Pufferscale: Rescaling HPC Data Services for High Energy Physics Applications

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

Stefan M. Wild
Argonne National Laboratory
Lemont, IL, USA
wild@anl.gov

Sven Leyffer
Argonne National Laboratory
Lemont, IL, USA
leyffer@anl.gov

Robert Ross
Argonne National Laboratory
Lemont, IL, USA
rross@anl.gov

***Abstract*—User-space HPC data services are emerging as an appealing alternative to traditional parallel file systems, because of their ability to be tailored to application needs while eliminating unnecessary overheads incurred by POSIX compliance. The High Energy Physics (HEP) community is progressively turning towards such services to enable high-throughput accesses under heavy concurrency to billions of event data produced by instruments and consumed by subsequent analysis workflows. Such services would benefit from the possibility to be rescaled up and down to adapt to changing workloads, as experimental campaigns progress , in order to optimize resource usage. This paper formalizes rescaling a distributed storage system as a multi objective optimization problem considering three criteria: load balance, data balance, and duration of the rescaling operation. We propose a heuristic for rapidly finding a good approximate solution, while allowing users to weight the criteria as needed. The heuristic is evaluated with Pufferscale, a new rescaling manager for microservice-based distributed storage systems. To validate our approach in a real-world ecosystem, we showcase the use of Pufferscale as a means to enable storage malleability in the HEPnOS storage system for HEP applications.**

***Index Terms*—Distributed Storage System, Elasticity, Rescaling, Load balancing, High Energy Physics**

I. Introduction

User-space data services are becoming increasingly popular in the HPC community as an alternative to traditional file-based methods for data storage and processing [1], [2], [3]. Although file-based approaches (e.g., POSIX-compliant parallel file systems) have certain advantages, such as standard interfaces with well-understood semantics, ensuring these properties leads to drawbacks including suboptimal underlying data storage organization and high overheads to preserve POSIX semantics unneeded by many HPC applications [4]. In contrast, user-space data services can be tailored to their target applications to meet specific needs [1]. They are usually deployed on compute nodes alongside the applications that use them. Some of these services may need to remain deployed after the application has shut down and to be rescaled up and down to either conserve resources or adapt to changing workloads.

An example is of such data service is HEPnOS [1], [5], a distributed storage system for High Energy Physics (HEP) applications developed in the context of the SciDAC4 "HEP on HPC" joint project between Argonne and Fermilab [6], based on a methodology and software components developed by the Mochi project [7]. HEP applications at FermiLab have traditionally used files stored on parallel file systems to manage event data coming out of particle accelerator and through pipelines of post-processing applications. These files are written in CERN's ROOT format [8] or in HDF5 [9]. The file-based abstraction however hinders performance, since parallel file systems are generally not optimized for highly-concurrent, small random accesses. HEPnOS is designed to store billions of events produced by particle accelerator experiments and analysis workflows. Ultimately, users intend to deploy HEPnOS on a supercomputer for the duration of an experimental campaign (a few weeks to a few months), which comprises a series of applications, each of which consumes existing data and produces more data. As the experimental campaign progresses, some processing steps will require HEPnOS to span many compute nodes in order to sustain a high throughput under heavy concurrency. During other steps, HEPnOS should be scaled down so that some compute nodes allocated to it could be reassigned to other applications on the supercomputer.

Today HEPnOS is not *malleable*: it cannot be scaled up or down. Once deployed, all initially allocated nodes remain reserved until shutdown, which forces users to either (1) run short experimental campaigns, (2) deploy HEPnOS on a limited number of resources that may be inadequate for high workloads, or (3) overprovision it and waste resources that could have been used by other applications. Indeed shutting down HEPnOs and redeploying it at a different scale in between each application run would require temporarily exporting the stored data to a parallel file system, and ingesting it all back in before the next application starts, an operation that would be arguably too slow in practice.

Hence one requirement when designing the architecture of HEPnOS was to give it the ability to scale up or down *without being shut down*, depending on the needs of the applications that interact with it and on the needs of other, separate applications requesting time and resources on the supercomputer. Indeed as the experimental campaign progresses, some processing steps will require HEPnOS to be deployed across many storage nodes to sustain a high throughput under heavy concurrency; while HEPnOS should be scaled down during other periods so that compute nodes could be allocated to other users of the supercomputer.

HEPnOS is representative of a class of distributed storage

systems built by composing microservices [1]. These microservices each provide a reduced set of features, such as managing a database or handling remote accesses to nonvolatile memory devices. In such a design, rescaling consists of *migrating microservices and their associated resources* across a set of compute nodes. This capability to rescale poses a number of challenges.

Speed. Rescaling should be as fast as possible. Whether rescaling is done while the data service is actively being used by applications or when it is idle, fast rescaling will lead to better resource utilization overall by enabling other applications to reuse decommissioned resources (when the service is scaled down) and by speeding up the applications that use the service (when the service is scaled up).

Load balancing. Part of the data managed by the service may be "hotter" than others (e.g., accessed more frequently). In HEP campaigns, *selection* steps will go through a large set of events in parallel and identify some events with specific characteristics. These events will then repeatedly be used as input of subsequent processing steps. If most of these events are located in the same node, this node will become a bottleneck. Hence given that a load metric can be assigned to individual data items or groups of items, such a load should remain balanced across the active nodes on which the service runs (i.e., ideally, the I/O pressure should be uniformly distributed across those nodes). Note that this goal is different from that of *data balancing*, which aims to have each node store *the same amount of data*: in a perfectly data balanced configuration, nonuniform data accesses (corresponding to having some hot data more frequently accessed than other, colder data) may produce a load imbalance. Having a load balanced storage system mitigates hotspots, which in turn reduces I/O interferences, one of the root causes of performance variability in HPC applications [10], [11], [12], [13].

Why classical load rebalancing is not enough. An approach to address these challenges would be to use classical load-rebalancing strategies [14], which move data from the most-loaded servers to the least-loaded ones, starting with data with the highest load-to-data-size ratio. This strategy minimizes the amount of data to transfer, while balancing the load on the storage system.

Let us note, however, that *rescaling* cannot simply target *load rebalancing*: classical load rebalancing, as described above, may create data imbalance, leaving some nodes with either much higher or much lower volumes of data than other nodes, for instance when a few (or small) data items have a high load while a large number of (or larger) data items have a comparatively lower load. This data imbalance will slow down future rescaling operations: nodes hosting more data than others need more time to transfer out data during their decommission, thus slowing down the operation.

This paper aims to provide a comprehensive understanding of the factors that affect rescaling operations and to propose an effective approach to optimize their performance. **Our approach considers both load balance and data balance**. To enable efficient, *repeated* rescaling operations over a long period of time, one must jointly (1) optimize load balance, (2) minimize the duration of the current rescaling operation, and (3) ensure data balance to help speed up the subsequent rescaling operations. This problem is NP-hard [14] and cannot be solved exactly in a reasonable amount of time in the envisioned situations. A fast heuristic taking decisions in fractions of a second is required since rescaling can last only a few seconds.

This paper makes the following contributions.

Multiobjective problem (Sections III and IV). We formalize the rescaling problem above as a multiobjective optimization problem. We demonstrate the need to consider all three objectives by showing what happens when one or more of the objectives are overlooked by the rescaling algorithm.

Heuristic (Section V). We present a heuristic to manage data redistribution during a rescaling operation. This heuristic aims to reach a good trade-off between *load balance* and *data balance* for the final data placement and the duration of the rescaling operation.

Pufferscale (Section VI). We design and implement Pufferscale, a generic rescaling manager that can be used in microservice-based distributed storage systems. The roles of Pufferscale are to (1) track the data hosted on each node and records its size and load, (2) schedule the data migrations using the previous heuristic, and (3) start and stop microservices on compute nodes that are being respectively commissioned and decommissioned.

Evaluation (Section VII). We show the performance and usability of Pufferscale in experiments on the French Grid'5000 experimental testbed. We show in practice that one can consider both load balancing and data balancing with negligible impact on the quality of both and with only a 5% slowdown compared with strategies ignoring load balancing. Moreover, we showcase the use of Pufferscale with a combination of microservice components used in the HEPnOS storage systems, with the goal of enabling malleability in HEPnOS.

Additional aspects are discussed in Section VIII, followed by a conclusion in Section IX.

II. Related Work

Most distributed storage systems, such as Ceph [15] or HDFS [16] include tools to balance the amount of data hosted by each node. Some works such as that of Liao et al. [17] place the data uniformly from the beginning to avoid rebalancing.

Rocksteady [18] is a technique for scaling out RAMCloud clusters, it allows fast data migration while minimizing response times. This work is orthogonal to ours as it is focused on efficient point to point data transfers while ours determines which data transfers are needed by rescaling operations.

Operations to rebalance the data in the context of peer-to-peer storage systems have been extensively studied by Rao et al. [14], Ganesan et al. [19], and Zhu and Hu [20]. Similar techniques have been applied to distributed storage systems [21] and RAID systems [22]. However, all these works focus only on one criterion to balance: either the amount of data per node or the load (memory usage, CPU usage, etc.) and intend to do so as quickly as possible. Neither considers both objectives.

The multicriteria problem studied in this paper is close to that of rebalancing virtual machines on a cluster [23], [24], [25], [26], [27]. Each virtual machine needs some resources to perform nominally (CPU, memory, bandwidth, etc.), and each physical machine has limited capacity. Thus, virtual machines should be migrated to avoid overloading physical machines; in addition, as few virtual machines as possible should be migrated to limit the performance degradation. Arzuaga and

Kaeli [26] simplify the problem to one dimension by simply adding resource usage metrics and by balancing this sum. However, even if this sum of metrics is well balanced across the cluster, it may not be the case for each individual metric. Most works on balancing virtual machines [23], [24], [25], [27] are designed to keep the resource usage on each physical machine under a user-defined threshold while migrating as few virtual machines as possible. The work presented in this paper, however, aims to simultaneously load balance *and* data balance the cluster as fast as possible.

III. Formalization

In this section, we formalize the problem of data redistribution during a rescaling operation.

A. Rescaling operations

We consider two rescaling operations: *commissioning* (adding) storage servers or *decommissioning* (removing) storage servers. Rescaling a distributed storage system has to be done under one main constraint: objects initially stored must be available in the system at the end of the operation. In other words, no data can be lost during the operation.

In the context of this paper we assume that the storage service is directed to add or remove specific storage servers, and the task is to decide which data to move to meet our objective. The definition of this server selection policy is out of the scope of this work since it can depend on external factors, such as the arrival of new jobs that need to take over servers from the existing job. For the same reason, the list of servers to commission or decommission is assumed not to be known in advance.

B. Parameter description

We consider a homogeneous cluster. Let I be the set of storage servers involved in the operation (including servers that will be commissioned or decommissioned). We denote as I^- the set of storage servers to decommission and as I^+ the set of storage servers to commission. All nodes have a network bandwidth S_{net}. We assume (fast) in-memory storage; therefore the network is expected to be the bottleneck of the rescaling operation in this case. The alternative scenario where data storage causes a bottleneck is discussed in Section VIII-A. Each storage server has a storage capacity of C.

In this work, we focus on storage systems *without* data replication for two reasons: (1) HEPnOS, our targeted system, does not rely on data replication for fault tolerance, and (2) space constraints do not allow us to develop both cases.

We assume that storage servers can exchange collections of objects, which we call *buckets*, during the rescaling operations. Buckets are considered because rebalancing the storage system with a finer, object-level granularity would take too long due to the sheer number of objects. This strategy has been used in Ceph [15]. Each bucket j has a *size* ($Size_j$) and a *load* ($Load_j$), a generic, user-defined measure of the impact the bucket has on its host. For example, the load of a bucket of objects can be defined as the number of requests for the objects in the bucket over a time period. Let J be the set of buckets. Each bucket is initially stored on a single storage server. b^0 and b are matrices in $\{0,1\}^{|I|\times|J|}$ representing the placement of buckets before and (respectively) after the rescaling operation. $b^0_{ij} = 1$ if the bucket j is on the storage server i at the beginning of the operation. Similarly, $b_{ij} = 1$ if the bucket j is on the storage server i at the end of the operation.

C. Problem formalization

Our goal is to minimize at the same time the maximum load per node (load balancing), the duration, and the maximum amount of data per node (data balancing) (Eq. 1).

1) Load balance: The first objective is to reach a load-balanced distribution of buckets. We assume that each server is under a load equal to the sum of the load of the buckets that are placed on it. We denote as L_{max} (Eq. 2) the load of the most loaded server. A cluster is load-balanced when L_{max} is as low as possible. Although other metrics for load balance could be considered (*e.g.* the variance of the loads across servers or the entropy of the load distribution), using the maximum ensures that hotspots are avoided, while leaving some leeway to optimize the other objectives. Indeed, as long as the maximum load across the cluster does not increase, buckets can be transferred from servers to servers to optimize the data balance or left in place to reduce the duration of the operation.

2) Duration: The second objective is to minimize the duration of the rescaling operation T_{max} (Eq. 3). It is the maximum of the time needed to receive data and to send data across all servers. This reflects that most modern networks are full duplex and thus can send and receive data at the same time without interference. Latency is ignored since it is a negligible part of the time needed to transfer a bucket because of their large sizes.

3) Data balance: The third objective is data balancing: each server should host similar amounts of data. We denote as D_{max} (Eq. 4) the maximum amount of data on a single node. A cluster is data balanced when D_{max} is as low as possible.

4) Constraints: The objectives should be minimized while ensuring some constraints. Each node must have the capacity to host the data placed on it (Eq. 5). Each bucket must be placed on one and only one node (Eq. 6). No bucket should be hosted by decommissioned nodes (Eq. 7), since these nodes are leaving the cluster.

$$\text{Find } b \text{ minimizing } L_{max},\ D_{max}, \text{ and } T_{max}. \tag{1}$$

$$L_{max} = \max_{i\in I} \sum_{j\in J} b_{ij} Load_j \tag{2}$$

$$T_{max} = \max_{i\in I} \max\left(\sum_{\substack{j\in J \\ b_{ij}=1 \\ b^0_{ij}=0}} \frac{Size_j}{S_{net}}, \sum_{\substack{j\in J \\ b_{ij}=0 \\ b^0_{ij}=1}} \frac{Size_j}{S_{net}} \right) \tag{3}$$

$$D_{max} = \max_{i\in I} \sum_{j\in J} b_{ij} Size_j \tag{4}$$

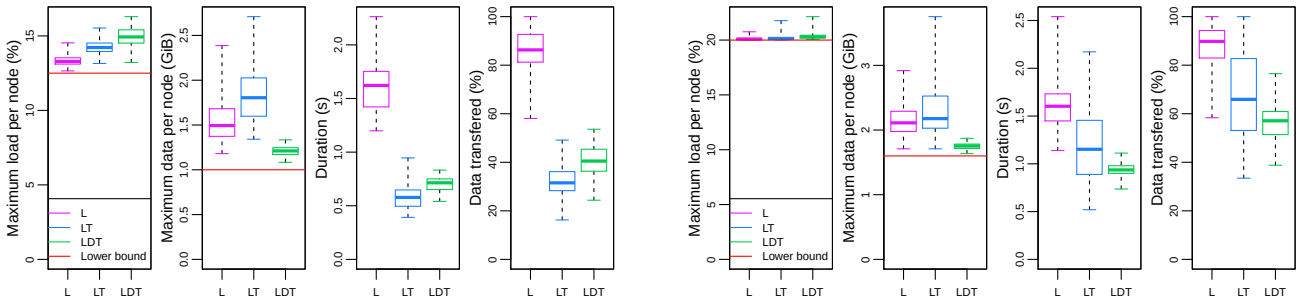
$$C \geq \sum_{j\in J} Size_j b_{ij} \qquad \forall i \in I \tag{5}$$

$$1 = \sum_{i\in I\setminus I^-} b_{ij} \qquad \forall j \in J \tag{6}$$

$$0 = \sum_{j\in J} b_{i^- j} \qquad \forall i^- \in I^- \tag{7}$$

IV. Need for Multiobjective Optimization

In this section, we discuss the multiple objectives we consider, their relevance, and their impact on the rescaling operations.



(a) Commission of 3 servers to a storage system with 5 initial servers

(b) Decommission of 3 servers out of a storage system with 8 servers

Fig. 1. Load balance, data balance, duration, and percentage of data transferred for the strategies L, LT, and LDT.[1]

A. Methodology

To study the relevance of each objective, we devise and evaluate several strategies for a rescaling operation and try to minimize various subsets of the objectives.

1) Problem size and cluster configuration: The optimal solutions for each strategy are obtained by using CPLEX, a solver for mixed-integer programming [28]. Because of the prohibitive compute time needed to get optimal solutions (1 h 40 min on average for one optimal solution on a cluster of up to 8 servers and 32 buckets), we focus in this section on smaller instances of the problem: only 16 buckets on up to 8 nodes. We consider 100 commissions with a cluster of 5 storage servers increased to 8 and 100 decommissions with a cluster of 8 storage servers reduced to 5. Each solution is obtained in 850 ms on average.

2) Distribution law: We generate buckets with sizes and loads that follow a normal distribution (standard deviation of 40%, for a total of 8 GB hosted on the cluster) to represent a bucket hosting hundreds of objects. Indeed, even if the load induced by a single file in a peer-to-peer setup is known to follow a Mandelbrot-Zipf distribution [29], the central limit theorem shows that the distribution of the load of a bucket can be approximated by a normal distribution with a standard deviation that decreases with the number of files in the collection. The same applies to the size of the buckets.

3) Initial data placement: The initial placement of the buckets on the nodes is obtained by executing sequentially 9 random rescaling operations (during each operation, the cluster is rescaled to a size randomly selected between 2 and 8 servers) using the same placement strategy as the one studied. This ensures that the initial data placement of the presented results is a consequence of the studied strategy. This warm up period is commonly used in simulations to reach a steady state from unlikely initial conditions [30].

B. Impact and relevance of each objective

Figure 1 respectively presents[1] the load balance, the duration, the data balance, and percentage of the stored data moved during the rescaling, for each of the three following strategies.

- **L:** Data is placed such that L_{max} is minimized.
- **LT:** The load balancing is relaxed; the data is placed so that L_{max} is within 10% of its optimal value (obtained with L) and T_{max} is minimized.
- **LDT:** The load balancing and data balancing are relaxed, and the duration is minimized. The data is placed so that L_{max} and D_{max} are within 10% of their optimal values, and T_{max} is minimized.

1) Load balance: In Figure 1, we can observe the performance of L, which focuses only on load balancing and is optimal for this criterion. The lower bound is below since it is estimated by the average load per node, and thus it is not a tight lower bound. However, the optimality of the load balance comes at the cost of many data transfers; *at least 85% of the data on the cluster was moved between nodes in half of the decommissions.* This is also reflected in the duration of operations: the median duration of the decommission is at least 1.8 times longer than with LDT.

These results can easily be explained by the fact that L solely focuses on finding an optimally load balanced data placement and is oblivious to the other objectives.

2) Duration: LT is a strategy focused on load balancing and fast rescaling. We observe in Figure 1(a) that most of the commissions are done about 3 times faster with LT than with L, since LT moves 3 times less data. Figure 1(b) shows that relaxing the load balancing helps speed decommission operations by a factor 1.4 compared with L. However, the strategy does not exhibit stable performance: the duration of operations can vary by up to 100%. The stability of the duration of rescaling operations can be obtained only by considering a third aspect: *data balance*.

3) Data balance: Data balancing is needed to speed up decommission operations and to stabilize the duration and performance of all rescaling operations.

In the case of a decommission, when the storage is data balanced, the duration of the operation is independent of the choice of the storage servers that are being removed: all storage servers host the same amount of data. If there is some data imbalance in the storage, the decommission could be faster if only storage servers hosting less data than average are decommissioned. Most likely, however, at least one storage server hosting more data than average is selected to be decommissioned, lengthening the duration of the operation.

In Figure 1(b) we observe that the duration of the decommission with LT can be up to 25% shorter than with LDT, but it can also be up to 2 times slower in some cases. Overall, LDT tries to satisfy all objectives and has less variability in duration, which can greatly help resource managers predict the performance of rescaling operations. However, it comes at the cost of higher load imbalance and longer commission

[1]Boxplots represent the min, first quartile, median, third quartile, and max.

operations (Figure. 1(a)): LDT requires more data movements to balance the data.

C. Why all three objectives are relevant

The three objectives are often mutually incompatible. For example, the fastest commission duration is not reachable if the cluster must be data balanced. Besides, there is no single hierarchy to order these objectives and minimize them one after another. Indeed, depending on the application, some of the objectives may not be as relevant as the others. For instance, a distributed storage that is not a bottleneck for the application using it may not need an ideal load balancing, but the job manager may need stable rescaling durations in order to anticipate the operations.

All three objectives should be taken into account when rescaling a distributed storage system. Thus, the rebalancing algorithm (which is also in charge of moving data out of decommissioned nodes) must consider load balance, duration, and data balance and find an equilibrium between them.

V. Heuristic

Calculating exact solutions for our multiobjective problem for realistic deployment scales is not feasible: it simply takes too long (see Sec. IV-A). Thus, a heuristic is needed to get fast approximations that are usable in practice.

A. Challenges

Traditional load rebalancing is usually a bi-objective optimization problem: the load must be balanced, but it should be done quickly. In the previous section, we showed that, to ensure efficient decommission for the long term, the data redistribution done during rescaling operations should consider three objectives simultaneously: load balance, data-balance and duration.

Like most multiobjective optimization problems, there is not just one optimal solution, thus the goal is to provide an acceptable trade-off. Moreover, because of the expected scale of the storage system (a few hundreds to thousands of nodes on a supercomputer) the rebalancing algorithm must compute a solution quickly. This makes computing an exact solution unusable (see Section IV-A). We need a fast heuristic that can be parametrized to provide solutions that balance the objectives according to the needs of each application.

B. Heuristic for the rescaling problem

The heuristic we design is a greedy algorithm inspired by the longest-processing-time-first rule and usual load-rebalancing mechanisms [31]. A greedy algorithm enables the transfer of buckets before a complete solution is computed.

It works in three steps: 1) estimate the target values for load-balancing, data-balancing, and duration; 2) select buckets that will not be moved (which we call "fixed" buckets); and 3) allocate the remaining buckets to destination storage servers (which may be the storage servers they are already on).

1) Determination of target values for metrics: The heuristic is designed to provide solutions that have their metrics as close as possible to target values; thus the computation of the target values is critical. We do not set the targets to 0 for two reasons, even if 0 is a relevant choice because we aim to minimize the objectives. First, setting realistic targets allows us to normalize the metrics with respect to these targets. Second, it prevents an imbalance between the objectives: none of the load balancing and data balancing metrics can reach 0 (since there are buckets on the storage, the lower bounds for these metrics is positive), but the duration can be 0 in case of a commission (leaving new nodes empty is valid). Thus, setting realistic targets helps avoid biases toward some objectives.

foreach *Storage server* i **do**
  $\forall j \in J$, set $b_{ij} = 0$;
  Let $J_i = \{j_1, j_2, ...\}$ be the buckets initially on i ordered by decreasing norm N (Eq. 14);
  Find the largest n such that $\sum_{k=1}^{n} Size_{j_k} \leq D_t$ and $\sum_{k=1}^{n} Load_{j_k} \leq L_t$;
  Allocate $\{j_1, ..., j_n\}$ to i: $\forall k \in [1, n], b_{ij_k} = 1$;
  Add $\{j_1, ..., j_n\}$ to J^a;

Algorithm 1: Fixing buckets

$$L_t = \frac{\sum_{j \in J} Load_j}{|I \backslash I^-|} \tag{8}$$

$$D_t = \frac{\sum_{j \in J} Size_j}{|I \backslash I^-|} \tag{9}$$

$$D_i = \frac{\sum_{j \in J} Size_j}{|I \backslash I^+|} \tag{10}$$

$$T_t = \begin{cases} \max\left(\frac{|D_t - D_i|}{S_{net}}, \frac{D_i}{S_{net}}\right) & \text{if } I^- \neq \emptyset \\ \max\left(\frac{|D_i - D_t|}{S_{net}}, \frac{D_t}{S_{net}}\right) & \text{if } I^+ \neq \emptyset \end{cases} \tag{11}$$

The target load per storage server L_t and the target amount of data per server D_t at the end of the rescaling operations are respectively the average load per storage server (Eq. 8) and the average amount of data per storage server (Eq. 9). The target duration T_t for the rescaling operations is more challenging to estimate. We approximate the initial data placements as perfectly data balanced; that is, each storage server initially hosts the same amount of data D_i (which can be computed by using Eq. 10) and will host D_t at the end of the operation. For a decommission (upper part in Eq. 11), the operation should last at least the time needed to empty the decommissioned servers of their buckets and at least the time needed for the remaining servers to receive those buckets. For a commission (lower part in Eq. 11), the duration is the maximum of the time needed to add enough buckets to new storage servers and the time required to send those buckets.

2) Fixing buckets: We call "allocated" a bucket for which the algorithm has determined a destination server. Among these buckets, we call "fixed" the ones that will not move from their current location. Let J^a be the set of allocated buckets during the execution of the algorithm. The fixing phase (Algorithm 1) consists of determining the buckets that will not be moved and adding them to J^a. The idea behind the fixing phase (Algorithm 1), is to avoid transferring the "largest" buckets. Buckets are ordered by decreasing norm N (Eq. 14), which is a combination of their load and size normalized by the total load and total size on the storage system (Eq. 12, Eq. 13). The fixing algorithm allocates to the node the largest buckets that fit within the target for load balancing and data balancing.

Sort unallocated buckets ($J \backslash J^a$) by decreasing N (Eq. 14);
foreach *Bucket* j **do**
  Let i^0 be the initial host of bucket j, $b^0_{i^0 j} = 1$;
  Find the server i that minimizes the penalty $P(i) + P(i^0)$ (computed assuming j has been allocated to i) ;
  Allocate j to i ($b_{ij} = 1$, $b^0_{i^0 j} = 0$);
  Add j to J^a;

Algorithm 2: Allocation of buckets

The remaining buckets will be allocated by Algorithm 2.

$$S_{Load} = \sum_{j \in J} Load_j \tag{12}$$

$$S_{Size} = \sum_{j \in J} Size_j \tag{13}$$

$$N(j) = \sqrt{\left(\frac{Load_j}{S_{Load}}\right)^2 + \left(\frac{Size_j}{S_{Size}}\right)^2} \tag{14}$$

3) Allocation of remaining buckets: The allocation phase (Algorithm 2) follows a greedy strategy: the buckets, taken in order of decreasing norm N (Eq. 14) are placed on the servers where so as to minimizes a penalty function (Eq. 15). The penalty is the sum of the cube of the value of each objective divided by their targeted value. Thus the bigger the value of an objective is compared with its targeted value, the higher is the penalty. The effect of this penalty function is to minimize the objective that is the least optimized.

$$P(i) = \left(\frac{\sum_{j \in J^a} b_{ij} Load_j}{L_t}\right)^3 + \left(\frac{\sum_{j \in J^a} b_{ij} Size_j}{D_t}\right)^3 + \frac{0.5}{(S_{net} T_t)^3} * \max\left(\sum_{\substack{j \in J^a \\ b^0_{ij}=0 \\ b_{ij}=1}} Size_j, \sum_{\substack{j \in J^a \\ b^0_{ij}=1 \\ b_{ij}=0}} Size_j\right)^3 \tag{15}$$

C. Enabling heuristic tuning

Since no ideal rebalancing exists for all situations, we added weights to provide the possibility to adapt the importance of each objective to the needs of the application. Let W_L, W_D, and W_T (such that $max(W_L, W_D) = 1$ and $W_T > 0, W_L > 0, W_D > 0$) be the weights for the load balancing, data balancing, and duration of the transfers, respectively. The higher the weight, the more important the objective.

$$L_i = \frac{\sum_{j \in J} Load_j}{|I \backslash I^+|} \tag{16}$$

$$L_{tw} = \begin{cases} \frac{L_t}{W_L} & \text{if } W_T < 1 \\ \frac{1}{W_L}((1 - \frac{1}{W_T})L_i + \frac{L_t}{W_T}) & \text{otherwise} \end{cases} \tag{17}$$

$$D_{tw} = \begin{cases} \frac{D_t}{W_D} & \text{if } W_T < 1 \\ \frac{1}{W_D}((1 - \frac{1}{W_T})D_i + \frac{D_t}{W_T}) & \text{otherwise} \end{cases} \tag{18}$$

$$T_{tw} = \frac{D_T}{W_T} \tag{19}$$

$$N_w(j) = \sqrt{\left(\frac{Load_j * W_L}{S_{Load}}\right)^2 + \left(\frac{Size_j * W_D}{S_{Size}}\right)^2} \tag{20}$$

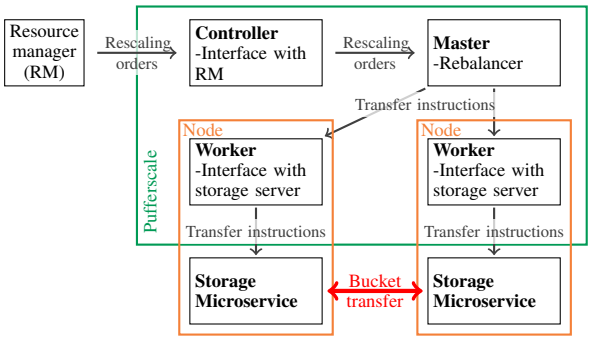


Fig. 2. Architecture of Pufferscale.

With these weights, we can adjust the target values in L_{tw}, D_{tw}, and T_{tw} for each objective. The target duration T_{tw} is simply scaled by its weight (Eq. 19) because the minimum for the duration is 0. But because the needed transfers to reach the targets L_t and D_t are estimated to last at least T_t, the targets for the load balancing and data balancing are adjusted as the weighted mean between the initial balancing and the targeted balancing. Then the data balancing and load balancing each are multiplied by their weights (Eq. 17 and Eq. 18). At least one of the weights W_L or W_D is set to 1 so that the heuristic aims to optimize at least one objective. A weighted norm is also used to order the buckets as shown in Eq. 20.

VI. Pufferscale

To evaluate our proposed heuristic, we implemented Pufferscale[1], a rescaling service developed in the context of the Mochi project [7]. This project aims at boosting the development of HPC data services thanks to a methodology based on the composition of building blocks that provide a limited set of features, accessible through remote procedure calls (RPC) and remote direct memory accesses (RDMA) and a threading/tasking layer [1]. As such, Pufferscale can be composed with other Mochi microservices to be integrated in various larger data services. Pufferscale's roles in such data services are to (1) keep track of the location of buckets, (2) schedule and request the migration of buckets from a node to another using the presented heuristic, and (3) request the deployment and shutdown of microservices on nodes that have to be respectively commissioned or decommissioned.

Pufferscale consists of the following components (Figure 2). The *controller* acts as an interface to send rescaling orders to the master from outside of the service. The *master* is the component that contains the heuristic and decides where each bucket should be placed during rescaling operations. The *workers* are the interfaces between the master and the *microservices* available on a given node. They are able to start and stop microservices on their node and forward the transfer instructions from the master to the corresponding microservices. The composition with the microservices is done by dependency injection. That is, the microservice registers callbacks that the workers can call to request the migration of a bucket or ask for information about the buckets present in the storage microservice.

[1]Pufferscale is available at gitlab.inria.fr/Puffertools/Pufferscale.

Pufferscale is not aware of the nature of the data that it is managing. It also does not handle data transfers itself. In the experiments presented in Section VII-D, the transfers are performed by REMI, Mochi's REsource Migration Interface [32], another microservice designed specifically to enable efficient file transfers across nodes using RDMA.

Pufferscale is written with about 3500 lines of C++ code and is implemented by using Mercury [33] for remote procedure calls and Argobots [34] for thread management.

VII. Evaluation

In this section, we evaluate the heuristic at a large scale using emulation. Then we showcase the use of Pufferscale to build the core of a real malleable storage system. For the first goal, we use a "dummy" storage microservice to evaluate Pufferscale's heuristic without making actual data transfers. We then use Pufferscale in a real-world setting, using the set of microservices used in the HEPnOS data service described in the introduction.

A. Scope of the evaluation

Note that a complete assessment of the advantages of rescaling a service would require evaluating (1) how fast a migration plan can be computed by Pufferscale; (2) how fast the rescaling and data migrations can be done; (3) how much of a performance speedup is obtained from running the application with a data service deployed at different scales; and (4) whether applications suffer from a slowdown during rescaling.

In this paper, due to space constraints and because of the focus on the scheduling heuristic, we restrict our evaluation to (1) and (2). The performance gains from the application perspective (3) depends on the application and will be studied in future works. As for whether applications suffer from a slowdown when a rescaling happens concurrently (4), this scenario is not envisioned by HEP use cases, where the service will be rescaled in between the execution of individual applications.

B. Evaluation of the heuristic

1) Setup: To evaluate the heuristic and its implementation at large, realistic scales, we emulate a storage system by implementing a "dummy" storage microservice that only transfers metadata about buckets but does not actually store or transfer data. Since no data is actually migrated between any two instances of this microservice, we estimate the duration of the rescaling using (Eq. 3). This setup allows us to scale up beyond the number of physical nodes available, by emulating multiple virtual storage servers on each physical node.

This distributed service can scale from 128 storage servers up to 2,048 storage servers on a 28-node cluster of the French Grid'5000 testbed [35]. Of the 28 nodes, one acts as a master, and one as a controller, and the other 26 each host up to 80 emulated storage servers. With this setup, we emulate 8,192 buckets with a load distribution following a normal law with 40% standard deviation proportionally adjusted to reach a total load of 100%. Similarly the amount of data stored in each bucket follows a normal law with 40% standard deviation for a total of 4,096 GB of data on the storage system. We distribute the buckets by doing 25 random rescaling operations as warm up with the same rescaling strategy as the one studied. Then, we consider the following rescaling scenarios: commission of 1,920 nodes to a storage system of 128 nodes, commission of 64 nodes to a storage system of 256 nodes, decommission of 1,920 nodes out of a storage system of 2,048 nodes, and decommission of 64 nodes out of a storage system of 320 nodes. Each rescaling operation is executed 100 times with 9 random rescalings between two recorded ones.

We record the load balance, the data balance, and the amount of data received and sent that allows us to estimate the duration of the data transfers under the best conditions.

In Figure 3 we compare the load balance, data balance, and duration of 3 strategies: `LDT` (optimization of all three objectives), `LT` (optimization of the load balance and of the duration of transfers), and `DT` (optimization of data balance and of the duration of transfers). Each of the strategies is obtained by modifying the weights in our heuristic.

We could not compare fairly our work with other works on rebalancing, because the latter do not comply with the strong constraints of the decommissions: nodes being decommissioned must have all their data transferred out to the other nodes, a constraint that is not enforced by other rebalancing algorithms. Moreover, to the best of our knowledge, there is no distributed storage system designed to be colocated with HPC applications that could serve as reference. Distributed storage systems designed to be colocated with applications exist in the cloud, like HDFS, but their rescaling mechanism relies on data replication (that is unneeded for our use-case) and are not optimized for speed [36], but to minimize their impact on application performance.

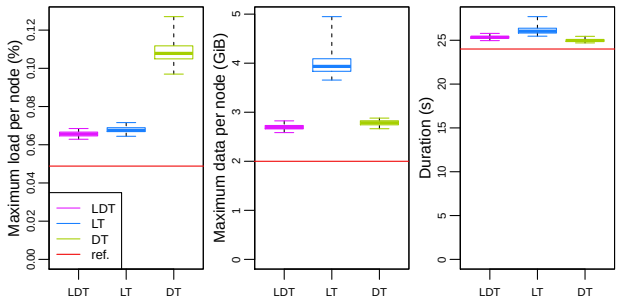
Instead, we added computed comparison points (*ref.*) on each of the graphs: for load balance and data balance we added the lower bounds (L_t (Eq. 8) and D_t (Eq. 9)). For duration, we added the lower bound of the duration of the transfers needed to transition from a perfectly data balanced storage system to another perfectly data balanced storage system (T_t, (Eq. 11)).

2) Results: Overall, the quality of the load balancing for `LT` and of the data balancing for `DT` compared with their respective lower-bounds depends on the average number of buckets on the servers at the end of the operation. It is on average within 2% when there are 128 servers after the rescaling operation (Figure 3(b)), and within 40% when there are 2,048 servers at the end of the operation (Figure 3(a)). Such a difference between the lower bounds and the obtained results is explained by the granularity of the load and data stored per node: the 8,192 buckets cannot be subdivided to perfectly balance the corresponding objective. The lower bounds, however, are the average per node, ignoring the granulary of the metrics.

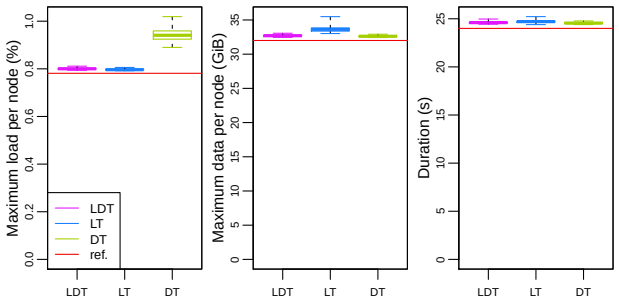
`LDT` combines of the benefits of the `LT` and `DT` strategies, without any major drawbacks. Its load balance is not significantly different from that of `LT`, and its data balance is similar to that of `DT`. It can be up to 5% slower than `DT` since `LDT` has more transfers to do in order to maintain both the data balance and the load balance.

Moreover, compared with `LT`, the range of duration (difference between the longest and shortest operation) of `LDT` is shorter by 32% to 70%, highlighting the fact that data balancing is needed to make operations faster (Figure 3(a) and Figure 3(d)) and have a more stable, and thus more predictable rescaling duration.

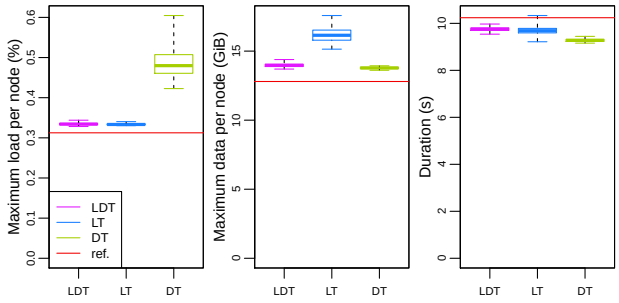
The computation of the heuristic was done in at most 612 ms



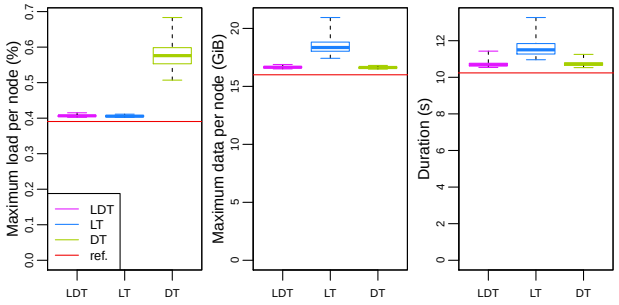
(a) Commission of 1920 to a cluster of 128



(b) Decommission of 1920 to a cluster of 2048

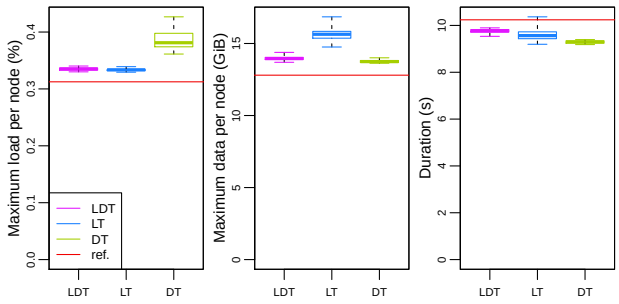


(c) Commission of 64 to a cluster of 256

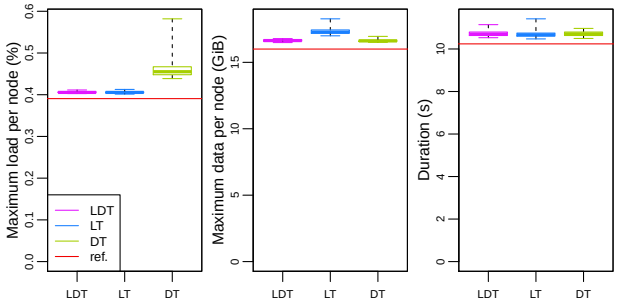


(d) Decommission of 64 to a cluster of 320

Fig. 3. Maximum load, amount of data per node, and duration of different rescaling operations.



(a) Commission of 64 to a cluster of 256



(b) Decommission of 64 to a cluster of 320

Fig. 4. Maximum load, amount of data per node, and duration of different rescaling strategies starting from a load and data balanced data placement.

for the commission of 1,920 nodes while the data transfers themselves lasted for 25 s. The impact of this delay can be reduced by starting the transfers as soon as the destination is decided by the heuristic.

C. Impact of the initial data balance

1) Setup: Using the same setup as in the previous experiment, we conduct a different set of measurements. After a warm-up of 25 operations that follow the `LDT` strategy, we switch the strategy and perform one rescaling operation. This ensures that the bucket placement before the last operation is data balanced and load balanced. Each measure is repeated 50 times with newly generated bucket sizes and loads.

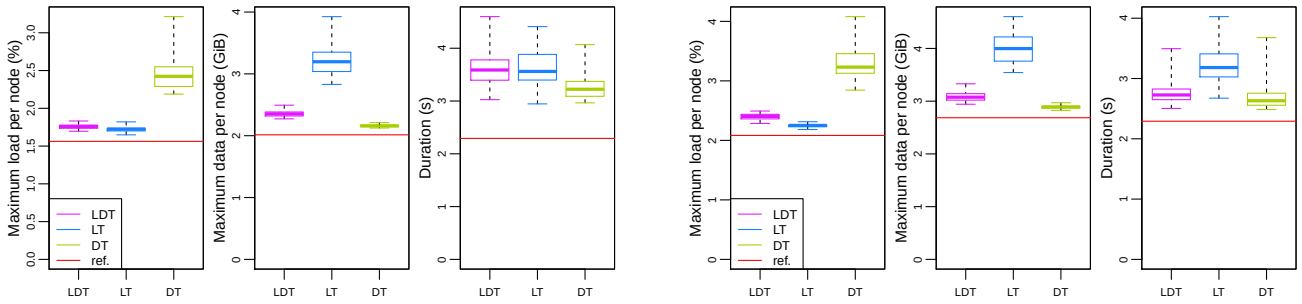
2) Results: By looking at the results of the commission (Figure 4(a)) and comparing them with those of the previous experiment (Figure 3(c)), we can see that starting from a data balanced bucket placement has no impact on the commission operation for any of the strategies.

In contrast, we can observe that an initially data balanced bucket placement has an important impact on the duration of the decommission operation (Figure 4(b) and Figure 3(d)): all strategies have a similar duration. This shows that, independently of any other value, enforcing data balance is required to improve the duration of rescaling operation on the long term.

D. Pufferscale in HEPnOS

In this section, we showcase the use of Pufferscale with a composition of microservices corresponding to the HEPnOS use case described in the introduction. We composed SDSKV [37], an in-memory, single-node key-value store acting as a storage microservice, REMI, a microservice designed to efficiently transfer files between nodes, and Pufferscale, to build the base of an elastic version of HEPnOS. Contrary to the previous experiments, databases are transferred between nodes, and the duration of the rescaling operations is recorded.

The rescaling operations of this composition were evaluated on the *paravance* cluster of the Grid'5000 testbed. This cluster is composed of 72 nodes each with 16 cores, 128 GiB of RAM, and a 10 Gbps network interface. At the maximum size, 64 nodes were used to host databases, and another one



(a) Commission of 16 to a cluster of 48

(b) Decommission of 16 from a cluster of 64 servers

Fig. 5. Maximum load, amount of data per node, and duration for two rescaling operations of the composition of Pufferscale, SDSKV, and REMI.

node served as controller to issue rescaling orders. Hosted by the SDSKV microservice instance were 256 databases, acting as buckets, each with a load following a normal distribution (with 40% standard deviation) and a size following a normal distribution with a mean of 512 MiB and a standard deviation of 40%. Similarly to the previous evaluation (Section VII-B), we distributed the databases by doing 25 random rescaling operations as warmup. Then, each rescaling operation was executed 50 times with 9 random rescalings in between two recorded ones. We add the same references on the figures, with the difference that the network bandwidth is the one recorded when benchmarking RDMAs on the cluster: 900 MiB/s. The network bandwidth is not maximized because of the emulation of RDMAs over a TCP network as well as some overhead in libfabric's socket provider used by Mercury.[2]

The load balance, data balance, and duration of the rescaling operations are presented in Figure 5. Comparing `LT` and `LDT`, we observe trends similar to that of Section IV: because `LDT` considers the data balance, its decommission operations are on average 13% faster than `LT`, but there are no significant changes for the commission operations. Even if the data balancing of `LDT` is on average 10% worse than `DT`, it does not have a significant impact on the duration of the decommissions: both strategies ensure similar durations. The large duration variability is due to the data transfers (neither the network nor the scheduling of the databases transfers is perfect, which adds interferences). The variability is likely to be increased by the emulation of RDMAs over a TCP network. In the case of the commission, `LDT` is slower than `DT` because of a higher number of database transfers that induces more network interferences.

An overall conclusion of all these experiments is the following: it is worth considering data balancing in addition to load balancing for rescaling storage systems. This helps reduce the rescaling duration with a negligible impact on load balance when sending data is the bottleneck of the rescaling, without any negative impact on the duration in the other cases.

VIII. Discussion

In this section, we discuss some assumptions made on Pufferscale, as well as some aspects related to generalizing the approach.

A. Storage bottleneck

$$T_{max} = \max_{i \in I} \left(\sum_{\substack{j \in J \\ b_{ij}=0 \\ b^0_{ij}=1}} \frac{Size_j}{S_{write}} + \sum_{\substack{j \in J \\ b_{ij}=1 \\ b^0_{ij}=0}} \frac{Size_j}{S_{read}} \right) \tag{21}$$

$$T_t = \begin{cases} \max\left(\frac{|D_t - D_i|}{S_{write}}, \frac{D_i}{S_{read}}\right) & \text{if } I^- \neq \emptyset \\ \max\left(\frac{|D_i - D_t|}{S_{read}}, \frac{D_t}{S_{write}}\right) & \text{if } I^+ \neq \emptyset \end{cases} \tag{22}$$

The objective T_{max} is written under the assumption of a network bottleneck. However, the work of this paper can easily be adapted to the case of a storage bottleneck. The particularity of storage devices is that they cannot sustain simultaneous reads and writes at maximum speed thus the duration of the I/O operations has to be modeled as the sum of the time taken to read data and the time taken to write data. Therefore, Eq. 3 should be replaced with Eq. 21, and Eq. 11 should be replaced with Eq. 22.

B. Generality of Pufferscale

Though Pufferscale was motivated by the need for rescaling the data service in HEP workflows, it can be used more generally in any service based on the Mochi components [1], and its principles could be applied to other user-space data services. For example, we are planning to adapt it to FlameStore, a storage system for caching deep neural networks.

C. Adjusting the weights

Giving the user the possibility to adjust the weights of the heuristic enables the user to tune the heuristic to the needs of the application. Users could also fine-tune these weights using some training runs in which the weights are adjusted in some outer loop, e.g., using derivative-free optimization solvers (DFO solvers) such as POUNDERS [38].

For instance, data balance is required mostly for decommissions. Thus, if few decommissions will be required, it makes sense to reduce the importance of the data balance.

If the load and the size of buckets are volatile, the load balance and data balance could be relaxed, since they will quickly change after the rescaling operation.

[2]https://ofiwg.github.io/libfabric/master/man/fi_provider.7.html: "Socket [...] This provider is not intended to provide performance improvements over regular TCP/UDP sockets, but rather to allow developers to write, test, and debug application code even on platforms that do not have high-speed networking."

D. Bucket replication

In this work we focused on the case where buckets are not replicated, which is often the case in state-of-the-art user-level data services, such as HEPnOS. If the storage system replicates buckets for fault tolerance, the focus of the heuristic should be different: data balance would be less important since bucket replication can be leveraged both to balance the load across the nodes and to speed up rescaling operations. This would be a means to greatly reduce the risk to experience bottlenecks on servers sending data, which reduces the importance of data balance. This is an open direction for future work.

IX. Conclusion

This paper formalizes the problem of rescaling a distributed storage system while simultaneously considering three optimization criteria: load balance, data balance, and duration of the rescaling operation. Since computing an exact solution to this multiobjective optimization problem takes too long, we introduce a heuristic that helps find a good approximate solution in a much shorter time. Users are provided with the possibility to weight each criterion as needed to reach the desired trade-off across the three criteria. To evaluate our heuristic, we introduce Pufferscale, a generic rescaling manager for microservice-based distributed storage systems. Our large-scale evaluation of the proposed heuristic with Pufferscale exhibits the importance of maintaining data balance in order to systematically ensure fast rescaling when data reading generates a bottleneck, with no drawbacks in the other cases. We showcase the use of Pufferscale as a means to enable storage malleability in HEPnOS, a real-world microservice-based storage system. The study of this problem under the assumption of data replication is 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] M. Dorier, P. Carns, K. Harms, R. Latham *et al.*, "Methodology for the rapid development of scalable HPC data services," in *Parallel Data Storage & Data Intensive Scalable Computing Systems*, 2018.
[2] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, "DeltaFS: Exascale File Systems Scale Better without Dedicated Servers," *Parallel Data Storage Workshop*, 2015.
[3] J. M. Wozniak, P. E. Davis, T. Shu, J. Ozik *et al.*, "Scaling deep learning for cancer with advanced workflow storage integration," in *IEEE/ACM Machine Learning in HPC Environments*, 2018.
[4] M. I. Seltzer and N. Murphy, "Hierarchical file systems are dead," in *Conference on Hot Topics in Operating Systems*, 2009.
[5] xgitlab.cels.anl.gov/sds/HEPnOS, accessed March 29, 2019.
[6] computing.fnal.gov/hep-on-hpc/, accessed March 29, 2019.
[7] www.mcs.anl.gov/research/projects/mochi/, accessed March 29, 2019.
[8] R. Brun and F. Rademakers, "Root—an object oriented data analysis framework," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 389, no. 1-2, pp. 81–86, 1997.
[9] The HDF Group. (1997-NNNN) Hierarchical Data Format, version 5. Http://www.hdfgroup.org/HDF5/.
[10] J. Lofstead, F. Zheng, Q. Liu, S. Klasky *et al.*, "Managing variability in the IO performance of petascale storage systems," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.
[11] Q. Liu, N. Podhorszki, J. Logan, and S. Klasky, "Runtime I/O re-routing+ throttling on HPC storage," in *HotStorage*, 2013.
[12] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *IEEE Parallel and Distributed Processing Symposium*, 2014.
[13] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the root causes of cross-application I/O interference in HPC storage systems," in *IEEE Parallel and Distributed Processing Symposium*, 2016.
[14] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in structured P2P systems," in *International Workshop on Peer-to-Peer Systems*, 2003.
[15] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *USENIX Symposium on Operating Systems Design and Implementation*, 2006.
[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," *IEEE Symposium on Mass Storage Systems and Technologies*, 2010.
[17] J. Liao, Z. Cai, F. Trahay, and X. Peng, "Block placement in distributed file systems based on block access frequency," *IEEE Access*, 2018.
[18] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman, "Rocksteady: Fast migration for low-latency in-memory storage," in *ACM Symposium on Operating Systems Principles*, 2017.
[19] P. Ganesan, M. Bawa, and H. Garcia-Molina, "Online balancing of range-partitioned data with applications to peer-to-peer systems," in *Conference on Very Large Data Bases*, 2004.
[20] Y. Zhu and Y. Hu, "Efficient, proximity-aware load balancing for DHT-based P2P systems," *IEEE Transactions on Parallel and Distributed Systems*, 2005.
[21] H.-C. Hsiao, H.-Y. Chung, H. Shen, and Y.-C. Chao, "Load rebalancing for distributed file systems in clouds," *IEEE Transactions on Parallel and Distributed Systems*, 2013.
[22] A. Miranda and T. Cortes, "CRAID: Online RAID upgrades using dynamic hot data reorganization," in *FAST*, 2014.
[23] G. Khanna, K. Beaty, G. Kar, and A. Kochut, "Application performance management in virtualized server environments," in *IEEE Network Operations and Management Symposium*, 2006.
[24] T. Wood, P. J. Shenoy, A. Venkataramani, M. S. Yousif *et al.*, "Black-box and gray-box strategies for virtual machine migration," in *NSDI*, 2007.
[25] A. Singh, M. Korupolu, and D. Mohapatra, "Server-storage virtualization: integration and load balancing in data centers," in *ACM/IEEE conference on Supercomputing*, 2008.
[26] E. Arzuaga and D. R. Kaeli, "Quantifying load imbalance on virtualized enterprise servers," in *WOSP/SIPEW International Conference on Performance Engineering*, 2010.
[27] H. Shen, "RIAL: Resource intensity aware load balancing in clouds," *IEEE Transactions on Cloud Computing*, 2017.
[28] "IBM ILOG CPLEX Optimization Studio (version 12.8.0.0)," https://www.ibm.com/products/ilog-cplex-optimization-studio, 2018.
[29] M. Hefeeda and O. Saleh, "Traffic modeling and proportional partial caching for peer-to-peer systems," *IEEE/ACM Transactions on Networking*, 2008.
[30] P. S. Mahajan and R. G. Ingalls, "Evaluation of methods used to detect warm-up period in steady state simulation," in *Proceedings of the 2004 Winter Simulation Conference, 2004.*, vol. 1. IEEE, 2004.
[31] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, 1969.
[32] xgitlab.cels.anl.gov/sds/remi, accessed March 29, 2019.
[33] mercury-hpc.github.io/, accessed March 29, 2019.
[34] argobots.org, accessed March 29, 2019.
[35] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez *et al.*, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, 2013.
[36] N. Cheriere, M. Dorier, and G. Antoniu, "Pufferbench: Evaluating and Optimizing Malleability of Distributed Storage," in *Parallel Data Storage & Data Intensive Scalable Computing Systems*, 2018.
[37] xgitlab.cels.anl.gov/sds/sds-keyval, accessed March 29, 2019.
[38] S. M. Wild, "Chapter 40: POUNDERS in TAO: Solving Derivative-Free Nonlinear Least-Squares Problems with POUNDERS," in *Advances and Trends in Optimization with Engineering Applications*. SIAM, 2017.