Emulation of Storage Performance in Testbed Experiments with Distem

Abdulqawi Saif, Alexandre Merlin, Olivier Dautricourt, Maël Houbre, Lucas Nussbaum, Ye-Qiong Song

Emulation of Storage Performance in Testbed Experiments with Distem

Abdulqawi Saif*, Alexandre Merlin*, Olivier Dautricourt*†, Maël Houbre†, Lucas Nussbaum*, Ye-Qiong Song*
* Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
† Telecom Nancy, École d'ingénieur du numérique F-54000, Nancy, France
{firstname.lastname}@loria.fr

***Abstract*—Together with the CPU and the network, storage plays an essential role in the overall performance of applications, especially in the context of big data applications, that can deal with enormous datasets. However, testbeds have scarce support for experiments involving storage performance. Experimenters can use the storage devices provided on the testbed directly, but (1) they might not provide the suitable performance characteristics for their experiments; (2) it might leave an uncontrolled bias over the experiments' results. In this paper, we explore the feasibility of using Linux's *control groups* to emulate I/O performance for testbed experiments. We then use it in the *Distem* emulator to create a customizable I/O experimental environment. Using *Distem*, we perform experiments on *Hadoop* to highlight the advantage of emulating I/O performance. Results obtained from a cluster of 25 nodes show how the performance of *Hadoop* changes according to emulated storage performance.**

***Index Terms*—experimentation, I/O performance, emulation, testbeds, experimental environments**

I. Introduction

Testbeds are a key asset for experimental research in computer science. They generally provide access to a large number of resources, trying to be representative of the hardware that is or was (or sometimes will be) available. However, despite this variety in resources characteristics (and their performance), experimenters remain limited to what is made available by testbeds, which might not match their needs. For instance, experimenting on a testbed offering high-speed networking and modern storage is not convenient if the target system is supposed to run on commodity hardware with moderate networking performance and rotational storage devices. The emulation of resources' performance can come to rescue in order to provide suitable environments for experiments.

Although emulation is widely applied for some resources such as CPUs, networks, and memory during experimentation, emulating I/O performance is not common in the testbeds' community. One could wonder about the behavior of big data systems on different scenarios where the I/O performance varies. However, almost no experimental tool can be used to achieve such experiments. Indeed, the existing I/O emulation tools focus on the proportional sharing of I/O throughput [1] [2] [3], targeting balanced or fair resources allocation rather than the experimentation use cases.

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.

The increasing importance of data makes it crucial to build experimental frameworks that enable emulation of I/O performance, in order to be able to investigate the I/O behavior of Big Data systems. Firstly, I/O operations are centric for this kind of systems, so testing them at scale under heterogeneous I/O performance can help with revealing potential performance issues, or getting a better understanding of their I/O behaviors. Secondly, emulating the I/O performance creates more testing configurations for experiments and avoids a bias about storage resources. For example, one could tune the performance of an *SSD* to act like an *HDD* in order to determine how the target system reacts accordingly.

In this paper, we firstly investigate the feasibility of using Linux's *control groups* (a.k.a *cgroups*) to emulate the I/O performance for testbed experiments. *cgroups* is intensively explored to achieve process isolation and containerization [4], [5] on the scale of a single node, but not yet examined to emulate the I/O performance for testbed experiments. We then use that technology to provide an I/O emulation service through extending the *Distem* emulator [6], which is available on different testbeds [7] such as *Grid'5000*, *CloudLab*, and *Chameleon*. Through that contribution[1], *Distem* allows 1) imposing static limitations at the start of experiments or 2) changing the I/O performance of involved nodes dynamically over time. We then perform experiments over *Hadoop* to highlight the advantage of emulating the I/O performance at scale. Results from a cluster with heterogeneous I/O performance show that *Hadoop* schedules map-reduce jobs on nodes (including those with moderate I/O performance).

The rest of this paper is organized as follows. Section 2 presents the *Distem* emulator briefly. Section 3 describes the challenges of using *cgroups* to emulate the I/O performance and how that emulation service is provided by *Distem*. Section 4 describes the experiments done on *Hadoop*. Section 5 discusses the related work while Section 6 concludes.

II. The Distem Emulator

The *Distem* emulator is an experimental framework enabling experimenters to build customized virtual experimental environments over physical infrastructure by leveraging the *Linux Containers* technology. *Distem* allows the creation of

[1] Scripts and results at: https://github.com/LeUnAiDeS/DistemIOEmulation

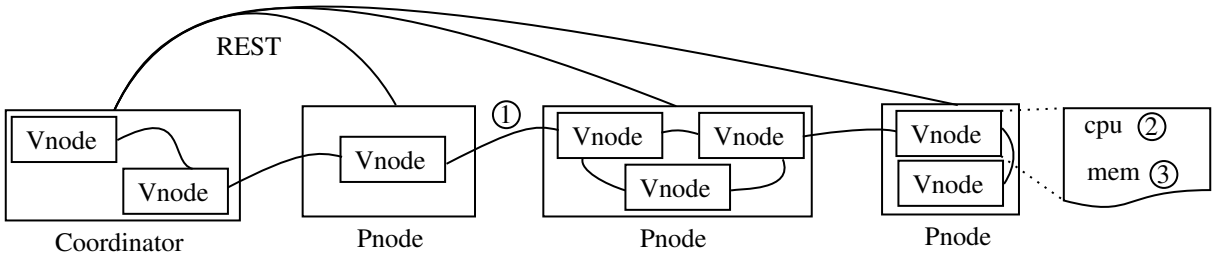

Fig. 1. An experimental environment created by the *Distem* emulator. It consists of eight virtual nodes (*vnodes*) on top of four physical nodes connected using a *REST API*. Experimenters can emulate the network performance, the CPU and the memory performance for any virtual node or all of them

one or several virtual nodes (*vnode*) over a physical node regarding the experiment needs.

Distem selects a physical node among the experimental infrastructure to be a coordinator in order to control the whole experimental environment. It receives the experimenter's requests and then forwards them to the corresponding *physical nodes*. The communication between the coordinator and the nodes is done via a *REST API* since every physical node hosts a *Distem* daemon. Each daemon also helps to control the set of local virtual nodes. Figure 1 shows an experimental environment created by the *Distem* emulator.

Creating a suitable experimental environment is not the only service offered by *Distem*. The emulator also gives experimenters the possibility to customize the environment by emulating the network, the CPU, and the memory performance. This is not only useful to increase the number of experimental configurations that can be obtained over the same infrastructure, but also ensuring to have a suitable environment for the target system under test.

Alternatives like *Mininet* [8] allow to create an experimental environment in a similar way as *Distem*. However, *Mininet* is limited to single-machine deployments (even if prototypes for multi-node Mininet exist), and thus does not scale to large experiments like Distem does. Also, it mainly focuses on networking experiments. Since scalability is a typical goal of testbed I/O experiments, *Distem* is more suitable to be extended for emulating I/O performance, thanks to its native distributed architecture.

III. I/O Performance Emulation

I/O performance emulation implies altering the allocated I/O throughput for one application or a group of applications. Limiting the discourse to the I/O operations on Linux, the suitable place for any I/O emulation approach is the Linux kernel as it hosts all the components in charge of treating the I/Os (e.g., the memory management system and several layers of the I/O stack such as the *filesystem* and the block I/Os). In contrast, when it comes to evaluating if the emulated throughput is applied or not, this should be done by measuring the I/O throughput from *userspace*, similarly to what target applications would go when generating I/O traffic.

The memory is involved in the management of I/O operations. Data could be written to the *page cache* instead of being committed to the storage devices immediately (e.g., using asynchronous I/Os). Similarly, data that was already read previously is kept in the *page cache*, resulting in much faster access in subsequent reads. Given that, when emulating I/O performance, it is essential to control the memory size of target machines, thus controlling the size of the *page cache*. The memory limitation, in turn, covers two main cases that possibly happen during the I/O emulation: 1) the interactions between the target applications and the memory during the execution of their I/O operations, and 2) the kernel's *write-back I/Os* used for writing the data permanently into disks (more details in the following paragraphs).

Any I/O operation has at least two main actors: the I/O process which initiates that operation and the Linux kernel which treats them. I/O processes select a suitable *system call* to deliver their I/O requests to the Linux kernel. The most-known *system calls* are *read & write* with many variations such *pread/pwrite*, *readv/writev*, etc. These *system calls* determines the I/O method that the application follows (e.g., sync I/O & vectored I/O), and the way that the kernel treats them. However, using any of them leads to one of four data communication cases. These cases are described as follows, regarding the I/O emulation context.

a) Read-from-disk: A straightforward operation where the *userspace* process initiates its request and communicates directly with the I/O device. The operation starts at the time of issuing the request, and it ends when the corresponding process is notified about the termination of data retrieval. Since the memory does not have any impact on this operation, it is sufficient to inject a desired I/O throughput value at the level of the block I/O layer to emulate the I/O performance.

b) Read-from-cache: When the I/O requests arrive at the *virtual file system* (*VFS*) layer on the I/O stack, the *VFS* checks if the data is already available in memory. In such a case, the *VFS* avoids further communication with lower layers of the I/O stack and retrieves the data from memory. In this case, injecting an I/O throughput value at the level of the block I/O layer has no sense as the I/O requests are served before reaching that layer. The I/O operation should proceed at the normal memory speed (without any slowdown caused

by emulation).

c) Write-to-disk-synchronously: In this case, I/O requests are supposed to bring feedback to the corresponding process after their termination, i.e., when the data is written permanently into the storage device. Like the *read-from-disk* case, this is a direct operation where the block I/O layer is the suitable place to adjust the I/O throughput value.

d) Write-to-cache: Writing data asynchronously can lead to activate this option, no matter which *system call* is used. Although this can lead to lost data if the machine is shut down, it accelerates the I/O activities of the *userspace* process. The I/O requests write the data to the *page cache* and return immediately. Then, the kernel notifies the corresponding process in *userspace* that the I/O operation is terminated while the data is still in memory and not yet written into the storage device. Asynchronously, the kernel will then proceed with the *write-back* operations in order to transfer the data from the *page cache* into the storage device. This step represents a challenge in case of measuring the I/O performance since the kernel writes the data into the storage support behind the scenes.

The read and write throughputs should be distinguished when controlling the I/O performance. This granularity provides the basis to separate the study of reading and writing behaviors for target applications, as well as evaluating the influence of read-optimized or write-optimized storage devices. Users should be able to limit the read throughput, the write throughput or both according to the needs of experiments.

In the next section, we study the feasibility of using *cgroups* to emulate the I/O performance for testbed experiments.

A. Emulating I/O performance with cgroups

Linux *cgroups* is a technology used at the scale of a single machine to control that machine's resources. It allows users to isolate target processes into one or several groups and then to define some constraints to limit their resources' usage.

cgroupV1, which appeared in Linux v2.6.24, proposes many controllers like *memcg* for memory performance and *blkio* for I/O resources. A given process should be added to several groups, each corresponding to a different resource (e.g., memory, I/O, CPU), to control its resources' usage. In other words, no more than one controller can be activated on a given group.

Altering the I/O throughput using *cgroupV1* can be done using the *blkio* controller. However, the limitations set by the *memcg* controller for the *page cache* management are ignored: instead, the whole memory of the host system is used for the *page cache*, resulting in performance that is higher than expected. This leads to wrong measurements (higher I/O throughput) except for direct I/O operations (because they bypass the *page cache*).

cgroupV2, included in Linux v4.5, addresses the drawback of its predecessor by having a more flexible structure. It permits to activate several controllers over the same group. The memory limitations become active as the memory controller works together with the block I/O controller on the same group, adjusting the *page cache* size of the target group. On one side, the block I/O controller assigns an *inode* number to the group that possesses the process which sends I/O requests. On the other side, the memory controller applies its limitation while processing the *dirty pages* of that group. This helps to overcome the *write-back* issue during writing operations, leading to obtaining correct measurements of I/O throughputs.

Technically, the support of *cgroup*'s *write-back* is limited to certain *filesystem* such as *ext2*, *ext3*, *ext4*, and *btrfs*. On other *filesystems*, *write-back* I/Os are assigned to the root *cgroup*. However, this is not a problem in practice as other filesystems are not interesting in our use case.

B. Experimental Validation

Both versions of *cgroups*' are evaluated to determine if they are usable for storage performance emulation, by applying various workloads and limitations and examining the resulting behaviour. The flexible I/O benchmark (*Fio* v3.12) is used. It provides several *IO engines*, each stressing different set of I/O *system calls* (e.g., *sync IOengine* uses *read & write syscalls*). Our experiments apply a set of I/O limitations following a full-factorial design of 5 experimental factors: 1) the I/O methods with the following *Fio's IOengines* as values: *Sync*, *Psync*, *Pvsync*, *Pvsync2*, *Posixaio*, and *Mmap*, 2) the type of workload including *read*, *write*, *randread*, *randwrite*, *readwrite*, and *randreadwrite*, 3) the I/O accessibility (direct and buffered I/Os), 4) the storage device (*HDD* and *SSD*), and 5) different sizes of files. The performance of the storage devices that are used in these experiments is as follows: $197\,\mathrm{MB/s}$ & $710\,\mathrm{kB/s}$ as direct read and write I/O throughputs of *HDD*, while the corresponding *SSD* performance is $119\,\mathrm{MB/s}$ & $108\,\mathrm{MB/s}$ (measurements obtained using *Fio*). The write bandwidth on HDD might seem low but is expected since *Fio* performs random writes. The *SSD* performance seems not very high compared with the *HDD* performance, but this is due to the low-default block size applied by *Fio* during tests ($4\,\mathrm{kB}$) and concerns only direct I/Os. Although using different set of files to test each storage device, we apply two memory limitations accordingly ($1\,\mathrm{GB}$ for *SSD* experiments and $128\,\mathrm{MB}$ *HDD* experiments).

The obtained results of *cgroupV1* are not coherent regarding the applied I/O limitations, especially for the buffered I/Os, for the reasons mentioned above. In contrast, results of *cgroupV2* are more consistent. Due to a limitation of space, two experiments are chosen to be discussed here. However, more results are in the paper's GitHub repository (URL on the first page).

Figure 2 and figure 3 show the results of two experiments performed using cgroupV2. Both figures represent in percentage the ratio of obtained I/O throughputs to the used I/O limitations. Indeed, quantifying the I/O throughput indicates to what extent the used limitations are considered. Additionally, the dashed line in all figures is set to 130% to represent the bandwidth measurements that are too high. If that line is reached, this indicates that the corresponding limitation is not applied.

Figure 2 compares the direct and buffered I/Os on an *SSD*. In case of direct I/Os, one can see that all I/O limitations are

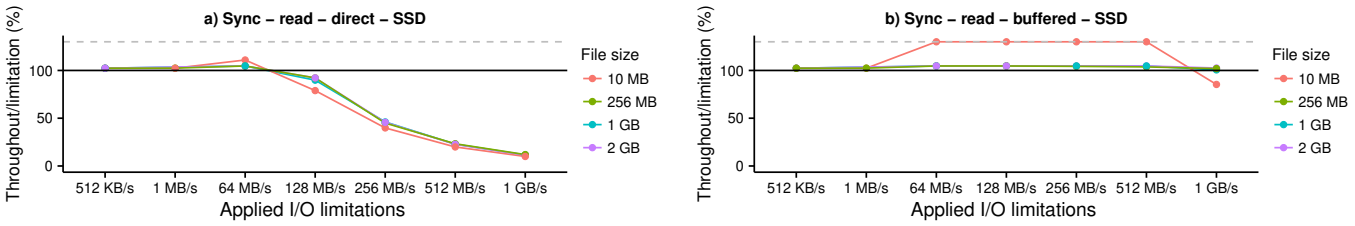

Fig. 2. Ratio of measured vs defined I/O throughput for a) direct & b) buffered read workload over an *SSD*, using *Fio*'s *sync IOengine*

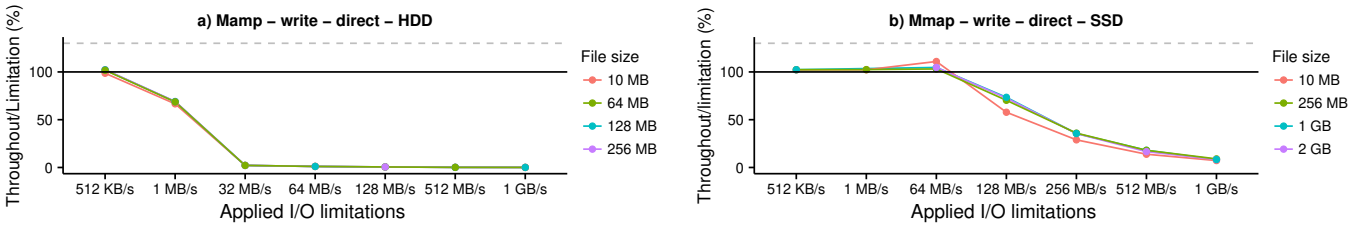

Fig. 3. Ratio of measured vs defined I/O throughput for direct write workload over a) an *HDD* & b) an *SSD* using *Fio's mmap IOengine*

efficient. The emulated I/O throughputs are respected for the points of $512\,\mathrm{kB/s}$, $1\,\mathrm{MB/s}$, and $64\,\mathrm{MB/s}$ included. However, the rest of I/O limitations achieves lower I/O throughputs since they are limited by the storage device's performance ($119\,\mathrm{MB/s}$ in this case). Figure 2-b shows that *cgroupV2* is capable to emulate the I/O performance for buffered I/Os. It shows that the emulated I/O performance corresponds to the measured I/O performance for almost all tested files, even for files that fit into memory. For instance, reading the $256\,\mathrm{MB}$ file with a rate of $1\,\mathrm{GB/s}$ reports a correct measurement near $1\,\mathrm{GB/s}$ despite of the fact that the file is smaller than the applied memory ($1\,\mathrm{GB}$). However, the result of the $10\,\mathrm{MB}$ file shows that it is read at once, ignoring the defined I/O limitations. Hence, the two files ($10\,\mathrm{MB}$ & $256\,\mathrm{MB}$) are not similarly treated, showing sophisticated caching behaviors which requires more investigation.

Figure 3 shows results when performing direct write I/Os using *mmap* over both storage devices. It indicates that the performance of storage device influences the applied I/O limitations. In the *HDD* experiment, the measured throughput decreases starting from the point of $1\,\mathrm{MB/s}$ (*HDD* direct write throughput is $710\,\mathrm{kB/s}$). Similarly, the I/O throughput of *SSD* experiments decreases starting from the point of $128\,\mathrm{MB/s}$ which is very close to the *SSD* performance ($108\,\mathrm{MB/s}$).

C. I/O Emulation in Distem

According to the results shown in the previous section, we extended *Distem* to leverage the potential of *cgroupV2*. Integrating this into Distem makes it possible to go beyond controlling the resources of a single machine. It allows emulating the I/O performance for dozens to thousands of experimental nodes simultaneously, creating more testing configurations for large-scale systems and emulating the end-environment performances. *Distem* provides two methods to emulate the experiments' I/O performance:

a) Statically: Users can emulate the I/O performance of all experimental *vnodes* or a subset of them at any moment during the experiment timeline. This is useful to study the behaviors of the target systems under many scenarios. For instance, each *vnode* can have different I/O throughput during the experiment. This can be done through sending a request using the *Distem client*, providing the following information: the *vnode(s)* to control its I/O performance, the desired values of read & write I/O throughput or both, and on which storage device the control should be applied.

b) Dynamically, based on time events: This emulation mode allows users to achieve several changes of I/O performance during the same experiment. Users can define several time points to alter the I/O performance of the selected *vnodes* automatically. This emulation mode is useful when emulating a performance degradation over fine-grained intervals or switching the I/O performance between *vnodes*. However, it is up to users to choose the time points to stress the system during the experiment. Hence, knowing the system's behaviors in advance is required in order to decide when to alter the performance.

IV. Experiments

This section describes a series of experiments done on the *Hadoop* framework to highlight the usefulness of emulating the I/O performance for testbed experiments. It starts by describing the experiments' goals and setup before presenting the obtained results.

A. Experiments' hypothesis

These experiments have two goals. They aim at determining the behaviors of *Hadoop* after several changes in the I/O performance of one or several nodes of the *Hadoop* cluster. They also illustrate how to mitigate the impacts of testbeds storage resources over results. Indeed, if the storage performance is emulated, experiments' results should be comparable despite the nature of the storage devices in use: *HDDs* or *SSDs*.

TABLE I
EXPERIMENTS DESCRIPTION AND THEIR EXPECTED RESULTS IN TERMS OF THROUGHPUT AND EXECUTION TIME

Experiments	Description	Expected results
Exp 1	I/O read and write throughputs are set up to $120\,\mathrm{MB/s}$ for all datanodes. Ref. experiment	$t_{exec} = t1$, $th_r = d \times n/t1$
Exp 2	I/O read and write throughputs are set up to $5\,\mathrm{MB/s}$ for one data node	$t_{exec} = t_2$, $th_r = d/t2$
Exp 3	I/O read and write throughputs of all datanodes are limited to $60\,\mathrm{MB/s}$	$t_{exec} = t_3$, $th_r = (d \times n/t3)$
Exp 4	I/O read and write throughputs of all datanodes are limited to $30\,\mathrm{MB/s}$	$t_{exec} = t_4$,$th_r = (d \times n/t4)$
Exp 5	I/O read and write throughputs of half nodes are limited to $60\,\mathrm{MB/s}$, the other half to $30\,\mathrm{MB/s}$	$t_{exec} = t_5$ while $t3 < t5 < t4$

In general, big data systems such as *Hadoop* are susceptible to CPU and memory resources while scheduling their map-reduce tasks. However, we are not sure if they take into account the I/O performance of their nodes during jobs' execution. Provided that, five experiments are designed to spot *Hadoop* behaviors, in order to validate this hypothesis. We expect that *Hadoop* will be influenced by emulating the I/O performance of its nodes, incurring a delayed execution time according to the applied I/O limitations. Our expectations are formulated in terms of execution time and I/O throughput for each experiment separately. To define our experiments, we suppose that 1) n is the number of *datanodes* in a *Hadoop* cluster, 2) the initial value of the read and write I/O throughput is $120\,\mathrm{MB/s}$ for each *datanode*, and 3) each *datanode* has d GB of data to be read/written regarding the workload. Table I describes the defined experiments and their expected results.

B. Experimental Setup

The experiments are performed on the *Grid'5000* testbed [9]. Twenty-five machines are used as infrastructure. Each machine is equipped with two Intel Xeon E5-2630 V3 CPUs (8 cores/CPU), $128\,\mathrm{GB}$ of RAM, and a $10\,\mathrm{GB/s}$ Ethernet interface. Also, every node has a $186\,\mathrm{GB}$ *SSD* and a $558\,\mathrm{GB}$ *HDD* connected as a *JBOD*. Debian 9 is deployed on top of these nodes with *ext4* as a *filesystem* and *CFQ* as an I/O scheduler. The *Distem* emulator is then used to deploy its virtual nodes (*vnodes*) over those 25 physical nodes with one-to-one mapping, i.e., one *vnode* over a physical node. This mapping simplifies the deployment process since it does not consider a local resource sharing of I/O performance.

Hadoop v2.8.4 is installed on the *vnodes* with *YARN* as its default map-reduce framework. A cluster of one *namenode* and twenty-four *datanodes* is created on top of the *Distem* environment. All *Hadoop*'s default configurations are maintained except the number of mappers and reducers per job. All *datanodes* are configured to have at maximum one mapper per job. Indeed, we do not allow to have two mappers per job, in order to simplify the throughput calculation process, ensuring that all nodes are busy in a given time during the experiment. Additionally, only half of the nodes are allowed to have one reducer at maximum, i.e., half of the nodes are not allowed to perform reduce tasks. Otherwise, it is not possible to control where to put each reducer as this depends on the resources availability and the internal algorithm of *YARN*. For instance, if a *datanode* is configured to have a moderate I/O throughput and the last job's reducer is to be run on that *datanode*, this leads to non-reproducible results since it is not guaranteed that this reducer will always run on that *datanode*.

TABLE II
AVERAGES OF EXPERIMENTS' EXECUTION TIME

	Exp 1	*Exp 2*	*Exp 3*	*Exp 4*	*Exp 5*
HDD	8 min	21.10 min	24.3 min	84 min	51.3 min
SSD	4.5 min	19.6 min	20.6 min	91 min	47 min

The *Terasort* benchmark is used as the main workload for this experiment. Before each execution, the benchmark's data generator is firstly used to generate $5\,\mathrm{GB}$ of data for each *datanode*, resulting to have $120\,\mathrm{GB}$ as a total data for the cluster. *Terasort*'s map-reduce job is then executed on the generated data, and the execution time of experiments is calculated as the time elapsed between the command execution and its termination. *Hadoop* is restarted between the executions and the cache is cleaned out too. Finally, each experiment is performed ten times to increase results reliability.

C. Results

Figure 4 and Table II provide execution time of all experiments on *HDDs* and *SSDs*. The results show that the execution time changes regarding the emulated I/O throughput in each experiment. The time is always stated as the time of the *datanode(s)* with the lowest I/O throughput. For instance, the second experiment has one *datanode* with an emulated I/O throughput of $5\,\mathrm{MB/s}$ while the I/O throughput of the rest of *datanodes* is $120\,\mathrm{MB/s}$. The obtained execution time of this experiment when performed on *HDDs* is $21.10\,\mathrm{min}$. This value corresponds to the execution time of the *datanode* with the moderate I/O throughput. We verify that by calculating the approximate throughput from the obtained execution time. We obtain $4\,\mathrm{MB}/s$ which is very close to the applied throughput on the limited *datanode* ($5\,\mathrm{MB/s}$). Similarly, this analysis is correct even in case of having several classes of emulated I/O throughputs. For example, the execution time of the fifth *SSD* experiment (12 *datanodes* with an I/O throughput of $60\,\mathrm{MB/s}$ & 12 *datanodes* with an I/O throughput of $30\,\mathrm{MB/s}$) is influenced by the group of nodes with the lower I/O performance. The overall throughput of that experiment is $21.7\,\mathrm{MB}/s$, which is relatively close to $30\,\mathrm{MB}/s$.

One can see in figure 4 that performance varies similarly for both storage types thanks to the defined I/O emulation. However, there are still some differences between *SSDs* and

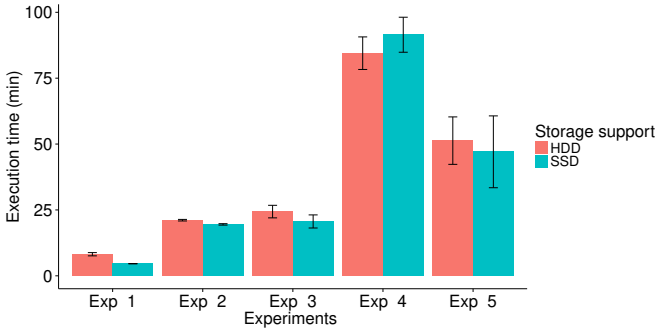

Fig. 4. Averages of experiments' execution time. Applying the same I/O limitations on both storage types produces comparable results.

HDDs, especially for the first experiment. We suspected that this is either due to 1) the fact that the applied I/O throughput ($120\,\mathrm{MB/s}$) is close to the peak performance of those HDDs (measured at $133\,\mathrm{MB/s}$); 2) the fact that I/O emulation only affects the throughput of the storage device, but not its latency, and that SSDs provide lower latency than HDDs. The same figure also shows that the variability is increased between repetitions of the fourth and fifth experiments. We suspected that this is due to different decisions being made by *YARN* during the scheduling of the last reducer job. This reducer may run early if a *datanode* becomes available or it may be executed at later stages with delayed execution time.

V. Related Work

Several studies tried to manage the I/O performance for enhancing the *QoS* of I/O applications. They use either *cgroups* or other methods to do so. Using *cgroups*, Suk *et al.* [1] allocate the I/O dynamically based on the needs of virtual machines. Their tool provides feedback to *cgroups*, which in turn, uses that information to alter the I/O throughput accordingly. However, their tool cannot be used to apply I/O limitations, and it is not distributed to manage the I/O throughputs over several machines. Huang *et al.* [2] contributes an I/O regulator to fairly allocate the I/O throughput for big data frameworks. They claimed that *cgroups* does that unfairly, so the I/O regulator measures the I/O load of *datanodes* to adjust the *IOPS*. However, the I/O regulator is a Hadoop-specific solution. Also, it cannot be used to apply I/O limitations. Ahn *et al.* [3] proposed a weight-based allocation for I/O throughput. Their solution uses a dynamic I/O throttling based on a prediction of future I/Os of containers. However, it is enhanced for the proportional sharing of throughput and not for applying I/O limitations.

Other studies proposed I/O schedulers to improve the *QoS* of I/O applications. Zhang *et al.* [10], [11] proposed an interposed I/O scheduling framework (*AVATAR*) to guarantee a proportional sharing of I/O throughput. *AVATAR* does not scale for several machine usage as it focuses on managing the I/Os for local and concurrent processes. Xu *et al.* [12] proposed a scheduler for *HDFS* to control the I/O throughput for map-reduce jobs internally. However, their solution is Hadoop-specific. Despite the scheduler's ability to work on several *HDFS* nodes, it manages the nodes' I/Os locally, in isolation.

To summarize, no experimental framework was proposed to emulate the I/O performance of testbed experiments. Almost all described frameworks are designed to share the I/O throughput fairly between I/O processes and are unsuited to cases where I/O limitations should be applied differently from one node to another, to perform evaluations under like-end environment circumstances.

VI. Conclusions

Testbed I/O experiments have different goals and architectures. Forcing them to follow the storage resources' performance leads unintentionally to have incompatible testing configurations and to leave a bias over experiments' results. In this paper, we explored the idea of customizing the I/O performance to overcome that issue, giving experimenters more control over testbed resources, and opening new directions to evaluate under suitable testing configurations. After demonstrating that Linux's *cgroups* is suitable for emulating the I/O performance for the dominant I/O workloads, we implement that technology in the *Distem* emulator to provide an I/O emulation service at scale. The advantage of emulating the experiments' I/O performance is highlighted via a *Hadoop* map-reduce experiment, on *HDDs & SSDs*, that shows that the performance of Hadoop varies under different and heterogeneous I/O throughput configurations.

Future research should investigate the ability of *Distem* to emulate more complex I/O experiments. It should consider more complex configurations with concurrent processes per *cgroups*, and folding multiple virtual nodes per physical machine, enabling larger-scale experiments.

References

[1] P. Sukheepoj and N. Nupairoj, "Adaptive i/o bandwidth allocation for virtualization environment on xen platform," in *ICIIP*, 2017.

[2] D. Huang, J. Wang, Q. Liu, X. Zhang, X. Chen, and J. Zhou, "Dfs-container: achieving containerized block i/o for distributed file systems," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[3] S. Ahn, K. La, and J. Kim, "Improving i/o resource sharing of linux cgroup for nvme ssds on multi-core systems." in *HotStorage*, 2016.

[4] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: a state-of-the-art review," *IEEE Transactions on Cloud Computing*, 2017.

[5] C. Pahl and B. Lee, "Containers and clusters for edge cloud architectures–a technology review," in *FiCloud*. IEEE, 2015.

[6] L. Sarzyniec, T. Buchert, E. Jeanvoine, and L. Nussbaum, "Design and evaluation of a virtual experimental environment for distributed systems," in *PDP*, 2013.

[7] C. Ruiz, E. Jeanvoine, and L. Nussbaum, "Porting the Distem Emulator to the CloudLab and Chameleon testbeds," Inria, Research Report RR-8955, Sep. 2016. [Online]. Available: https://hal.inria.fr/hal-01372050

[8] M. Team, "Mininet," 2014.

[9] D. Balouek *et al.*, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, 2013.

[10] J. Zhang, A. Sivasubramaniam, A. Riska, Q. Wang, and E. Riedel, "An interposed 2-level i/o scheduling framework for performance virtualization," in *ACM SIGMETRICS Performance Evaluation Review*, 2005.

[11] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel, "Storage performance virtualization via throughput and latency control," *ACM Transactions on Storage (TOS)*, vol. 2, no. 3, pp. 283–308, 2006.

[12] Y. Xu and M. Zhao, "Ibis: interposed big-data i/o scheduler," in *Proceedings of the 25th ACM HPDC*, 2016.