Bursting the Cloud Data Bubble: Towards Transparent Storage Elasticity in IaaS Clouds

Bogdan Nicolae
IBM Research, Ireland
bogdan.nicolae@ie.ibm.com

Pierre Riteau
University of Chicago, USA
priteau@uchicago.edu

Kate Keahey
Argonne National Laboratory, USA
keahey@mcs.anl.gov

***Abstract*—Storage elasticity on IaaS clouds is an important feature for data-intensive workloads: storage requirements can vary greatly during application runtime, making worst-case over-provisioning a poor choice that leads to unnecessarily tied-up storage and extra costs for the user. While the ability to adapt dynamically to storage requirements is thus attractive, how to implement it is not well understood. Current approaches simply rely on users to attach and detach virtual disks to the virtual machine (VM) instances and then manage them manually, thus greatly increasing application complexity while reducing cost efficiency. Unlike such approaches, this paper aims to provide a transparent solution that presents a unified storage space to the VM in the form of a regular POSIX file system that hides the details of attaching and detaching virtual disks by handling those actions transparently based on dynamic application requirements. The main difficulty in this context is to understand the intent of the application and regulate the available storage in order to avoid running out of space while minimizing the performance overhead of doing so. To this end, we propose a storage space prediction scheme that analyzes multiple system parameters and dynamically adapts monitoring based on the intensity of the I/O in order to get as close as possible to the real usage. We show the value of our proposal over static worst-case over-provisioning and simpler elastic schemes that rely on a reactive model to attach and detach virtual disks, using both synthetic benchmarks and real-life data-intensive applications. Our experiments demonstrate that we can reduce storage waste/cost by 30–40% with only 2–5% performance overhead.**

***Keywords*-cloud computing; elastic storage; adaptive resizing; I/O access pattern prediction**

I. Introduction

Infrastructure clouds (Infrastructure-as-a-Service, or IaaS clouds) [1] are increasingly gaining popularity over privately owned and managed hardware. One of the key features driving their popularity is *elasticity*, that is, the ability to acquire and release resources on-demand in response to workloads whose requirements fluctuate over time. However, elasticity presents a different optimization opportunity: rather than molding a problem to fit a fixed set of resources in the most efficient way as is the case in traditional high performance computing (HPC) centers, we now fit resources – from a flexible and extensible set – to the problem.

To date, most of the efforts have focused on exploiting the elasticity of computational resources, ranging from localized virtual clusters [2], [3] to approaches that facilitate elasticity across cloud federations [4], [5]. However, *elasticity of storage* has gained comparatively little attention, despite continuous explosion of data sizes and, as a response, the rise of data-intensive paradigms and programming models (such as MapReduce [6] and its vast ecosystem) that are highly scalable and capable of processing massive amounts of data over short periods of time.

In this context, *a growing gap is forming between the actually used storage and the provisioned storage*. Since traditional IaaS platforms offer little support to address storage elasticity, users typically have to manually provision raw virtual disks that are then attached to their virtual machine (VM) instances. All details related to the management of such raw disks, including what size or type to pick, how to use it (e.g., with what file system) and when to attach/detach a disk are handled manually and increase the application complexity. In response, users often simply over-provision storage, an action that leads to unnecessarily tied-up resources and, since the user has to pay for all the provisioned storage, also to overpaying. Thus, this gap significantly contributes to rising storage costs, adding to the costs caused by natural accumulation of data.

As a consequence, there is a need for an elastic storage solution that narrows the gap between the required and provisioned storage described above. For this to be possible, three major requirements need to be addressed. First, in order to minimize wasted storage space, elasticity needs to be implemented in a highly dynamic fashion, such that it can adapt to large fluctuations over short periods of time and match the provisioned storage space to the needs of the application as closely as possible. Second, it must exhibit low performance overhead, such that it does not lead to a significantly longer application runtime that threatens performance requirements or incurs extra costs that offset the savings gained by using elastic storage. Third, elasticity must be achieved in a *transparent* fashion, such that it hides all details of raw virtual disk management from the users and facilitates ease-of-use.

This paper contributes such a transparent elastic storage solution that presents a unified storage space to the VM in the form of a regular POSIX file system that hides all the details of attaching and detaching virtual disks. Our approach is designed to deal with data-intensive workloads that exhibit large fluctuations of storage space requirements over short periods of time. In addition to technological choices, the main difficulty in this context is to anticipate the application intent and proactively attach and detach disks such as to minimize the wasted storage without significant performance overhead. To

this end, we propose a prediction scheme that correlates different I/O statistics in order to optimize the moment when virtual disks should be attached or detached without compromising normal application functionality by prematurely running out of space.

Our contributions can be summarized as follows:

- We describe requirements and design considerations that facilitate transparent elasticity for cloud storage. In particular, we show how to leverage multi-disk aware file systems to circumvent the difficulty of resizing virtual disks on-the-fly. To this end, we advocate for a predictive scheme that anticipates near-future storage space requirements based on fine granularity-monitoring (Section III-A)
- We show how to apply these design considerations in practice through a series of building blocks (along with their associated algorithmic description and implementation) that integrate with a typical IaaS cloud architecture. (Sections III-B, III-C and III-D)
- We evaluate our approach in a series of experiments conducted on dozens of nodes of the Shamrock experimental testbed, using both synthetic benchmarks and real-life applications. In this context, we demonstrate a reduction in waste of storage space of 33% for microbenchmarks and 42% for applications, all of which is possible with minimal (2-5%) performance overhead. (Section IV)

II. Related work

Extensive work exists on elasticity of computational resources, with focus on various aspects including: responsiveness to job submissions patterns and performance acceleration [7], automated monitoring and workload adaptation for OpenStack [8], elasticity of virtual clusters on top of IaaS clouds [3] and wide area cloud federations [4], [5].

With respect to storage, compression [9] and other space reduction techniques can be used to reduce associated costs. However, such approaches deal with actually used data and do not directly address the gap between actually used data and provisioned space. Thus, building blocks that facilitate elasticity of storage are crucial. Means to conveniently create and discard virtual disks of fixed sizes that can be freely attached and detached to running VM instances are supported by both open-source platforms [10] and commercial IaaS clouds [11]. Disk arrays (in various RAID configurations) have long been used by storage servers in order to aggregate the storage space of multiple disks. Although growing and shrinking of RAID volumes is possible, this is a lengthy and expensive operation because it requires rebuilding the entire RAID. While several efforts have been made to improve this process [12], [13], [14], such an approach is not feasible in our context where we need to grow and shrink storage over short periods of time. On the other hand, approaches that manage multiple disks at file system level have demonstrated scalability and low resizing overhead [15]. We note in this context our own previous work on multi-versioning [16], of interest especially if leveraged to reduce remove overhead: by writing into new snapshots and serving reads from old snapshots that potentially include the disk to be removed, blocking during reads can be completely avoided until all content has been copied to the remaining disks. At this point, a simple atomic switch to the newest snapshot is enough to complete the remove operation transparently.

Approaches that aim at automated control of storage elasticity have been proposed before. Lim et al. [17] address elastic control for multi-tier application services that allocate and release resources at coarse granularity, such as virtual server instances of predetermined sizes. In this context, the focus is on adding and removing entire storage nodes and rebalancing data across remaining nodes in order to optimize I/O bandwidth and CPU utilization.

Storage correlations have been explored before at various granularity. Several efforts analyze correlations at the file level either in order to detect access locality and improve prefetching [18] or to conserve energy in a multi-disk system without sacrificing performance [19]. Other efforts go one level below and focus on smaller (i.e. block-level) granularity, under the assumption that it would enable additional disk-level optimization opportunities in the area of storage caching, prefetching, data layout, and disk scheduling [20]. Our approach on the other hand focuses on correlations that help predict storage utilization, without insisting on any particular storage unit.

Prediction of I/O and storage requirements have been attempted from the perspective of both storage space utilization and behavior anticipation. For example, Stokely et al. developed forecasting methods to estimate storage needs in Google datacenters [21]. In this context, the focus is on long term prediction (the order of months), which is insufficient for adapting to short term fluctuations that can happen in as little as the order of seconds. Anticipation of I/O behavior has been realized mainly by identifying and leveraging I/O access patterns [22], [23]. Such access pattern-based analysis could be interesting to explore in our context, as a complement that facilitates bandwidth-elastic capability in addition to storage space elasticity.

Our own work focuses on a specific aspect of elasticity: minimizing waste of storage space transparently without performance degradation. To our best knowledge, *we are the first to explore the benefits of elasticity under such circumstances*.

III. System design

This section presents the design and implementation of our approach.

A. Requirements and Design Considerations

Transparent online elasticity via multi-disk aware file system: Storage typically is provisioned on IaaS clouds in the form of virtual disks that can be attached and detached from running VM instances. While this model provides low-level control over the storage resources, by itself it has limited potential for elasticity: a virtual disk is often provisioned by using a predefined initial size, with no capability of *online*

resizing (i.e., while being attached to a running VM instance that potentially uses the disk during the resize operation). Further, a virtual disk typically is used via a file system rather than as a raw block device. Therefore, in order to provide disk elasticity, a corresponding resize of the file system is necessary as well. This operation is usually not possible in an online fashion: the file system needs to be unmounted, resized offline and then mounted again. Since our goal is to achieve *transparent* elasticity, the online capability is crucial in our context. Thus, attempting to resize virtual disks themselves presents a problem both at the level of the disk itself and at the level of the file system.

To address both issues simultaneously, we propose to leverage a *multi-disk aware file system* that is able to aggregate the storage space of multiple virtual disks into a single pool. Using this approach, we can start with a small virtual disk of fixed size and then add or remove additional fixed-size virtual disks (which we will refer to as *increments*) as needed. While simple as a principle, such an approach still presents the challenge of adding and removing disks in an online fashion. Two main requirements arise in this context: (1) scalability with respect to the number of disks (i.e. the file system should not become slower as more disks are added) and (2) minimal overhead on application performance (i.e. adding and removing disk should not slow down the file system). Fortunately, as explained in Section III-D, building blocks that fulfill these requirements are readily available.

Dynamic adaptation to space utilization using fine-grained monitoring and preventive reserve: The capability to add and remove virtual disks to form an elastic pool of raw storage space is by itself useful only as long as it is leveraged in a way that matches application requirements. However, this ability to *dynamically adapt* to the application requirements is challenging, especially in the context of data-intensive applications that frequently exhibit large fluctuations of storage space utilization over short periods of time. If there is not enough time to react and add a new disk to the file system, the application will run out of space and either fail or slow down. This scenario is unacceptable as a trade-off for reducing the waste of storage space. Thus, we aim to guarantee *correctness* for our approach, which in our context means that the application should behave when using elastic storage in the same way as it would when using an infinitely large storage.

To achieve this goal, we propose to monitor changes in utilization at fine-granularity in order to enable the system to accurately capture the access pattern and adapt accordingly. Although such an approach helps alleviate the problem of fluctuations over short periods of time, by itself it is not enough because applications often check for free space and change their behavior if their expectation is not met, even before attempting an operation that risks failing due to running out of space. To deal with this issue, we propose to use a *preventive reserve*, namely, keep an extra amount of storage space available at all times, beyond the immediate needs. This not only solves the expectation issue, but also helps with the fluctuations, especially if there are dramatic changes between two consecutive probings.

Minimal waste of storage space and performance overhead using prediction: While a large enough reserve can guarantee correctness, letting it grow too much defeats the purpose of being economical. Furthermore, even in an ideal scenario where no storage space is wasted, poorly choosing the moment to add and remove virtual disks can lead to performance degradation and longer application runtimes, which in turn lead to an increase in operational costs. Thus, it is important to take both reserve and this timing into consideration when adapting to the space utilization.

To address this issue, we argue for a predictive scheme capable of anticipating near-future space requirements and other favorable circumstances that can be used to optimize the moment when to add and remove disks. Such a scheme ultimately helps our approach satisfy correctness with a smaller reserve while minimizing the waste and the performance overhead. More specifically, we propose to go beyond just looking at the storage space utilization itself and look at correlations between several I/O parameters. One such correlation that we found particularly helpful is the amount of data written in the near past: under high write pressure, it is likely that the free space as reported by the file system does not accurately reflect the real free space, due to pending flushes that were not yet committed. Thus, factoring the amount of written data into the prediction helps avoid worst-case scenarios where the reported free space suddenly jumps by an amount proportional to the written amount. Furthermore, under high write pressure, fluctuations in space utilization are more likely to exhibit larger variability over short periods of time. To address this issue, we propose to adapt the frequency of probing in order to increase reactivity: we monitor parameters at finer granularity when the write pressure grows higher and back off to coarser granularity when the pressure falls lower. Finally, we also use I/O pressure (both read and write) to decide when to remove a disk: under the assumption that a high I/O pressure makes a removal expensive in terms of performance overhead, we avoid this operation as long as the pressure stays high (note that delaying a removal does not affect correctness, unlike the case when a new disk needs to be added). We describe these considerations in more detail in Section III-C.

B. Architecture

The simplified architecture of our approach is depicted in Figure 1. We assume that the VMs are deployed on an IaaS cloud that enables users to provision raw storage as virtual disks. Furthermore, we assume that the cloud hypervisor is capable of dynamically attaching and detaching virtual disks to the VM instances (a feature that is standard in most production-ready hypervisors). Finally, we also assume a cost model that charges users for utilization at fine time granularity, which can be as little as the order of seconds (providers increasingly push towards finer granularity, currently as low as the order of minutes, e.g. RackSpace [24]).

Once deployed on the IaaS cloud, the VM instance initializes a *multi-disk aware file system* that is exposed to the

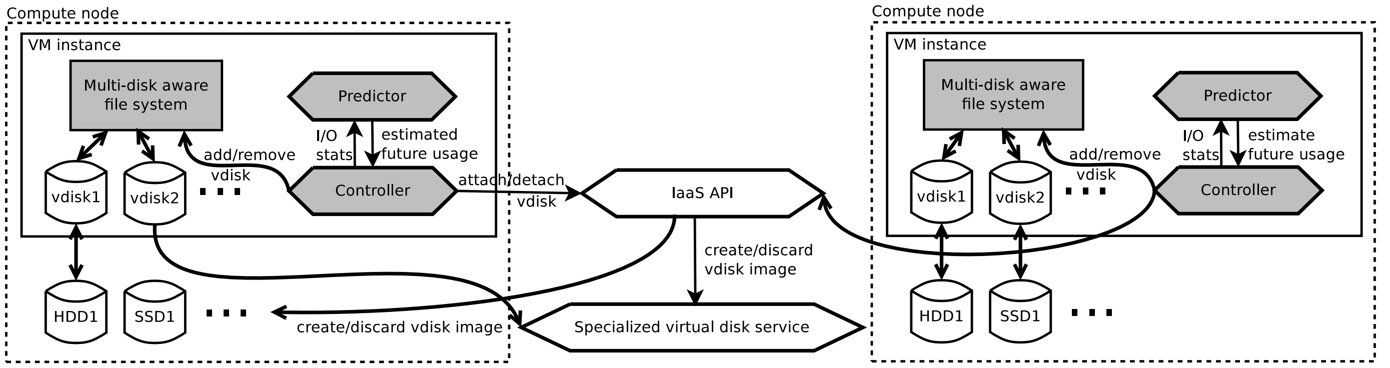


Fig. 1. Integration of our approach into an IaaS cloud architecture; components that are part of our design are highlighted with a darker background.

users using a regular POSIX mountpoint and implements the requirements described in Section III-A. At the same time, it launches the *predictor* and the *controller*. The controller is responsible for monitoring the system via a range of sensors, applying policies, and enactment of suitable actions (i.e., attaching or detaching disks). The frequency of monitoring depends on the intensity of I/O (as detailed in Section III-C); each sensor collects information about free space utilization and other parameters. This information is then passed to the predictor which uses it to estimate near-future storage space utilization. Based on this estimate, the controller decides its next action, which can either be to request a new virtual disk and then add it to the file system pool, or to remove a virtual disk from the pool and then ask for it to be discarded.

How to provision a virtual disk is open to a wide range of choices: virtual disk images of various formats (e.g. raw, QCOW2 [25]) stored either locally on the physical disks of the hosting node or remotely; dedicated physical disks exported as block devices (again locally available or exported through, e.g., iSCSI); specialized virtual disk services, such as Amazon EBS [11]; or our own previous work [26].

C. Predictor and Controller

In this section, we introduce an algorithmic description of the predictor and controller. By convention, identifiers in italic capitals represent constants. These are as follows: $INIT$ is the size of the initial virtual disk; INC is the size of newly added virtual disks (although more complex models are possible that enable growing and shrinking in variable increments, for the purpose of this work we consider the increment as a constant); $TMAX$ and $TMIN$ represent respectively the coarsest and the finest granularity at which the controller probes for I/O statistics of the file system; R represents the reserve of space; AD (add delay) represents the expected time to add a new virtual disk (and is set to a conservative value); finally RD (remove delay) represents the amount of time that must elapse after a new virtual disk was added before a remove is permitted, which is needed in order to enable the newly added virtual disk to become fully integrated into the file system.

The main loop of the controller is listed in Algorithm 1.

Algorithm 1 Controller

```
1: size ← ADD_DISK(INIT)
2: next ← TMAX
3: window ← 2 · TMAX
4: while true do
5:     stats ←GET_FS_STATS()
6:     pred ← PREDICT(stats, window, 2 · next + AD)
7:     if pred − used > R or size < pred + R then
8:         ts ← CURRENT_TIMESTAMP()
9:         if next > TMIN then
10:            next ← next/2
11:            window ← window + TMAX
12:        end if
13:        if size < pred + R then
14:            size ← ADD_DISK(INC)
15:        end if
16:    else
17:        if next < TMAX then
18:            next ← 2 · next
19:            window ← window − TMAX
20:        end if
21:        ct ← CURRENT_TIMESTAMP()
22:        if size > pred + INC + R and ct > ts + RD
   and not under I/O pressure then
23:            size ← REMOVE_DISK(INC)
24:        end if
25:    end if
26:    SLEEP(next)
27: end while
```

The interactions with the multi-disk aware file system takes place through ADD_DISK and REMOVE_DISK, both of which are blocking operations that return the new size of the disk pool. In a nutshell, the controller constantly probes for new I/O statistics since the last query (using GET_FS_STATS) and then passes these statistics to the predictor in order to find out an estimation of near-future usage ($pred$), after which it uses this estimation to take action. Two additional variables aid the prediction: the interval of probing ($next$) and the $window$, which represents how much time into the past the predictor

should look in order to anticipate near-future usage. Under high uncertainty (i.e., when the difference between prediction and actual usage is large) or when the file system is close to getting filled up, the intuition is that we need to be "more careful" and thus we probe twice as frequently (but not at finer granularity than $TMIN$) and we need to look more into the past ($window$ increases). When the opposite is true, the interval of probing doubles (up to $TMAX$) and the window decreases. Thus, the notion of "near-future" becomes more concretely $2 \cdot next + AD$, because in the worst case, it might happen that $next$ halves and a new disk is added, causing additional AD overhead.

Based on the prediction, if the current size of the disk pool is not enough to cover the near-future utilization ($size < pred + R$), then a new disk is added. Conversely, if the size is large enough to cover a removal of an increment ($size > pred + INC + R$), then there is potential to remove a disk. However, we set two additional conditions for actual disk removal: (1) there is no I/O pressure (such that a removal does slow the application), and (2) the file system has spent enough time (RD) in a non-uncertain state to make a removal safe.

Note that the reserve R is assumed constant, which implies some a priori knowledge about application behavior and requirements. when R is unknown, our algorithms require an extension (e.g., start with a large R that gradually is reduced when enough information about the past access pattern was collected to justify taking bigger risks). However, this aspect is outside the scope of this work.

Algorithm 2 Predictor

1: **function** PREDICT($stats, window, future$)
2: $t \leftarrow$ CURRENT_TIMESTAMP()
3: **if** $stats.used > (t_{max}, s.used) \in History$ **then**
4: **for all** $(t_i, s_i) \in History | t_i + window < t$ **do**
5: $History \leftarrow History \setminus \{(t_i, s_i)\}$
6: **end for**
7: **else if** $stats.used < (t_{max}, s.used) \in History$ **then**
8: $(lt, lstats) \leftarrow (t_{max}, s) \in History$
9: $History \leftarrow \emptyset$
10: **end if**
11: $History \leftarrow History \cup \{(t, stats)\}$
12: **if** $|History| > 1$ **then**
13: $(t_i, s_i) \leftarrow (t_{max}, s) \in History$
14: $(t_j, s_j) \leftarrow (t_{max}, s) \in History \setminus \{(t_i, s_i)\}$
15: $extra \leftarrow future \cdot (t_i - t_j)/(s_i.wb - s_j.wb)$
16: $(a, b) \leftarrow$LINREGRESS($(t, s.wb) \in History$)
17: $extra \leftarrow max(extra, a \cdot future + b)$
18: **else**
19: $extra \leftarrow future \cdot (t - lt)/(stats.wb - lstats.wb)$
20: **end if**
21: **return** $stats.used + s.wb_{max} - s.wb_{min} + extra$
22: **end function**

The predictor is listed in Algorithm 2. It centers around the idea of keeping a history of recent statistics labeled with the corresponding timestamp ($History$) that gets updated according to the utilization. More specifically, if there is no change in utilization, then we assume the worst case (i.e. all written data to the disk might represent new data that was not flushed yet) and keep accumulating statistics. Otherwise, if we see an increase in utilization, then we assume that write operations were at least partially flushed, so we discard all entries in the history that are older than the $window$. Finally, if we observe a decrease in utilization, we assume it is safe to discard the whole history, under the intuition that the application will move to a new phase and thus change its behavior.

Once the history has been updated as described above, the predictor calculates the estimated utilization in the near future ($extra$). In this context, the most relevant parameter to consider is the amount of written data (wb). The calculation relies on a conservative approach that takes the maximum of two evaluations: (1) the most probable evolution based on the whole history of writes (entries denoted $(t, s.wb)$), and (2) a possible short term evolution based only on the latest entries in the history ($(t_i, s_i.wb)$ and $(t_j, s_j.wb)$). The reasoning behind (2) is the fact that (1) alone might not capture write-intensive bursts that follow right after a period of write inactivity, thus presenting the risk of unexpectedly running out of space. To calculate (1), we use *linear regression* [27]. As mentioned in the previous paragraph, the amount of free space might not accurately reflect all previous writes due to pending flushes. Thus, as a final step, we take an additional measure of caution and increase our prediction by the number of written bytes starting from the earliest moment recorded in the history up to the present ($s_{max}.wb - s_{min}.wb$), in order to cover the worst case where all written data corresponds to new data. Once all steps have finished, the final predicted value is returned.

D. Implementation

In this section, we briefly introduce a prototype that implements the components presented in Section III-B.

We rely on *Btrfs* [15] to fulfill the role of the multi-disk aware file system. Our choice was motivated by several factors. First, Btrfs implicitly supports online adding and removing of new disks and can do so in a scalable fashion. Second, thanks to its B-Tree centered design, it can efficiently mask the overhead of adding and removing disks asynchronously in the background, causing minimal performance degradation for the application. Third, Btrfs is part of the official Linux kernel and is widely accepted as a viable candidate to replace the current generation of file systems (such as *ext4*).

The predictor and the controller were implemented as a Python daemon. We found the rich ecosystem of libraries around Python to be particularly helpful: the psutil package offers out of the box support to get per-disk I/O statistics, while the scipy package implements several optimized numerical algorithms and techniques, including linear regression.

We also note certain non-trivial aspects related to the attaching of virtual disks, in particular how to detect inside the guest when the disk is recognized by the kernel. To this end, we rely on pyudev, which implements an accessible interface

to libudev, including asynchronous monitoring of devices in a dedicated background thread.

IV. Evaluation

This section presents the experimental evaluation of our approach.

A. Experimental Setup

Our experiments were performed on the *Shamrock* testbed of the Exascale Systems group of IBM Research in Dublin. For the purpose of this work, we used a reservation of 30 nodes interconnected with Gigabit Ethernet, each of which features an Intel Xeon X5670 CPU (6 cores, 12 hardware threads), HDD local storage of 1 TB and 128 GB of RAM.

We simulate a cloud environment using *QEMU/KVM* 1.6 [28], [29] as the hypervisor. On each node, we deploy a VM that is allocated two cores and 8 GB of RAM. Each VM instance uses a locally stored QCOW2 file as the root partition, with all QCOW2 instances sharing the same backing file through a NFS server. The guest operating system is a recent Debian Sid running the 3.10 Linux kernel. Both the root disk and any other virtual disks that are added or removed dynamically, use the virtio driver for best performance. The process of adding and removing virtual disks from VM instances is handled directly through the hypervisor monitor, using the device_add and the device_remove command respectively. Each virtual disk that is part of the Btrfs pool is hosted as a RAW file on the local HDD. To avoid unnecessary caching on both the host and the guest, any newly added virtual disk has host-side caching disabled (cache=none). Furthermore, the network interface of each VM uses the virtio driver and is bridged on the host with the physical interface in order to enable point-to-point communication between any pair of VMs.

B. Methodology

We compare four approaches throughout our evaluation:

1) Static worst-case pre-allocation: In this setting, a large, fixed-sized virtual disk is attached to each VM instance from the beginning, in order to cover all storage requirements throughout the runtime of the VM. Inside the guest, a Btrfs file system is created and mounted on this single large virtual disk. This setting corresponds to a typical static worst-case pre-allocation that is the most widely used on IaaS clouds when a user might mount e.g., an EBS partition. We will call this setting prealloc and use it as a baseline for our comparisons.

2) Incremental additive using free space reserve: In this setting, a small 1 GB virtual disk is initially attached to the VM instance and used as a Btrfs file system, same as in the previous setting. The file system usage is monitored by using a fixed window of 5 seconds throughout the runtime of the VM. Whenever the remaining free space is smaller than a predefined fixed amount, a new virtual disk is attached to the VM instance and added to the Btrfs pool. This predefined fixed amount corresponds to the *reserve*, as explained in Section III-A. Throughout our experiments, we pick the reserve such that it corresponds to the minimal amount that satisfies correctness (i.e. leads to failure-free execution that does not generate out-of-space and other related errors). The size of each new virtual disk (which we denote *increment*) is fixed at 2 GB. We denote this setting as reactive−add.

3) Incremental add-remove using free space reserve: This setting is similar to the previous setting, except that it also removes the last added virtual disk from the Btrfs pool whenever the free space grows higher than the reserve and the increment size (in order for the free space not to shrink below the reserve size after removal). We denote this setting as reactive−addrm. Both this setting and the previous setting were chosen in order to underline the importance of prediction (as featured by our approach) in minimizing the wasted space throughout the application runtime.

4) Incremental add-remove using our approach: We start in this setting from the same initial Btrfs configuration (1 GB virtual disk) and use the same increment size (i.e. 2 GB). However, the monitoring granularity and decision when to attach/detach a virtual disk are based on our adaptive predictor, as detailed in Section III-C. We fix $TMIN = 1s$, $TMAX = 5s$, $AD = 10s$, $RD = 60s$. We denote this setting as predictive−addrm.

These approaches are compared based on the following metrics:

- *Impact on application performance* is the difference in performance observed due to the overhead of attaching and detaching virtual disks dynamically compared to the baseline (i.e., prealloc). Ultimately, this metric reveals how much longer the VMs need to stay up and running (and thus potentially generate extra compute costs) as a compensation for reducing storage costs. This metric is based on application completion time; clearly, lower values are better.
- *Allocated and used storage* is the total allocated/used storage space of the Btrfs pools of all VM instances at a given time. These metrics are relevant to determine how much storage space is wasted.
- *Cumulated waste* is the accumulation of the difference between the allocated and the used storage, as the application progresses in time. This metric is expressed in $GB \times hours$ (GBh) and is calculated in the following fashion: the runtime is divided at fine granularity in 5 second intervals, in order to accurately capture fluctuations over short periods. Note that we intentionally assume a cost model that charges users at fine granularity in order to explore the limits of our approach. Given this assumption, for each interval, the sum of all space differences corresponding to the participating VM instances is calculated and converted to GBh. Finally, the sums are accumulated as the application progresses from one interval to the next. This metric is relevant because it directly corresponds to the extra unnecessary costs incurred by provisioning unused storage space. Again, lower values are better.
- *Expected usage* is the expected storage space utilization in

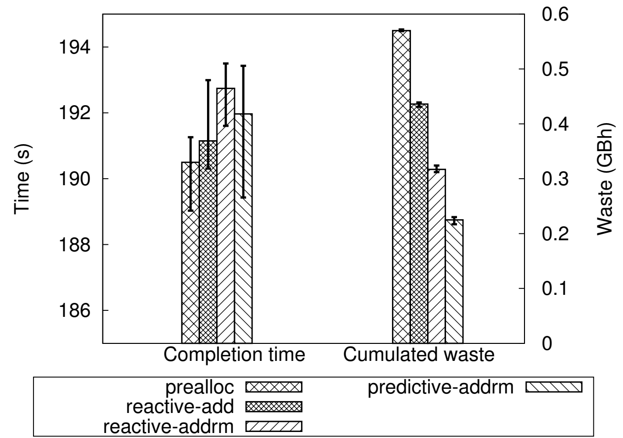
the near future (based on the previous experience), including the reserve. For reactive−addrm and reactive−add, which do not use prediction, it simply represents the used space plus the reserve. For predictive−addrm, it represents the predicted usage plus the reserve. When multiple VMs are involved, we calculate the expected usage in the following fashion: we divide the runtime in 5-second intervals and sum up the expected usage of all individual VMs for each interval. This metric is important because it shows how accurate our prediction compares to the other approaches. Naturally, a value as close as possible to the actual usage is preferred.

C. Microbenchmarks

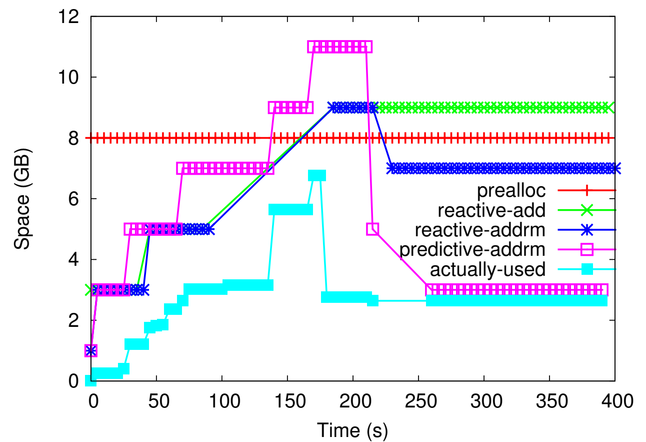
Our first series of experiments aims to push all approaches to the limit in order to better understand the trade-offs involved in attaching and detaching virtual disks dynamically. To this end, we implemented a benchmark that writes a large amount of data over a short period of time, waits for the data to be flushed to the file system and then scans through the generated dataset while keeping a predefined amount of it and discarding the rest. This is a common pattern encountered in data-intensive applications, especially those of iterative nature that refine a dataset until a termination condition is reached [30]. More specifically, the benchmark works as follows: we use dd to continuously generate files of 128 MB until we reach more than 6.5 GB (which ideally should trigger an addition of 3 virtual disks of 2 GB in addition to the initial disk of 1 GB). After the data is flushed (using sync), we proceed to read all generated data while at the same time removing files for a total of 4 GB, which makes the remaining data fit into the initial disk plus an additional 2 GB disk.

For this set of experiments we deploy a single VM and run the benchmark three times for each of the approaches, averaging the results. To enable a fair comparison, an important condition that we set is to achieve failure-free execution on all three attempts using a minimal reserve of space. Thus, we run the experiments and gradually increase the reserve until we satisfy this condition for all approaches. Our findings are as follows: both reactive approaches require a reserve of 2 GB to survive the initial write pressure, while our approach accurately predicts near-future requirements and is able to achieve failure-free execution with a reserve of 0. For prealloc, we fix the initial disk size at 8 GB, enough to fit any additional space requirements beyond useful data (e.g. system reserved space, metadata, etc.).

Results are depicted in Figure 2. First, we focus on completion time (left hand side of Figure 2(a)). As can be observed, all approaches perform closely to the baseline (prealloc): an increase of less than 2% in completion time is observable, which leads to the conclusion that the process of attaching and detaching virtual disks can be efficiently masked in the background by Btrfs. These results are consistent throughout all three runs: the error bars show a difference of less than 1% in both directions. As expected, prealloc is the fastest, followed by reactive−add. Although minimal, a wider gap is observable



(a) Completion time and cumulated waste (lower is better, error bars represent min and max)



(b) Allocated space (lower is better)

Fig. 2. Benchmark: single VM writes 6.5 GB worth of data in files of 128 MB, waits for the data to be flushed, then reads it back while deleting 4 GB of it

between reactive−add and reactive−addrm, hinting at larger disk remove overhead.

Figure 2(b) helps understand these results better by depicting the variation of allocated space in time. This variation directly corresponds to adds (rise) and removes (fall) and also shows their overhead in terms of how long the operation takes: sudden rise/fall means low overhead, gentle rise/fall means higher overhead. As can be observed, our approach is more aggressive in predicting future storage space requirements, since it relies on information about past written data, which in this case amounts to large quantities. Thus, it decides to add virtual disks earlier to the Btrfs pool, which enables it to behave correctly without a reserve. This is noticeable due to a higher slope for additions and the presence of flat regions that approximate the used space much better as opposed to the reactive approaches, where a flat region is missing completely between the addition of the third and fourth disk. We suspect that the gentle slope exhibited by the reactive approaches

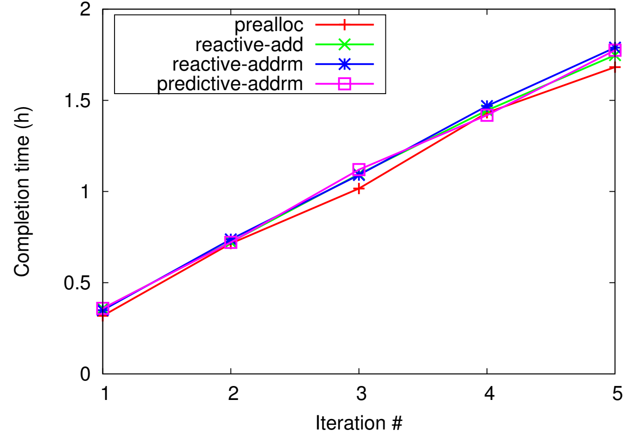
is caused by spending more time in a state where the file system is close to being full, explaining the high add overhead and thus the need for a high reserve. Ultimately, this effect combined with a delayed removal of virtual disks enables predictive−addrm to finish slightly faster than reactive−addrm, despite adding more disks overall.

The benefits of eliminating the need for a reserve thanks to prediction are especially visible when observing the cumulated waste (right hand side of Figure 2(a)). The cumulated waste is calculated for the duration of the benchmark plus a "cool-down" period, to enable the file system to stabilize and complete all pending asynchronous operations (removes in particular; total duration is 400s). As expected, prealloc generates the largest waste of space at almost 0.6 GBh. Next is reactive−add, which manages to save almost 0.17 GBh. It is followed by reactive−addrm, which thanks to its last removal saves an additional 0.1 GBh. The winner is predictive−addrm: because of accurate prediction and lack of reserve, it manages to remove all extra allocated disks. This amounts to an additional 0.1 GBh compared to reactive−addrm, which represents a relative reduction of 33% and brings the total reduction compared to prealloc to 66%.

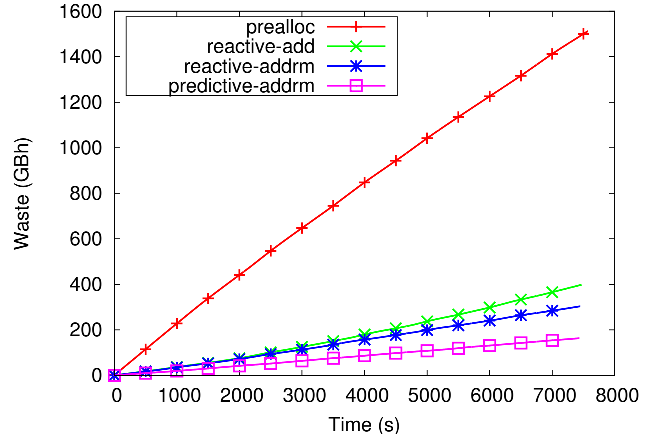
D. Case Study: MapReduce K-Means

Our next series of experiments focuses on real-life data-intensive scenarios. As an illustrative application, we use *K-Means* [31], which is widely used in a multitude of contexts: vector quantization in signal processing, cluster analysis in data mining, pattern classification and feature extraction for machine learning, and so forth. It aims to partition a set of multi-dimensional vectors into k sets, such that the sum of squares of distances between all vectors from the same set and their mean is minimized. This is typically done by using iterative refinement: at each step the new means are calculated based on the results from the previous iteration, until they remain unchanged (with respect to a small epsilon). K-Means was shown to be efficiently parallelizable and scales well using MapReduce [32], which makes it a popular tool to analyze large quantities of data at large scale. Furthermore, due to its iterative nature, it generates fluctuating storage space utilization. This fact, combined with the inherent scalability, makes K-Means a good candidate to illustrate the benefits of our proposal.

For the purpose of this work, we use the K-Means implementation of the PUMA set of Hadoop benchmarks [33], which applies K-Means on a real-life problem: clustering movies according to their ratings from users. The experiment consists in deploying a Hadoop cluster (Hadoop version 1.2.1) of 30 VMs (1 jobtracker/namenode and 29 tasktrackers/datanodes), each on a dedicated node. All Hadoop data (both HDFS data and intermediate data) is configured to be stored on a Btrfs file system that we use to compare each of the four approaches mentioned in Section IV-B. Once the Btrfs file system is mounted, in the first phase the input data (standard 30 GB movie database that is part of the PUMA benchmarks) is copied into HDFS. Then, five iterations are computed starting from this input data. Each iteration consists of two parts: the K-Means computation itself and the extraction of the new centroids at the end of the computation for the next iteration. To speed up the extraction, which in the original implementation is done in a serial fashion on the master (and thus does not scale for our purposes), we expressed this process itself as MapReduce grep job.



(a) Completion time (lower is better)

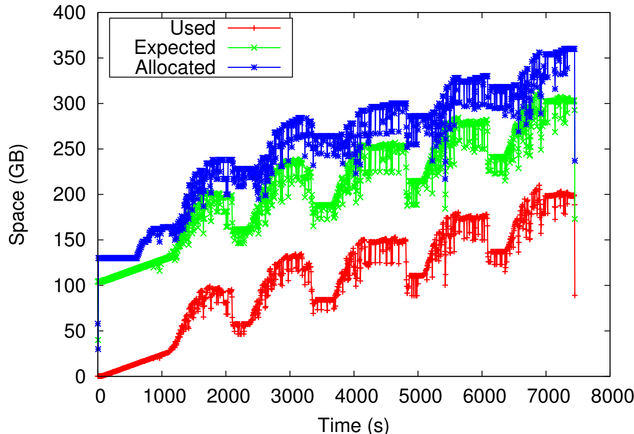


(b) Cumulated waste (lower is better)

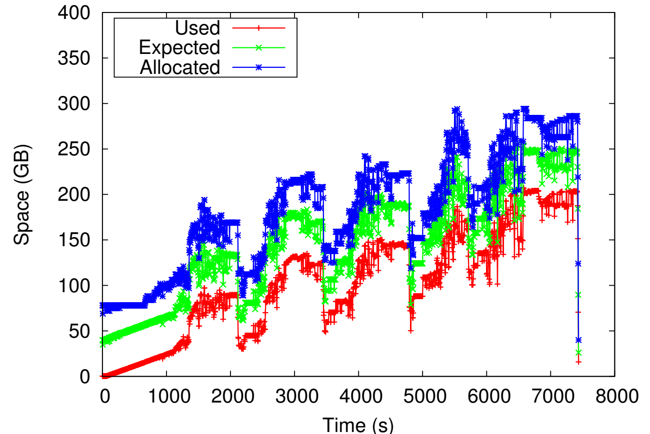
Fig. 3. K-Means: trade-off between achieved performance and waste of storage space for a Hadoop cluster made out of 30 VMs

The experiment is repeated three times for each of the four approaches and the results are averaged. As in the case of the microbenchmarks, we first established the minimal reserve of space necessary to achieve a failure-free execution on all three runs. Both reactive approaches require a reserve of 4 GB, while predictive−addrm can handle a 1.5 GB reserve thanks to prediction. For prealloc, we fix the initial disk size at 32 GB, which is the maximum observed throughout the lifetime of any of the VM instances.

As can be observed in Figure 3(a), the completion times for all four approaches are again close, demonstrating that Btrfs efficiently handles attaching and detaching of virtual disks in an asynchronous fashion. More specifically, the performance



(a) Reactive incremental add-remove (reactive−addrm) with a preventive reserve of 4 GB



(b) Predictive incremental add-remove (predictive−addrm) with a preventive reserve of 1.5 GB

Fig. 4. K-Means: aggregated evolution of used, expected and allocated storage space for a Hadoop cluster made out of 30 VMs

overhead of reactive−addrm and predictive−addrm is 6.3% and 5.5% respectively when compared to prealloc. This is a small price to pay when considering the large reduction in cumulated waste: at the extreme, reactive−addrm manages a five-fold reduction, while predictive−addrm manages an almost ten-fold reduction (which itself is 42% relative to reactive−addrm). We note the small relative difference in cumulated waste between reactive−add and reactive−addrm, which can be traced back to the fact that a large reserve limits the opportunities of disk removal.

To understand why predictive−addrm reduces the waste almost twice as much as reactive−addrm, we depict in Figure 4 the evolution of used, expected and allocated storage space for both approaches. As can be observed, both approaches have a similar storage space utilization pattern that clearly delimits the initial phase where the input data is copied into HDFS (steady growth in the beginning) and the K-Means phase with its five iterations (a "bump" for each iteration). Thanks to the accuracy of our prediction scheme and the resulting small required reserve, the expected utilization (Figure 4(b)) is much closer to the real utilization than in the case of reactive−addrm (Figure 4(a)). Notice the amplified effect of accumulating a large reserve for reactive−addrm: the difference between expected and used space grows to 100GB, which for our approach stays well below 50 GB. Ultimately, this large difference in expected utilization enables a much more flexible allocation and removal of virtual disks for predictive−addrm: the allocated space stays throughout the application runtime much closer to the expected utilization and exhibits steeper fluctuations compared to reactive−addrm, which in turn explains the reduction in cumulated waste.

V. Conclusions

The ability to dynamically grow and shrink storage is crucial in order to close the gap between provisioned and used storage. Even now, due to lack of automated control of provisioned storage resources, users often over-provision storage to accommodate the worst-case scenario, thereby leading to waste of storage space and unnecessary extra costs. Thus, a solution that adapts to data-intensive workloads and handles growing and shrinking of storage *transparently* to *minimize wasted space* while causing *minimal performance overhead* is an important step towards leveraging new cloud capabilities.

In this paper we have described such a solution in the form of a regular POSIX file system that operates with virtual disks of small fixed sizes, while hiding all details of attaching and detaching such disks from VM instances. Rather than relying solely on a reactive model, our approach introduces a prediction scheme that correlates different I/O statistics in order to optimize the moment when to attach and detach virtual disks. This scheme lowers wasted storage space without compromising normal application functionality by prematurely running out of space.

We demonstrated the benefits of this approach through experiments that involve dozens of nodes, using both microbenchmarks and a widely used, real-life, data-intensive MapReduce application: *K-Means*. Compared with traditional static approaches that over-provision storage in order to accommodate the worst-case scenario, we show reduction of wasted storage space over time that ranges from 66% for microbenchmarks up to 90% for K-Means. We also quantify the importance of prediction: compared with a simple reactive scheme, our approach reduces cumulative waste by 33% for microbenchmarks and 42% in real-life for K-Means. All these benefits are possible with minimal performance overhead: compared to static worst-case over-provisioning, we show a performance overhead of less than 2% for microbenchmarks and around 5% for K-Means.

Seen in a broader context, our results demonstrate that the concept of elastic storage is not only efficient but also cost-effective. This observation potentially removes a constraint from the development of systems that up to now have been

designed to work with fixed storage space: while total storage available from a cloud provider is fixed, in many configurations that fixed value will be many times higher than what can be reached in practice by specific applications. In addition, this observation also makes the implementation of systems with small average but high worst-case storage requirements potentially cheaper. Overall, a logistical constraint (fixed storage) has become a cost constraint (how much storage an application can afford in practice).

This work can be extended in several directions. One direction concerns our predictor: we plan to investigate how leveraging additional correlations and access patterns can further improve the accuracy of our near-future predictions. Another direction is bandwidth elasticity, namely, how to allocate more/less I/O bandwidth according to workload requirements in such way as to consume as little bandwidth as possible. Such an approach has potential especially in the context of multi-tenancy, enabling, for example, a cloud provider to oversubscribe available I/O bandwidth without violating quality-of-service constraints. Other interesting questions also arise in the context of data partitioning and reliability schemes that relate to storage elasticity.

Acknowledgments

The experiments presented in this paper were carried out using the Shamrock cluster of IBM Research, Ireland. This material is based in part on work supported in part by the Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: Towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, Jan. 2009.

[2] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *SC '11: Proc. 24th International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, USA, 2011, pp. 49:1–49:12.

[3] M. Caballer, C. De Alfonso, F. Alvarruiz, and G. Moltó, "EC3: Elastic cloud computing cluster," *J. Comput. Syst. Sci.*, vol. 79, no. 8, pp. 1341–1351, Dec. 2013.

[4] K. Keahey, P. Armstrong, J. Bresnahan, D. LaBissoniere, and P. Riteau, "Infrastructure outsourcing in multi-cloud environment," in *FederatedClouds '12: Proceedings of the 2012 workshop on cloud services, federation, and the 8th OpenCirrus summit*, San Jose, USA, 2012, pp. 33–38.

[5] R. N. Calheiros, A. N. Toosi, C. Vecchiola, and R. Buyya, "A coordinator for scaling elastic applications across multiple clouds," *Future Gener. Comput. Syst.*, vol. 28, no. 8, pp. 1350–1362, Oct. 2012.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *6th Symposium on Operating Systems Design and Implementation*, 2004, pp. 137–149.

[7] P. Marshall, K. Keahey, and T. Freeman, "Elastic site: Using clouds to elastically extend site resources," in *CCGrid'10: Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Melbourne, Australia, 2010, pp. 43–52.

[8] L. Beernaert, M. Matos, R. Vilaça, and R. Oliveira, "Automatic elasticity in OpenStack," in *SDMCMM '12: Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*. Montreal, Quebec, Canada: ACM, 2012, pp. 2:1–2:6.

[9] B. Nicolae, "On the benefits of transparent compression for cost-effective cloud data storage," *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, vol. 3, pp. 167–184, 2011.

[10] S. A. Baset, "Open source cloud technologies," in *SoCC '12: Proceedings of the 3rd ACM Symposium on Cloud Computing*, San Jose, USA, 2012, pp. 28:1–28:2.

[11] "Amazon Elastic Block Storage (EBS)," http://aws.amazon.com/ebs/.

[12] J. L. Gonzalez and T. Cortes, "Increasing the capacity of RAID5 by online gradual assimilation," in *SNAPI '04: Proceedings of the international workshop on Storage network architecture and parallel I/Os*, Antibes Juan-les-Pins, France, 2004, pp. 17–24.

[13] W. Zheng and G. Zhang, "FastScale: Accelerate RAID scaling by minimizing data migration," in *FAST'11: Proceedings of the 9th USENIX conference on File and Storage Technologies*, San Jose, USA, 2011.

[14] C. Wu and X. He, "GSR: A global stripe-based redistribution approach to accelerate RAID-5 scaling," in *ICPP '12: Proceedings of the 41st International Conference on Parallel Processing*, Pittsburgh, USA, 2012, pp. 460–469.

[15] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The linux b-tree filesystem," *Trans. Storage*, vol. 9, no. 3, pp. 9:1–9:32, Aug. 2013.

[16] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation data management for large scale infrastructures," *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, pp. 169–184, Feb. 2011.

[17] H. C. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," in *ICAC '10: Proceedings of the 7th international conference on Autonomic computing*, Washington DC, USA, 2010, pp. 1–10.

[18] P. Xia, D. Feng, H. Jiang, L. Tian, and F. Wang, "FARMER: A novel approach to file access correlation mining and evaluation reference model for optimizing peta-scale file system performance," in *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, Boston, MA, USA, 2008, pp. 185–196.

[19] M. Iritani and H. Yokota, "Effects on performance and energy reduction by file relocation based on file-access correlations," in *EDBT-ICDT '12: Proceedings of the 2012 Joint EDBT/ICDT Workshops*. Berlin, Germany: ACM, 2012, pp. 79–86.

[20] Z. Li, Z. Chen, and Y. Zhou, "Mining block correlations to improve storage performance," *Trans. Storage*, vol. 1, no. 2, pp. 213–245, May 2005.

[21] M. Stokely, A. Mehrabian, C. Albrecht, F. Labelle, and A. Merchant, "Projecting disk usage based on historical trends in a cloud environment," in *ScienceCloud '12: Proceedings of the 3rd International Workshop on Scientific Cloud Computing*, Delft, The Netherlands, 2012, pp. 63–70.

[22] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun, "I/O acceleration with pattern detection," in *HPDC '13: Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, New York, USA, 2013, pp. 25–36.

[23] J. Oly and D. A. Reed, "Markov model prediction of I/O requests for scientific applications," in *ICS '02: Proceedings of the 16th international conference on Supercomputing*, New York, USA, 2002, pp. 147–155.

[24] "RackSpace," http://www.rackspace.com/.

[25] "The QCOW2 Image Format," https://people.gnome.org/~markmc/qcow-image-format.html.

[26] B. Nicolae, J. Bresnahan, K. Keahey, and G. Antoniu, "Going back and forth: Efficient multi-deployment and multi-snapshotting on clouds," in *HPDC '11: 20th International ACM Symposium on High-Performance Parallel and Distributed Computing*, San José, USA, 2011, pp. 147–158.

[27] N. Draper and H. Smith, *Applied regression analysis*, ser. Probability and mathematical statistics. New York: Wiley, 1966.

[28] F. Bellard, "QEMU, a fast and portable dynamic translator," in *ATEC '05: Proceedings of the 2005 USENIX Annual Technical Conference*, Anaheim, USA, 2005, pp. 41–46.

[29] "KVM: Kernel Based Virtual Machine," http://www.linux-kvm.org/.

[30] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "The HaLoop approach to large-scale iterative data analysis," *The VLDB Journal*, vol. 21, no. 2, pp. 169–190, Apr. 2012.

[31] H.-H. Bock, "Clustering methods: A history of K-Means algorithms," in *Selected Contributions in Data Analysis and Classification*, ser. Studies in Classification, Data Analysis, and Knowledge Organization. Springer Berlin Heidelberg, 2007, pp. 161–172.

[32] W. Zhao, H. Ma, and Q. He, "Parallel K-Means clustering based on MapReduce," in *CloudCom '09: Proceedings of the 1st International Conference on Cloud Computing*, Beijing, China, 2009, pp. 674–679.

[33] "PUMA: Purdue MapReduce benchmarks suite," http://web.ics.purdue.edu/~fahmad/benchmarks.htm.