Transparent Throughput Elasticity for IaaS Cloud Storage Using Guest-Side Block-Level Caching

Bogdan Nicolae, Pierre Riteau, Kate Keahey

Transparent Throughput Elasticity for IaaS Cloud Storage Using Guest-Side Block-Level Caching

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 a crucial feature in the age of data-intensive computing. However, the traditional provisioning model of leveraging virtual disks of fixed capacity and performance characteristics has limited ability to match the increasingly dynamic nature of I/O application requirements. This mismatch is particularly problematic in the context of scientific applications that interleave periods of I/O inactivity with I/O intensive bursts. In this context, overprovisioning for best performance during peaks leads to significant extra costs because of unnecessarily tied-up resources, while any other trade-off leads to performance loss. This paper provides a transparent solution that automatically boosts I/O bandwidth during peaks for underlying virtual disks, effectively avoiding overprovisioning without performance loss. Our proposal relies on the idea of leveraging short-lived virtual disks of better performance characteristics (and thus more expensive) to act during peaks as a caching layer for the persistent virtual disks where the application data is stored. We show how this idea can be achieved efficiently at the block-device level, using a caching mechanism that leverages iterative behavior and learns from past experience. We demonstrate the benefits of our proposal both for microbenchmarks and for two real-life applications using large-scale experiments.**

Keywords-**cloud computing, storage elasticity, adaptive I/O, virtual disk, block-level caching**

I. Introduction

Elasticity (i.e., the ability to acquire and release resources on-demand as a response to changes of application requirements during runtime) is a key feature that drives the popularity of infrastructure clouds (Infrastructure-as-a-Service, or IaaS, clouds). To date, much effort has been dedicated to studying the elasticity of computational resources, which in the context of IaaS clouds is strongly related to the management of virtual machine (VM) instances [1], [2], [3]: when to add and terminate instances, how many and what type to choose, and so forth. *Elasticity of storage* has gained comparatively little attention, however, despite the fact that applications are becoming increasingly data-intensive and thus need cost-effective means to store and access data.

An important aspect of storage elasticity is the management of I/O access throughput. Traditional IaaS platforms offer little support to address this aspect: users have to manually provision raw virtual disks of predetermined capacity and performance characteristics (i.e., latency and throughput) that can be freely attached to and detached from VM instances (e.g. , Amazon Elastic Block Storage (EBS) [4]). Naturally, provisioning a slower virtual disk incurs lower costs when compared with using a faster disk; however, this comes at the expense of potentially degraded application performance because of slower I/O operations.

This trade-off has important consequences in the context of large-scale, distributed scientific applications that exhibit an iterative behavior. Such applications often interleave computationally intensive phases with I/O intensive phases. For example, a majority of high-performance computing (HPC) numerical simulations model the evolution of physical phenomena in time by using a bulk-synchronous approach. This involves a synchronization point at the end of each iteration in order to write intermediate output data about the simulation, as well as periodic checkpoints that are needed for a variety of tasks [5] such as migration, debugging, and minimizing the amount of lost computation in case of failures. Since many processes share the same storage (e.g., all processes on the same node share the same local disks), this behavior translates to periods of little I/O activity that are interleaved with periods of highly intensive I/O peaks.

Since time to solution is an important concern, users often overprovision faster virtual disks to achieve the best performance during I/O peaks and underuse this expensive throughput outside the I/O peaks. Since scientific applications tend to run in configurations that include a large number of VMs and virtual disks, this waste can quickly get multiplied by scale, prompting the need for an elastic solution.

This paper introduces an elastic disk throughput solution that can deliver high performance during I/O peaks while minimizing costs related to storage. Our proposal relies on the idea of using small, short-lived, and fast virtual disks to temporarily boost the maximum achievable throughput during I/O peaks by acting as a caching layer for larger but slower virtual disks that are used as primary storage. We show how this approach can be efficiently achieved in a completely *transparent* fashion by exposing a specialized block device inside the guest operating system that hides all details of virtual disk management at the lowest level. Our approach effectively enables any higher-level storage abstraction to take advantage of throughput elasticity, including those systems not originally designed to operate on IaaS clouds. In effect, we cast throughput elasticity as a block-device caching problem where performance is complemented by cost considerations.

Our contributions can be summarized as follows:

- We introduce the concept of throughput elasticity as well as a series of requirements, cost model and design considerations that guide our approach to providing it transparently at the block-device level. In particular, we describe a caching strategy that adapts to iterative behavior and learns from past experience in order to maximize the I/O boost during peaks, while minimizing the usage of fast virtual disks. (Section III-A)
- We show how to apply these design considerations in practice through a series of building blocks (and their corresponding implementation as a prototype) that run inside the VM instances of the users and interact with a typical IaaS cloud architecture. (Sections 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 real-life reductions of up to 65% of storage costs at the expense of minimal performance overhead (3.3%) compared with overprovisioning. (Section IV)

II. Related work

Hybrid file systems [6], [7] and associated caching strategies [8], [9] have long been used to combine multiple devices of different types (e.g., SSDs, HDDs, nonvolatile memories). Generally, however, they are designed to use fixed storage resources available as physical hardware in order to improve access performance. Unlike our approach, such strategies are not concerned with being economical and minimizing resource usage. Furthermore, if the application needs a storage abstraction that is not file-based, then a file system introduces unnecessary overhead.

In the area of caching, Wang et. al. studied the problem of dynamically selecting the caching policy under varying workloads [10]. The caching framework selects a caching policy and reconfigures the storage system on the fly based on access traces gathered and analyzed during application runtime. An aspect of adaptation to access pattern is also explored by our previous work [11], however, the focus is on scalable virtual disk on-demand image content delivery at large scale. Also with respect to caching strategies, an increasingly popular target is the newer generation of flash-based devices. For example, in [12], special hybrid trees are proposed to organize and manipulate intervals of cached writes. Although orthogonal to our own work, such efforts provide valuable insight in terms of how caching is handled on the host that exposes virtual disks to the VMs, which may influence how to best leverage guest-level caching if additional information is available.

Moving upward in the storage stack at the virtualization level, several approaches aim to accelerate the throughput of virtual disks at hypervisor level or below. Jo et al. [13] proposed a hybrid virtual disk based on a combination of an SSD and an HDD. Contrarily to our solution, the SSD in their approach is read-only and used only to improve read performance to the template VM image. All write operations are sent to the HDD to avoid degrading the performance of flash storage. Using a fast device to cache both reads and writes from a read-only virtual disk snapshot is possible using copy-on-write and/or mirroring [14], [15], [16]. However, one of the disadvantages in this context is fragmentation, for which specialized strategies might be necessary [5].

How to virtualize bandwidth in a cloud environment was explored at various levels. S-CAVE developers [17] propose to leverage the unique position of the hypervisor in order to efficiently share SSD caches between multiple VMs. Similarly, vPFS [18] introduces a bandwidth virtualization layer for parallel file systems that schedules parallel I/Os from different applications based on configurable policies. Unlike our approach, the focus in this context is bandwidth isolation between multiple clients, as opposed to elasticity.

Storage elasticity on IaaS clouds was explored at coarse granularity by Lim et al. [19] for multitier application services, with a focus on how to add and remove entire storage nodes and how to rebalance data accordingly. Higher level service processing acceleration was described in [20]. This work introduces an elasticity aspect in the form of a series of algorithms to scale the cache system up during peak query times and back down to save costs. LogBase [21] is another elastic storage effort that employs a log-structured database system targeted at write-intensive workloads. Unlike our approach, the goal is to to improve write performance and simplifies recovery. Chen et al. introduced *Walnut* [22], an object store that provides elasticity and high availability across Yahoo!'s data clouds and is specifically optimized for the data-intensive workloads observed in these clouds: Hadoop [23], MObStor (unstructured storage similar to [24]), PNUTS [25], and so forth. The main goal is sharing of hardware resources across hitherto siloed clouds of different types, offering greater potential for intelligent load balancing and efficient elastic operation, while simplifying the operational tasks related to data storage. Our own previous work [26] focused on storage elasticity from the perspective of space utilization, aiming to adapt transparently to growing/shrinking data sizes by means of a POSIX-compatible file system that automatically adds and removes virtual disks accordingly.

The work we present here focuses on a specific aspect of elasticity: minimizing the waste caused by overprovisioning of throughput transparently without performance degradation. To our best knowledge, *we are the first to explore the benefits of elasticity under such circumstances*.

III. System design

In a nutshell, our proposal relies on a simple core idea: the use of small and short-lived virtual disks of high-throughput capability (with higher price per gigabyte) to *transparently* boost the I/O performance during peak utilization of slower (and cheaper per gigabyte) virtual disks that are continuously used by the application to accumulate persistent data. We use the former as a ephemeral caching device and the latter as a backing device. When the caching device is in operation, it acts as a read/write caching layer at the block level that uses an adaptive mechanism to asynchronously flush dirty blocks back to the backing device. Besides the technical challenges related to achieving transparency efficiently at the block level, the main challenge in this context is the focus on cost reduction, which brings a novel perspective to the otherwise well-studied caching domain. Several critical questions arise: How large should the virtual disk acting as a cache be? When and for how long do we need it? What caching strategy should we use?

To answer these questions, we introduce a series of design considerations formulated in response to the problem we study. We give a general description of these design considerations (Section III-A), show how to adopt them in a typical IaaS cloud (Section III-C), and briefly describe a prototype that implements our approach (Section III-D).

A. Design considerations

Our proposal relies on three key design principles:

1) ***Transparent block level caching****:* Storage is typically provisioned on IaaS clouds in the form of virtual disks that are created by using a predefined size and performance characteristics (i.e., throughput). Although the virtual disks can be freely attached to and detached from running VM instances, this degree of elasticity is hard to leverage directly at the application level in order to deal with fluctuating I/O throughput requirements: data would have to be constantly migrated to/from a slower/faster device, thereby generating high performance and cost overheads that are unacceptable if a large amount of data accumulates during runtime or the fluctuations happen over short periods of time. Even if such an approach were feasible, applications often do not leverage virtual disks directly but rely on storage abstractions (e.g., a file system) that were not designed to add/remove disks on the fly. Thus, it is desirable to handle throughput elasticity in a *transparent* fashion at the lowest level.

In response to this need, we propose a solution that works at the *block level*. Specifically, we expose a block device in the guest operating system that replaces the virtual disk normally leveraged by users directly, using it as a *backing device* that all I/O is redirected to. When I/O throughput utilization rises above the utilization threshold (UT), a second, faster virtual disk (referred to as the *caching device*) is provisioned to act as a caching layer for the backing device, temporarily boosting I/O throughput until the threshold falls below UT. At this point, the data from the caching device is flushed to the backing device, and the caching device is removed, with I/O passing directly to the backing device again. Such an approach enables transparency not only from the user's perspective but also from the cloud provider's perspective: it works at the guest level and does not require changes to the virtualization infrastructure or provisioning model.

2) ***Adaptive flushing of dirty blocks****:* A solution that alternates between a fast and a slow virtual disk to achieve elasticity suffers from poor I/O performance and is unsustainable, because an increasingly larger set of accumulated data needs to be migrated between the two devices. On the other hand, a solution based on caching dramatically reduces the amount of data movements, because only the most recently used blocks are involved (we call these blocks "hot"). Even when considering only the "hot" blocks, however, the constant movement between the backing device and the caching device naturally steals bandwidth from both devices, effectively limiting the potential to boost the I/O throughput at full capacity.

To address this issue, we propose to make the caching device act like a regular block-level read/write cache but with a custom dirty block commit strategy. More specifically, during a read operation, any requested blocks that are not already available on the caching device are first fetched from the backing device and written to the caching device by using an LRU (least recently used) eviction strategy. Then, the read operation is fully redirected to the caching device. In the case of a write operation, all dirty blocks are initially written to the cache only and are later committed to the backing device.

The strategy to commit dirty blocks works in two phases. In the first phase, we use a mechanism that closely resembles writeback and prioritizes application I/O: it flushes dirty blocks asynchronously to the backing device only when spare bandwidth is available or when the caching device is full and needs to evict. Once the required I/O throughput drops to a level that the backing device is able to sustain on its own, a transition to the second phase is initiated, in which the priority is reversed: the flushing process proceeds at full speed at the expense of application I/O. At the same time, only reads are allowed from the caching device starting from this moment onwards, with writes bypassing the caching device and being redirected to the backing device. We refer to this two-phase strategy as "dynamic writeback."

Note that finding the right moment to reverse the priority is important: if it happens too soon, the application will suffer a performance penalty because of background flushes. If it happens too late, the caching device stays up longer than necessary and thus incurs extra costs. In order to deal with this issue, the decision of when to reverse the priority is based on a configurable amount of time ID (inactivity delay), which represents how much the application's I/O throughput needs to stay below UT before we initiate the reverse of priority. Since the flush process is prioritized after the reverse and writes bypass the caching device, eventually all dirty blocks will be committed to backing device. At this point, one can safely detach the caching device and remove its corresponding virtual disk.

3) ***Access-pattern-aware cache sizing****:* Using a caching device can be expensive, particularly since while it is active the user is charged for *both* the caching device and the backing device. Thus, the caching device cannot be arbitrarily large for two reasons.

First, the flushing of dirty data does not happen instantly after the decision was made to proceed to the second phase of the commit strategy, thus delaying the moment when the caching device can be safely removed. Second, since dirty data tends to accumulate proportionally to the cache size, a large caching device is likely to cause a long flush delay. On the other hand, if the cache size is too small, flushing may be forced prematurely, limiting the potential to boost application I/O at full capacity. Thus, it is important to automatically optimize the cache size specifically for the access pattern of the application observed during the I/O-intensive phase.

To optimize the size of the caching device, we propose to leverage the predominantly repetitive I/O behavior of large-scale scientific applications in order to learn from the experience of the previous I/O-intensive phases for which a caching device was in use. More specifically, we start with a large cache size for the first time when an I/O boost is needed and then monitor the cache utilization. If the caching device was used only partially, then we decrease its size for the next I/O-intensive phase down to the size that was actually used. Similarly, if too much flushing was forced prematurely during the first phase of the commit strategy (in which application I/O is prioritized), we increase the cache size for the next I/O-intensive phase by the amount that had to be evicted.

B. Cost model

We assume a cost model that charges users for utilization at fine grain that can be as little as the order of seconds. This approach is already adopted in the cost model: for instance, Google Compute Engine charges persistent disks at a granularity of seconds [27]. Since we are dealing with different types of virtual disks, we approximate a realistic cost by defining utilization as a function of both the size of the virtual disk and its throughput characteristics (i.e., reserved bandwidth).

Note that the utilization is based on reserved bandwidth, which relates to the high-end spectrum of cloud offerings such as the Object Storage offered by *IBM SoftLayer* [28] or provisioned Elastic Block Store (EBS) volumes offered by *Amazon* [4]. This is different from the more popular “classic EBS” model where users are not offered any bandwidth guarantees and are charged per IOPS instead. We justify the need to rely on reserved bandwidth because our primary target is tightly coupled HPC applications that run at large scale and are known for their susceptibility to *system noise amplification* [29] (i.e., the bulk-synchronous nature makes processes sensitive to delays that affect all other processes, which in our context means that a slow virtual disk attached to a VM instance causes slowdown of all other VM instances where the application processes are running). Thus, reasoning in terms of average throughput (as is the case of classic EBS volumes) is not feasible in our context.

To quantify the utilization in accordance with the requirements mentioned above, we introduce a metric called *adjusted storage accumulation*, which reflects the total cost that accumulates over a period of time t for a VM instance as a result of storage use. We assume that for every time unit of utilization, a virtual disk of size N and reserved bandwidth B incurs a cost of $N \cdot B$. Thus, if a single virtual disk is attached to a VM instance for the whole duration t, the total cost is $C(t) = B \cdot N \cdot t$. When using both a backing device and a caching device to implement our approach, the backing device will continuously contribute to the total cost, as in the case of a single virtual disk, whereas the caching device will contribute only while it is used and proportionally to its dynamically adjusted size. More formally, this approach can be expressed as follows.

$$C(t) = B_b \cdot N \cdot t + B_c \cdot \int_0^t M(x) \cdot dx$$

N and B_b are the size and bandwidth of the backing device, respectively; B_c is the bandwidth of the caching device; and $M(x)$ is the size of the caching device at a given moment x (0 if the caching device is not used at that moment). We express the bandwidth in MB/s, the size in GB, and the time in seconds, which results in a combined metric unit that we call *adjusted GB seconds* (denoted AGBs). For the rest of this paper, we use this unit to express the cost.

C. 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 can dynamically attach and detach virtual disks to the VM instances (a standard feature in most production-ready hypervisors).

Once booted, the VM instance initializes an *adaptive block device* that uses a nonexpensive virtual disk of limited throughput as the *backing device*. The adaptive block device is exposed inside the guest OS as a standard block device and can be leveraged as such (e.g., it can be formatted by using a file system). Once the adaptive block device is running, a *controller* daemon collects I/O statistics about it at fine granularity (e.g., sustained throughput); and, based on these statistics, it implements the design principles described in Section III-A. Specifically, during an I/O-intensive phase, it interacts with a standardized *IaaS API* in order to attach a virtual disk of optimized size (based on the experience from the previous I/O intensive phases) that will act as the *caching device*. Once the VM instance has recognized the caching device, it incorporates the corresponding guest-level block device into the adaptive block device. All interactions between the caching device and the backing device are handled transparently by our adaptive block device based on the principles mentioned in Section III-A. Once the I/O-intensive phase completes, the controller signals the adaptive block device to start flushing. After the flushing has completed, it detaches the corresponding virtual disk from the VM instance using the IaaS API and destroys the disk, thus stopping the accumulation of storage costs due to caching.

How to provision a virtual disk is open to a wide range of choices: various types of devices (e.g., HDDs, SSDs, RAM-disks) can be directly leveraged by the hypervisor and exposed as virtual disks inside the VM instance. If devices need to be shared, another option is to use virtual disk images of various formats (e.g., raw, QCOW2 [14]), hosted either locally or remotely on different types of devices. Virtual disks also may

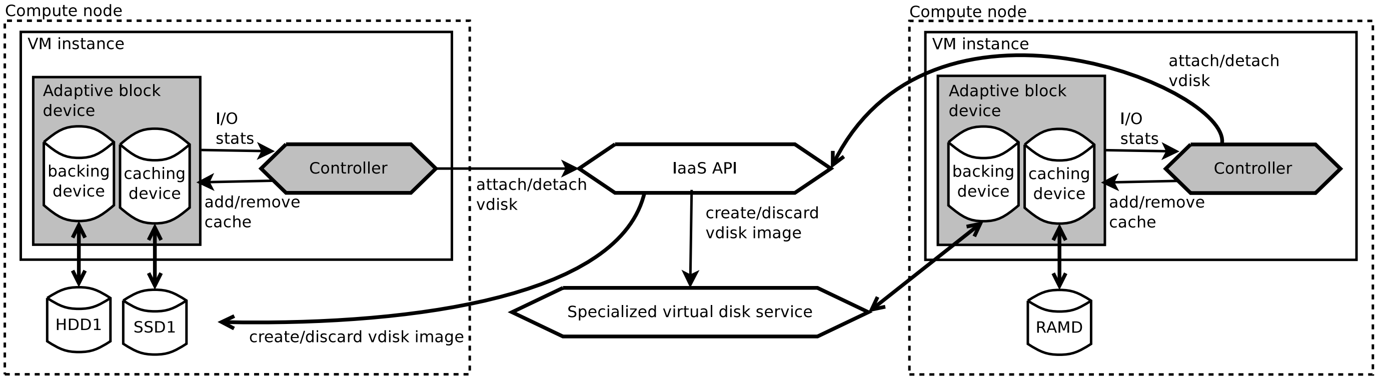


Fig. 1. Integration of our approach into an IaaS cloud architecture; components that are part of our design are highlighted with a darker background.

be provided by specialized services, such as Amazon EBS [4] or our own previous work [16]. Our approach is agnostic to any of these choices as long as they are handled through a standardized IaaS API.

D. Implementation details

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

We rely on *Bcache* [30] to implement the *adaptive block device*. Our choice was motivated by several factors. First, it offers out-of-the-box support to cache hot blocks of designated devices on other devices, while offering support to activate/deactivate caching in an online fashion. Second, it is highly configurable and offers detailed statistics about I/O utilization. In particular, the ability to control the caching strategy and the interaction between the backing device and caching device was crucial in enabling the implementation of the design principles presented in Section III-A. Third, it is implemented at the kernel level and is specifically designed to minimize performance overhead. Since we need to handle another level of indirection on top of the virtualized nature of the backing and caching device, this aspect is important in our context. Furthermore, it is part of the official Linux kernel and thus enjoys widespread exposure and adoption.

The *controller* was implemented as a Python daemon. We rely on *psutil* to get per-disk I/O statistics. The interaction with Bcache is implemented directly through the *sysfs* interface. We note also certain nontrivial aspects related to the management 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. Experimental evaluation

This section presents the experimental evaluation of our approach. We introduce the experimental setup and methodology (Section IV-A), and discuss results for microbenchmarks (Section IV-B) and two real-life HPC applications (Section IV-C and Section IV-D).

A. Experimental setup

Our experiments were performed on the *Shamrock* testbed of the Exascale Systems group of IBM Research in Dublin. For this work, we used a reservation of 32 nodes interconnected with Gigabit Ethernet, each of which features an Intel Xeon X5670 CPU (12 cores), HDD local storage of 1 TB, and 128 GB of RAM.

We simulate a cloud environment using *QEMU/KVM* 1.6 as the hypervisor. The VM instances run a recent Debian Sid (3.12 Linux kernel) as the guest operating system. 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.

We compare three approaches throughout our evaluation:

Static preallocation using a slow virtual disk: In this setting, a large, fixed-size virtual disk is created on the local HDD of each host as a RAW file and attached to each VM instance after booting. The maximum I/O bandwidth of the virtio driver is fixed by using the hypervisor monitor, and the host-side caching is disabled to avoid interference. After booting the VM instances, all virtual disks are formatted by using the *ext4* file system, and the corresponding mount points are used for all I/O generated during the experiments. We refer to this setting as *static-slow*.

Static preallocation using a fast virtual disk: This setting is similar to the one described above, except that the fixed-sized virtual disk is hosted as a RAW file in a RAM-disk. Again, the maximum bandwidth available to the guest operating system is fixed. However, it is several times higher than in the previous case and is intended to simulate a faster device, such as an SSD. We refer to this setting as *static-fast*.

a) Transparent throughput elasticity using our approach: In this setting, we use a virtual disk with properties identical to those in the *static-slow* case as a backing device. Whenever more bandwidth is needed, a new virtual disk with properties identical to those in the *static-fast* case is used as a caching device to temporarily boost I/O throughput. The size of the caching device, as well as the moment when to attach and detach it, is automatically determined by our approach during runtime (as explained in Section III-A). Furthermore, the utilization threshold UT is set to 30%, and the inactivity delay ID is set to 30 s. We refer to this setting as *adaptive*.

These approaches are compared based on the following metrics:

- *Performance overhead* is the difference in performance observed between *static-fast*, which is used as a baseline for the best possible performance, and the other two approaches that leverage slower virtual disks. In the case of microbenchmarks, performance refers to the sustained I/O throughput as perceived by the application. In the case of real-life applications, performance refers to the completion time, which measures the overall end-impact of each approach on the application runtime.
- *Total adjusted storage accumulation* is the sum of the adjusted storage accumulation for all VM instances involved in the experiment. It is used to quantify storage-related costs for the entire application deployment according to the cost model introduced in Section III-B.
- *Evolution of I/O activity* represents the total I/O bandwidth utilization (due to reads and writes to virtual disks) for all VM instances. In the *static-fast* and *static-slow* cases, it overlaps with the sustained I/O throughput as perceived by the application. In the *adaptive* case, it measures all background I/O activity to the backing device and caching device (which can be higher than *static-fast* when both devices are active simultaneously). This metric is important for studying how the compute phases interleave with the I/O phases and how the backing device interacts with the caching device during this interleaving.

B. Microbenchmarks

Our first series of experiments focuses on the I/O performance of all three approaches in synthetic settings. For this purpose, we use *Bonnie++*, a standard I/O benchmarking tool that measures read, write, and rewrite throughput when using 32 KB blocks (default value used in the experiments). We focus on these values because they are the most representative of real-life large-scale scientific applications (as opposed to byte-by-byte read/write throughput or other file-system related statistics that are reported).

We run the following experiment: a single VM instance is booted and Bonnie++ is launched three times in a row, with a 120 second pause between each run. We repeat the experiment for each approach three times and record the average of the relevant Bonnie++ statistics. The memory allocated to the VM is fixed at 2 GB. The virtual disk settings are as follows: the size of the backing device is fixed at 10 GB, with a reserved bandwidth of 60 MB/s for both *static-slow* and *adaptive* and a reserved bandwidth of 128 MB/s for *static-fast*. For *adaptive*, the reserved bandwidth of the caching device is fixed at 128 MB/s.

The Bonnie++ statistics are depicted in Figure 2(a). Since our approach automatically adjusts the size of the caching device after the first iteration, we depict the results for the first run and the remaining two runs separately. The initial cache size is set to 10 GB, which is lowered by our approach to 6 GB for the consecutive iterations. We denote the first iteration *adaptive-first*, and the average of the second and third iteration *adaptive-rest*.

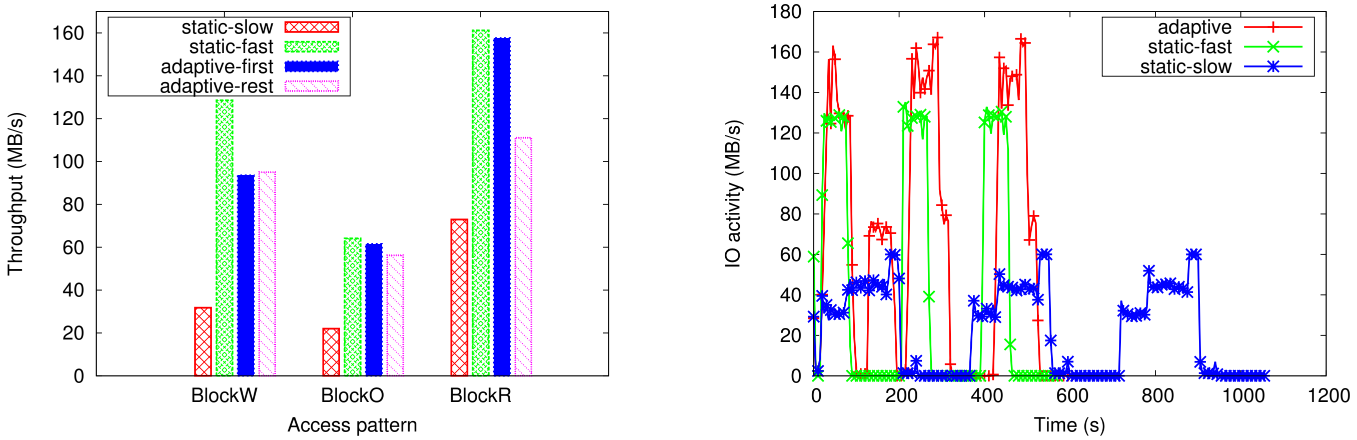
As can be observed, in the case of *static-slow*, the backing HDD limits the write throughput to 33 MB/s (out of 60 MB/s), which is not the case for *static-fast*, where the maximum write throughput of 128 MB/s can be achieved). At approximately 100 MB/s, both adaptive approaches have an overhead of 30% of write throughput compared with *static-fast*. With respect to overwrite throughput, all approaches suffer performance degradation compared with a simple write, which is due to the read, seek back, and write cycle employed by Bonnie++ for each block. The large gap between *static-slow* and the rest is still present; however, this time the overhead between the two adaptive approaches and *static-fast* is almost negligible. With respect to read throughput, a considerable increase is present for all approaches as a result of caching. Also, for the first time, a visible difference is noticeable between the two adaptive approaches: *adaptive-first* is comparable to *static-fast* whereas *adaptive-rest* has an overhead 25%. This overhead is due to the smaller size of the caching device, which forces writeback to the backing device earlier (and thus causes reads from the caching device).

To understand the interaction between the backing device and the caching device better, we depict the evolution of I/O activity (as measured at five-second granularity) in Figure 2(b). As expected, both static approaches have a highly deterministic behavior, with a flatter and elongated pattern observable for *static-slow*. In the case of *adaptive*, an initial burst is observable for the first run, which is followed by a smaller secondary burst. This secondary burst corresponds to cache flushing after ID elapsed that triggers the detach request. Thanks to the automatic adjustment of the size of the caching device to a smaller value, we observe earlier writeback. This causes a shorter flush burst that is fused into the primary burst, effectively enabling the caching device to be detached sooner (which reduces utilization cost).

C. Case study: CM1

Our next series of experiments evaluates the behavior of our system for *CM1*, a real-life HPC application that is a three-dimensional, nonhydrostatic, nonlinear, time-dependent numerical model suitable for idealized studies of atmospheric phenomena. This MPI application is used to study small-scale processes that occur in the atmosphere of the Earth (such as hurricanes) and is representative of a large class of HPC bulk-synchronous stencil applications that exhibit an iterative behavior of alternating between a compute phase and an I/O-intensive phase to write intermediate output results and checkpoints (used to restart from in case of failures).

For this work, we have chosen as input data a 3D hurricane that is a version of the Bryan and Rotunno simulations [31]. We run the simulation of this 3D hurricane on 32 VMs, with each VM hosted on a separate physical node and equipped with 11 virtual cores (out of which 10 are reserved for CM1 and 1 reserved for guest OS overhead). Furthermore, 1 core is



(a) Throughput for block-write (BlockW), block-overwrite (BlockO), and block-read (BlockR)

(b) I/O activity (reads and writes from backing device and, if applicable, caching device)

Fig. 2. Bonnie++: I/O performance under different access patterns and its corresponding I/O activity

reserved for the hypervisor. Each VM is allocated 18 GB of RAM, enough to fill the need of the MPI processes. Thus, the overall setup totals 320 MPI processes that generate a heavy load on the underlying nodes. The output/checkpointing frequency is set at 50 simulation time steps, out of a total of 160 time steps. This setup leads to the following access pattern: right after initialization, the application dumps the initial state, which causes a first I/O-intensive phase. After that there follow three more I/O-intensive phases of higher magnitude that are interleaved with computational phases.

The settings are as follows: the size of the backing device is fixed at 50 GB, while the reserved bandwidth is fixed at 33 MB/s for *static-slow* and *adaptive* (according the maximum write throughput observed in microbenchmarks in order to avoid overprovisioning). The bandwidth in the case of *static-fast* is fixed at 128 MB/s. The caching device has a bandwidth of 128 MB/s and an initial size of 10 GB. Each experiment is repeated five times for all three approaches, and the results are averaged.

Performance results are summarized in Table I. As can be observed, speeding the I/O phase can lead to a significant boost in overall completion time: compared with *static-fast*, which is used as a baseline, *adaptive* has a small overhead of 3.3%, which contrasts with the large overhead of 23% observed for *static-slow*. These results are significant both directly (the users want a minimal time to solution) and indirectly (longer runtimes mean the VMs need to stay up longer and thus generate more costs).

TABLE I
CM1 (NUMERICAL MODEL DESIGNED FOR IDEALIZED STUDIES OF ATMOSPHERIC PHENOMENA): PERFORMANCE RESULTS

Approach	Completion Time	Overhead
static-slow	1471s	23%
static-fast	1190s	–
adaptive	1231s	3.3%

Even if the overhead of *adaptive* is small, it is justifiable only if the storage space and bandwidth utilization can be significantly lowered according to the cost model introduced in Section III-B. To quantify these costs, we compute the adjusted storage accumulation for all three approaches. More specifically, for each VM instance i, in the case of *static-slow* we have $C_i(t) = 33 \cdot 50 \cdot t$, while for *static-fast* we have $C_i(t) = 128 \times 50 \cdot t$. For *adaptive*, we have

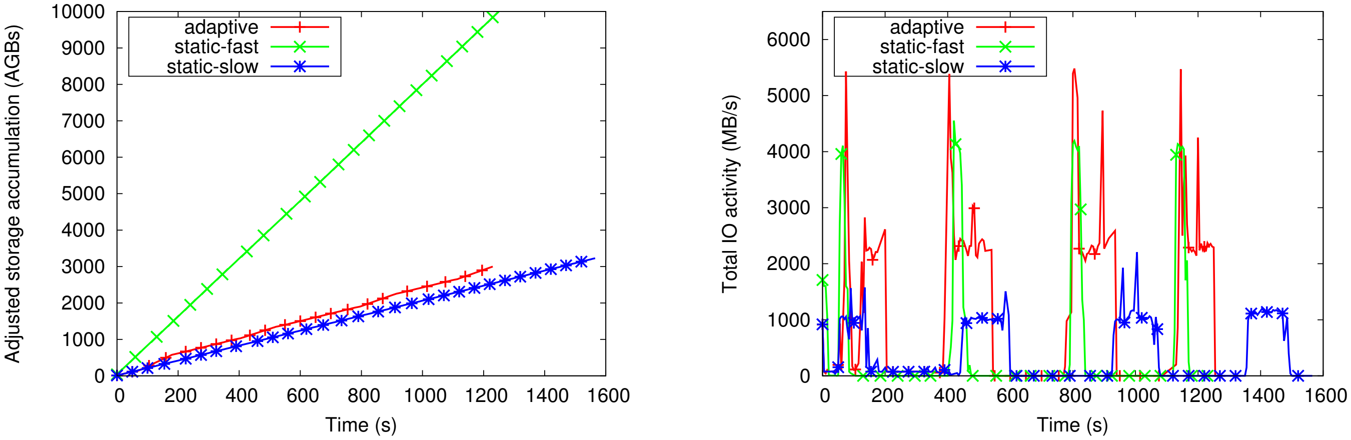
$$C_i(t) = 33 \cdot 50 \cdot t + 128 \cdot \int_0^t M_i(x) \cdot dx.$$

$M_i(x)$ is the size of the caching device for VM instance i at moment x and is automatically determined by our approach (0 if the caching device is not in use). To facilitate the calculation of $C_i(t)$ in practice, we assume a discretization of time at five-second granularity (i.e., we probe for the value $M_i(x)$ every five seconds and assume it stays constant during this interval). Since we have a total of 32 VM instances, the total cost in each of the three cases is

$$TC(t) = \sum_{i=1}^{32} C_i(t).$$

TC is expressed in AGBs (introduced in Section III-B) and is depicted in Figure 3(a). One can see a large gap between *static-slow* and *static-fast*, which steadily grows as the application progresses in time. However, not only is *adaptive* very close to *static-slow*, but overall it even manages to reduce the total adjusted storage accumulation by almost 7% because the application finishes faster. Compared with *static-fast*, this amounts to a reduction of 65% in cost, which is a large gain for the price of 3.3% performance overhead.

The evolution of total I/O activity in depicted in Figure 3(b). To calculate it, we divide the time (x axis) in five-second intervals and sum the I/O throughput (expressed in MB/s) observed during each interval for all VM instances (y axis).



(a) Evolution of total adjusted storage accumulation for all 32 VM instances (lower is better)

(b) Total I/O activity (reads and writes from all backing devices and, if applicable, caching devices of all 32 VM instances)

Fig. 3. CM1 (numerical model designed for idealized studies of atmospheric phenomena): a real-life HPC application that runs on 32 VMs (each on a different node) using 10 MPI processes/node

Since the application is bulk-synchronous, the I/O-intensive phases and the compute intensive phases overlap for all VM instances at the same time, leading to a clear delimitation in terms of I/O activity for all three approaches.

In the case of *static-fast* and *static-slow*, the I/O phases are short but of high amplitude and, respectively, longer but of lower amplitude. These results directly correspond to the ability of the underlying backing device to handle a fixed amount of I/O activity using a high and, respectively, low reserved bandwidth. With our approach, the behavior is more complex: the first I/O phase of the application generates a high I/O burst, followed by a clearly visible flush period. Since the first I/O phase is less intensive than the rest (for all three approaches), our approach picks a 4 GB cache device for the second phase, which then is increased to 6 GB for the last two phases. All I/O phases except the first exhibit an I/O burst that is fused to the flush period, with only a small difference noticeable between the second phase and the third, which itself is almost identical to the fourth. Intuitively, this hints at the ability of our approach to optimally adjust the cache size based on the previous I/O-intensive phases.

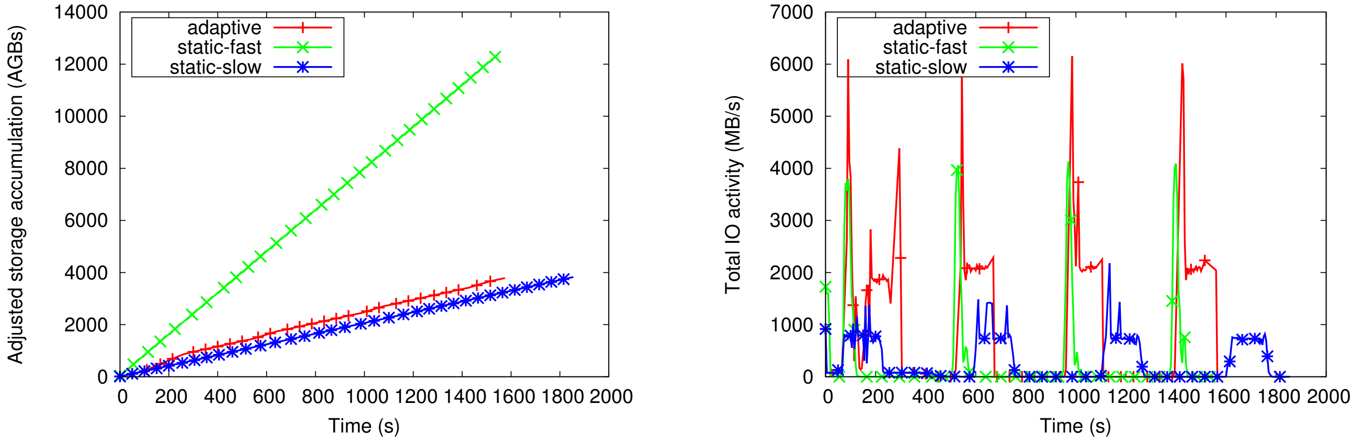
D. Case study: LAMMPS

A second real-life HPC application we use to demonstrate the benefits of our approach is *LAMMPS* [32], a large-scale atomic/molecular massively parallel simulator. LAMMPS can be used to model atoms or, more generically, particles at the atomic, meso, or continuum scale. Such modeling is useful in understanding and designing solid-state materials (metals, semiconductors), soft matter (biomolecules, polymers), and coarse-grained or mesoscopic systems. Similar to CM1, it exhibits an iterative behavior of alternating between a compute phase and an I/O-intensive phase to write intermediate output results and checkpoints, which are used for restart in case of failures.

For this work, we have chosen as input data a 3D Lennard-Jones melting scenario. Melting, the phenomenon of phase transition from a crystalline solid state to a liquid state, is one of the most important phase transformations in the processing and applications of materials, playing an important role in materials science and engineering [33]. As in the case of CM1, we run the simulation on 32 VMs, with each VM hosted on a separate physical node and equipped with 11 virtual cores, each of which corresponds to a physical core, with the remaining physical core reserved for the hypervisor. Inside each VM, 10 of the virtual cores are reserved for MPI processes, while the remaining core is reserved for operating system overhead. Each MPI process is responsible for a 20x160x160 subdomain. Thus, the total deployment amounts to 320 MPI processes that are evenly spread over 32 VMs and process a 6400x160x160 grid. The output/checkpointing frequency is set at 30 simulation time steps, out of a total of 100 time steps. This results in an initial I/O- intensive phase to dump the initial state, followed by three I/O- intensive phases interleaved with computational phases.

The settings are identical to the setup used in Section IV-C for CM1: the size of the backing device is fixed at 50 GB, with a reserved bandwidth of 33 MB/s for *static-slow* and *adaptive*. The bandwidth of the backing device is fixed at 128 MB/s for *static-fast*. The caching device has a bandwidth of 128 MB/s and an initial size of 10 GB. Each experiment is repeated five times for all three approaches, and the results are averaged.

Performance results are depicted in Table II, where *static-fast* is used as a baseline for the fastest possible completion time. As can be observed, using a slow backing device in the case of *static-slow* leads to a significant increase (15.5%) in overall completion time when compared with *static-fast* because of the longer I/O phases. On the other hand, *adaptive* successfully reduces the overhead of the I/O phases to such extent that the overall increase in completion time becomes



(a) Evolution of total adjusted storage accumulation for all 32 VM instances (lower is better)

(b) Total I/O activity (reads and writes from all backing devices and, if applicable, caching devices of all 32 VM instances)

Fig. 4. LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator): a real-life HPC application that runs on 32 VMs (each on a different node) using 10 MPI processes/node

negligible (i.e., around 1%).

TABLE II
LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator): Performance results

Approach	Completion Time	Overhead
static-slow	1680s	15.5%
static-fast	1454s	–
adaptive	1476s	1.01%

For this negligible increase in completion time, our approach achieves massive reductions in cost when compared with *static-fast* and even a moderate reduction in cost when compared to *static-slow*. These results are depicted in Figure 4(a) as the total adjusted storage accumulation for all instances ($TC(t)$). For all three approaches, $TC(t)$ is calculated in the same way as is described in Section IV-C (since we use the same configuration as in the case of CM1).

More specifically, *adaptive* reduces the storage accumulation by more than 70% when compared with *static-fast* and marginally by 1% compared with *static-slow* (thanks to the fact that it finishes faster), effectively making it a double winner over *static-slow* in terms of both performance and cost.

To understand how the interaction between backing devices and caching devices contributes to the results, we again analyze the total I/O activity. In Figure 4(b) a pattern similar to the case of CM1 is observable: *static-fast* and *static-slow* exhibit a short but intense I/O burst vs. a longer but milder I/O burst, with our approach successfully attaching the caching device at the right time and adapting its size based on the past experience (demonstrated by the shorter and more regular flush periods after the initial burst for the last three I/O-intensive phases when compared with the initial I/O-intensive phase where the optimal cache size is unknown).

V. Conclusions

The ability to transparently adapt to the fluctuations of I/O throughput requirements in order to reduce the costs associated with overprovisioning faster (and more expensive) virtual disks is crucial in the context of large-scale scientific applications for two reasons: (1) such applications tend to have an iterative behavior that interleaves computational phases with I/O-intensive phases, which leads to extreme fluctuations in I/O requirements that make overprovisioning particularly wasteful; and (2) such applications run in configurations that include a large number of VMs and associated virtual disks, which amplifies the waste due to scale.

In this paper we have described a solution that relies on a specialized block device exposed inside the guest operating system in order to intercept all I/O to a potentially slow virtual disk and enhance its throughput during I/O peaks by leveraging an additional faster, short-lived virtual disk that acts as a caching layer. Our approach offers low-level transparency that enables any higher-level storage abstraction to benefit from throughput elasticity, including those not originally designed to run on IaaS clouds. Furthermore, it relies solely on standard virtual disks to operate, which offers a high degree of compatibility with a wide range of cloud providers.

We demonstrated the benefits of this approach through experiments that involve dozens of nodes, using both microbenchmarks and two representative real-life HPC applications: *CM1* and *LAMMPS*. Compared with static approaches that overprovision fast virtual disks to accommodate the I/O peaks, our approach demonstrates a reduction of storage cost in real life (using a cost model that charges users proportionally to disk size and reserved bandwidth) of 66%–70%, all of which is possible with a negligible performance overhead (1%–3.3%) when compared with the fastest and more expensive solution. Furthermore, our findings show that using slow virtual disks to minimize storage costs is not the optimal solution: because of higher performance degradation (15%–23%), applications end up using the cheap virtual disks for longer, thereby resulting in an overall increase in cost compared with our approach (1%–7%).

Encouraged by these initial results, we plan to develop this work in several directions. One straightforward extension is to explore how to leverage multiple virtual disks, potentially in striping configuration, in order to boost I/O throughput during peaks. Another interesting idea could be to explore putting the backing device itself in stand-by during compute-intensive phases and to use a smaller caching device of equal bandwidth capability to serve most I/O requests, under the assumption that it makes sense to pay for the penalty of reattaching the backing device in order to deal with cache misses and evictions. Such an approach has the potential to further reduce the costs. In a more broader sense, an interesting topic to explore is how to achieve the minimum cost and what relationship it has to the performance. Especially interesting in this context is the problem of minimizing the cost for a given time to solution: this may involve a different elasticity strategy that accepts higher performance degradation as long as it manages to run the application in less than the given time to solution.

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 U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References

[1] 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.

[2] 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 Conference on Cluster, Cloud and Grid Computing*, Melbourne, Australia, 2010, pp. 43–52.

[3] S. Niu, J. Zhai, X. Ma, X. Tang, and W. Chen, "Cost-effective cloud hpc resource provisioning by building semi-elastic virtual clusters," in *SC' 13: Proc. 26th International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, USA, 2013, pp. 56:1–56:12.

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

[5] B. Nicolae and F. Cappello, "BlobCR: Virtual disk based checkpoint-restart for HPC applications on IaaS clouds," *Journal of Parallel and Distributed Computing*, vol. 73, no. 5, pp. 698–711, May 2013.

[6] A.-I. A. Wang, G. Kuenning, P. Reiher, and G. Popek, "The conquest file system: Better performance through a disk/persistent-ram hybrid design," *Trans. Storage*, vol. 2, no. 3, pp. 309–348, Aug. 2006.

[7] S. Qiu and A. L. N. Reddy, "Nvmfs: A hybrid file system for improving random write in nand-flash ssd," in *MSST' 13: Proc. 38th International Conference on Massive Storage Systems and Technology*, Lake Arrowhead, USA, 2013, pp. 1–5.

[8] B. C. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Storage-aware caching: Revisiting caching for heterogeneous storage systems," in *FAST'02: Proc. 1st USENIX Conference on File and Storage Technologies*, 2002, pp. 5–5.

[9] G. Wu, X. He, and B. Eckart, "An adaptive write buffer management scheme for flash-based ssds," *Trans. Storage*, vol. 8, no. 1, pp. 1:1–1:24, Feb. 2012.

[10] Y. Wang, J. Shu, G. Zhang, W. Xue, and W. Zheng, "Sopa: Selecting the optimal caching policy adaptively," *Trans. Storage*, vol. 6, no. 2, pp. 7:1–7:18, Jul. 2010.

[11] B. Nicolae and M. Rafique, "Leveraging Collaborative Content Exchange for On-Demand VM Multi-Deployments in IaaS Clouds," in *Euro-Par '13: 19th International Euro-Par Conference on Parallel Processing*, Aachen, Germany, 2013, pp. 305–316.

[12] Y.-H. Chang, P.-Y. Hsu, Y.-F. Lu, and T.-W. Kuo, "A driver-layer caching policy for removable storage devices," *Trans. Storage*, vol. 7, no. 1, pp. 1:1–1:23, Jun. 2011.

[13] H. Jo, Y. Kwon, H. Kim, E. Seo, J. Lee, and S. Maeng, "Ssd-hdd-hybrid virtual disk in consolidated environments," in *Euro-Par'09: Proc. 15th International Conference on Parallel Processing*, Delft, The Netherlands, 2010, pp. 375–384.

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

[15] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield, "Parallax: Virtual disks for virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 4, pp. 41–54, Apr. 2008.

[16] 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.

[17] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou, "S-cave: Effective ssd caching to improve virtual machine storage performance," in *PACT '13: Proc. 22nd International Conference on Parallel Architectures and Compilation Techniques*, Edinburgh, UK, 2013, pp. 103–112.

[18] Y. Xu, D. Arteaga, M. Zhao, Y. Liu, R. J. O. Figueiredo, and S. Seelam, "vPFS: Bandwidth virtualization of parallel storage systems," in *MSST' 12: Proc. 38th International Conference on Massive Storage Systems and Technology*, Pacific Grove, USA, 2012, pp. 1–12.

[19] H. C. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," in *ICAC '10: Proc. 7th International Conference on Autonomic computing*, Washington DC, USA, 2010, pp. 1–10.

[20] D. Chiu, A. Shetty, and G. Agrawal, "Elastic cloud caches for accelerating service-oriented computations," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. New Orleans, USA: IEEE Computer Society, 2010, pp. 1–11.

[21] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "Logbase: A scalable log-structured database system in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 1004–1015, Jun. 2012.

[22] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears, "Walnut: A unified cloud object store," in *SIGMOD '12: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. Scottsdale, USA: ACM, 2012, pp. 743–754.

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

[24] "Amazon Elastic Block Storage (S3)," http://aws.amazon.com/s3/.

[25] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008.

[26] B. Nicolae, P. Riteau, and K. Keahey, "Bursting the cloud data bubble: Towards transparent storage elasticity in iaas clouds," in *IPDPS '14: Proc. 28th IEEE International Parallel and Distributed Processing Symposium*, Phoenix, USA, 2014.

[27] Google, "Google Compute Engine Pricing," https://developers.google.com/compute/pricing, 2014.

[28] "SoftLayer," http://www.softlayer.com/.

[29] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *SC '10: Proceedings of the 23rd ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. New Orleans, USA: IEEE Computer Society, 2010, pp. 1–11.

[30] "BCache," http://bcache.evilpiepirate.org.

[31] G. H. Bryan and R. Rotunno, "The maximum intensity of tropical cyclones in axisymmetric numerical model simulations," *Journal of the American Meteorological Society*, vol. 137, pp. 1770–1789, 2009.

[32] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, no. 1, pp. 1–19, Mar. 1995.

[33] C. Das and J. Singh, "Melting transition of confined lennard-jones solids in slit pores," *Theoretical Chemistry Accounts*, vol. 132, no. 4, 2013.