio Netti, Micha Müller, Axel Auweter, et al. “From facility to application sensor data: modular, continuous and holistic monitoring with DCDB”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–27. cit. on p. 53

[Nix22] [SW] NixOS, *NixOps* 2022. LIC: LGPL-3.0. URL: `https://github.com/NixOS/nixops`, SWHID: ⟨`swh:1:dir:1096fd578c5f898d1b883239bbffc1ba4d5562c1;origin=https://github.com/NixOS/nixops`⟩. cit. on p. 119

[Nix23] [SW] NixOS, *nixpkgs* 2023. URL: `https://github.com/nixos/nixpkgs`, SWHID: ⟨`swh:1:dir:3ae910dfaee09a77c726f69f7c6519488e2bd9c3;origin=https://github.com/NixOS/nixpkgs`⟩. cit. on p. 114

[OAR23] [SW] OAR-Team, *NUR-Kapack* 2023. URL: `https://github.com/oar-team/nur-kapack`, SWHID: ⟨`swh:1:dir:1a6970dbce78a86062648a7c76978f674f136607;origin=https://github.com/oar-team/nur-kapack`⟩. cit. on p. 114

[oar23a] [SW] oar-team, 2023. URL: `https://github.com/oar-team/cigri`, SWHID: ⟨`swh:1:dir:f8b6df45300534e9807addc7ec125cd72b92c139;origin=https://github.com/oar-team/cigri`⟩. cit. on p. 23

[oar23b] [SW] oar-team, 2023. URL: `https://github.com/oar-team/oar3`, SWHID: ⟨`swh:1:dir:ab502da78090e4d77ee363ce7ccc2f6ea65c2d92;origin=https://github.com/oar-team/oar3`⟩. cit. on pp. 23, 70

[oar23c] [SW] oar-team, 2023. URL: https://github.com/oar-team/colmet, SWHID: ⟨swh:1:dir:eef757f42afc1c6032921ea4f56e4999aa2fd63d;origin=https://github.com/oar-team/colmet⟩. cit. on p. 53

[Ope] OpenWhisk. *OpenWhisk*. URL: https://openwhisk.apache.org/ (visited on Apr. 25, 2023). cit. on p. 13

[Ope13] LLC OpenStack. "OpenStack". In: *Apache Licence* 2 (2013), p. 86. cit. on p. 139

[Pag23] Rosa Pagano. *Controlling unused resources for digital sobriety*. forthcoming. Nov. 2023. cit. on p. 66

[Paw+00] Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow. "The NFS version 4 protocol". In: *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*. Citeseer. 2000. cit. on p. 145

[Pon+18] Vitchyr Pong, Shixiang Gu, Murtaza Dalal, and Sergey Levine. "Temporal Difference Models: Model-Free Deep RL for M odel-Based Control". In: (2018). arXiv: 1802.09081 [cs.LG]. cit. on p. 68

[Poq17] Millian Poquet. "Simulation approach for resource management". PhD thesis. Université Grenoble Alpes, 2017. cit. on p. 101

[PRD20] Barry Porter, Roberto Rodrigues Filho, and Paul Dean. "A survey of methodology in self-adaptive systems research". In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE. 2020, pp. 168–177. cit. on pp. 15, 16

[Prz+22] Bartłomiej Przybylski, Maciej Pawlik, Paweł Żuk, et al. "Using unused: non-invasive dynamic FaaS infrastructure with HPC-whisk". In: *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2022, pp. 1–15. cit. on pp. 13, 81

[Ram+19] Srinivasan Ramesh, Swann Perarnau, Sridutt Bhalachandra, Allen D Malony, and Pete Beckman. "Understanding the impact of dynamic power capping on application progress". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 793–804. cit. on p. 101

[RMS17] Eric Rutten, Nicolas Marchand, and Daniel Simon. "Feedback control as MAPE-K loop in autonomic computing". In: *Software Engineering for Self-Adaptive Systems III. Assurances: International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*. Springer. 2017, pp. 349–373. cit. on p. 17

[Ros+20] Daniel Rosendo, Pedro Silva, Matthieu Simonin, Alexandru Costan, and Gabriel Antoniu. "E2Clab: Exploring the Computing Continuum through Repeatable, Replicable and Reproducible Edge-to-Cloud Experiments". In: *Cluster 2020 - IEEE International Conference on Cluster Computing*. Kobe, Japan, Sept. 2020, pp. 1–11. DOI: 10.1109/CLUSTER49012.2020.00028. URL: https://hal.science/hal-02916032. cit. on p. 109

[Rui+15] Cristian Ruiz, Salem Harrache, Michael Mercier, and Olivier Richard. "Reconstructable Software Appliances with Kameleon". en. In: *ACM SIGOPS Operating Systems Review* 49.1 (Jan. 2015), pp. 80–89. DOI: 10.1145/2723872.2723883. URL: https://dl.acm.org/doi/10.1145/2723872.2723883 (visited on June 12, 2020). cit. on pp. 69, 111

[RW18] David Randall and Christopher Welser. *The Irreproducibility Crisis of Modern Science. Causes, Consequences, and the Road to Reform*. en. New York: National Association of Scholars, 2018. URL: https://www.nas.org/reports/the-irreproducibility-crisis-of-modern-science. cit. on p. 107

[ser23] [SW] serokell, 2023. URL: https://github.com/serokell/deploy-rs, SWHID: ⟨swh:1:dir:d28ae84202e328ef087cff85d9c6748ad9b19080;origin=https://github.com/serokell/deploy-rs⟩. cit. on p. 119

[SG19] Aadirupa Saha and Aditya Gopalan. "Combinatorial bandits with relative feedback". In: *Advances in Neural Information Processing Systems* 32 (2019). cit. on p. 97

[She+17] Stepan Shevtsov, Mihaly Berekmeri, Danny Weyns, and Martina Maggio. "Control-theoretical software adaptation: A systematic literature review". In: *IEEE Transactions on Software Engineering* 44.8 (2017), pp. 784–810. cit. on pp. 20, 66

[Shv+10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The hadoop distributed file system". In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, pp. 1–10. cit. on p. 13

[SK05] David Skinner and William Kramer. "Understanding the causes of performance variability in HPC workloads". In: *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.* IEEE. 2005, pp. 137–149. cit. on p. 3

[SK18] Victoria Stodden and Matthew S Krafczyk. "Assessing reproducibility: An astrophysical example of computational uncertainty in the HPC context". In: *Proceedings of the 1st Workshop on Reproducible, Customizable and Portable Workflows for HPC at SC*. Vol. 18. 2018. cit. on p. 109

[SMM18] Josef Spillner, Cristian Mateos, and David A Monge. "Faaster, better, cheaper: The prospect of serverless scientific computing and hpc". In: *High Performance Computing: 4th Latin American Conference, CARLA 2017, Buenos Aires, Argentina, and Colonia del Sacramento, Uruguay, September 20-22, 2017, Revised Selected Papers 4*. Springer. 2018, pp. 154–168. cit. on p. 13

[Sou+19] Abel Souza, Mohamad Rezaei, Erwin Laure, and Johan Tordsson. "Hybrid resource management for HPC and data intensive workloads". In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2019, pp. 399–409. cit. on p. 13

[SRI16] Supreeth Shastri, Amr Rizk, and David Irwin. "Transient guarantees: Maximizing the value of idle cloud capacity". In: *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2016, pp. 992–1002. cit. on p. 12

[ST17] Tim Shaffer and Douglas Thain. “Taming metadata storms in parallel filesystems with metaFS”. In: *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*. 2017, pp. 25–30. cit. on p. 170

[Sta+04] David A Stainforth, Myles R Allen, David Frame, et al. “Climateprediction. net: a global community for research in climate physics”. In: *Environmental online communication*. Springer, 2004, pp. 101–112. cit. on p. 11

[Sta+18] Emmanuel Stahl, Agustin Gabriel Yabo, Olivier Richard, et al. “Towards a control-theory approach for minimizing unused grid resources”. In: *Proceedings of the 1st International Workshop on Autonomous Infrastructure for Science*. 2018, pp. 1–8. cit. on p. 25

[Sto+21] Mathieu Stoffel, François Broquedis, Fré déric Desprez, and Abdelhafid Mazouz. “Phase-TA: Periodicity Detection and Characterization for HPC Applications”. In: *HPCS 2020-18th IEEE International Conference on High Performance Computing and Simulation*. IEEE. 2021, pp. 1–12. cit. on p. 100

[Sze10] Miklos Szeredi. “FUSE: Filesystem in userspace”. In: *http://fuse. sourceforge. net* (2010). cit. on p. 170

[Tar+23] Ahmad Tarraf, Alexis Bandet, Francieli Boito, Guillaume Pallez, and Felix Wolf. “FTIO: Detecting I/O Periodicity Using Frequency Techniques”. In: *arXiv preprint arXiv:2306.08601* (2023). cit. on p. 100

[Ter+17] Théophile Terraz, Alejandro Ribes, Yvan Fournier, Bertrand Iooss, and Bruno Raffin. “Melissa: Large Scale In Transit Sensitivity Analysis Avoiding Intermediate Files”. In: *The International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing)*. Denver, United States, Nov. 2017, pp. 1–14. URL: https://hal.inria.fr/hal-01607479. cit. on p. 127

[toh22] [SW] tohojo, *Overview — Flent: The FLExible Network Tester* 2022. URL: https://flent.org/, SWHID: ⟨swh:1:dir:b31ce4d599147041ab4bb658e521620f3525a059;origin=https://github.com/tohojo/flent⟩. cit. on p. 136

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. “Distributed computing in practice: the Condor experience”. In: *Concurrency and computation: practice and experience* 17.2-4 (2005), pp. 323–356. cit. on p. 81

[Twe] Tweag.io. *What problems do flakes solve?* https://www.tweag.io/blog/2020-05-25-flakes/. Accessed: 2023-04-04. cit. on p. 115

[Wan+21] Yawen Wang, Kapil Arya, Marios Kogias, et al. “Smartharvest: Harvesting idle cpus safely and efficiently in the cloud”. In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 1–16. cit. on p. 12

[Wei+06] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. “Ceph: A scalable, high-performance distributed file system”. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 307–320. cit. on p. 154

[XKC] XKCD. *Compiling*. https://xkcd.com/303/. cit. on p. 107

[Yab+19] Agustin Gabriel Yabo, Bogdan Robu, Olivier Richard, Bruno Bzeznik, and Eric Rutten. “A control-theory approach for cluster autonomic management: maximizing usage while avoiding overload”. In: *2019 IEEE Conference on Control Technology and Applications (CCTA)*. IEEE. 2019, pp. 189–195. cit. on p. 25

[YJG03] Andy B. Yoo, Morris A. Jette, and Mark Grondona. “SLURM: Simple Linux Utility for Resource Management”. en. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, et al. Vol. 2862. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. DOI: 10.1007/10968987_3. URL: http://link.springer.com/10.1007/10968987_3 (visited on Nov. 22, 2021). cit. on pp. 13, 125, 129, 142

[Zak+22] Farid Zakaria, Thomas RW Scogland, Todd Gamblin, and Carlos Maltzahn. “Mapping out the HPC dependency chaos”. In: *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2022, pp. 1–12. cit. on p. 170

[zen] zenodo. *Zenodo*. Accessed: 2023-03-30. URL: https://zenodo.org/. cit. on p. 109

List of Figures

List of Tables

Abstract

High-Performance Computing (HPC) systems have become increasingly more complex, and their performance and power consumption make them less predictable. This unpredictability requires cautious runtime management to guarantee an acceptable Quality-of-Service to the end users. Such a regulation problem arises in the context of the computing grid middleware CiGri that aims at harvesting the idle computing resources of a set of cluster by injection low priority jobs. A too aggressive harvesting strategy can lead to the degradation of the performance for all the users of the clusters, while a too shy harvesting will leave resources idle and thus lose computing power. There is thus a tradeoff between the amount of resources that can be harvested and the resulting degradation of users jobs, which can evolve at runtime based on Service Level Agreements and the current load of the system.

We claim that such regulation challenges can be addressed with tools from Autonomic Computing, and in particular when coupled with Control Theory. This thesis investigates several regulation problems in the context of CiGri with such tools. We will focus on regulating the harvesting based on the load of a shared distributed file-system, and improving the overall usage of the computing resources. We will also evaluate and compare the reusability of the proposed control-based solutions in the context of HPC systems.

The experiments done in this thesis also led us to investigate new tools and techniques to improve the cost and reproducibility of the experiments. We will present a tool named NixOS-compose able to generate and deploy reproducible distributed software environments. We will also investigate techniques to reduce the number of machines needed to deploy experiments on grid or cluster middlewares, such as CiGri, while ensuring an acceptable level of realism for the final deployed system.

Résumé

Les systèmes de calcul haute performance (HPC) sont devenus de plus en plus complexes, et leurs performances ainsi que leur consommation d'énergie les rendent de moins en moins prévisibles. Cette imprévisibilité nécessite une gestion en ligne et prudente, afin garantir une qualité de service acceptable aux utilisateurs. Un tel problème de régulation se pose dans le contexte de l'intergiciel de grille de calcul CiGri qui vise à récolter les ressources inutilisées d'un ensemble de grappes via l'injection de tâches faiblement prioritaires. Une stratégie de récolte trop agressive peut conduire à la dégradation des performances pour tous les utilisateurs des grappes, tandis qu'une récolte trop timide laissera des ressources inutilisées et donc une perte de puissance de calcul. Il existe ainsi un compromis entre la quantité de ressources pouvant être récoltées et la dégradation des performances pour les tâches des utilisateurs qui en résulte. Ce compromis peut évoluer au cours de l'exécution en fonction des accords de niveau de service et de la charge du système.

Nous affirmons que de tels défis de régulation peuvent être résolus avec des outils issus de l'informatique autonomique, et en particulier lorsqu'ils sont couplés à la théorie du contrôle. Cette thèse étudie plusieurs problèmes de régulation dans le contexte de CiGri avec de tels outils. Nous nous concentrerons sur la régulation de la récolte de ressources libres en fonction de la charge d'un système de fichiers distribué partagé et sur l'amélioration de l'utilisation globale des ressources de calcul. Nous évaluerons et comparerons également la réutilisabilité des solutions proposées dans le contexte des systèmes HPC.

Les expériences réalisées dans cette thèse nous ont par ailleurs amené à rechercher de nouveaux outils et techniques pour améliorer le coût et la reproductibilité des expériences. Nous présenterons un outil nommé NixOS-compose capable de générer et de déployer des environnements logiciels distribués reproductibles. Nous étudierons de plus des techniques permettant de réduire le nombre de machines nécessaires pour expérimenter sur des intergiciels de grappe, tels que CiGri, tout en garantissant un niveau de réalisme acceptable pour le système final déployé.