# Mitigating Shared Storage Congestion Using Control Theory

Thomas Collignon, Kouds Halitim, Raphaël Bleuse, Sophie Cerf, Bogdan Robu, Éric Rutten, Lionel Seinturier, Alexandre van Kempen

**To cite this version:**

Thomas Collignon, Kouds Halitim, Raphaël Bleuse, Sophie Cerf, Bogdan Robu, et al.. Mitigating Shared Storage Congestion Using Control Theory. UCC 2025 - 18th IEEE/ACM International Conference on Utility and Cloud Computing, Dec 2025, Nantes, France. pp.1-10, ⟨10.1145/3773274.3774277⟩. ⟨hal-05368563⟩

**HAL Id: hal-05368563**

**https://hal.science/hal-05368563v1**

Submitted on 18 Nov 2025

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

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

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

# Mitigating Shared Storage Congestion Using Control Theory

Thomas Collignon
Qarnot Computing
Montrouge, France
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
Lille, France
thomas.collignon@qarnot-computing.com

Kouds Halitim
Univ. Grenoble Alpes, Inria, CNRS, LIG
Grenoble, France
kouds.halitim@inria.fr

Raphaël Bleuse
Univ. Grenoble Alpes, Inria, CNRS, LIG
Grenoble, France
raphael.bleuse@inria.fr

Sophie Cerf
Univ. Grenoble Alpes, Inria, CNRS, LIG
Grenoble, France
sophie.cerf@inria.fr

Bogdan Robu
Univ. Grenoble Alpes, CNRS, Grenoble INP, GIPSA-lab
Grenoble, France
bogdan.robu@gipsa-lab.grenoble-inp.fr

Éric Rutten
Univ. Grenoble Alpes, Inria, CNRS, LIG
Grenoble, France
eric.rutten@inria.fr

Lionel Seinturier
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
Lille, France
lionel.seinturier@inria.fr

Alexandre van Kempen
Qarnot Computing
Montrouge, France
alexandre.vankempen@qarnot-computing.com

## Abstract

Efficient data access in High-Performance Computing (HPC) systems is essential to the performance of intensive computing tasks. Traditional optimizations of the I/O stack aim to improve peak performance but are often workload specific and require deep expertise, making them difficult to generalize or re-use. In shared HPC environments, resource congestion can lead to unpredictable performance, causing slowdowns and timeouts. To address these challenges, we propose a self-adaptive approach based on Control Theory to dynamically regulate client-side I/O rates. Our approach leverages a small set of runtime system load metrics to reduce congestion and enhance performance stability. We implement a controller in a multi-node cluster and evaluate it on a real testbed under a representative workload. Experimental results demonstrate that our method effectively mitigates I/O congestion, reducing total runtime by up to 20% and lowering tail latency, while maintaining stable performance.

## CCS Concepts

• **Hardware → Sensors and actuators**; • **Computer systems organization → Cloud computing**; *Client-server architectures*; *Real-time system architecture*.

## Keywords

Cloud Computing, Control Theory, Storage, HPC, Congestion, Cluster

## 1 Introduction

High-Performance Computing (HPC) systems play an increasingly crucial role in scientific discovery, complex simulations, fast prototyping, and other computationally intensive tasks. Performance in these systems heavily depends on efficient data access, making fast and scalable parallel I/O a critical requirement for many HPC applications. To meet this demand, the HPC I/O stack has grown highly complex, spanning multiple layers—including hardware, filesystems, and middleware—each with thousands of tunable parameters (see [4] for a comprehensive survey). Traditional optimization approaches focus on maximizing peak I/O performance through exhaustive parameter tuning, sophisticated benchmarking tools [13], and, more recently, machine learning techniques [2, 14]. While such tuning can yield significant performance gains, it is usually workload-specific, requires deep expertise, and does not generalize easily across diverse HPC environments.

In the context of shared HPC infrastructures (e.g., cloud-based HPC), multiple users and applications may compete for limited storage resources, leading to resource congestion that can significantly degrade system-wide performance [8]. Storage congestion results in highly unpredictable performance, causing random slowdowns and timeouts. In many cases, unpredictability can be more problematic than consistently lower but stable performance. In this context, we argue that preventing performance degradation due to resource congestion can sometimes be more important than maximizing peak I/O speed for individual applications. Several studies follow this approach by trying to mitigate I/O interferences [6, 15].

However, this requires an *a priori* knowledge about the I/O patterns [3] of the applications, which is hardly available.

To mitigate congestion, we propose a coarse-grained, self-adaptive approach that relies on control theory. Rather than fine-tuning each layer of the I/O stack, our method dynamically regulates client-side I/O rates using real-time system load metrics. By formulating the problem within a feedback-control framework, we abstract away much of the stack's inherent complexity. This paper makes the following contributions:

- We present a generic control-theoretic architecture for designing a controller that employs an actuator–sensor pair to dynamically adjust load on shared storage, ensuring more stable and predictable performance.
- We describe the implementation of a simple proportional–integral (PI) feedback controller on a computing cluster, including guidelines for tuning it to achieve effective regulation and noise reduction.
- We provide experimental results from a deployment on a real testbed with a representative write-intensive workload.
- We discuss paths we have identified for further improvements, including noise reduction, approaches beyond simple PI control e.g., adaptivity to workload variations, and fully distributed designs.

The rest of the paper is organized as follows. Section 2 presents the background and related works. Section 3 details the design and implementation of the controller. Section 4 describes how we evaluated our system and presents the experimental results. Section 5 proposes discussions and identified perspectives before concluding the paper.

## 2 Background and Motivation

HPC systems are designed to solve large and complex problems by coordinating multiple computing nodes. These computing nodes work collaboratively on tightly coupled tasks, frequently exchanging intermediate data and accessing common datasets. To enable this collaboration, they rely on a shared storage space that must deliver both high throughput and low latency. This shared storage space is usually accessed by the clients via a networked filesystem such as NFS, CephFS, LustreFS, etc.

As an example, Figure 1 depicts an overview of the I/O path in the case of the NFS protocol. NFS allows clients to access files stored on a remote server as if they were local. The NFS client sends file operation requests (open, read, write) over the network to the NFS server, which manages the shared storage space. The server translates these requests into local file system operations. At a lower level, the block I/O layer handles scheduling and queuing of read/write requests, ensuring efficient access to storage. Finally, these requests reach the physical storage devices (e.g., HDDs, SSDs), where data is stored and retrieved, then sent back through the stack to the client. The efficiency of this networked filesystem is critical: any bottleneck can slow down the entire computation.

### 2.1 I/Os and Congestion

The I/O stack is composed of multiple layers, from the application and file system down to device drivers and storage hardware. Each layer introduces its own policies and optimizations, such as buffering, caching, and scheduling, to improve performance [4]. However, these mechanisms interact in complex ways, making overall system behavior difficult to predict. When workloads are dynamic and resource contention arises, these interactions can amplify congestion and degrade performance. I/O congestion can stem from multiple sources: the client, the network, the server, or a combination of these [6, 15]. Even more challenging, congestion at one layer can propagate to others, creating contention elsewhere and producing cascading performance issues.

In this paper, we argue that I/O congestion can be mitigated through dynamic regulation of client-side I/O rates driven by a feedback loop. We advocate that this feedback loop should adopt an end-to-end perspective, managing performance across the entire path from right under the application to the storage device. By abstracting away the complexity of intermediate layers, such an approach enables coordinated adaptation that addresses congestion holistically rather than through layer specific optimizations.

This approach enables to support a wide range of hardware platforms and system architectures, which fine-tuning every layer of the stack would prevent. In addition, it avoids relying on prior knowledge about I/O profiles or access patterns (see [3] for a comprehensive survey) that might be hard to obtain as these are determined by users at submission. This uncertainty prevents us from pre-configuring system parameters effectively.

### 2.2 Feedback Loop

Feedback regulation has been widely used to dynamically monitor computing systems' performance and adjust their behavior to meet desired objectives [10]. Examples include the MAPE-K loop in autonomic computing [11] and control-theory-based feedback mechanisms [7, 9].

In our system, a closed-loop control mechanism is employed as shown in the bottom of Figure 1, where we continuously monitor performance at run-time through appropriate sensors (metrics), detailed in Section 3.1. These measurements are compared against a predefined reference state, that generates an error signal, quantifying deviation from desired behavior.

To correct the deviation, the controller uses an actuator to dynamically adjust system settings in response to the error signal. The actuator executes the control action, allowing the system to adapt at run-time. This feedback loop operates continuously to achieve the control objectives specified in Figure 2, ensuring that the system meets user-defined performance criteria: (i) *reference tracking*, which ensures that the system's output, represented by the black curve in the figure, closely follows the desired reference value with minimal *steady-state error* – the residual discrepancy at $t \to \infty$ between reference and actual output, (ii) the system must achieve a fast and stable response, meaning the output should

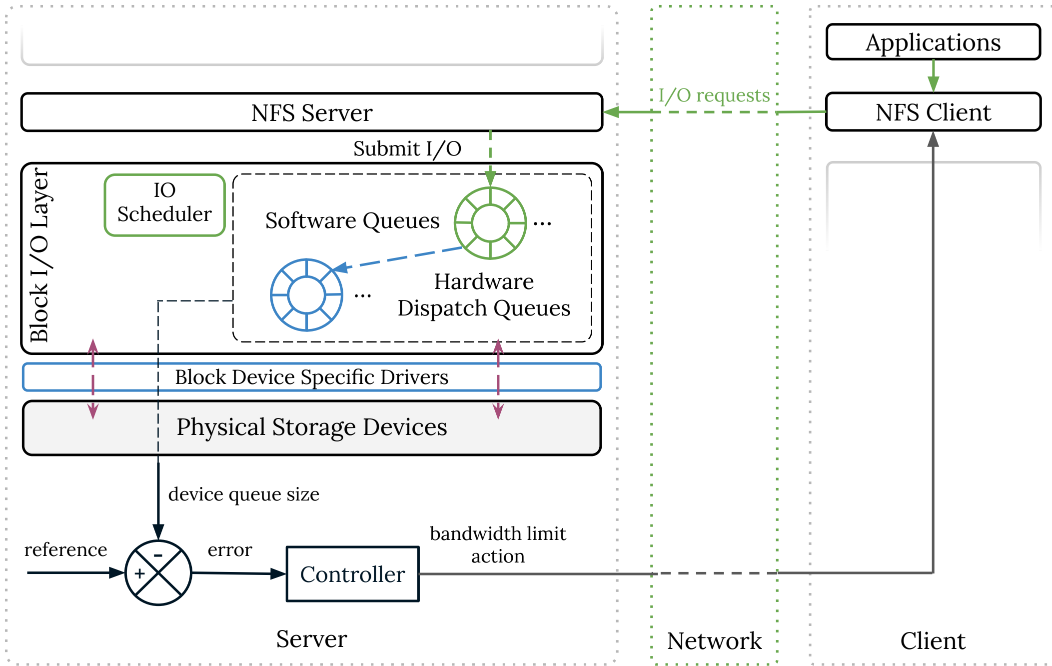



**Figure 1: Overview of the control on the I/O path on computing cluster**

reach the reference value with a *short settling time*, defined as the time required for the output to remain within 5% of its final value, with minimal oscillations and *overshoot*, meaning the system avoids rising excessively above the reference when the input changes.

By achieving these objectives, the system can autonomously optimize performance, maintain stability, and respond efficiently to workload variations and unexpected disturbances. The translation of those objectives in the I/O congestion regulation problem that we consider is explained in the next section, where we present the necessary steps in building such a controller.

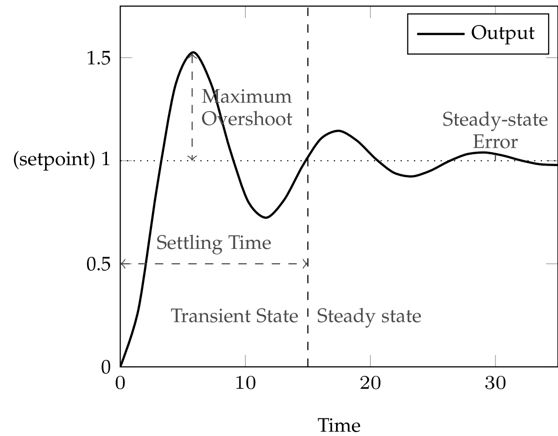



**Figure 2: Graphical explanation of the different properties of a controlled system (stability, settling time, overshoot) [12]**

## 3 Controller Design

This section describes the design and implementation of a controller that dynamically regulates client-side I/O rates to reach desired storage activity objectives. It details the common methodology for building a controller, adapted to the selected system. We first need to define the goals that we want to achieve, possible metrics that would help us characterize the disk activity of the server (see Section 3.1. We then have to select the actions that we will apply on the system to achieve the defined goals. Our choice is explained in Section 3.2. We discuss how these choices affect the implementation of the controller in our cluster environment in Section 3.3. After making those choices, it is possible to derive the model that the controller will use, defining the relationship between variations of actions and the selected system metrics. This is explained in Section 3.4, and reported experimentally in Section 4.2. Once a model is elected, we can design the controller, as in choosing which type of controller is appropriate in our context and what are the right parameters for it to obtain good results. This is described in Section 3.5. Once this is done, it is then possible to test the controller on a real context and validate it, as shown in Section 3.6.

Following this methodology, we obtain a control stack that we describe in Figure 1 and that we will reference throughout this section.

### 3.1 Choosing the sensors

Building a controller requires to identify a sensor, to quantify the congestion level of the shared storage. This measure should be done on the server side where the storage device resides. Control Theory requires the chosen sensor to fit multiple criteria for the controller design, e.g. continuous values, precision, etc. Dealing with I/Os also adds some time granularity or delay requirements as the implied I/O mechanisms are fast. The sensor should reflect as closely as possible the state of the storage device at a given time. Looking at the complex I/O stack, we focus on the layers closest to the disk as they are the first to be affected by disk congestion. We opt for not using the disk utilization rate as a congestion indicator because a rate of 100% does not necessarily mean that the disk is congested, simply that it is treating requests at its maximum speed, which is actually good for performance. Instead, we use the size of the dispatch queue of the targeted block device, on the block layer, to measure disk utilization and congestion level [2]. This queue fills up when I/O requests are waiting for the targeted disk to treat them, indicating that too many requests are sent over to the server. This sensor is illustrated on the left side of Figure 1 and is an input of the feedback loop.

### 3.2 Choosing the actuators

We then select an actuator, responsible of dynamically adjusting parameters on the clients to throttle executing workloads.

As well as for the sensor, there are requirements for the actuators that we should try to maintain. Mainly, the actuator needs to have an impact on the system and the measurement that we chose to qualify its state (i.e. dispatch queue size). It also preferably needs to have a short and stable response time and to support a continuous range of command values. On this cluster environment, we have to consider the network layers in the I/O stack. We make the choice to place the actuator on the client side, so that we prevent the network links from getting saturated, and act early on the I/O path to have a better impact on the congestion that we want to avoid. We choose to dynamically adjust an outgoing bandwidth limit on the client nodes as the action. We set it with the Linux tool `tc`, which allows to define a bandwidth limit using the *Token Bucket Filter*[1] algorithm that happens on lower layers of the kernel to manipulate the network packet queues. This actuator can be seen on client side in Figure 1. It does the transition from our controller to the system, by setting the bandwidth limit of the client.

### 3.3 Control architecture

The sensor and controller being on the server node, and the actuators being on the client nodes, we need to establish a one-way communication from the server to the clients in order to share the desired bandwidth limit action. For that, the controller sends the actions in a multicast group to which daemons running on the clients register. These daemons are responsible for updating the bandwidth limit on the host they are when they receive an action from the server.

For this approach, and because we study a specific workload that is run across all the clients, we apply the same action on all the clients. We are planning to study the relevance of this choice and how our controller setup can be adapted to support multiple concurrent workloads. We start a reflection about this in Section 5.2.

### 3.4 Deriving the model

In control system design, an open-loop model describes how the system evolves in response to its inputs without any feedback correction. Studying the open loop is important because it shows how the system naturally behaves, whether it is stable, and which parts of its behavior matter most for control. The chosen model should accurately capture the essential dynamics while remaining simple enough for effective control design. To use linear control theory, we approximate the system to a first-order linear model in discrete time, given by:

$$q(k+1) = a \cdot q(k) + b \cdot \mathrm{bw}(k) \tag{1}$$

where $q(k)$ is the dispatch queue size at time $k$, $\mathrm{bw}(k)$ is the bandwidth value at time $k$, and $a, b$ are identified system parameters.

### 3.5 Controller Implementation

We can now select what type of controller we want to implement. This choice is driven by the properties that we want to achieve (stability, responsiveness, etc.). Some controllers are less complex than others, considering or not past system states, or predicting future ones. Some require a model, or are only data-driven. They are also more or less sensitive to system perturbations such as network fluctuations, or other tasks targeting the shared storage that would affect our measurements while not being considered in our model. In this work, we do not consider perturbations.

For the control strategy, we use a discrete-time Proportional-Integral (PI) controller [1], chosen for its balance between responsiveness and stability. Its simple yet effective design makes it a widely adopted solution, with most industrial systems relying on it. The PI controller computes the next action to be applied on the system based on the error between the measured value of the sensor and its reference value $e[k]$. The proportional term provides immediate correction based on deviations (current error signal $e[k]$), ensuring fast response (reducing settling time). The integral term accumulates past errors to eliminate steady-state error (performance drift from target) over time as illustrated in the following equation:

$$\mathrm{bw}(k) = K_p e(k) + K_i T_s \sum_{j=0}^{k} e(j) \tag{2}$$

Where $K_p$ and $K_i$ are the PI controller gains, $e(k)$ is the error at discrete time step k, where k is an integer index corresponding to multiples of the sampling time $T_s$.

[1] https://linux.die.net/man/8/tc-tbf

The sampling time significantly affects both system stability and performance. If the sampling time is too short, the system becomes unstable due to large signal variations and sensitivity to random noise, while a larger one would cause measurement imprecision due to average on long periods. Setting the sampling time to $T_s = 300ms$ has experimentally shown a good trade-off.

Tuning of the controller gains $K_p$ and $K_i$ directly influences the performance objectives – reference tracking, stability, overshoot, and settling time explained in Section 2.2. Proper tuning also improves robustness, enabling the system to handle disturbances and varying workloads more effectively.

To tune the PI controller, we adopt a classic model-based methodology [5] that, given the approximated linear dynamic model of the system, maps the design specifications into the proportional and integral gains $K_p$ and $K_i$:

$$\begin{cases} K_p &= \frac{a-r^2}{b} \\ K_i &= \frac{1-2r\cos\theta+r^2}{b} \end{cases} \tag{3}$$

where:

$$\begin{aligned} r &= \exp\left(-\frac{4T_s}{K_s}\right) \\ \theta &= \pi \, \frac{\log r}{\log M_p} \end{aligned} \tag{4}$$

Here, $a$ and $b$ are the system parameters from Eq. (1) and $T_s$ is the sampling time, while $r \in (0, 1)$ and $\theta \in (0, \pi)$ correspond to the target transient response determined by the settling time $K_s$ and the overshoot $M_p$.

### 3.6 Validating the controller

Once every previous step is done, it is possible to finally implement the controller and validate it on the real system. Details on how we implement the sensor, actuator and controller are presented in Section 4.1. Validation is done by applying a wide range of control targets and see how the system reacts to them in terms of convergence, response time, overshoot, etc.

## 4 Control Experiments and Results

The objective of this section is to determine the benefits of using a control theory approach for mitigating congestion issues related to shared storage accesses in the context of a computing workload. Throughout the experiment, a synthetic workload was employed to reproduce the behavior of a write-intensive workload, such as checkpointing, which is widely used in HPC. Checkpointing consists in freezing computations in order to save the state of a task on a physical drive, allowing to restart the task from this state if needed. This type of workload is critical to the global performance of tasks, but it also has an overhead on their execution because of the pauses that are required to do the backup. On top of this, it involves all the computing nodes of a task and is well known for causing congestion on storage because of the large quantity of data that needs to be written.

The objective of the conducted experiments is to determine whether it is possible to improve performances of write-intensive workloads (as a first approach) by mitigating the congestion of the storage server using Control Theory. We present in this section the implementation of a controller on a real testbed by following the methodology that was explained in the previous section.

The source code for reproducing the following experiments can be found on this Software Heritage archive [2].

We firstly define the experimental setup that we use for the experiments in Section 4.1. Then, we present the results of the open loop experiment for the system identification in Section 4.2. We present the experimental validation of our controller in Section 4.3 and the importance of choosing the right controller gains in Section 4.4. We then study its impact on workload performances such as run time and stability (Section 4.5) and tail latency (Section 4.6) to conclude on the interest of using control theory in congestion regulation.

### 4.1 Experimental setup

The Grid'5000[3] testbed is used to deploy and test the controller. For each experiment, we reserve $n = 16$ computing nodes and one server from the same experimental cluster. The results were obtained on the *ecotype* cluster, in which every host has the same configuration, described in Table 1. A NFSv4 partition is shared across the $n$ clients so that the workloads executed on the clients are targeting the storage server.

For the write-intensive workload, we define a FIO job with sequential write I/Os using large block size and file size that is going to be executed simultaneously on all the computing nodes. The precise FIO settings are described in Listing 1. This workload entirely fills up the dispatch queue when no bandwidth restriction is applied on the clients. We say that the queue (or the system) is saturated.

We use Python for the entire control system by defining multiple main classes for root components of the control (ControllerPI, Sensor, Actuator). They are used to define sub classes, specific to a control configuration, depending on the chosen metrics. This makes the deployment of new controllers easier. We develop a sensor of the length of the dispatch queue using the `time_in_queue` metric from the *sysfs* stat file (`/sys/block/<dev>/stats`). For the actuator we need a sender and a receiver as the actions are sent through multicast. The receiver side consists in waiting for new messages in the group and in replacing the previous TBF limit with the new bandwidth value.

The controller computes the control parameters $K_p$ and $K_i$ from the previously established model of the system and the desired properties of the controller, including settling time and overshoot (i.e. the maximum peak value of the system response measured from the objective). It then starts the main loop, consisting of polling sensors and using actuators with the computed action.

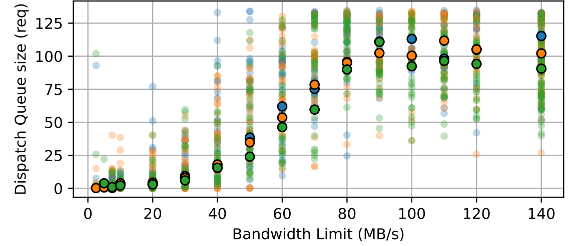
[2] swh:1:dir:0ba46bc22e2f1bed6a049c274ce401453f8c9d51
[3] https://www.grid5000.fr

**Table 1: Configuration of the ecotype cluster of Grid'5000**

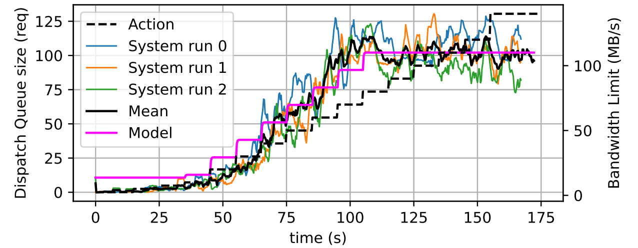
| Nodes | CPU | | | | Memory | Storage | Network |
|---|---|---|---|---|---|---|---|
| | # | Name | Cores | Architecture | | | |
| 48 | 2 | Intel Xeon E5-2630L v4 | 10 cores/CPU | x_86_64 | 128 GiB | 400 GB SSD | 10 Gbps (SR-IOV) |

**Listing 1: FIO job configuration**

```
rw=write
size=4g
bs=1024k
numjobs=4
ioengine=libaio
iodepth=16
```




**(a) Identification experiments to model Static behavior of the dispatch queue size (y-axis) for fixed values of bandwidth limit input (x-axis). Average dispatch queue size for each bandwidth limit action is displayed for each of the 3 runs (one color per run) for better visualization.**




**(b) Identification experiments to model dynamic behavior of the dispatch queue size (left y-axis): The model (magenta line) is fitted to the average (black line) of three dispatch queue response to bandwidth limit input variations (dashed black line). We filter the noise of the system with a rolling average over 10 points on the real data and use the effective range of actions to fit the model. The model fits well for certain operation region but relatively poorly for others**

**Figure 3: Open-Loop Experiments used for Identification – No Feedback or Control: The upper plot illustrates the system's static behavior under fixed values of input, while the lower plot depicts its dynamic response to input varying during run-time.**

## 4.2 System analysis

The first step before building the type of controller that we chose (PI controller) is to build a model of the studied system, that is, to find a relationship between action and system variations. The system identification results are shown in Figure 3. To understand the structure of the real system we observe the data pattern in the static behaviour (Fig. 3a) of the dispatch queue size for fixed values of bandwidth limit input.

We apply a wide range of actions (client bandwidth limits) following an increasing step function and measure the system metric (dispatch queue size) response while executing the selected workload on every client, over multiple iteration. We then fit the first-order linear model described in Section 3 after filtering the noise with a Savitzky-Golay filter over the collected measures. Also, the data where the queue is saturated and empty are excluded from the fitting phase so that the obtained model better captures the behavior of the queue. On Figure 3b, we obtain a model (magenta line) that is good-enough to build a controller, it overall follows the behavior of the real size of the dispatch queue while having a decent response time to the action.

## 4.3 Control results

This section presents the results obtained with the controller. The goal is to evaluate how effectively our control system can manage to reach any desired system state (level of dispatch queue) in a tolerable amount of time and overshoot.

For this experiment, we change the control target over time according to a step function. We measure the dispatch queue size (system metric) over time during the execution of the workload and compare it to the desired target. Results are shown in Figure 4, where the top plot shows the dispatch queue size measured by the sensor as well as a filtered signal with a rolling average, and the control target that that has to be reached. We also plot the average of the system for each segment of fixed targets in order to better visualize the steady state error and accuracy of the control. The bottom plot shows the action computed by the controller over time.

We can see that on average, the system follows the desired target despite the noisy behavior. Thanks to the averaged system values, we observe that the steady state error to the target is negligible. The response time is also very short but the noise makes it complicated to determine its value.

About the presence of noise in the results, we can look at the open loop results (Figure 3b), on which the same rolling averages were applied, and say that the system metric is inherently noisy, so it is to be expected that we observe

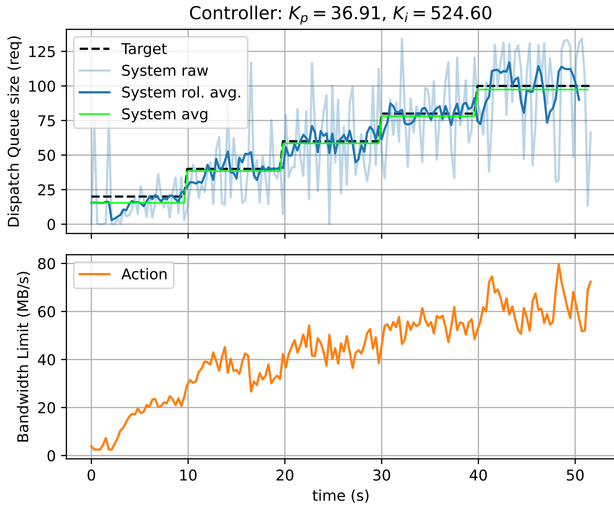



**Figure 4: Control results - Top plot represents the dispatch queue size (raw and rolling average) response to control target changes over time. Green line is the average of the system over fixed control targets, showing that the controller manages to reach the desired target on average. Bottom plot is the bandwidth limit action decided by the controller over time**

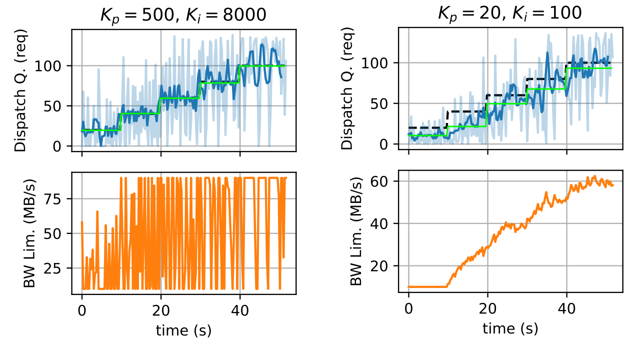
similar noise during control. We will be discussing the noise mitigation in Section 5.1.

From these results, we see that it is possible to make the system reach any specific state within the tolerated range of targets (limited by the size of the dispatch queue), in a short time. Despite some imprecision of the model and the measured noise, the controller is effective in the context of I/O request regulations, which have a variable and unpredictable behavior.

## 4.4 Tuning parameters despite noise

In this subsection, we study the importance of the controller gains ($K_P$ and $K_I$) in determining control quality, with respect to response time, oscillations, noise, and steady-state error. Figure 5 shows the results of multiple controller configurations. We observe that the action as well as the system response can be very noisy, as in Subfigure 5a, while still reaching the desired targets on average. Conversely, the action and response can be much less noisy but poorly accurate with respect to the target, with a very slow response, as shown in Subfigure 5b where the control gains are very small. This highlights the importance of finding optimal control gains to mitigate noise while maintaining system responsiveness.

We argue that the tuning methodology described in Section 3.5 could yield different behavior for two reasons: first, the real system is more complex and it approximated as a linear system around an operating point; second, the system is noisy in open loop, and this noise makes it difficult to be robust and achieve the desired closed-loop specifications.




**(a) Noisy control results** **(b) Slow control response and poor reference tracking**

**Figure 5: Control results with multiple control gain configurations. This shows the impact of the controller gains in the quality of the control.**

The results shown in Figure 4 were obtained with closed-loop specifications of $M_p = 0.02$ and $K_s = 1.4, \mathrm{s}$.

## 4.5 Performance benefits

In this section, we investigate the usefulness of a controller on mitigating the performance degradation caused by congestion. To do this, we run the same FIO workload as previously mentioned without any control and collect the `job_runtime` metric of FIO on all the clients. The `job_runtime` metric given in the output of FIO only considers the effective part of the workload, excluding the initialization (file creation, etc.) and cleanup phases (flushing I/O requests, etc.). We do the same measurements with a controller for which we define a fixed dispatch queue size target, and we test different values of that target. We repeat each run 5 times to remove any variability and build interpretation of the results.

Figure 6 shows the results of the execution without control (Baseline) in comparison to controlled executions with different configurations. We plot the job run time for all the clients and runs for the baseline and every control configuration. The mean of all the run times as well as the 10 and 90 percentiles are also displayed. Please note that the outliers across all the runs are not part of a single run, meaning that the disparity in the run times is part of the workload. We see that the choice of the target makes the overall run time of the FIO job vary. Under specific target choice (i.e. 70 or 80 requests in the dispatch queue), the average run time of the jobs on all clients is shorter than for the baseline for a best case scenario of a 20% average run time improvement with a fixed target of 80 requests. This shows that it is possible to improve the performances of a workload by choosing the right target value.

When looking at the disparity of the points, we can pretend that a good control target choice also leads to a reduced range of run time values across concurrent workloads, improving the overall performance of the workload and stability. Choosing

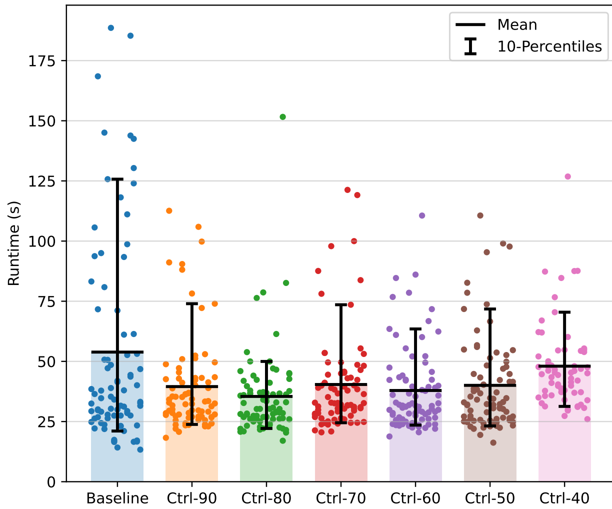



**Figure 6: Control performance overhead in comparison of a normal execution of the same FIO workload. Each color represents the execution of the FIO job with a controller whose target is set to a constant dispatch queue size ("Ctrl-90" is for the control with a 90 requests target).**

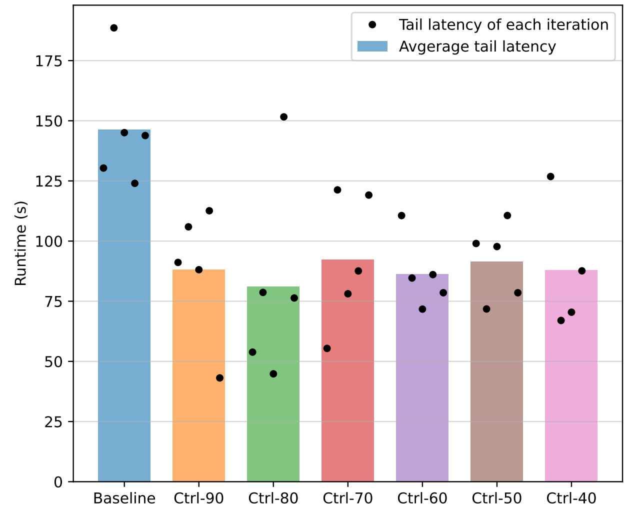



**Figure 7: Tail latency of the baseline and multiple control target configuration for each iteration and on average.**

the right target leading to performance improvement is still under investigation and will be discussed in Section 5.2.

### 4.6 Tail latency improvements

When considering concurrent processes across multiple hosts, a common performance metric is the tail latency – the longest run time across all clients – of the distributed jobs. For a checkpointing process, it corresponds to the pausing time of a task need for a checkpoint. Here, we use the same results as previously mentioned to study the tail latency benefits of our control solution. In Figure 7, the tail latency of every iteration is displayed as well as an average over all the iterations. For all the tested targets, the tail latency is smaller than the baseline's. A reduction of 35% of the tail latency of the baseline is achieved with a fixed control target of 70. It is possible to note that reducing the dispatch queue target, which implies to reduce the bandwidth limit of clients, can lead to less variability in the tail latency. This also illustrates the randomness of the consequences of storage congestion on workload performances.

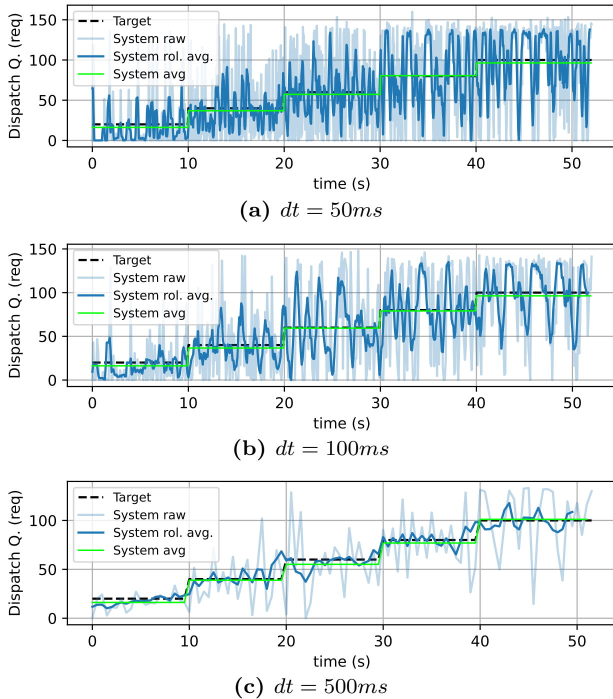
## 5 Discussions and Identified Perspectives

In this section, we discuss some choices and observations that we encountered throughout the experiments, as well as clearly identified perspectives on how to improve our work and results. Section 5.1 addresses the topic of noise mitigation. In Section 5.2, we discuss the portability and the workload adaptivity of the control that we implemented and ways to improve them. Finally, in Section 5.3, we discuss the structural choices that we made in terms of control strategies and hypothesis, and explore possible improvements.

### 5.1 Noise mitigation

Throughout the results that we showed in earlier sections, we observe a relatively important noise. Despite showing that this noise is not a response to a poorly defined controller (with similar noise being present during the open loop), possible ways exist to mitigate it. One first approach is to filter the noise out at different levels of the control stack. We implemented an average window on the measurement, or on the measured error. This led to smoother control action depending on the size of the window, but it induces delay in the response as previous measurements are used at each iteration. Furthermore, we sometimes observed oscillations with larger time periods than the noise. In future experiments, we will try new filters such as a Kalman filter, which is widely used in the control theory community.

We also studied the impact of sampling time on the control result. Figure 8 compares the control results with the same controller and different sampling times. We see that the noise is effectively smaller as the sampling time is larger, but it comes with a longer response time. This was to be expected as the dispatch queue sensor is in fact based on an average between two consecutive measurements of a base metric (the `time_in_queue` metric). There is a trade-off to find between the accuracy of our measure and its resolution. If the sampling time is too short, the measure will be sensitive to any small variation in our base metric, leading to increased noise and possibly false values. On the opposite, selecting a long sampling time leads to large period averages of the base metric, hiding the smaller high-frequency changes in the system and delaying the system variations.




(a) $dt = 50ms$

(b) $dt = 100ms$

(c) $dt = 500ms$

**Figure 8: Same controller with different sampling time. The noise amplitude changes with the sampling time.**

A possible workaround to avoid having that much noise in the measures but still have a reactive controller would be to have a dynamic sampling time, where the controller would use a short sampling time on target changes and short dispatch queue bursts, and a longer one as the system is stable. This would probably help reduce the noisy behavior on longer phases.

In the end, we see in Figure 8 that we manage to reach the desired targets on average for any sampling time. Although the noise is often wanted to be attenuated, it might not be necessary in our case. In the context of I/O operations, where there is a lot of variability and unpredictability, with fast-changing I/O loads and bursts, it is likely that filtering the noise out would not be of any interest nor possible.

## 5.2 Portability and workload adaptivity

One of our objectives with this controller solution is to be agnostic to both the architecture and the workload. Our control system is already independent of the architecture it is executing on, the only requirement for deploying the controller on another cluster is to run the open loop experiment and system identification because the change of hardware will likely change the orders of magnitude of the selected metrics and the control parameters.

About the workload adaptivity of our control system, more reflections need to be made. Until now, we have been doing the open loop and control with a single and identical workload on every client. This workload is also using only one kind of I/O request (write-only requests, fixed block size and I/O depth, etc.), which makes it convenient for building a working controller. We plan to consider more complex workload distributions and variety over clients. Considered strategies include (i) Changing the selected system metric to also consider characterizing properties of I/O requests such as the block size, the type of request (read, write, etc.) or the amount of nodes working for the same job, so that the defined model is more robust and supports more (preferably all) kinds of workload (ii) Using a type of controller that is model-agnostic, meaning that it would only be based on collected data of previous iterations. This would help dealing with the wide variety of workloads. (iii) Moving the controller on the client side for having multiple, separate actions on the clients, this would make the actions more adapted to the system. On top of this, we want to keep the controller simple to use, with the smallest human intervention possible, which consists up to this date in starting a process that will run an initial open loop phase to divide the model and build the controller, and then start it as a Linux service.

The question of the choice of the optimal control target still remains. It can be found manually, similarly to the study that we did in Section 4.5, but that is not a preferable solution since this value would be different for any configuration and workload. Also, having a constant target might not work for workloads with multiple I/O phases, which might require a dynamic target, likely determined by a new performance metric that would be used at run time, such as workload progression, or a more elaborate controller that would activate during intensive I/O phases.

## 5.3 Control architecture choices

When choosing to have a single workload for every client, we could easily deploy a server-side controller that would decide the allowed outgoing bandwidth of the clients, without differentiating them. This leads to acceptable results in terms of reaching a desired control target, and theoretically maintain fairness across clients in regard to that control. Such action on the system should firstly contain over-consuming clients without degrading the less active ones. When running diverse workloads on clients, it is to be expected that the optimal bandwidths to be allocated to them are going to be different. This rises the question of changing the controller design to support distributed actions.

As previously mentioned, one possible architecture would be to have a controller on each client, with each controller living independently and making decision for the client it is running on. The used model for the controllers could use server and client side metrics. The server dispatch queue measure could be sent over to clients, in the same way that the action is currently being shared, and additional metrics about I/O requests properties being sent by the NFS client could be measured directly on the clients with tools such as

`eBPF`[4] or `tcpdump`[5]. This would allow to have information on the current state of the storage and on the I/O activity of a client to better determine a suitable bandwidth limit.

In this theoretical setup, we would hope that the independent controllers lead to a stable system behavior that respects control objectives. But it is also possible that the resulting global action of all the controllers will not be appropriate for reaching the control objective, leading to over-throttling and performance degradation, or the opposite. Such control could then benefit from techniques such as agreement protocols or synchronization to allow the system to converge faster towards the desired state.

## 6 Conclusion and Future Works

In this paper, we addressed the challenges of I/O congestion and performance degradation in distributed clusters by proposing an approach using Control Theory that abstracts the complexity of the I/O stack. We detailed the implementation of a PI controller of the disk activity of a storage server that dynamically adjusts the outgoing bandwidth attributed to computing nodes to reach desired dispatch queue objectives on the server.

Experimental results carried out on the Grid'5000 testbed validated our approach for a typical write-intensive workload (similar to checkpointing). We reached up to 20% run time improvement on average and 35% tail latency reduction for specific fixed control targets, compared to an uncontrolled baseline execution of the same workload.

This study demonstrates that Control Theory is well suited for managing complex I/O dynamics in HPC.

Future works will aim to compare our approach to existing solutions such as Machine Learning or scheduling (e.g. sequentially executing the clients' workloads, removing any degradation caused by concurrence) as well as making the controller handle heterogeneous and more complex workloads. This will require working on more elaborate control strategies such as moving and distributing the controller on the clients, improving the robustness and workload adaptivity of the control model or using model-free approaches.

## Acknowledgments

Our work is done in the context of the Inria – Qarnot Pulse project (see https://www.inria.fr/en/pulse). It is supported by the ANRT (Association nationale de la recherche et de la technologie) with a CIFRE fellowship granted to Thomas Collignon. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several universities as well as other organizations (see https://www.grid5000.fr).

## References

[1] Karl J Astrom. 1995. PID controllers: theory, design, and tuning. *The international society of measurement and control* (1995).

[2] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. 2019. Optimizing I/O Performance of HPC Applications with Autotuning. *ACM Trans. Parallel Comput.* 5, 4, Article 15 (March 2019), 27 pages. doi:10.1145/3309205

[3] Jean Luca Bez, Suren Byna, and Shadi Ibrahim. 2023. I/O Access Patterns in HPC Applications: A 360-Degree Survey. *ACM Comput. Surv.* 56, 2, Article 46 (Sept. 2023), 41 pages. doi:10.1145/3611007

[4] Francieli Zanon Boito, Eduardo C. Inacio, Jean Luca Bez, Philippe O. A. Navaux, Mario A. R. Dantas, and Yves Denneulin. 2018. A Checkpoint of Research on Parallel I/O for High-Performance Computing. *ACM Comput. Surv.* 51, 2, Article 23 (March 2018), 35 pages. doi:10.1145/3152891

[5] Rakesh P. Borase, D. K. Maghade, S. Y. Sondkar, and S. N. Pawar. 2020. A review of PID control, tuning methods and applications. *International Journal of Dynamics and Control* 9, 2 (July 2020), 818–827. doi:10.1007/s40435-020-00665-4

[6] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. 2014. CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 155–164. doi:10.1109/IPDPS.2014.27

[7] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolas d'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, et al. 2015. Software engineering meets control theory. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 71–82.

[8] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. 2015. Scheduling the I/O of HPC Applications Under Congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 1013–1022. doi:10.1109/IPDPS.2015.116

[9] Quentin Guilloteau, Sophie Cerf, Raphaël Bleuse, Bogdan Robu, and Eric Rutten. 2024. Under Control: A Control Theory Introduction for Computer Scientists. In *ACSOS 2024 - 5th IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS 2024)*. IEEE, Aahrus, Denmark, 1–10. https://hal.science/hal-04666859

[10] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. 2004. *Feedback Control of Computing Systems*. John Wiley & Sons, Inc., Hoboken, NJ, USA.

[11] Eric Rutten, Nicolas Marchand, and Daniel Simon. 2018. Feedback Control as MAPE-K loop in Autonomic Computing. In *Software Engineering for Self-Adaptive Systems III. Assurances*. Lecture Notes in Computer Science, Vol. 9640. Springer, 349–373. doi:10.1007/978-3-319-74183-3_12

[12] Stepan Shevtsov, Mihaly Berekmeri, Danny Weyns, and Martina Maggio. 2017. Control-Theoretical Software Adaptation: A Systematic Literature Review. *IEEE Transactions on Software Engineering* PP (05 2017), 1–1. doi:10.1109/TSE.2017.2704579

[13] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *login Usenix Mag.* 41 (2016). https://api.semanticscholar.org/CorpusID:56553130

[14] Abdul Jabbar Saeed Tipu, Pádraig Ó Conbhuí, and Enda Howley. 2022. Artificial neural networks based predictions towards the auto-tuning and optimization of parallel IO bandwidth in HPC system. *Cluster Computing* 27, 1 (Dec. 2022), 71–90. doi:10.1007/s10586-022-03814-w

[15] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Robert B. Ross, and Gabriel Antoniu. 2016. On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems. In *IPDPS*. IEEE, 750–759. doi:10.1109/IPDPS.2016.50

[4] https://ebpf.io

[5] https://www.tcpdump.org/manpages/tcpdump.1.html