Pufferbench: Evaluating and Optimizing Malleability of Distributed Storage

Nathanaël Cheriere, Matthieu Dorier, Gabriel Antoniu

Pufferbench: Evaluating and Optimizing Malleability of Distributed Storage

Nathanaël Cheriere
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
nathanael.cheriere@irisa.fr

Matthieu Dorier
Argonne National Laboratory
Lemont, IL, USA
mdorier@anl.gov

Gabriel Antoniu
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
gabriel.antoniu@inria.fr

***Abstract*—Malleability is the property of an application to be dynamically rescaled at run time. It requires the possibility to dynamically add or remove resources to the infrastructure without interruption. Yet, many Big Data applications cannot benefit from their inherent malleability, since their colocated distributed storage system is not malleable in practice. Commissioning or decommissioning storage nodes is generally assumed to be slow, as such operations have typically been designed for maintenance only. New technologies, however, enable faster data transfers. Still, evaluating the performance of rescaling operations on a given platform is a challenge in itself: no tool currently exists for this purpose.**

We introduce Pufferbench, a benchmark for evaluating how fast one can scale up and down a distributed storage system on a given infrastructure and, thereby, how viably can one implement storage malleability on it. Besides, it can serve to quickly prototype and evaluate mechanisms for malleability in existing distributed storage systems. We validate Pufferbench against theoretical lower bounds for commission and decommission: it can achieve performance within 16% of them. We use Pufferbench to evaluate in practice these operations in HDFS: commission in HDFS could be accelerated by as much as 14 times! Our results show that: (1) the lower bounds for commission and decommission times we previously established are sound and can be approached in practice; (2) HDFS could handle these operations much more efficiently; most importantly, (3) malleability in distributed storage systems is viable and should be further leveraged for Big Data applications.

***Index Terms*—Distributed Storage System Malleability, Benchmark, Pufferbench.**

I. Introduction

Malleable applications or services, supporting the possibility to be dynamically rescaled at runtime, are ideal for an execution on cloud platforms. The core principle of cloud computing is to enable users to quickly get as many resources as they need and to release some when they are not needed. Thus, users have the opportunity of constantly optimizing performance and cost with respect to changing workloads. This is typically refered to as *cloud elasticity*. When taking a closer look at how it is applied in practice today, it can be noticed that in most cases cloud users have the possibility to rent a fixed number of resources for a certain of time specified in advance, then they run their service or application; in practice, when they need more resources, they often have to stop their service, rent more resources, then redeploy the service on the new configuration. That is, there is still a gap to fill to adequately support malleable applications and services through dynamic resource rescaling without service interruption.

To illustrate this limitation, let us focus on Big Data frameworks such as Hadoop or Spark, which are malleable by design. They often however rely on a distributed storage system (DSS) that is *not malleable*. This lack of malleability leaves the user with two options: 1) deploy the DSS in a static set of resources that is separate from those running the data processing tasks, which prevents the application from performing any local data accesses; 2) deploy the DSS within the application's resources on a static set of resources, which prevents any kind of malleability altogether.

The reason why malleability has not been widely adopted for DSS is that it involves migrating large amounts of data, which is potentially long. A DSS generally has to ensure three properties when malleability is used: 1) no data can be lost, 2) fault tolerance must be guaranteed, and 3) the placement of data across the storage nodes should be balanced. Thus, in general, *DSS malleability is assumed to be too slow for practical use and is traditionally reserved for maintenance purposes.* Yet, today, thanks to many technology improvements, data transfers are not as slow as they used to be.

To know whether DSS malleability would be useful *for a given workload* or *on a given platform*, one should be able to estimate the duration of both commission (adding nodes) and decommission (releasing nodes) operations. Indeed, having quick operations for the malleability is essential to benefit from the malleability. Quickly decommissioning unused nodes reduces the overall cost of running an application, while quickly commissioning nodes allows the amount of resources to dynamically adjust to a volatile workload.

In previous work [1], [2],[1] we have provided theoretical lower bounds for the duration of rescaling operations (the commission and decommission operations). These lower bounds are useful because they indicate whether the malleability would be too slow on a given platform (i.e., in the case these lower bounds are too high from a practical perspective). However, if the lower bounds are acceptable, it is still difficult to know *in practice* how fast malleability can really be

[1]The lower bound for the duration of the decommission has been published and is available in [1]. A research report concerning the lower bound of the commission is available [2]. A journal paper with all results has been submitted and is under review.

supported on a specific platform. For this, one must deploy an actual DSS, determine an efficient configuration for the rescaling operations, generate data, and record the duration of the operations. Moreover, the obtained results may not be accurate since rescaling operations implemented in current DSS are often not optimized for speed but to limit their impact on application execution.

In this paper, we address the problem of evaluating malleability by introducing *Pufferbench*,[2] a modular benchmark developed to efficiently measure the duration of commission and decommission operations on a given platform. To this purpose, Pufferbench emulates a DSS, executing only the inputs and outputs needed for a rescaling operation.

Pufferbench has been designed with two goals in mind:

1) Evaluate the viability of DSS malleability on a given platform. Pufferbench provides the duration of rescaling operations on a platform, regardless of the DSS that would use these operations.
2) Help optimize migration mechanisms in order to improve the malleability of a specific DSS. Pufferbench is independent from any DSS and thus can be used to quickly prototype and test custom data migration mechanisms (algorithms, network transfers, storage management) on a simpler code before implementing them into a real DSS.

In the second case, Pufferbench also verifies the correctness of the custom migration algorithms by checking that the postconditions (number of replicas, data distribution, etc.) are satisfied.

First, we validate Pufferbench against the lower bounds established in our previous works. According to our evaluation, Pufferbench achieves performances that are on average only 16% slower than the lower bound for the decommission and 7% for the commission. From these results, we also conclude that the previously determined lower bounds are sound.

In a second set of experiments we illustrate the usefulness of Pufferbench by using it to measure how fast the rescaling operations of Hadoop's distributed storage system (HDFS) could be if they were optimized for speed. Experiments show that even if the performance of Pufferbench varies greatly depending on the operation (commission or decommission) and the type of storage (in RAM or on disk), *it is always faster than HDFS.* This suggests that the algorithms used for commission and decommission in HDFS can substantially be improved. In particular, for the commission operation, HDFS can be sped up by as much as a factor 7.

The remainder of this paper is organized as follows. In Section II we introduce the context and the related work. Pufferbench is presented in Section III and validated in Section IV. In Section V we detail how we used Pufferbench to measure how fast the rescaling operations of HDFS could be, and we present the experimental results in Section VI. We discuss the results in Section VII and conclude in Section VIII.

[2]Pufferbench is available at https://gitlab.inria.fr/Puffertools/Pufferbench

II. Context

Malleability has gained interest because it improves resource utilization on shared cluster-based platforms while saving energy and money. Many frameworks provide support for malleability of computing resources [3], [4], [5]. Big Data processing frameworks such as Hadoop [6] and Spark [7] are malleable: new computation nodes can be added and removed without having to shut down the framework. However, the lack of malleability of the underlying storage system greatly limits the overall malleability of the application. In Hadoop's case, the underlying HDFS storage system is not malleable; thus the malleability of Hadoop can hardly be used in practice. The reserved cluster cannot be smaller than the size of the HDFS cluster; thus, shrinking the Hadoop deployment less than this size is pointless since HDFS and Hadoop are sharing the same nodes. Conversely, the deployment cannot grow too large without incurring a high load on HDFS and without losing the benefits of data locality. This limitation is not specific to HDFS. DSS need to be able scale to up and down to ensure consistent performance especially when the applications using them are rescaled.

A. Related Works

Most distributed and parallel file systems such as Ceph [8] or HDFS [9] include both commission and decommission operations, mainly for maintenance purposes. Their rescaling operations are optimized mostly to reduce their impact on the performance of the applications, rather than to reduce the duration of the operation (which can be understood).

A few DSS are built around a restricted form of malleability: they have a pool of machines available but can shut some of them down to save energy. Three systems have been developed with this strategy: Rabbit [10], Sierra [11], and SpringFS [12]. They have a common limitation: the shutdown nodes are not fully released. They host data on their permanent storage drive that will be updated when they rejoin the storage cluster, and they must be available for fault tolerance.

The SCADS Director [13] is a resource manager designed to ensure service-level objectives. It chooses to add or remove nodes, where and when to move data, and the number of replicas each file needs. The SCADS Director adds malleability to the SCADS file system [14]. Its authors have focused their evaluation on the point of view of applications that use their storage system. They evaluate whether their system is able to follow the workload to scale up and down and whether performing data migration has an impact on the service-level objectives. They do not, however, evaluate the performance of the rescaling operations themselves. Their goal is to be the least impactful to users, not necessarily the fastest.

Lim, Babu, and Chase [15] propose a resource manager based on HDFS. This resource manager chooses when to add and remove nodes and the parameters of the rebalancing operations. However, it simply uses HDFS as it is and does not focus on its efficiency. Both [13] and [15] focus on ways to leverage malleability rather that on improving it. They are therefore orthogonal and complementary to this work.

B. Challenge

The first step towards implementing real, efficient malleability in existing or in future DSS is to demonstrate that it will be fast enough to be used in practice. It is difficult, however, to evaluate the performance of data migration operations on a given platform. In our previous work [1], [2], we provided lower bounds for the duration of these operations; however, these lower bounds are based on strong hypotheses: uniformity of the hardware, perfect load balancing, absence of latency, etc. These hypotheses are rarely met in practice. To accurately measure the duration of the rescaling operations, an evaluation on the actual target platform is needed.

The process of evaluating the performance of malleability with a real DSS is time consuming and not necessarily accurate: the rescaling operations of existing DSS may not be optimized for speed nor for the needs of the workload and the platform.

To address these needs, we introduce Pufferbench, a benchmark specifically designed to improve the process of evaluating storage malleability on cluster-based platforms. Pufferbench is not a distributed file system, nor does it rely on a specific one.

III. Pufferbench

A. Overview of Pufferbench

Pufferbench is implemented as an MPI application that emulates the rescaling operations of a DSS by doing all I/O operations that are needed during such a rescaling operation: data accesses to/from a local storage device (which may be local memory) and across the network. It is a master/workers application with MPI rank 0 acting both as master and worker and all other ranks acting as workers.

Its execution involves four steps.

1) **Migration planning:** Pufferbench's master node applies the migration algorithm chosen in its configuration file to compute the sequence of I/O operations (writing/reading to/from local device, sending/receiving to/from other nodes) that each node needs to execute to complete the migration. The trace of I/Os is then sent to worker nodes to be replayed.
2) **Data generation:** All nodes running Pufferbench generate data on their local storage device to be able to read from their local storage.
3) **Execution:** All Pufferbench nodes execute their respective sequence of I/O, recording timing and statistics. In particular, the overall duration of the rescaling operation is measured.
4) **Statistics aggregation:** Statistics collected by each node are gathered by the master node and output to the user.

While executing the data migration, Pufferbench makes sure that the following properties are met:

1) No data is ever lost.
2) The data replication factor is the same before and after the migration.
3) The distribution of data across nodes remains balanced.

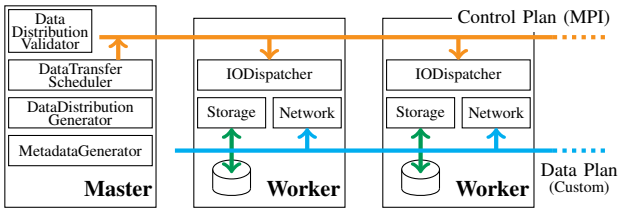


Fig. 1: Components of Pufferbench and their interactions.

Hence Pufferbench not only evaluates the performance of rescaling operations, it also assesses the correctness of the migration algorithm.

B. Customizable Components

In order to match as many platforms and systems as possible, Pufferbench has been designed with modularity in mind. With a simple change in its configuration file, the main components of the system can be switched for custom ones.

1) Master node components: Three of these components are used exclusively by the master node (Fig. 1). In order of action, they are the following.

1) The **MetadataGenerator** generates the basic metadata of the dataset initially present on the emulated storage system. This set of metadata takes the form of a set of pairs *(object id, size)*. Tuning this component enables choosing between various data sizes (e.g., many small objects, few large objects, random size uniformly distributed across a range, gaussian). By changing this component for a custom one, users can plug in data sizes that best match their workload.
2) The **DataDistributionGenerator** takes this metadata set as input and assigns each object to as many virtual storage nodes as necessary to meet the required replication factor. The output of this component is a data distribution map associating each virtual node id with the list of *(object id, size)* that this virtual node manages. Two implementations of this component are provided by default: the first one places data randomly across the nodes, ensuring only the replication factor; the second one balances the load across the nodes. Changing this component enables matching a particular DSS's placement policy or evaluating new ones.
3) The **DataTransferScheduler** is the core of Pufferbench. It takes as input the previously generated placement map as well as the desired migration (e.g. "commissioning 3 nodes") and produces a sequence of I/O operations (read, write, send, receive) that each virtual node has to replay in order to accomplish this migration. The default DataTransferScheduler redistributes the data randomly but maintains a level of load balancing and carefully chooses the nodes reading and sending the data in order to mitigate the bottlenecks of the operations: receiving and writing the data for the decommission, and reading the data for the commission. Customizing the DataTransferScheduler allows the user to test new

migration algorithms and evaluate their performance before implementing them in a real DSS.

Those components are responsible for the simulation of various DSS. For example, one can simulate HDFS with a MetadataGenerator that generates mostly chunks of 128 MiB and a DataDistributionGenerator that will replicate the chunks three times and place them onto random nodes, as HDFS does.

In addition to these three components, the master node includes a **DataDistributionValidator** component that cannot be customized. This component takes as input a data distribution map that represents the placement of objects on the nodes. From this map, it checks that the data migration is valid, that is, that it respects the requirements listed in Section III-A. This validation is done twice in order to control the data distribution maps: before and after the execution of the DataTransferScheduler. Running the DataDistributionValidator twice ensures that the customized DataDistributionGenerator and DataTransferScheduler are both satisfying their requirements.

To evaluate the validity of a particular migration algorithm, Pufferbench can be executed on a single node and stop at the validation step instead of effectively replaying the I/O operations.

2) Worker nodes components: Three components are used by all nodes (including the master) to replay the I/Os.

1) The **Storage** component makes the interface with the local backend storage device by providing the `read` and `write` functions from/to the storage device. By default we provide a Storage component that stores its data in memory, a Storage component that stores its data in a local disk drive, and a Storage component that also stores its data in a local disk drive but ignores the file system cache. Users can plug their own Storage component to, for example, use the custom interface of a particular backend device.
2) The **Network** component provides the `send` and `receive` methods used to transfer object between nodes. Pufferbench's default Network component relies on MPI's nonblocking send and receive functions (`MPI_Isend` and `MPI_Irecv`) so that these operations can complete in parallel with other operations. By default, up to 500 send/receive can proceed concurrently, each of them transferring at most 8 MiB, although these figures can be configured. Once again, users can plug their own Network component, for example to use other network interfaces, or RDMA.
3) The **IODisptacher** component takes as input the sequence of I/O operations received from the master's DataTransferScheduler component and dispatches the operations to the Storage and Network components for execution.

C. Using Pufferbench to Evaluate a Platform

Pufferbench can be used with its default components to evaluate the potential for storage malleability on a given platform. This can be done with a simple default configuration, which uses the migration algorithms described in Section IV. Pufferbench replays all I/Os needed to commission or decommission nodes on the target platform. The duration of each operation is recorded along with various other metrics.

Since the measurements are done on a real platform, the recorded performance is reachable by any DSS available on that platform, provided that DSS is optimized for the rescaling operations.

Some platforms feature heterogeneous hardware. While the current version of Pufferbench does not handle such a case, we plan to add it in the future.

D. Using Pufferbench to Evaluate Data Migration Algorithms

The modularity of Pufferbench allows any user to evaluate and optimize the algorithms used for the rescaling operations (commission and decommission). Thus, Pufferbench can easily be used to optimize and evaluate data migration mechanisms in an existing DSS without modifying it.

Users can plug in custom algorithms for data migration, which can replace the default ones provided by Pufferbench. To this purpose, Pufferbench embeds a component that checks that the plugged-in algorithm yield a valid migration plan. In particular, it checks the replication factor of all objects and evaluates the load balancing of the final distribution by computing the average, minimum, maximum, median, and standard deviation of the amount of data held by each node and the same set of statistics of their number of objects. These statistics enable the user to check whether the final data distribution is more, less, or equally load balanced than the original one.

Moreover, writing commission and decommission algorithms can be done in significantly fewer lines of codes than in an actual DSS because of the abstraction. For instance, the commission and decommission algorithms used in the following section are written in 350 lines of C++ overall.

IV. Validation against the Lower Bounds

In this section, we evaluate the performance of Pufferbench against the lower bounds established for the rescaling operations in previous works [1], [2].

A. Experimental Setup

All measurements were done on the French Grid'5000 [16] experimental testbed. Experiments on decommission were done on the *paravance* cluster in Rennes, while experiments on commission were done on the *grisou* cluster in Nancy. Both clusters feature the same type of node: Dell PowerEdge R630 with Intel Xeon E5-2630 v3 Haswell 2.40 GHz (2 CPUs/node, 8 cores/CPU), 128 GiB of RAM, and two 558 GiB HDD. They are all connected with a 10 Gb/s Ethernet network to a common Cisco Nexus 6000 switch (for *paravance*) or a Cisco Nexus 9508 (for *grisou*).

Pufferbench emulates a DSS that initially hosts 50 GiB per node. Ten measurements per configuration of Pufferbench were done. The results are represented by box plots showing the minimum, the first quartile, the median, the third quartile, and the maximum duration of the rescaling operation.

B. Evaluation of Pufferbench against the Lower Bounds

The lower bounds [1], [2] for the duration of the rescaling operations have been built on strong hypotheses, such as an all-to-all network topology and the absence of latency in disk accesses. In practice, these hypotheses are not met. The following sections describe to what extent our experimental setup respects the hypotheses. In order to safely evaluate any result against theoretical lower bounds, the experimental conditions should be such that practical constraints only increase (and never decrease) the duration of the rescaling operations.

1) Hypotheses on the hardware:

- *Cluster homogeneity*: The cluster should be composed of identical nodes. In practice, although the clusters we used are homogeneous, performance variations can be observed across nodes. For instance, the maximum bandwidth of the drives while reading varies between 195 MiB/s and 207 MiB/s on the cluster used for the experiments. The parameters of the lower bound (maximum disk read speed, maximum disk write speed, and maximum network bandwidth) have been set with the maximum measured values prior to the experiments.
- *Ideal network*: The theoretical lower bounds ignore the network latency and the potential interference that can happen. An all-to-all topology is assumed, with identical bandwidth between any two nodes. In practice, all the nodes of our clusters are connected to a common switch through a 10 Gb/s Ethernet link. Hence this switch may become a bottleneck.
- *Ideal storage backend*: The theoretical lower bounds ignore the latency (seek times of the drives) and assume that the drives always read and write at their maximum speed. In practice, seek times and read/write contention add to the duration of the execution.

Although the theoretical hypotheses are not met, the differences between the experimental setup and the hypotheses only increase the duration of the commission and decommission operations. This ensures that the lower bounds keep their property of lower bounds even with the relaxed hypotheses.

2) Hypotheses on the components: With the components, we can make sure that the other hypotheses and objectives of the lower bounds are met.

The DataDistributionGenerator ensures the other three hypotheses of the lower bounds:

- *Load balancing*: The nodes host the same (or similar) amount of data.
- *Data replication*: Each object stored in the cluster is replicated r times.
- *Uniform data distributions*: Each set of r distinct nodes has some data that is replicated only on these r nodes. The amount of such data should be the same for every such set.

The DataTransferScheduler, which implements the migration algorithms, ensures that the following objectives are met at the end of the rescaling operation:

- *No data loss*: No data is lost during the operations.

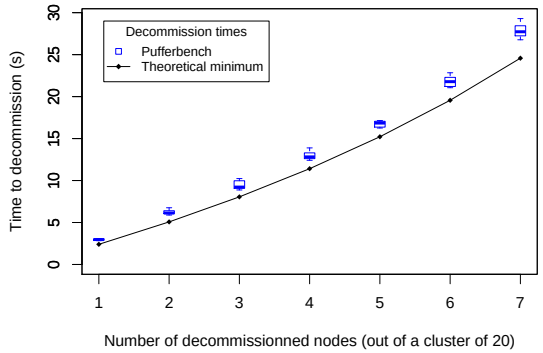


Fig. 2: Time needed to decommission nodes, with storage in memory. Nodes initially host 50 GiB of data on average. Comparison with lower bound.

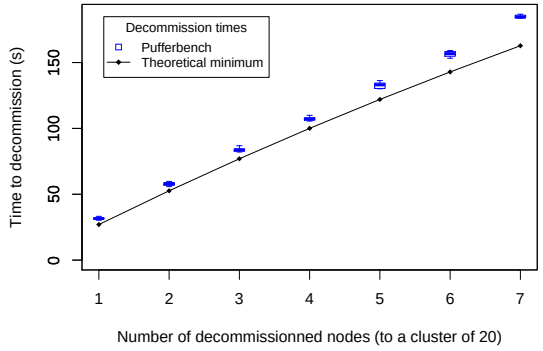


Fig. 3: Time needed to decommission nodes, with storage on drive. Nodes initially host 50 GiB of data on average. Comparison with lower bound.

- *Maintained data replication*: The replication factor is the same as the initial one.
- *Load balancing*: The nodes host the same amount of data.

The last objective, which is to have *uniform data distribution* at the end of the rescaling operation, is relaxed since it is achieved with random data placement. This does not impact the lower bound in most cases except when, during the commission, reading from disk becomes the bottleneck (commission of more than 22 nodes in the following measurements). However, the randomness ensures that the data distribution generated is close to the objective.

Overall, the experimental setup is such that the lower bounds are valid for the average duration of the commission and decommission. The lower bounds can safely be used to evaluate the performance of Pufferbench configured as detailed in this section.

C. Results

Figure 2 shows the performance of Pufferbench against the lower bounds when decommissioning nodes from a cluster of 20 nodes. Each node initially hosts 50 GiB of data in memory.

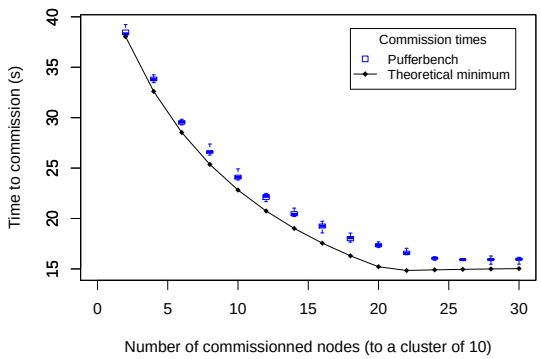


Fig. 4: Time needed to commission nodes, with a storage in memory. Nodes initially host 50 GiB of data on average. Comparison with lower bound.

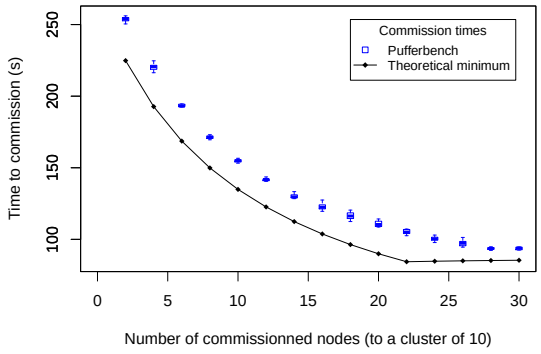


Fig. 5: Time needed to commission nodes, with a storage on drive. Nodes initially host 50 GiB of data on average. Comparison with lower bound.

On average, the decommission times of Pufferbench are 16% longer than the lower bound.

Figure 3 presents the decommission with storage on disk. On average, Pufferbench is 11% slower than the lower bounds.

In both cases, the difference between the lower bounds and Pufferbench is due to the fact that Pufferbench runs on real hardware. The lower bounds consider only the maximum bandwidth of the network and the storage. They ignore the latency and any interference. Pufferbench takes all this into account as it replays all I/Os needed to decommission nodes.

Figure 4 shows the performance of Pufferbench when commissioning nodes into a cluster of initially 10 nodes with storage in memory. On average, Pufferbench is 7% slower than the lower bounds.

Figure 5 presents the results of the commission when the storage is on disk. In this case, Pufferbench is 16% slower than the lower bounds.

The main reason for the difference in performance between the storage in memory and the storage on disk drives is the latency to access data on disks, as well as the disks not exhibiting uniform performance across nodes and across requests (peak read speed varied between 195 MiB/s and 207 MiB/s across nodes).

In both cases, we observe that when 20 nodes are added, the difference between the lower bound and the results of Pufferbench is the largest. This is due to the fact that stragglers appear. In this case, the effect of the stragglers is clear because all the nodes become bottlenecks: the 20 new nodes must finish writing their data in about 15 s, but the old nodes must also finish reading and sending their data in 15 s. Thus, any node straggling has an impact on the overall performance. Stragglers appear because of the variability of hardware performance.

> We show that Pufferbench is able to emulate rescaling operations that are close to the theoretical lower bounds (at most 16% of difference on average). Such a difference with the theoretical lower bounds can be attributed to the fact that the hardware does not match the hypotheses of the model. Moreover, these results also show that the lower bounds themselves are realistic and that the optimal duration of the rescaling operations are within 16% of the lower bounds.

V. Use Case: HDFS

In the following subsections we evaluate the performance of commission and decommission operations in HDFS and compare the results with Pufferbench in order to assess whether HDFS's rescaling algorithms could be optimized.

A. Goals

We focus on HDFS [9] because it is a DSS that is widely used and that has both the commission and decommission operations already implemented. These operations are used in practice mainly for maintenance. In our previous works, we showed that the rescaling operations of HDFS could be greatly accelerated, provided that the estimated lower bound is correct. For example, in the case of decommission, with the storage done on disk, HDFS is three times slower than the lower bound. With Pufferbench, we now have a tool to confirm this in practice.

In this section and the following, our goal is to evaluate how fast rescaling operations could be under the constraints of HDFS. In particular we show that, with a different migration mechanism (algorithms, data transfers, and disk management), HDFS could be much faster at commissioning and decommissioning nodes. Thus, we implement in Pufferbench rescaling mechanisms moving the same objects (chunks of 128 MiB) and aiming for the same final data distribution. Doing so, we can compare the performance of the rescaling operations of HDFS and those of Pufferbench as they produce the same results from the same initial situation.

Note that we did not recreate the commission and decommission mechanisms of HDFS in Pufferbench for a simple reason: the algorithms used by HDFS are dynamic. Data transfers are rescheduled every few seconds to match the actual progress, while Pufferbench assumes that all transfers are scheduled at the beginning of the rescaling operation.

B. How Pufferbench Emulates HDFS

Because we want to create a data distribution similar to that of HDFS, the MetadataGenerator and DataDistributionGenerator components in Pufferbench are configured to generate 128 MiB chunks of data that are replicated three times across randomly selected nodes.

1) New decommission algorithm: Since the data migration mechanism used by HDFS is not optimized for speed, we implemented our own DataTransferScheduler in Pufferbench to try to achieve the best performance rather than imitating HDFS's algorithm.

As shown in [1], the bottleneck in the decommission is receiving and writing the data to storage. Indeed, thanks to data replication, not only do the nodes being decommissioned have the data, some other nodes have them as well.

The amount of data written on each new node is determined by the data placement, which is random but with some load balancing. Nodes are classified as either overloaded or underloaded. Nodes being decommissioned are overloaded since they should host no data. Replicas are randomly moved from overloaded nodes to underloaded ones, provided that no two replicas of the same chunk end up on a same node.

When using in-memory storage, no optimization can be made: each node has data to write and can read data simultaneously without interference. In the case of on-drive storage, each replica must be read once and then forwarded to nodes on which it should be written. The replica can be read from any node hosting it, not only in-decommission nodes. Thus, some drive load-balancing is done to choose which node has to read data: all nodes should have balanced amounts of disk I/Os (taking into account the fact that disks from in-decommission nodes will only read, while others can read and write).

2) New commission algorithm: As shown in [2], two bottlenecks arise during the commission operation: reading the data from the old nodes and writing it to the new nodes.

The data placement is the same as for the decommission.

The main bottleneck that can be mitigated during the commission is when reading data from the nodes initially present. Since existing nodes initially have all the data, each replica to be moved is read once from an existing node and then exclusively forwarded between new nodes.

3) Replay components: The Storage and Network components are the ones provided by default in Pufferbench (in-memory and disk-based for Storage, MPI-based for Network). We disabled caching in the underlying local file system in order to match the configuration used by HDFS.

Compared with HDFS, there is a large improvement in the disk I/Os done by the Storage component. HDFS reads and writes data onto the drives by blocks of 4 KiB and lets the file system cache optimize the operations. The Storage component in Pufferbench buffers the data until the data for the whole object is receive and then writes it. Overall, Pufferbench's storage component writes and reads larger chunks to/from the drives, optimizing the drive bandwidth usage.

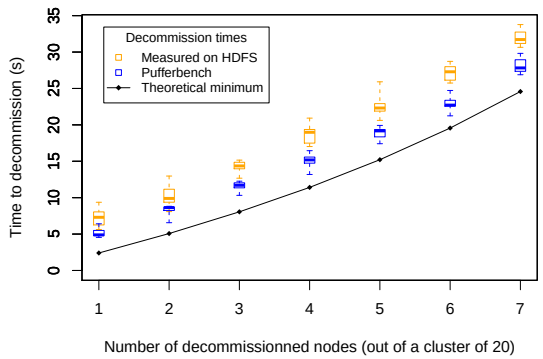


Fig. 6: Time needed to decommission nodes, with a storage in memory. Nodes initially host 50 GiB of data on average.

VI. Experimental Results

In this section, we compare the performance of HDFS and Pufferbench emulating an optimized HDFS during rescaling operations as presented in Section V.

A. Experimental Setup

The experimental platform is the same as the one presented in Section IV-A.

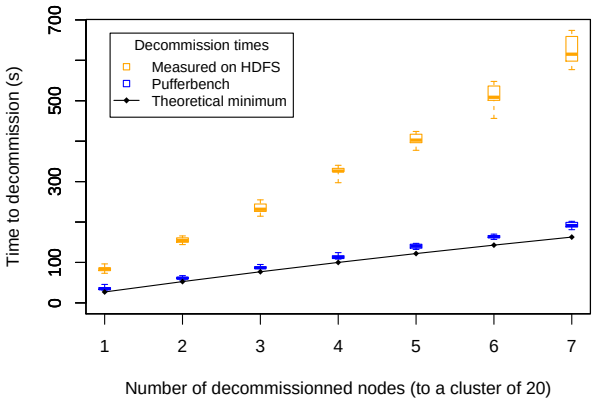
HDFS and Hadoop 2.7.3 were deployed on the nodes. The replication factor is left unchanged to 3. The configuration of HDFS was adjusted in order to remove the limits on the bandwidth usage of the rescaling operations. The commission is divided in two steps. First, new HDFS workers join the cluster; then a rebalancing operation is started. For the decommission, the built-in mechanism is used. The initial size of the cluster is 20 for the decommission operations and 10 for the commission operation. All nodes in the initial clusters host 50 GiB of data before each rescaling operation.

The lower bounds are different depending on whether the network or the storage is the bottleneck for the data transfers. In order to create a situation in which HDFS has a storage bottleneck, the data is simply stored on the disks with the file system's cache limited to 64 MiB. For the situation in which the network is the bottleneck, the data of HDFS was stored on a RAMDisk, effectively storing all the data in memory.

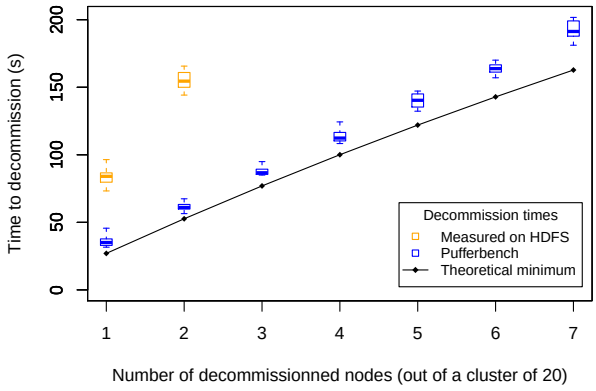
Each measurement was repeated 10 times except for the commission with storage on drive, for which time constraints limited the number of repetitions to 5.

B. Potential Speed of Decommission in HDFS

Figure 6 presents the results of the decommission when the data is stored in memory. Pufferbench is faster than HDFS. HDFS is on average 23% slower than Pufferbench and up to 40% slower in some cases. The difference with the lower bound is due mainly to the initial load-balancing hypothesis (the nodes should all host exactly 50 GiB) not being met. Because HDFS and Pufferbench randomly distribute the data on the nodes, initial load balancing is indeed not guaranteed. Thus, some nodes receive more data than determined by the lower bound. For instance, a node can initially host 45 GiB of

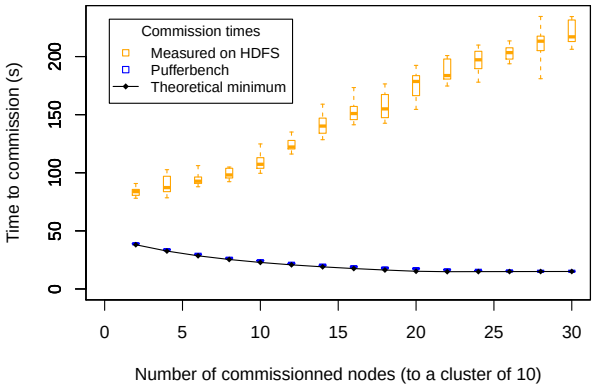


(a) Global situation

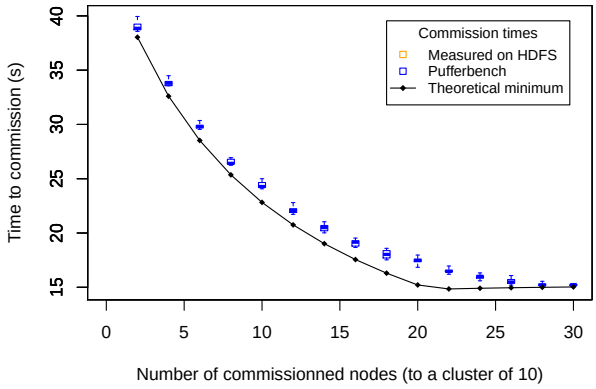


(b) Details of Pufferbench

Fig. 7: Time needed to decommission nodes, with storage on disk. Nodes initially host 50 GiB of data on average.

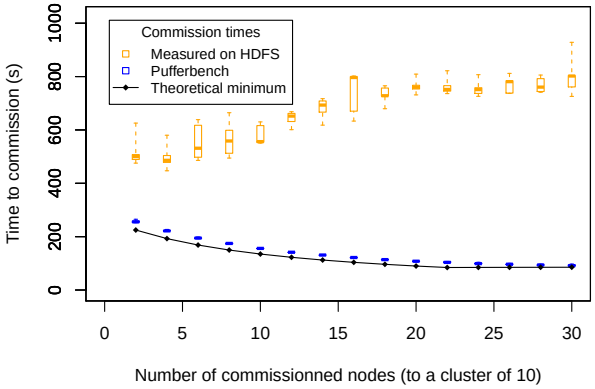


(a) Global situation

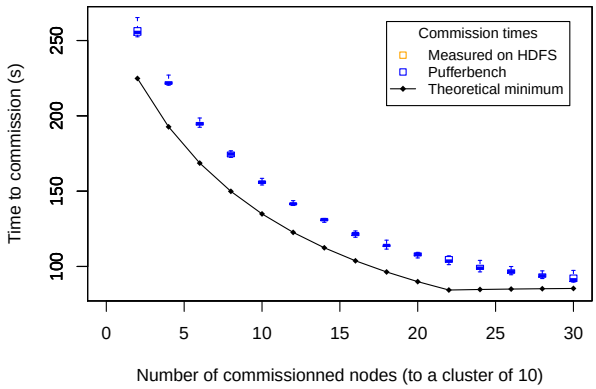


(b) Details of Pufferbench

Fig. 8: Time needed to commission nodes, with storage in memory. Nodes initially host 50 GiB of data on average.



(a) Global situation



(b) Details of Pufferbench

Fig. 9: Time needed to commission nodes, with storage on disk. Nodes initially host 50 GiB of data on average.

data instead of 50 GiB and have to host 60 GiB at the end of the decommission. Because of the initial load imbalance, this node will have to receive 15 GiB instead of 10 GiB, and the decommission will take longer.

In the case where data is stored on drive, Pufferbench is able to decommission nodes faster than can HDFS (Fig. 7). On average, HDFS decommissions nodes 2.8 times more slowly than does Pufferbench. The problem of initial load imbalance observed in Figure 6 is still present, but the migration scheduler of Pufferbench configured for these experiments mitigates it when balancing the read load across drives. The drives that have more data to write will spend less time reading.

C. Potential Speed of Commission in HDFS

In the case of the commission with storage in memory (Fig. 8), Pufferbench is on average 8 times faster than HDFS and up to 14 times when a large number of nodes are added at the same time.

Note that the commission mechanism in HDFS is not optimized for speed but is made to minimize the impact of the commission on the overall cluster.

Pufferbench also exhibits good performance when commissioning nodes with storage on drive (Fig. 9). On average, Pufferbench is 2.79% faster than HDFS.

D. Low overhead of Pufferbench

Experiments with both HDFS and Pufferbench highlight another aspect about Pufferbench: it provides results quickly. Its speed is related not only to the improved rescaling operations but also to the reduced overhead.

To measure the duration of the commission with storage in memory in HDFS, one has to start HDFS, start Hadoop, generate enough data, start the commission, and stop the system. The whole operation took 39 h for the results presented in Fig. 8, with only 13 h spent commissioning nodes; it is an overhead of 26 h.

In contrast, the measurements with Pufferbench lasted for 2 h and 53 min with 2 h commissioning nodes. The overhead of Pufferbench was only 53 min, about 30 times lower than the overhead of HDFS. Indeed, Pufferbench only has to allocate some memory in order to be able to replay a commission, and it is able to quickly switch between measurements.

> With the help of Pufferbench, we showed that the rescaling operations of HDFS could realistically be sped up by 23% on average in the case of the decommission with in-memory storage and by up to 8 times on average and up to 14 times in the case of in-memory commission.

VII. Discussion

In this section, we discuss various points about Pufferbench and its usage.

A. Ideal Setup for the Validation of Pufferbench

The experimental setup used to validate Pufferbench is favorable to fast rescaling operations. In particular, all objects had a size of 128 MiB in order to read, send, receive, and write large sequential chunks of data. This optimized the I/Os for both the network and the local storage.

Thus, the performance should degrade with smaller objects, and Pufferbench should then be used to optimize algorithms, storage, and network transfers to efficiently migrate small objects.

B. HDFS's Case

In the case of HDFS, we observe that the optimized algorithms for the migration are not the only factor of improvement. Pufferbench's network usage is better than HDFS, but the most important part is the backend drive usage: Pufferbench reads and writes full chunks (128 MiB) sequentially, and this approach improves the read and write bandwidth during the decommission operations by at least a factor of 2 compared to HDFS.

Moreover, we show that HDFS's migration mechanism can be sped up by a factor of as much as 14, in particular when commissioning many nodes at once.

The drive management of HDFS can be improved by taking advantage of the amount of memory available on the nodes. HDFS writes and reads data by chunks of 4 KiB. Using larger values (e.g, 128 MiB in the case of Pufferbench) is enough to greatly speed up the decommission of nodes.

Two main aspects of the commission could be improved. First, the algorithm used by HDFS easily accumulates delays: the rebalancing is scheduled by waves of data transfers, and each wave must be completed before the next one starts. This should be replaced with an algorithm that maintains a constant transfer of data between nodes. Second, some buffering should be used in order to read only once each chunk of data from the drives when sending the data to multiple destinations.

The modularity of Pufferbench allows users to test and improve easily all the relevant components used during the rescaling operations: scheduler, network, and storage.

C. Pufferbench and Lower Bounds

Compared against the lower bounds, the results obtained with Pufferbench have the advantage of providing more accurate commission and decommission times. First, it actually replays I/Os so the characteristics of the hardware (network latency, network interferences, disk seek times, disk throughput) are taken into account, but it also evaluates an implementation of the operations.

The lower bounds have the advantage of fixing what is theoretically possible: provided that the hypotheses are met (in particular the initial load balancing), the operations cannot be faster than the lower bound. The lower bound can be a good comparison point when the platform is not available, whereas Pufferbench gives more accurate results when the platform is available.

However, the results show that the lower bounds previously determined are realistic and that performance close to these lower bounds can be reached.

D. Limiting the Impact of Rescaling Operations on Application Performance

The rescaling operations of DSS are used primarily as maintenance operations: permanently increasing the size of a cluster and safely removing faulty nodes. Thus, the rescaling operations are rarely optimized for speed.

In HDFS, for example, both the rebalancing (for the commission) and decommission operations have options in the configuration to limit the bandwidth they can use in order to reduce their impact on concurrently running applications.

One can implement such a limitation in Pufferbench simply by implementing a custom Network component that limits the bandwidth usage. The modularity of Pufferbench allows experimenting with multiple limitations (global disk bandwidth, disk read and/or write bandwidth, network bandwidth, network send and/or receive bandwidth, etc.).

Thus, Pufferbench can also be used to optimize other aspects of the rescaling operations.

VIII. Conclusion

Efficient rescaling operations are needed in order to use the malleability of distributed storage systems. Thus, in this work, we introduced Pufferbench, a modular benchmark with two goals. First, it can measure how fast the rescaling operations can be done in practice on a given platform. With this, the administrators of the platform can decide whether using DSS malleability worth it. Second, it can be used to fine-tune all components involved in data migration (scheduler, storage, and network). Modifying an existing DSS is a strenuous task. Pufferbench enables an easy prototyping and testing of the data migration mechanisms before implementing them in any DSS.

By validating Pufferbench against the lower bounds, we show that one can implement rescaling operations with performance within 16% of the lower bound, in practice. Moreover, this result highlights the fact that the lower bounds are realistic and that the optimal duration for rescaling operations is within 16% of the lower bound.

With Pufferbench, we show that the rescaling operations of HDFS can be greatly sped up, by as much as a factor 14 in some cases.

These results strengthen the idea that malleability in distributed storage systems is viable and should further be studied in order to benefit data-intensive applications.

Implementing efficient commission and decommission into a real distributed storage system and using the prototype to evaluate the benefits of malleability with real use cases is a challenge left for future work.

Acknowledgment

We would like to thank Kevin Harm (ANL) for providing valuable feedback on this paper, and Gail Pieper (ANL) for proofreading it.

The work presented in this paper is the result of a collaboration between the KerData project team at Inria, and Argonne National Laboratory, in the framework of the Data@Exascale Associate team, within the Joint Laboratory for Extreme-Scale Computing (JLESC, https://jlesc.github.io).

Experiments presented in this paper were carried out on 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).

This material is based upon work supported by the U.S. Department of Energy, Office of Science under contract DE-AC02-06CH11357.

References

[1] N. Cheriere and G. Antoniu, "How Fast Can One Scale Down a Distributed File System?" in *BigData 2017*, 2017.

[2] N. Cheriere, M. Dorier, and G. Antoniu, "A Lower Bound for the Commission Times in Replication-Based Distributed Storage Systems," Inria Rennes - Bretagne Atlantique, Research Report 9186, Jun. 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01817638

[3] S. S. Vadhiyar and J. J. Dongarra, "SRS: A Framework for Developing Malleable and Migratable Parallel Applications For Distributed Systems," *Parallel Processing Letters*, vol. 13, no. 2, pp. 291–312, 2003.

[4] L. V. Kale, S. Kumar, and J. Desouza, "A Malleable-Job System for Timeshared Parallel Machines," *IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[5] J. Buisson, F. André, and J. Pazat, "A Framework for Dynamic Adaptation of Parallel Components," *International Conference Parallel Computing*, pp. 1–8, 2005.

[6] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, inc., 2009.

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," *HotCloud*, vol. 10, no. 10, p. 95, 2010.

[8] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *7th Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006, pp. 307–320.

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," *IEEE Symposium on Mass Storage Systems and Technologies*, pp. 1–10, 2010.

[10] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan, "Robust and Flexible Power-Proportional Storage," *ACM Symposium on Cloud Computing*, pp. 217–228, 2010.

[11] E. Thereska, A. Donnelly, and D. Narayanan, "Sierra: Practical Power-Proportionality for Data center Storage," *Conference on Computer Systems*, p. 169, 2011.

[12] X. Lianghong, C. James, K. Elie, T. Alexey, G. Nitin, K. Michael, and G. Gregory, "SpringFS: Bridging Agility and Performance in Elastic Distributed Storage," *USENIX Conference on File and Storage Technologies*, pp. 243–255, 2014.

[13] B. Trushkowsky, P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "The SCADS Director: Scaling a Distributed Storage System under Stringent Performance Requirements," *USENIX Conference on File and Storage Technologies*, pp. 163–176, 2011.

[14] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh, "SCADS: Scale-Independent Storage for Social Computing Applications," in *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research*, 2009.

[15] H. C. Lim, S. Babu, and J. S. Chase, "Automated Control for Elastic Storage," *International Conference on Autonomic Computing*, pp. 1–10, 2010.

[16] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding Virtualization Capabilities to the Grid'5000 Testbed," in *Cloud Computing and Services Science*, 2013, vol. 367, pp. 3–20.