Is it Worth Relaxing Fault Tolerance to Speed Up Decommission in Distributed Storage Systems?

Nathanaël Cheriere, Matthieu Dorier, Gabriel Antoniu

Is it Worth Relaxing Fault Tolerance to Speed Up Decommission in Distributed Storage Systems?

Nathanaël Cheriere*, Matthieu Dorier†, Gabriel Antoniu*
*Univ. Rennes, Inria, CNRS, IRISA, Rennes, France, nathanael.cheriere@irisa.fr, gabriel.antoniu@inria.fr
†Argonne National Laboratory, Lemont, IL, USA, mdorier@anl.gov

***Abstract*—Efficient resource utilization is a major concern for large-scale computer platforms. One method used to lower energy consumption and operational cost is to reduce the amount of idle resources. This can be achieved by using malleability, namely, the possibility for resource managers to dynamically increase or decrease the amount of resources of jobs while they are running.**

Decommissioning (i.e., removing from the cluster) the idle nodes as soon as possible allows the resource manager to quickly reallocate those nodes to other jobs. Challenges appear when such nodes host part of a distributed storage system. Such storage systems may need to transfer large amounts of data before releasing the nodes, in order to ensure data availability and a certain level of fault tolerance.

In this paper, we model and evaluate the performance of the decommission operation when relaxing the level of fault tolerance (i.e., the number of replicas) during this operation. Intuitively, this is expected to reduce the amount of data transfers needed before nodes are released, and thus allow nodes to be returned to the resource manager faster. We quantify theoretically how much time and resources are saved by such a *fast decommission* strategy compared with a standard decommission that does not temporarily reduce the fault-tolerance level. We establish lower bounds for the duration of the different phases of a fast decommission. We use the lower bounds to estimate when fast decommission would be useful to reduce the usage of core-hours and when not. We implement a prototype for fast decommission and experimentally validate the lower bounds on the duration of the operation and confirm in practice our theoretical findings.

***Index Terms*—Decommission, Fault Tolerance, Malleable Distributed Storage System, Elastic Distributed Storage System**

I. Introduction

Improving the resource utilization of a platform is a challenge for its administrators. It directly links to better cost effectiveness and higher productivity. One approach used to improve resource utilization is to reduce the number of idle nodes. This can be achieved by multiple ways, such as careful scheduling, precise prediction of the required resources, or job malleability.

Job malleability is the possibility for jobs to have their resources resized at run time by the resource manager. Thus, unused resources can be returned to the resource manager and either shut down or reassigned to other jobs. Applications also benefit from malleability; unused resources are released, thus reducing the energy consumption and costs attributed to that application at the same time as the core-hour usage is decreased. Moreover, the capability to dynamically adjust the number of resources available to a job allows the job to match the workloads in order to have constant quality of service, even if the workload is highly volatile. Many solutions have been proposed to add malleability to platforms and applications. Resource managers such as KOALA-F [1] and Morpheus [2], are able to manage malleable jobs, and various frameworks [3], [4], [5] enable the design of malleable applications.

Previous works focused on the malleability of computing resources, however; and no distributed storage systems have been designed to be malleable, even though many of them include both commission (adding nodes) and decommission (removing nodes) operations for maintenance purposes. Indeed, adding nodes to a distributed storage system or removing nodes from it involves many data transfers in order to balance the load. These data transfers are assumed to be too slow for practical use. The implemented rescaling operations (commission and decommission) are thus rightfully designed to limit their impact on application performance as much as possible and are not optimized for speed.

Having a truly malleable distributed storage system with fast rescaling operations would enable many features for applications that need to deploy a distributed storage system colocated with computation resources. In particular, applications could benefit from fast data accesses to a co-deployed distributed storage system and benefit from malleability.

- *Reduction of the core-hours cost:* When a storage node is not needed by the application anymore, it can be given back to the resource manager instead of being idle.
- *Ideal scalability:* The distributed storage system can expand and contract with the malleable application using it, ensuring consistent storage system performance.
- *Data isolation:* The data manipulated by an application can be located solely on the computing nodes used by the application and does not need to be stored on a shared storage cluster. This is consistent with the recent trends in HPC systems, which increasingly include local storage on computing resources.

With a fast decommission, the platform can claim resources back from applications quickly. This action allows the resource managers to mitigate unpredictable events, such as the submission of high-priority jobs or a sudden increase in the workload of some jobs, by quickly allocating new resources to the task.

In our previous work [6], we modeled node decommission in replication-based distributed storage systems and provided a theoretical lower bound for the duration of this operation. Knowing a lower bound for this operation can help in several

ways.

- The lower bound can be used to evaluate the performance of decommission mechanisms when designing a distributed storage system on a given physical platform: decommission would cost at least that lower bound on that platform.
- It can also help the resource scheduler make scheduling decisions. With it, the resource scheduler can anticipate some minimum duration needed to decommission nodes and make them available when they are needed.
- With the lower bound, a user can quickly estimate whether to use malleability on a given platform: if the lower bound is too high it means that the performance cost of supporting malleability may not be compensated by its expected benefits, so malleability would not be useful in such a case.
- The study of this operation highlights inherent bottlenecks that need to be mitigated for efficient implementations.

In our previous work, we assumed that the level of fault tolerance of the storage system should not be weakened during the decommission; if the system is configured to keep k replicas of an object at all times, the number of replicas of that object during the decommission should never be strictly less than k. It also means that the decommissioned nodes can be given back to the resource manager only at the end of the decommission operation, since all objects need to be sufficiently replicated on the remaining nodes.

This is an opportunity for optimization. As long as no data is lost, decommissioned nodes can be returned to the resource manager sooner. We denote this strategy as *fast decommission*. It is composed of three phases. During the *data-safekeeping* phase, the system ensures that at least one replica of each object is present on the remaining nodes, transferring objects if needed. Then, during the *node release* phase, the decommissioned nodes are given back to the resource manager. Missing replicas are recreated during the *system stabilization* phase.

With this strategy, the decommissioned nodes are effectively made available for other jobs faster than they are with standard decommission. However, fast decommission comes at the cost of weakened fault tolerance during the system stabilization phase: not all objects have their required number of replicas until the stabilization finishes.

The idea of trading fault tolerance for performance is not new, and is, in fact, rather intuitive. The main contribution of this paper is to show, *mathematically and experimentally*, that *this intuition is incorrect in many situations*. Our goal is to make a step forward in better understanding the actual trade-off that exists between the duration of the decommission and its impact on fault tolerance. To this end, we provide theoretical lower bounds for the two main phases of fast decommission: the data-safekeeping and the system stabilization phases (the node release is assumed to be instantaneous). We also implement fast decommission in Pufferbench [7] [8], a benchmark designed to study the commission and decommission mechanisms of distributed storage systems in practice.

The lower bounds highlight interesting results. As expected, the nodes decommissioned with the fast decommission mechanism are released in a fraction of the time needed by the standard decommission. For distributed storage systems, however, the phase of system stabilization that comes after the release of the decommissioned nodes lengthen the duration of the whole operation. When the bottleneck of the operation is the network, the whole operation lasts as long as the standard decommission. In the case of a storage bottleneck, the duration of the operation is longer than that with the standard decommission: fewer resources are available for the stabilization phase, and thus the operation is longer.

The experimental results obtained with Pufferbench confirm the trends obtained from the lower bounds. In particular, the decommission times obtained are on average within 10% of the lower bounds when the storage is the bottleneck and are on average within 40% of the lower bounds when the network is the bottleneck.

We also compared the number of core-hours needed for the fast decommission and the standard decommission mechanisms. In the case of a network bottleneck, using fast decommission always leads to a reduction in core-hour usage. The gain in core-hours increases with the number of decommissioned nodes. In the case of a storage bottleneck, however, no gain in the usage of core-hours is realized unless many nodes are decommissioned at once.

Overall, fast decommission can be an interesting trade-off for the designer of distributed storage systems when the network is the bottleneck: the consumption of core-hours is reduced compared with that of standard decommission while the duration stays the same for the distributed storage system. Moreover, depending on the network bandwidth, the stabilization phase, during which the fault tolerance is weakened, can be short. However, the trade-off is less relevant in the case of a storage bottleneck: the whole operation is longer than the duration of the standard decommission, and there is no gain in core-hours unless many nodes are decommissioned at once. Fast decommission may even be detrimental depending on the bandwidth of the storage devices. If the storage is slow, such as a hard drive, the fault tolerance will be weakened for the duration of the long stabilization phase, increasing the probability of losing data.

This paper is organized as follows. Related work is presented in Section II. Assumptions detailed in Section III are used to build the lower bounds in Section IV. In Section V, we compare the experimental run times of an implementation of fast decommission with the lower bounds. We discuss the results in Section VI and present our conclusions in Section VII.

II. Related Works

Many distributed and parallel file systems, such as Ceph [9] and HDFS [10], include a decommission mechanism. However, it is available primarily for maintenance purposes. Hence it is understandably optimized to reduce the performance

impact of the decommission on other jobs. It is not meant to be fast.

Some distributed storage systems enable some form of malleability: some of the machines of the cluster on which they are deployed can be shut down to save energy. Rabbit [11], Sierra [12], and SpringFS [13] are examples of such a system. They have two main limitations. First, the shutdown nodes still store data and may be turned back on at any time; thus they cannot be given back to the resource manager for use by another job. Second, the nodes that can be shut down are determined by the distributed storage system and not by the resource manager.

Some resource managers are able to manage malleable distributed storage systems. The SCADS Director [14], for example, is a resource manager designed to ensure service-level objectives. It can add or remove storage nodes and move the data, as well as the number of replicas needed for each file. The SCADS Director adds malleability to the SCADS file system [15]. Its authors focus their evaluation on the ability of the system to maintain its service-level objective, however, and not on the performance of the rescaling operations themselves.

Lim, Babu, and Chase [16] propose a resource manager based on HDFS. This resource manager chooses when to add and remove nodes and the parameters of the operations. However, it simply uses HDFS "as is" and does not focus on efficiency. Both Trushkowsky et al. [14] and Lim et al. [16] focus on ways to leverage malleability rather that on improving it. Therefore their work is complementary to this paper.

In a previous work [6], we provided lower bounds for the time of a standard decommission in replication-based storage systems. This standard decommission required that the requested number of replicas for each object be maintained throughout the decommission operation. In this paper, we relax this constraint. We accept that the number of replicas of the stored objects drops below its normal value during the decommission. Of course, we still require that no data be lost and that the replication factor be brought back to its normal value at the end of the operation. This method of decommission trades better resource utilization for temporarily higher vulnerability to faults. Indeed, a storage node can be given back to the resource manager as soon as at least one replica of each of its data objects exists on remaining nodes. Until new replicas are created, however, a crash may lead to data loss. While this trade-off is not new, this paper aims to model it precisely.

III. Context and Problem Statement

In this section, we start by stating what the targeted storage systems are. We then define the fast decommission operation and the assumptions used in order to compute the lower bounds of its duration.

A. Targeted distributed storage systems

Many similarities exist between the fast decommission mechanism and the crash of a node followed by the recovery of the system. In both cases, some storage nodes become unavailable, and some data has to be recreated on the remaining nodes.

While several methods exist to recreate the missing data, in this paper we consider only the distributed storage systems that use *data replication*. This crash recovery mechanism is popular: it is used in HDFS [10], Rabbit [11], and Sierra [12], among others. It has the advantages of being simple and highly parallel: most of the remaining nodes share some replicas with the crashed nodes and thus can quickly restore the replication level to its initial value. Moreover, little CPU power is required for this technique.

We do not consider *full-node replication*, in which sets of nodes host exactly the replicas of the same objects, since the recovery mechanism is fundamentally different. *Erasure coding*, used in systems such as Pelican [17], is not considered either. With erasure coding, CPU power is needed to recreate missing data. Thus a mathematical model of the CPU would be required in order to compute a lower bound on the duration of such operations.

Another major recovery mechanism, *lineage*, also requires CPU power. Its principles greatly differ from those of data replication. When a node crashes, the data that is lost is recreated by executing again the jobs that generated it. Modeling the lower bounds of such an operation would require knowledge of the jobs that generated the data, however, and we therefore do not consider this recovery mechanism in this paper. Lineage is used in Tachyon [18] and is tightly coupled to Spark [19] in order to regenerate the data.

Despite the similarities between fault tolerance mechanisms and fast decommission, the fault tolerance mechanism cannot be used directly to decommission nodes. The fault tolerance mechanism of a distributed storage system has an upper limit on the number of nodes that can crash simultaneously. The fast decommission mechanism is an intentional operation. Hence, even if more nodes are decommissioned than the number of replicas, this decommission mechanism will prevent the loss of data by first making sure that at least one replica of each object exists in the remaining nodes.

B. Problem definition

We consider a replication-based distributed storage system deployed on a cluster of N nodes. Each node initially hosts an amount of data D. Each of the objects stored in the system is replicated r times. The resource manager requests the decommission of x arbitrarily chosen nodes.

A fast decommission is done in three main steps.

1) **Data-safekeeping:** During the data-safekeeping phase, the objects that are stored only on the leaving nodes have a replica transferred to remaining nodes to ensure that no data is lost during the operation.
2) **Nodes release:** The leaving nodes are given back to the resource manager. They no longer participate in the distributed storage.
3) **System stabilization:** The missing replicas are recreated by the remaining nodes to recreate the target replication degree.

We define the time to availability t_{avail} as the lower bound of the date at which the data-safekeeping terminates. The stabilization time t_{stab} is the lower bound on the time at which the whole process terminates; t_{stab} is obtained assuming that the leaving nodes participated only in the data-safekeeping phase and were all removed from the cluster at time t_{avail}.

C. Assumptions on the cluster infrastructure

We make three assumptions concerning the hardware of the cluster to provide comprehensive lower bounds.

Assumption 1: Homogeneous cluster

All nodes have the same characteristics, in particular the same network throughput (S_{Net}) and storage write and read throughputs (S_{Write}, S_{Read}).

Assumption 2: Ideal network

The network is full duplex, data can be sent and received with a throughput of S_{Net} at any time, and there is no contention.

Assumption 3: Ideal storage system

The writing speed is not higher than the reading speed ($S_{Write} \leq S_{Read}$). The storage device must share its I/O time between reads and writes and thus cannot sustain simultaneous reads and writes at maximum speed (during any span of time t, if a time $t_{Read} \leq t$ is spent reading, the storage cannot write for more than $t - t_{Read}$, and conversely).

Assumption 3 holds for most modern storage devices. Moreover, we assume that all resources are available to the decommission and that *either the network or the storage is the bottleneck.*

The network is the bottleneck if it limits the storage in any situation: $S_{Net} < S_{Read}$. Conversely, the storage is the bottleneck if it cannot read or write fast enough the data received and sent on the network: $\frac{S_{Read}S_{Write}}{S_{Read}+S_{Write}} < S_{Net}$. There exists a situation in which both storage and network are bottlenecks for different nodes. Although not presented in this paper this situation can be studied by extending the presented approach. We leave this for future work.

D. Assumptions on the initial data distribution

The initial data distribution is important for the performance of the decommission. Thus we make some assumptions in this respect.

Assumption 4: Even data distribution

All N nodes initially host the same amount of data D.

Assumption 5: Uniform data replication

Each object stored in the storage system is replicated on $r \geq 2$ distinct nodes. The probability of finding a given object on a node is uniform and independent of the node.

Assumption 6: Uniform data distribution

The probability of finding a given object on all the nodes in a set of r distinct nodes is uniform and independent of the chosen set.

These assumptions reflect the ideals of the load-balancing policies implemented in many state-of-the-art distributed file systems such as HDFS [10] and RAMCloud [20]. We assume that the data is initially in an ideal load-balanced state.

E. Formalizing the problem

At the end of the decommission operation, the data distribution on the remaining nodes should satisfy the following objectives.

Objective 1: No data loss

No data can be lost during the decommission.

Objective 2: Maintenance of an even data distribution

All nodes host the same amount of data D'.

Objective 3: Maintenance of a uniform data distribution

All sets of r distinct nodes host the same amount of exclusive data, independently of the choice of the r nodes.

These objectives ensure that the load balancing is ideal at the end of the decommission.

All the listed assumptions and objectives are common with standard decommission. The difference between both decommission strategies comes from Objective 4.

Objective 4: Maintenance of the replication factor

Each object stored on the storage system is replicated on r distinct nodes.

The fault tolerance requirements are relaxed during the execution of the decommission. Instead of ensuring the replication factor of the objects at any time during decommission, the replication factor is required to be at its initial level only at the end of the decommission. This relaxation is the main difference between the assumptions of this work and the ones for the lower bounds of the standard decommission established in our previous work [6]. The purpose of this paper is to quantify theoretically how many resources are saved by relaxing this constraint and how fast such a decommission process can be, compared with a standard decommission.

IV. Lower Bounds

In this section, we establish the lower bounds for the duration of the data-safekeeping and stabilization phases (the node release phase is assumed to be instantaneous). Because of space limitations, all proofs are provided in a separate research report.[1]

A. Data to move

Because data should not be lost during a decommission (Objective 1), a minimum amount of data has to be moved from the leaving nodes to the remaining ones. The objects to move are the ones that have all their replicas on the leaving nodes and that would have been lost had these nodes all been removed at the same time. Thus, we first compute the probability, p_i, for an object to have exactly i replicas on the leaving nodes. From it, we deduce the minimum amount of data to transfer to remaining nodes D_{avail}.

[1] https://hal.archives-ouvertes.fr/hal-01943964

$$p_i = \begin{cases} 0 & \text{if } i > r, \\ \frac{\binom{r}{i}\binom{N-r}{x-i}}{\binom{N}{x}} & \text{for } i \leq r. \end{cases} \quad \textbf{(Def. 1)}$$

$$D_{avail} = \begin{cases} NDp_r/r & \text{if } x \geq r \\ 0 & \text{in other cases.} \end{cases} \quad \textbf{(Def. 2)}$$

D_{stab} is the lower bound of the amount of data to move in order to recreate all replicas from the leaving nodes onto the remaining nodes. It is the amount of data that was initially present on the leaving nodes and includes D_{avail}.

$$D_{stab} = xD \quad \textbf{(Def. 3)}$$

Both D_{avail} and D_{stab} are lower bounds of their respective metric, they are obtained assuming that objects stored on the nodes can be divided as needed to perfectly balance the data on each node.

B. Case 1: Bottleneck at network level

In this section we assume that the network is the bottleneck for the data transfers required by the data-safekeeping and stabilization phases. The network is the bottleneck if it limits the storage in any situation ($S_{Net} < S_{Read}$).

1) Time to availability: During the data-safekeeping phase, only the leaving nodes send data to the remaining ones. As defined by Assumption 2, the network is ideal without interference, and each node can send and receive data with a bandwidth S_{Net} at the same time. Two possible bottlenecks may appear, however: either sending data from the leaving nodes or receiving the data on the remaining nodes.

Thus, the time to availability t_{avail} depends on the number of nodes leaving the cluster x, the amount of data to move D_{avail}, and the bandwidth of the network S_{Net}. We express t_{avail} as follows.

$$t_{avail} = \begin{cases} \frac{NDp_r}{rxS_{Net}} & \text{if } x \leq N/2 \\ \frac{NDp_r}{r(N-x)S_{Net}} & \text{otherwise.} \end{cases} \quad \textbf{(Prop. 1)}$$

2) Stabilization time: The safekeeping phase has the priority over the stabilization phase; decommissioned nodes have to be released as fast as possible. However, depending whether the bottleneck of the safekeeping phase is receiving or sending data, the stabilization can happen at the same time as the safekeeping without slowing down the later. Indeed, when the leaving nodes sending data are the bottleneck of the data-safekeeping phase, the remaining nodes do not have their network bandwidth saturated by the reception of the data. Thus, data exchanges needed to stabilize the storage can start before the end of the safekeeping phase without slowing it.

We denote as t_{over} the time gained on the duration of the stabilization phase by starting it before the end of the safekeeping phase.

$$t_{over} = \begin{cases} \frac{(N-2x)NDp_r}{rx(N-x)S_{Net}} & \text{if } x \leq N/2 \\ 0 & \text{otherwise.} \end{cases} \quad \textbf{(Prop. 2)}$$

From this, we obtain the time needed to stabilize the distributed storage system t_{stab}. The stabilization phase can use all available resources only after the end of the safekeeping phase, however, some overlap between the two phases reduces the duration of the stabilization by t_{over}. Thus, t_{stab} is defined by prop. 3.

$$t_{stab} = t_{avail} - t_{over} + \frac{D_{stab} - D_{avail}}{S_{Recv}} = \frac{xD}{(N-x)S_{Net}}. \quad \textbf{(Prop. 3)}$$

3) Observations: The lower bound for the whole operation (t_{stab}) is exactly the lower bound for the standard decommission established in our previous work [6] (in which the replication factor is maintained). Thus, one can relax the fault tolerance to release nodes faster (the fewer the‘ core-hours used, the better the overall platform utilization) without any difference in the length of the operation compared with standard decommission.

We also infer that keeping the leaving nodes after they have transferred the data needed for the fast decommission does not speed the duration of the operation: in all cases, receiving data on the remaining nodes is the bottleneck. It would, however, have an impact on the ability of the cluster to service read requests. The network is completely saturated by the stabilization, servicing any request would slow it.

In Figure 1, we observe the differences between a standard decommission and a fast decommission; with the fast decommission, the nodes are released in a fraction of the time needed to decommission nodes while maintaining the replication factor.

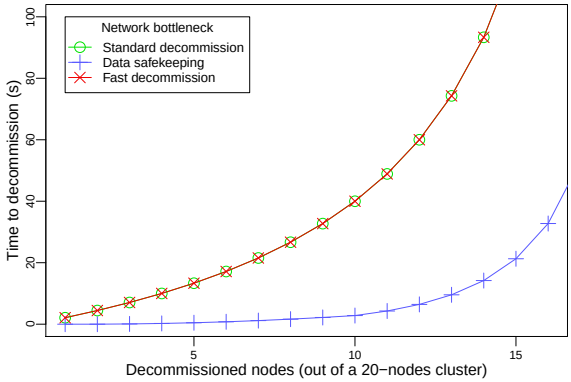

Fig. 1. Lower bounds for the duration of the data-safekeeping phase and fast decommission compared with the lower bound of the standard decommission, in case of a network bottleneck. Each node initially hosts 50 GiB of data; the network has a throughput of 1.25 GiB/s (10 Gib/s).

C. Case 2: Bottleneck at storage level

When the storage is the bottleneck, the situation is different because of the limitations of the storage devices (Assumption 3): data cannot be read and written at the same time. The storage is a bottleneck if it cannot read and write all the data received and sent on the network during the same period of time ($\frac{S_{Read}S_{Write}}{S_{Read}+S_{Write}} < S_{Net}$).

1) Time to availability: The limitations on the storage, however, do not have any impact on the time to availability since leaving nodes only have to read data, and remaining nodes only have to write it. Thus, the time to availability

depends on the data to move during the data-safekeeping phase D_{avail} and the reading and writing speeds of the storage devices S_{Read} and S_{Write}.

$$t_{avail} = \begin{cases} \frac{NDp_r}{rxS_{Read}} & \text{if } x \leq \frac{NS_{Write}}{S_{Read}+S_{Write}} \\ \frac{NDp_r}{r(N-x)S_{Write}} & \text{otherwise.} \end{cases}$$

(Prop. 4)

2) Stabilization time: Similar to the first case, when the bottleneck of the operation is reading data from the leaving nodes, the storage of the remaining nodes is not saturated: these nodes can read or write more data without slowing down the data-safekeeping process. Thus, the remaining nodes can exchange data to start the stabilization before the data-safekeeping finishes and without impact on the time to availability.

Each remaining node has some time t_{over} to exchange data with other remaining nodes in the data-safekeeping phase.

$$t_{over} = \begin{cases} \frac{(N-x)S_{Write}-xS_{Read}}{x(N-x)S_{Read}S_{Write}} & \text{if } x \leq \frac{NS_{Write}}{S_{Read}+S_{Write}} \\ 0 & \text{otherwise.} \end{cases}$$

(Prop. 5)

We determine S_{eff}, the effective writing speed on the cluster when the remaining nodes exchange data among themselves. S_{eff} is not simply the product of the number of remaining nodes by their individual writing speed. Indeed, to exchange data among themselves, remaining nodes also must read data.

To avoid reading multiple times data from storage devices with low read bandwidth, many systems use buffering. The data read is stored in memory (that has a higher bandwidth) and then sent to has many destinations as needed. The buffering relies on the bandwidth of the memory being a few times higher than the bandwidth of the storage device. We denote as R the ratio of data read to data written on the storage device during the stabilization.

$$R = \begin{cases} 1 & \text{in case of in-memory storage,} \\ \frac{\sum_{i=1}^{r-1} p_i}{(r-1)p_r+\sum_{i=1}^{r-1} ip_i} & \text{otherwise.} \end{cases}$$

(Prop. 6)

With the ratio R we deduce S_{eff}. Storage devices have their operation time divided between reads and writes (they cannot read and write at the same time). The cluster must also avoid imbalances between the data read and written. If too much data is read compared with the data written, the amount of memory needed to store it before writing it will increase. On the contrary, if too little data is read, the system will slow since storage devices will have to wait for data to write. Thus, the ratio of data read on data written during any given duration should be equal to R. From this we deduce S_{eff}.

$$S_{eff} = \frac{(N-x)S_{Write}S_{Read}}{S_{Read}+RS_{Write}}$$

(Prop. 7)

From the speed at which data is effectively exchanged on the cluster during the stabilization (Prop. 7), the amount of data to write (Def. 2 and 3), the duration of the overlap of data-safekeeping and stabilization (Prop. 5), and the time to availability (Prop. 4), we deduce the stabilization time t_{stab}.

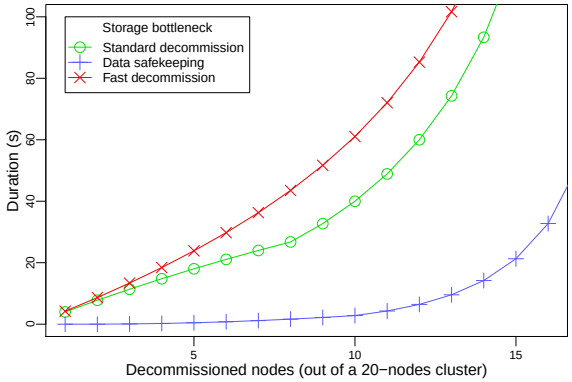

Fig. 2. Lower bounds for the duration of the data-safekeeping phase and fast decommission compared with the lower bound of the standard decommission, in case of a storage bottleneck. Each node initially hosts 50 GiB of data, and the reading and writing speed of the storage is set to 1.25 GiB/s.

$$t_{stab} = \frac{D}{N-x}\left(\frac{R}{S_{Read}}+\frac{1}{S_{Write}}\right)\left(x-\frac{Np_r}{r}\right)+\frac{NDp_r}{r(N-x)Sw}$$

(Prop. 8)

3) Observations: In the case of a storage bottleneck, the data-safekeeping phase and thus the effective decommission of the leaving nodes can be completed a lot faster than with the standard decommission. It is done, however, at the cost of a long stabilization phase: the leaving nodes were reading data in the case of a standard decommission, reading that must be done by remaining nodes in the case of a fast decommission. This situation implies that, contrary to the case of a network bottleneck, the longer the leaving nodes stay in the cluster, the faster the stabilization is. The stabilization cannot be faster than the standard decommission since the standard decommission is the extreme case in which the leaving nodes stay until the end of the stabilization.

During a fast decommission, the storage devices are fully saturated. Thus, servicing any request can only slow the decommission.

In Figure 2, we show the lower bounds for the duration of a standard decommission and of the data-safekeeping and stabilization phases of a fast decommission. Decommissioned nodes are available in a fraction of the time needed for a standard decommission. However, it comes at the cost of having the distributed storage system unable to operate for a longer time.

D. Core-hour usage

We study the core-hours usage of the strategies as it is linked to the financial and energetical cost of the active nodes during the operation. Comparing the consumption of core-hours equates to comparing the cost of the strategies.

In Figure 3 we compare the usage of core-hours for the standard decommission and the fast decommission in the case of a network bottleneck. Since the numbers are based on the lower bounds for the duration of the operations, the figure represents the lower bound for the core-hour consumption. We observe that using the fast decommission mechanism

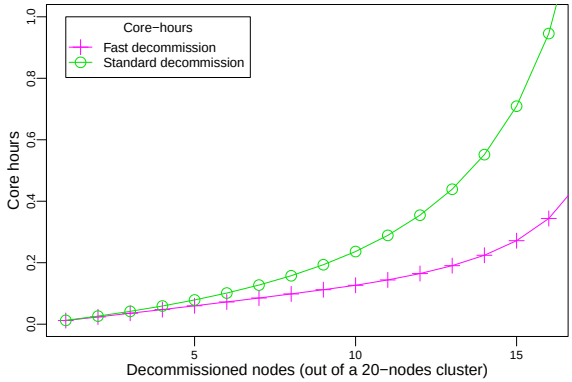

Fig. 3. Lower bounds for the number of core-hours used during a fast decommission compared with the standard decommission in the case of a network bottleneck. Each node initially hosts 50 GiB of data, and the network bandwidth is set to 10 Gb/s to match the values observed on the hardware used for the experiments in Section V.

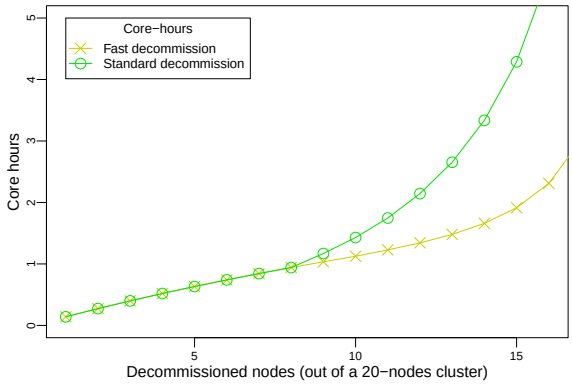

Fig. 4. Lower bounds for the number of core-hours used during a fast decommission compared with the standard decommission in the case of a storage bottleneck. Each node initially hosts 50 GiB of data, and the storage bandwidths are set to 207 MiB/s reading and 199 MiB/s writing to match the values observed on the hardware used for the experiments in Section V.

always reduces the core-hours consumption when the network is the bottleneck. Moreover, the gain increases greatly with the number of decommissioned nodes, and more than 50% of the core-hours consumption can be saved when many nodes are decommissioned at once.

In Figure 4 we compare the core-hours needed for the decommission in the case of a storage bottleneck. When few (less than $NS_{Write}/(S_{Read}+S_{Write})$, less than 8 in this case) nodes are decommissioned at once, there are no benefits in using the fast decommission compared with the standard decommission. When many nodes are decommissioned simultaneously, however, the core-hours consumption can be reduced by more than 50%.

V. Experimental Validation

In this section, we use Pufferbench to study the fast decommission mechanism in practice.

A. Implementing fast decommission in Pufferbench

Pufferbench [7] is a modular benchmark designed to evaluate how fast one can rescale a distributed storage system on a given infrastructure. We implemented the fast decommission mechanism in Pufferbench. Pufferbench computes and recreates on the hardware all the I/O that are requiered for a rescaling operation. It emulates a distributed storage system for the duration of a rescaling operation.

The leaving nodes transfer to the remaining ones only the data that is exclusively on them with high priority. The remaining nodes have to recreate the missing replicas; however, the operation is done with a lower priority. The leaving nodes can leave the cluster only after the data is on the storage device; they cannot leave if the data is only buffered in memory.

We also made sure that the implementation of the fast decommission matches the assumptions presented in Section III in order to be able to safely compare the lower bounds and the practical results.

B. Experimental setup

All measurements were done on Grid'5000 [21], the French experimental testbed. Experiments were done on the *grisou* cluster in Nancy. The cluster is composed of 51 nodes: 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. The nodes are all connected with a 10 Gb/s Ethernet network to a common Cisco Nexus 9508.

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

In order to create a network bottleneck, the data was stored in memory because it has a bandwidth (6 GiB/s reading, 3 GiB/s writing) significantly higher than the network's bandwidth. Similarly, in order to generate a storage bottleneck, the data was stored on the drives of the nodes (207 MiB/s reading, 199 MiB/s writing).

For each configuration (bottleneck and number of decommissioned nodes), ten measures of decommission times were done for fast decommission and for standard decommission.

In this evaluation, we use a 20-node cluster as the initial size, to show that the lower-bounds match the behavior observed in practice. For other scales of cluster, the analytical results should be used to determine the relevance of the fast-decommission.

C. Decommission when the network is the bottleneck

Figure 5 shows the duration of the data-safekeeping and stabilization phases when the network is the bottleneck. The standard decommission has been added for comparison.

Compared with the lower bounds, the time to availability is on average 37% slower, while the stabilization time is 32% slower. For the same configurations, the standard decommission is, on average, 22% slower than its lower bound. Note that the lower bounds cannot be reached in practice: they assume an absence of latency and permanent maximum throughput from both the storage and the network.

When few nodes are decommissioned (less than 6), the difference in duration between the two strategies is negligible.

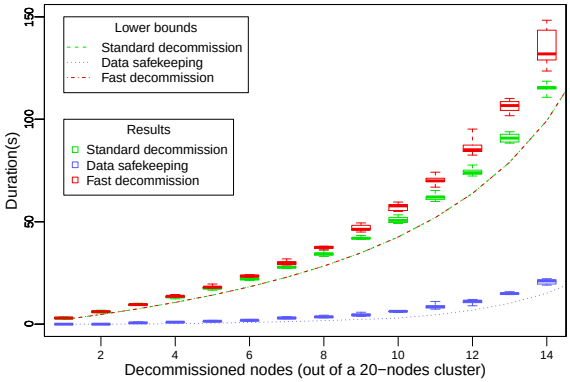

Fig. 5. Duration of the data-safekeeping phase, fast decommission and normal decommission obtained with Pufferbench and compared with the lower bounds, in the case of a network bottleneck. Each node initially hosted 50 GiB of data.

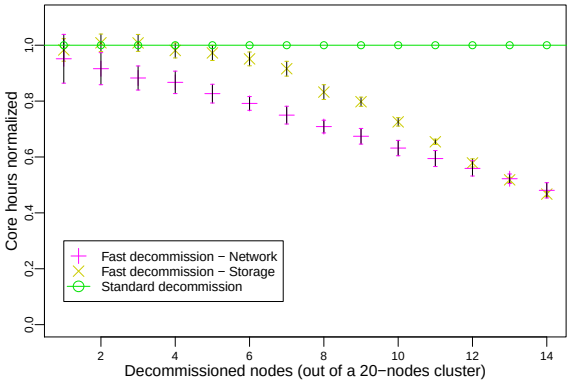

Fig. 6. Core-hours needed to do a fast decommission on a cluster of 20 nodes normalized by the standard decommission with the same bottleneck. The standard deviation has been added.

When many nodes are decommissioned at once, however, there is a large difference between the standard decommission and the time to stabilization. For example, the fast decommission is 12% slower than the standard decommission when 14 nodes are decommissioned. The reason for this difference is the stress on the network induced by the fast decommission. Indeed, during the fast decommission, the remaining nodes have to send and receive data at the maximum bandwidth speed in order to stabilize the system quickly. During the standard decommission, however, the sending load is distributed not only on the remaining nodes but also on the leaving nodes, reducing the overall load on each node. This difference does not appear on the lower bounds because we assume that the network is ideal (Assumption 2).

Figure 6 shows the number of core-hours consumed by decommission normalized by standard decommission. In most cases, using the fast decommission reduce the usage in the number of core-hours. The gain in core-hours increases with the number of decommissioned nodes. When most of the nodes are decommissioned at once, the fast decommission uses only 50% of the core-hours required by the standard decommission. The predictions about the core-hour usage (see Section IV-D) are confirmed by these results.

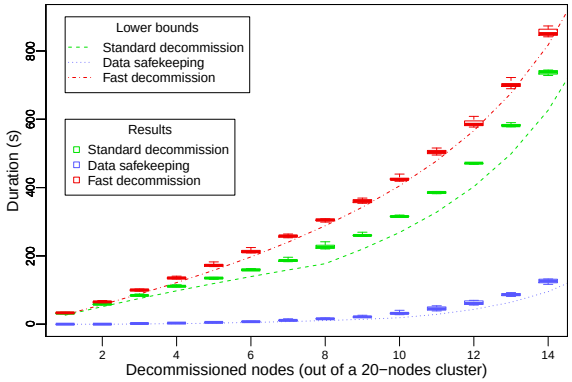

Fig. 7. Duration of the data-safekeeping phase, fast decommission and normal decommission obtained with Pufferbench and compared with the lower bounds, in the case of a storage bottleneck. Each node initially hosted 50 GiB of data.

When the network is the bottleneck, using fast decommission is a relevant choice. Even if fault tolerance is temporarily reduced, the overall operation is slightly slower, but there is a substantial gain in core-hours saved. This gain increases with the number of decommissioned nodes.

D. Decommission when the storage is the bottleneck

The time to availability and the stabilization time obtained with Pufferbench in the case of a storage bottleneck are presented in Figure 7. The duration of the standard decommission has been added for reference.

On average, the time to availability is within 10% of its lower bound, while the stabilization time is within 9% of its lower bound. In comparison, the standard decommission is within 17% of its lower bound. From this, we deduce that the lower bounds are sound and can almost be reached in practice.

Figure 6 shows the number of core-hours needed for the whole operation normalized by the core-hours needed for a standard decommission. Using a fast decommission offers no benefit in core-hours when the number of decommissioned nodes is low. In this case, the core-hours needed to stabilize the system are canceling the benefits of releasing the decommissioned nodes earlier. When a large number of nodes are decommissioned at once, however, the gain in core-hours can reach 50%. These results are in line with the core-hours that lower bounds established in Section IV-D.

Depending on the scenario, using fast decommission when there is a storage bottleneck can be detrimental or risky. If most of the decommission concerns just a few nodes, the fast decommission is detrimental: the fault tolerance is affected, the whole operation is slower than a standard decommission, and there are no gains in core-hours usage. If many nodes are decommissioned at once, the gains in core-hours may be worth the longer operation and the risk taken. However, the whole operation takes a long time, during which the fault tolerance is not ensured; thus there is a greater risk of losing data due to a crash.

VI. Discussion

In this section, we discuss some aspects of the lower bounds.

A. Simple assumptions for general conclusions

The assumptions used in this work are voluntarily few and simple to enable general conclusions, not specific to a particular platform or implementation. These assumptions were used in previous works [6] [8], where they helped closely model the behavior of HDFS.

B. Upper bound versus lower bound

Defining an upper bound for the operation is hardly possible since nothing prevents a node from waiting for an arbitrarily long time before sending any data to its destination. In contrast, a lower bound can be used in many useful ways, as explained in Section I.

C. Using the lower bound as a model

As shown in Section V, an efficient implementation of fast decommission exhibits a performance that varies with the number of decommissioned nodes in a way similar to the lower bounds. Thus, *we can use the lower bound as a model* for the fast decommission to predict the duration of the different phases. For instance, in the case of the network bottleneck, the lower bounds can be used as models with a coefficient of determination of 0.979 for the time to preservation and 0.995 for the time to stabilization.

Once fitted to the decommission mechanism of a distributed storage system, the model can be useful for resource managers to estimate the duration of a decommission and evaluate whether it is interesting to decommission nodes, when to do so, and which nodes to decommission.

D. Preserving $k > 1$ replicas

For the lower bounds presented in Section IV, the fault tolerance is simply ignored during the decommission: only one replica of each object is required. However, one may want to be able to tolerate $0 < k-1 < r$ faults during the decommission. In this case, at least $k > 1$ replicas of each object must be preserved on the remaining nodes before the leaving nodes are released.

Lower bounds for this situation can be defined. In the case of a network bottleneck (Prop. 9 and Fig. 8), the time to stabilization is the same as the standard decommission which is also the time to stabilization when maintaining only one replica. For the time to availability, we notice that receiving the data is always the bottleneck, indeed, due to the uniform data distribution (Assumption 6), each and every node hosts some objects that are also stored by leaving nodes, and they can replicate them among themselves.

$$t_{avail} = \sum_{i=1}^{k} ip_{r-k+i} \frac{ND}{r(N-x)S_{Net}}$$
$$t_{stab} = \frac{xD}{(N-x)S_{Net}} \qquad \textbf{(Prop. 9)}$$

In the case of a storage bottleneck (Prop. 10 and 11, and Figure 9) the time to availability is longer than when keeping only one replica. On the other hand, the time to stabilization is shorter. Indeed, since the leaving nodes stay for a longer time, their drives are used to read data during a longer duration, eventually reducing the reading load on the drives of the remaining nodes. Note, however, that reaching the lower bound of the stabilization time when $k > 1$ is hardly possible in practice since all the data transferred during the preservation would have to be kept in a buffer to reduce the reading overhead during the stabilization, which induces very large memory buffers.

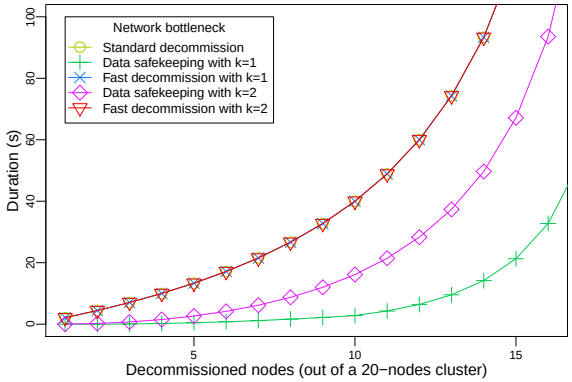

Fig. 8. Lower bounds for the duration of the data-safekeeping phase and fast decommission compared with the lower bound for the standard decommission in case of a network bottleneck for $k = 1$ and $k = 2$. Each node initially hosts 50 GiB of data, and the network bandwidth is set to 1.25 GiB/s.

Let R_{avail} be the ratio of the amount of data to read on the data to write during the data-safekeeping phase.

$$R_{avail} = \begin{cases} 1 & \text{for in-memory storage} \\ \frac{\sum_{i=1}^{k} p_{r-k+i}}{\sum_{i=1}^{k} p_{r-k+i}} & \text{in other cases.} \end{cases}$$

$$t_{avail} = \begin{cases} \sum_{i=1}^{k} ip_{r-k+i} \frac{D}{r} \frac{S_{Read}+R_{avail}S_{Write}}{S_{Write}S_{Read}} & \text{if } x < \frac{R_{avail}(N-x)S_{Write}}{S_{Read}} \\ \sum_{i=1}^{k} ip_{r-k+i} \frac{ND}{r(N-x)S_{Write}} & \text{in other cases.} \end{cases}$$

(Prop. 10)

Let R_{stab} be the ratio of the amount of data to read on the amount of data to write during the stabilization phase.

$$R_{stab} = \begin{cases} 0 & \text{in case of storage in-memory} \\ \frac{\sum_{i=1}^{r-k} p_i}{\sum_{i=1}^{r} ip_i - \sum_{i=1}^{k} ip_{r-k+i}} & \text{in other cases.} \end{cases}$$

$$t_{stab} = t_{avail} + \left(\sum_{i=1}^{r} ip_i - \sum_{i=1}^{k} ip_{r-k+i} \right) \frac{ND}{r} \frac{R_{stab}S_{Write} + S_{Read}}{(N-x)S_{Read}S_{Write}}$$

(Prop. 11)

VII. Conclusion

Efficient decommission is needed to leverage malleability in distributed storage systems. In this work, we study fast decommission, a decommission mechanism that makes the released nodes available to the resource manager as soon as possible by relaxing the fault tolerance. We provide lower bounds for the various steps required for this decommission,

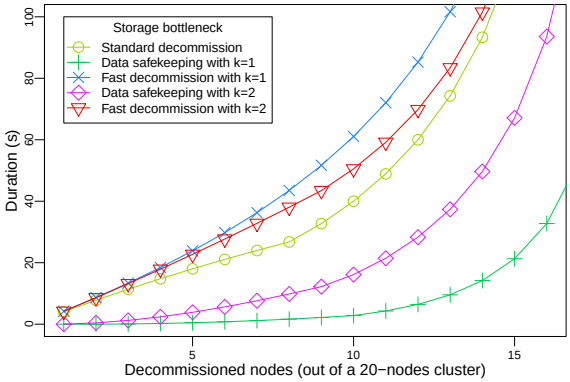

Fig. 9. Lower bounds for the duration of the data-safekeeping phase and fast decommission compared with the lower bound for the standard decommission in case of a storage bottleneck for $k = 1$ and $k = 2$. Each node initially hosts 50 GiB of data, and the reading and writing speed of the storage is set to 1.25 GiB/s.

and we validate them using a prototype implemented in Pufferbench.

We demonstrate that fast decommission allows to return the decommissioned nodes to the resource manager in a fraction of the time required by standard decommission. We show that in case of a network bottleneck, the duration of the whole operation is only slightly longer than for a standard decommission, while the core-hour usage is significantly reduced. In this situation, the choice of using fast decommission is relevant and can be considered by distributed storage system designers, or even decided at run time.

In the case of a storage bottleneck, however, using a fast decommission to release few nodes is detrimental to resource usage. The whole operation lasts longer than standard decommission; and although the released nodes are returned faster to the resource manager, there is no overall gain in core-hours.

In both cases, the more nodes are decommissioned at once, the higher the gain in core-hours. However, this also leads to higher risks since the fault tolerance is weakened for a longer period of time.

Using these lower bounds in order to design a resource manager fully aware of distributed storage system malleability is a challenge left for future work.

Acknowledgment

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] A. Kuzmanovska, R. H. Mak, and D. Epema, "KOALA-F: A Resource Manager for Scheduling Frameworks in Clusters," *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 592–595, 2016.

[2] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni, and S. Roa, "Morpheus: Towards Automated SLOs for Enterprise Clusters," *USENIX Symposium on Operating Systems Design and Implementation*, pp. 117–134, 2016.

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

[7] "Pufferbench," gitlab.inria.fr/Puffertools/Pufferbench.

[8] N. Cheriere, M. Dorier, and G. Antoniu, "Pufferbench: Evaluating and Optimizing Malleability of Distributed Storage," in *Proceedings of the 3rd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, 2018.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, A. Rowstron, P. England, R. Black, A. Donnelly, A. Glass, D. Harper, A. Ogus, E. Peterson, and A. Rowstron, "Pelican: A Building Block for Exascale Cold Data Storage," in *Operating Systems Design and Implementation*, 2014, pp. 351–365.

[18] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Reliable, Memory Speed Storage for Cluster Computing Frameworks," in *ACM Symposium on Cloud Computing*, 2014, pp. 1–15.

[19] 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.

[20] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2010.

[21] 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.