Towards scalable on-demand collective data access in IaaS clouds: An adaptive collaborative content exchange proposal

Bogdan Nicolae, Andrzej Kochut, Alexei Karve

Towards Scalable On-Demand Collective Data Access in IaaS Clouds: An Adaptive Collaborative Content Exchange Proposal

Bogdan Nicolae[a], Andrzej Kochut[b], Alexei Karve[b]

[a]*IBM Research, Ireland*
[b]*IBM Research, USA*

Abstract

A critical feature of IaaS cloud computing is the ability to quickly disseminate the content of a shared dataset at large scale. In this context, a common pattern is *collective read*, i.e., accessing the same VM image or dataset from a large number of VM instances concurrently. Several approaches deal with this pattern either by means of pre-broadcast before access or on-demand concurrent access to the repository where the image or dataset is stored. We propose a different solution using a hybrid strategy that augments on-demand access with a collaborative scheme in which the VMs leverage similarities between their access pattern in order to anticipate future read accesses and exchange chunks between themselves in order to reduce contention to the remote repository. Large scale experiments show significant improvement over conventional approaches from multiple perspectives: completion time, sustained read throughput, fairness of I/O read operations and bandwidth utilization.

Keywords: IaaS, scalable content dissemination, collective I/O, on-demand read access under concurrency, collaborative content exchange, adaptive prefetching, I/O access pattern awareness

1. Introduction

Infrastructure-as-a-Service (IaaS) cloud computing has matured over the years up to the point where it represents a cost-effective and sustainable environment for large-scale, data-intensive applications. Thanks to its low entry barrier, it is not only a viable solution for traditional HPC (high performance computing) and data-intensive applications ported on IaaS clouds, but it is also fueling a new generation of applications specifically written from scratch to take advantage of cloud capabilities, which is often referred to as *cloud-native* [9].

A particularly difficult challenge in this context is the *collective read* I/O pattern, i.e., provisioning a large number of inter-dependent VMs (we henceforth refer to as a *multi-deployment*) that concurrently read from the same VM disk image (e.g., boot and launch applications) or from a large dataset (e.g., shared input data). This pattern is not only encountered when running traditional HPC (high performance computing) and data-intensive applications, but it is also a central aspect of running cloud-native applications. This latter aspect is of particular interest because it involves an elasticity-centered design, focused on scale-out micro-services that share stateless data as needed. In this context, there are extreme agility requirements: micro-services need to grow or shrink by hundreds of VM instances at a time, either as a consequence of fluctuations in the workload itself or because of agile development platforms (e.g. *Asgard* [5]) that facilitate automated upgrades by constantly "baking" new versions of virtual disk images and replacing old multi-deployments based on them, sometimes even within hours.

Obviously, there is a need to minimize the provisioning time and guarantee scalability despite a growing number of VM instances, otherwise users will perceive a disruption in the normal application runtime and development cycle. At the same time, if the provisioning is inefficient, cloud providers can charge users for less time and lose potential profit, even to the point where this can interfere with the cloud business model itself. For example, many cloud providers

Email addresses: bogdan.nicolae@ie.ibm.com (Bogdan Nicolae), akochut@us.ibm.com (Andrzej Kochut), karve@us.ibm.com (Alexei Karve)

enable users to bid for idle cloud resources at lower than regular prices if they accept their lease to be terminated at any point without warning (e.g., spot instances [4]). In this case, users will not be interested to acquire such resources if they take too long to initialize.

Despite widespread need for efficient collective reads, little effort has been undertaken to improve their scalability. Current techniques often pre-broadcast the full VM image or dataset on the local storage devices of the compute nodes before allowing any reads. Since the sizes involved often reach the order of tens of GBs, a pre-broadcast can take in the order of tens of minutes or even hours [33], not counting the time spent to read the content afterwards. Most of the time, this approach is sub-optimal because of two reasons: (1) not all content is actually read; and (2) reads need to wait for the whole broadcast to finish. Thus, on-demand techniques have been introduced that avoid a pre-broadcast (e.g., copy-on-write techniques [12] that create and gradually populate a local snapshot of the image or dataset by reading only the necessary data from the remote source as needed). Such techniques have been shown to generate little overhead on application performance compared to the case when a local copy is available [8]. However, they saw comparatively little attention for collective reads due to the fact that they generate I/O contention to the repository, which limits the scalability even at moderate scale.

This paper contributes with a novel technique that aims to alleviate the aforementioned issue and improve the scalability of collective reads by enabling efficient decentralized on-demand access that avoids I/O bottlenecks caused by contention to the repository. To achieve this, we leverage the fact that other VM instances may already have read and cached the necessary pieces of the disk image or dataset locally and thus can act as alternative sources in addition to the repository. Using this approach, we build a collaborative scheme where VMs exchange information about the access pattern and pro-actively fetch the hot content that is needed or likely to be needed later, which greatly speeds up subsequent read requests and avoids the contention to the repository. Our approach can dynamically adapt to the access pattern: it increases the rate of fetches when remote accesses were successfully avoided, and backs off when the success rate starts dropping due to diverging access patterns.

This contribution extends our previous work on collaborative content exchange during on-demand data access [24], where we explored a push strategy that enables VM instances of the same multi-deployment to send the content of a shared disk image accessed during run-time to other VM instances, under the assumption that this content will be required later. Although effective at improving the average performance of I/O operations, such a strategy leads to high throughput variability between the VM instances and can cause excessive bandwidth consumption. Furthermore, we explored our previous strategy only in the context of a single access pattern. In this work we introduce a novel content exchange and access pattern adaptation strategy that adopts the opposite angle: it pushes only advertisements about the content of datasets accessed in an on-demand fashion, while relying on a pull-based approach to fetch the data itself. Using this new strategy, we manage not only to address the aforementioned issues, but also to efficiently handle a multitude of other popular collective I/O access patterns. We also show high adaptability to the degree of content overlap: when the access patterns diverge (e.g., each VM instance accesses different data), our approach detects this and refrains from collaboration, which avoids unnecessary bandwidth consumption and performance overhead.

We summarize our contributions as follows:

- We introduce a series of general principles that form a collective content exchange strategy optimized to handle the multi deployment pattern by enabling efficient sharing of virtual disk image templates in an on-demand fashion. (Section 3.1).

- We design a series of new algorithms that materialize the aforementioned principles based on our findings from our previous work [24]. The key novelty consists in propagating information about the on-demand access pattern between the VM instances, which can be then independently leveraged to optimize the collaborative content exchange based on the collective I/O access pattern (Sections 3.1).

- We propose a hypervisor-transparent implementation of this scheme as an independent FUSE module that can mount a raw remote backing file locally as a mutable snapshot. Furthermore, we show how this FUSE module can be integrated in a typical IaaS architecture. This is functionally equivalent to the broadcast technique but completely removes the broadcast overhead (Sections 3.4 and 3.3).

- We experimentally evaluate the benefits of our approach on the Shamrock testbed by performing multi-deployments on dozens of nodes (Section 4) using several different collective I/O access patterns.

2. Related Work

Approaches that enable collective reads broadly fall into two major categories: pre-broadcast and on-demand access.

Pre-broadcast techniques fully copy all content locally on all compute nodes before launching the VM instances themselves. This enables high I/O disk access performance inside the VM instances, because no remote access to the image repository or I/O competition with other VM instances is present. However, the broadcast step has a high overhead both in execution time and network traffic, which reduces the attractiveness of IaaS for short-lived jobs and is expensive for the provider (in terms of lost resources that could otherwise be charged for). Thus, reducing the broadcast overhead has been an active area of study, with proposals ranging from multi-cast [14] and application level broadcast-trees [2] to peer-to-peer protocols [28, 33].

At the other extreme are on-demand access approaches: VMs are instantiated on-the-fly by keeping the initial virtual disk image or dataset as a read-only snapshot on the remote repository. Then, a copy-on-write view of the read-only snapshot is created locally that stores all required modifications on the local storage devices, while accessing the content of the read-only snapshot on-demand only. Many hypervisors support this feature at block-level using copy-on-write virtual disk image formats (e.g., *qcow2* [12]). While this eliminates the need for pre-broadcast, it introduces I/O competition between VM instances because they share a single source for the content that needs to be accessed: the repository. Obviously, a centralized repository generates the highest contention, but is still a very popular choice due to simplicity [27]. Using a decentralized storage solution (such as a parallel file system [7, 29, 34] or a dedicated repository [20]) reduces contention thanks to striping, but is only partially effective in our case, because the VM instances often access the same pieces of data in the same order. In our previous work [22], we show how to alleviate this issue by means of adaptive prefetching, however I/O contention to the repository is still a potential problem for scalability.

Another emerging direction that relates to collective reads is user-level virtualization [35, 19]. The idea here is to use a minimally configured OS and virtual disk on top of which application packages, configuration files and user data are copied on-the-fly during boot time. In this context, the same content propagation principles that apply at low level (i.e., virtual disk blocks) can be used for pieces of data at higher level.

Finally, there are several ways to complement existing techniques with additional optimizations. A straightforward optimization is to use the local storage on the compute nodes as a caching layer [11]. While this does not improve first-time accesses, given the dynamicity of the cloud, there is a high chance that much of the content fetched from VM images and datasets during the lifetime of a VM instance can be reused for subsequent instances running on the same compute node. Another related effort [25] proposes to build copy-on-reference caches that are exposed as independent images and can be persisted. In effect, such a cache image captures the most frequently accessed blocks of its backing file and can be used to fill the gap between pre-propagation and on-demand access. However, given the large variety of VM image templates in a cloud, it is highly probable to quickly run out of local storage if full caching is attempted. Luckily, VM images share a large amount of content between each other, which makes de-duplication [15, 31] an effective tool to leverage local storage more efficiently.

Our own previous work [24] explores how to improve on-demand approaches by having a VM instance push the content it accessed during runtime to other VM instances, effectively avoiding I/O accesses to the shared virtual disk image read-only snapshot. However, despite being effective at improving the average performance of I/O operations, such a strategy leads to high throughput variability between the VM instances and can cause excessive bandwidth consumption.

This paper focuses on disseminating only the needed content on-the-fly, similar to on-demand techniques, but achieves this by advertising and fetching content between the VM instances, similar to peer-to-peer broadcast techniques. Although there are studies that analyze the effectiveness of sharing incentives and clustering during content exchange (especially in the context of *BitTorrent* [17, 18]), most of these aspects were tackled in the context of pre-broadcast, independently of an actual on-demand collective I/O pattern. The potential benefits of peer-to-peer content exchange that is guided by an actual on-demand streaming access pattern was explored before by efforts such as VM-Torrent [26], where the authors combine a profile-based execution prefetch with on-demand fetch. Like VMTorrent, our approach also adopts peer-to-peer content exchange for on-demand access. However, our work is different in that we emphasize access pattern awareness: how to advertise and learn information about the I/O access pattern on-the-fly,

without need for profiling information. This is a key ingredient in enabling adaptive prefetching that helps deliver high performance, scalability, fairness and low bandwidth utilization for a variety of access patterns.

3. System Design

This section describes the design principles and algorithms behind our approach (Sections 3.1 and 3.2), how to apply them in a cloud architecture(Section 3.3) and finally how to efficiently implement them in practice (Section 3.4).

Note that in the description of our approach, we focus on virtual disk images as the content accessed by collective reads. However, the techniques, the algorithms and the prototype we introduce can be applied without modifications to handle any generic unstructured dataset that can be represented as a sequence of bytes.

3.1. Design Principles

3.1.1. Copy-on-Reference Local Mirroring:

To facilitate on-demand VM disk image access, we leverage *copy-on-reference*, initially introduced for process migration in the V-system [32]. To this end, our approach exposes a private local view of the virtual disk image stored remotely on the VM repository to the hypervisor. We call this local view a *mirror*. From the perspective of the hypervisor, the local mirror behaves like the original and it is functionally equivalent to a local copy using pre-broadcast. The mirror is logically partitioned into fixed-sized *chunks*. Whenever the hypervisor needs to read a region of the image, all chunks covered by the region that are not already locally available are fetched from the original source and copied locally (i.e., "mirrored"). Once all contents is available locally, the read can proceed. Writes behave in a similar fashion, except for those chunks that are totally overwritten: in this case no remote fetch is necessary. Note that when a VM instance uses the mirror directly as its underlying virtual disk image, write operations will be treated as copy-on-write. However, this is by no means a requirement: the mirror can also be used as a local read-only backing image for multiple separate copy-on-write images. In this mode, the mirror does not need to handle writes and the VM instances can share and populate the mirror in an on-demand fashion. To cover both cases, we consider the most complex scenario when both reads and writes are possible.

3.1.2. Peer-to-Peer Chunk Advertisement and Exchange:

As explained in Section 2, on-demand access has a serious disadvantage as it generates I/O access contention to the remote repository where the VM disk image is stored. Although copy-on-reference limits this effect to first-time reads only (because the local mirror gradually becomes populated), by itself this is often not enough, as most access patterns need to read data only once (e.g., read configuration files during the boot process or sweep through an input data set in order to perform a computation). Thus, optimizing first-time reads becomes a prime concern. Since the VM instances of multi-deployments often follow a similar access pattern, a natural idea in this context is to enable the mirrors to talk to each other and "help" each other out in order to reduce the pressure on the remote repository. In this context, we propose to organize the mirrors in a peer-to-peer fashion, such that each mirror has a set of neighbors with whom it "gossips" about the access pattern: what chunks were locally mirrored and in what order. Based on this information, each mirror constructs a list of alternative locations for each chunk that can be used instead of the original source to fetch the missing chunks. To avoid I/O contention to the original source, using an alternate location if available to fetch missing chunks is preferred. This is done using a load-balancing strategy: if multiple neighbors advertised the same chunk, the one that has the least number of pending requests that need to be served is selected.

3.1.3. Prefetching based on I/O Access Jitter:

In a multi-deployment, even if the I/O access pattern is highly similar, there are slight delays between when a chunk is accessed for the first time by a VM instance vs. the rest of the VM instances. We refer to these delays as "jitter". Our chunk advertisement and exchange technique detailed above is exploiting jitter under the assumption that advertisements arrive in time to be able to construct a list of alternative locations for a chunk before it is actually read. However, many times the jitter is much larger than the time required to propagate advertisements (which is mostly latency-bound due to the tiny size of the advertisement). To illustrate this point, consider the boot phase of 120 VM instances that are part of the same multi-deployment (using a Debian Sid Linux distribution that boots from a 2 GB large virtual raw image striped in chunks of 256 KB [22]).

Table 1: Distribution of the jitter for all chunks read during the boot phase by more than one VM instance

Average delay	0s	0-1s	1-2s	2-3s	3-4s	4-5s	Over 5s
#Chunks	0	4	10	0	2	447	2

Table 1 shows how the jitter is distributed among all chunks accessed by more than one VM instance. As can be observed, for most chunks there is an average delay greater than 4s from the moment it was accessed the first time. Thus, jitter is not the consequence of a few "laggers" that tend to access a chunk later than everybody else, but rather a global phenomenon experienced by all VM instances. Given the high throughput that the networking infrastructure is able to sustain in a data-center, such a jitter gives enough time to send more than just advertisements. Thus, we propose to leverage the jitter to prefetch chunks that were advertised by neighbors, under the assumption that those chunks will be needed in the near future and can mirrored fully (or at least partially) before a read call is issued.

3.1.4. Access Pattern Aware Prefetch Throttling:

Prefetching however is not without drawbacks. Although the performance overhead of chunk transfers can be masked by decoupling prefetching from on-demand access and running it as a background process, it invariably leads to network bandwidth utilization. This steals away bandwidth from the application running inside the VM instance and might even impact the on-demand access bandwidth. Besides performance and bottleneck considerations, bandwidth is also an expensive resource and thus must not be wasted. Therefore, it is crucial to "focus the prefetching" such that it maximizes the prediction rate, and thus minimizes the bandwidth wasted on obtaining chunks that were never needed. To this end, we propose to monitor the success rate in terms of number of chunks that were fetched locally but not yet accessed (which we refer to as *unmatched*). When the number of unmatched chunks reaches a predefined threshold, we assume the VM has started to exhibit an access pattern that diverges from the rest of the neighborhood and as such it will avoid prefetching chunks until the number of unmatched chunks falls below the threshold. Using this scheme, each VM dynamically adapts to the access pattern in relationship to the other VMs, strengthening its exchanges when it senses a common pattern, and backing off when it senses a divergence. To avoid the case when the VMs converge again without lowering the amount of unmatched chunks accumulated in the past, we monitor how many chunk advertisements that lead to on-demand exchanges were held back from prefetching due to a high number of unmatched chunks. When the predefined threshold is reached, we assume that the access patterns converge again and thus we reset the number of unmatched chunks, which effectively re-activates prefetching.

3.2. Algorithmic Description

In this section, we zoom on the design principles presented in Section 3.1 by providing an algorithmic description. For simplicity, we insist only on the most important aspects, in particular how a read and a write is performed and how to decouple the peer-to-peer on-demand exchange from the prefetching strategy in order perform it asynchronously in the background.

The local mirror corresponding to the virtual disk image (denoted $Mirror$) is split into fixed sized chunks. Each chunk of the $Mirror$ can be in one of the five possible states (denoted $ChunkState$): $REMOTE$ (the chunk was not yet locally mirrored), $PREFETCH$ (the chunk is in the process of being prefetched from another mirror), $LOCAL$ (the chunk was successfully prefetched and mirrored locally, but was not yet read), $READ$ (the chunk was needed by a read operation) and $WRITTEN$ (the chunk was overwritten either totally or partially). Furthermore, there is a global count for *unmatched* chunks, initially set to zero. This count gets incremented every time a chunk is prefetched and decremented every time a chunk gets is read after being prefetched.

The READ operation is detailed in Algorithm 1. In a nutshell, it ensures that all chunks that intersect with the range $offset, size$ from $Mirror$ are locally available, after which it redirects the read request to the local mirror. More specifically, if a chunk is in the process of being prefetched, then it waits for the prefetching to finish. If the prefetching was not successful (i.e., it timed out), then it reverts to the remote repository. Reverting to the remote repository is not immediate: the read request for the chunk is accumulated in the $Repo$ set and handled only after all chunks were processed, which avoids waiting for repository unnecessarily. If a chunk is not locally available or in the process of being prefetched ($REMOTE$ state), then READ attempts to fetch it from another mirror that advertised it (all advertisements for a chunk are accumulated in the $Source$ set). If such a mirror (denoted $Peer$) exists, then READ

Algorithm 1 Read the range ($offset$, $size$) into $buffer$ from disk image

```
1:  function READ(buffer, offset, size)
2:      Repo ← ∅
3:      for all chunk ∈ Image such that chunk ∩ (offset, size) ≠ ∅ do in parallel
4:          if ChunkState[chunk] = PREFETCH then
5:              wait until ChunkState[chunk] = LOCAL
6:              if not timeout then
7:                  unmatched ← unmatched − 1
8:              else
9:                  Repo ← Repo ∪ {chunk}
10:             end if
11:             ChunkState[chunk] ← READ
12:         else if ChunkState[chunk] = REMOTE then
13:             if Source[chunk] ≠ ∅ then
14:                 select least loaded Peer ∈ Source[chunk]
15:                 fetch chunk from Peer and mirror it locally
16:                 advertise chunk to neighbors (asynchronously)
17:             end if
18:             if Source[chunk] = ∅ or fetch unsuccessful then
19:                 Repo ← Repo ∪ {chunk}
20:             end if
21:             ChunkState[chunk] ← READ
22:         else if ChunkState[chunk] = LOCAL then
23:             unmatched ← unmatched − 1
24:             ChunkState[chunk] ← READ
25:         end if
26:     end for
27:     for all chunk ∈ Repo do in parallel
28:         fetch chunk from repository and mirror it locally
29:         advertise chunk to neighbors (asynchronously)
30:     end for
31:     return read (offset, size) into buffer from Mirror
32: end function
```

Algorithm 2 Write the range ($offset$, $size$) from $buffer$ to disk image

```
1:  function WRITE(buffer, offset, size)
2:      for all chunk ∈ Image such that chunk ∩ (offset, size) ≠ ∅ do in parallel
3:          if chunk ⊄ (offset, size) and not mirrored then          ▷ Partially overwritten
4:              mirror chunk locally using READ
5:          end if
6:          ChunkState[chunk] ← WRITTEN
7:      end for
8:      return write (offset, size) from buffer to Mirror
9:  end function
```

Algorithm 3 Asynchronous chunk advertisement and exchange with other peers

```
1:  procedure BACKGROUND_EXCHANGE
2:      while true do
3:          msg ← listen for any message from any peer
4:          if msg = chunk received and ChunkState[chunk] = PREFETCH then
5:              mirror chunk locally
6:              ChunkState[chunk] ← LOCAL
7:              unmatched ← unmatched + 1
8:              advertise chunk to neighbors (asynchronously)
9:          end if
10:         if msg = chunk requested then
11:             send chunk to requester or notify WRITTEN state
12:         end if
13:         if msg = advertisement from Peer then
14:             Source[chunk] ← Source[chunk] ∪ {Peer}
15:             if ChunkState[chunk] = REMOTE and unmatched < Threshold then
16:                 ChunkState[chunk] = PREFETCH
17:                 ask Peer to send chunk
18:             end if
19:         end if
20:     end while
21: end procedure
```

picks the one that is the least loaded and fetch the chunk from it, otherwise it reverts to the remote repository. Note that least loaded refers to a local view, i.e. the *Peer* to which the current mirror has issued the fewest pending requests for which it is still waiting for an answer. This is a simple load balancing strategy that avoids asking the same *Peer* for all content if multiple sources are available. However, more complex strategies that include load balancing information in advertisements to build a global view are also possible.

In either case, one the chunk is obtained one way or another, its state becomes *READ*. Furthermore, if the chunk was successfully obtained by prefetching either during the read call itself or earlier, then the number of *unmatched* chunks is decremented. Once all chunks were processed this way, all chunks deferred to be read from the repository (which are part of the *Repo* set) are mirrored locally and advertised asynchronously to the neighbors. At this point, all chunks needed by the read operation are locally available and the *Mirror* can be used to fill the $buffer$ where the result is stored.

The WRITE operation, depicted in Algorithm 2 simply needs to make sure that all chunks being overwritten are properly marked with the *WRITTEN* state in order to recognize and avoid sending locally modified chunks to other mirrors. Furthermore, if a chunk is only partially written to (i.e., not fully included in the written range), the part that is outside of the written range needs to be filled with the original content. To this end, we rely on READ to mirror the chunk locally before applying the write. Note that there is a slight difference in the way READ works in this case: since the chunks will be at least partially overwritten, READ will not advertise them. Once all chunks covered by ($offset$, *size*) are handled as above, the *Mirror* is overwritten with the contents of $buffer$.

Finally, BACKGROUND_EXCHANGE is responsible to perform the collaborative chunk advertisement and exchange scheme asynchronously. This is detailed in Algorithm 3: in a nutshell, it listens for advertisements about new chunks from all its neighbors and whenever it receives one, it adds the originating *Peer* to the corresponding *Source*[*chunk*] set. Furthermore, if the number of unmatched chunks is less than *Threshold*, it puts the chunk in the *PREFETCH* state and asks *Peer* to send it. If the payload received from any peer is a request for a chunk, the reply will be sent only if it was not modified locally, otherwise it will instruct the requester to look for an alternative. Whenever a chunk is received as a reply, the *Mirror* will be updated with its content, the state will be updated to *LOCAL* and the number of unmatched chunks is incremented. Furthermore, the chunk is advertised asynchronously to all neighbors.

Note that we use a priority based mechanism to handle requests from other peers: replies to on-demand chunk

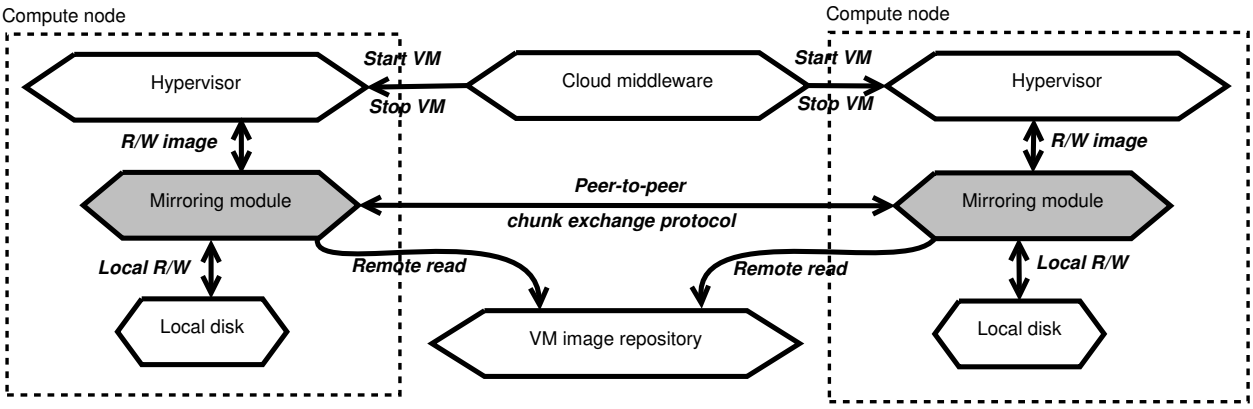


Figure 1: Cloud architecture that integrates our approach (dark background)

requests (as initiated by READ) take precedence over prefetch replies. Furthermore, the sending of requests is also prioritized: on-demand requests for a chunk take precedence over advertisements, which in turn take precedence over prefetch requests.

3.3. Architecture

We depict a simplified IaaS cloud architecture that integrates our approach in Figure 1. For better clarity, the building blocks that correspond to our own approach are emphasized with a darker background.

The *VM image repository* is the storage service responsible to hold the VM disk images that are accessed concurrently as read-only snapshots in a multi-deployment. The only requirement for the VM image repository is to be able to support random-access remote reads, which gives our approach high versatility to adapt to a wide range of options: centralized approaches (e.g., NFS server), parallel filesystems or other dedicated services that specifically target VM storage and management [20, 13].

The *cloud user* has direct access to the VM image repository and is allowed to upload and download VM images from it. Furthermore, the cloud user also interacts with the *cloud middleware* through a control API that enables launching and terminating multi-deployments. In its turn, the cloud middleware will interact with the *hypervisors* deployed on the compute node to instantiate the VM instances that are part of the multi-deployment.

Each *hypervisor* interacts with the local mirror of the VM disk image as if it were a full local copy of the VM disk image template. To facilitate this behavior, the *mirroring module* acts as a proxy that traps all reads and writes of the hypervisor and takes the appropriate action: it populates the local mirror on-demand only in a copy-on-reference fashion while using the peer-to-peer chunk exchange protocol described in Section 3.1 to pre-populate regions that are likely to be accessed in the future based on the collective access pattern trend.

3.4. Implementation

We implemented the mirroring module as file system in userspace on top of FUSE [1]. This has several advantages in our context: (1) it is transparent to the hypervisor (and thus portable); (2) it enables easy interfacing with any remote storage repository (since it is a userspace implementation) and (3) it is easy to integrate into existing cloud middleware, as it enables us to emulate a behavior that is functionally equivalent to pre-broadcast.

Furthermore, since the mirroring module relies on FUSE, the content that can be exposed to the upper layers is not limited to virtual disks and block devices: our implementation is highly generic and can be used to expose any regular file to a large number of concurrent readers. This opens two possibilities: first, the cloud provider could leverage it at host-level to build an advanced on-demand data dissemination service that is used by the VM instances indirectly. Second, a guest-level approach where the mirroring module is mounted directly by the application inside the VM instances is also possible in order to accelerate collective reads for user datasets when the cloud provider does not offer a native solution.

The peer-to-peer chunk exchange and prefetching strategy runs in its own thread, which communicates with the main FUSE thread through the mechanisms presented in Section 3.2. The communication between the mirroring modules is implemented on top of Boost ASIO [3], a high performance asynchronous event-driven library which is part of the Boost C++ collection of libraries. Since the chunk exchange and prefetching scheme is not a pre-condition for correctness (i.e. it is always possible to fall back to the original source in order to fetch the missing chunks), we have opted for a lightweight solution that performs gossiping through UDP communication channels. This has the potential to significantly reduce networking overhead at the cost of unreliable communication, which is a perfectly acceptable trade-off in our case.

4. Evaluation

This section evaluates the scalability of our approach experimentally for a series of collective read scenarios.

4.1. Experimental Setup

Our experiments were performed on the *Shamrock* testbed of the Exascale Systems group of IBM Research Ireland. For the purpose of 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 [6, 16] as the hypervisor. On each node, we deploy a VM that is allocated two cores and 8 GB of RAM. The guest operating system is a recent Debian Sid running the 3.10 Linux kernel and the corresponding backing image is stored on a NFS server that is accessible through the Gigabit Ethernet link. This configuration is typically encountered in practice. As an alternative, we also use *GlusterFS* [10] 3.6 instead of NFS to store the backing image in a striped fashion. The format of the image is RAW and its total size is 4 GB. 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. The I/O caching mode of QEMU/KVM is the default (i.e. cache=writethrough). We assume that the data accessed in parallel by all VM instances that are part of the multi-deployment is stored in the RAW image.

4.2. Methodology

We compare three approaches throughout our evaluation:

Full pre-broadcast to local storage. In this setting, the backing image is first broadcast from the NFS server to the local storage of each node that is hosting a VM. Once the local copy is available on every node, *QEMU/KVM* boots the VMs directly from the local copy and then the experiments are started. The broadcast phase itself is performed using *Scp-Tsunami*. This involves splitting the backing image into chunks and then propagating each chunk to all nodes using application-level broadcast trees. Once a node receives all chunks, it re-assembles the original backing image into a local copy, which is then used by *QEMU/KVM* to boot the VM instance. For the rest of this paper, we denote this approach scp−tsunami.

On-demand access using local copy-on-write. In this setting, the backing image residing on the NFS server or GlusterFS is used as a read-only template for locally stored copy-on-write images. More specifically, we rely on the *QCOW2* [12] image format (which is part of standard QEMU/KVM distribution) to create an initially empty local copy-on-write image on all nodes. Then, *QEMU/KVM* boots the VMs using the corresponding copy-on-write images as the root virtual disk, after which the experiments are started. We denote this approach nfs−qcow2 or gluster−qcow2, depending on the repository used to hold the backing image.

Adaptive collaborative content exchange using our approach. In this setting, the backing image (residing on the NFS server or GlusterFS) is mirrored locally on each node using the FUSE implementation presented in Section 3.4. Once the mirror was successfully mounted, *QEMU/KVM* boots the VMs directly from the mirror and then the experiments are started. With respect to the peer-to-peer topology, we opted for a circular double-linked list: we fix a predefined ordering of all nodes in a ring and link each mirroring module of a given node to the previous and next node in the ring. Thus, each mirroring module has three potential sources from where it can fetch chunks: its neighbors in the ring and the backing image. The chunk size is fixed at 32 KB. For the rest of this paper, we denote this setting ac−store. In

order to study the effectiveness of our adaptive prefetching strategy, we use a predefined unmatched chunks threshold which is fixed at 512. We compare this setting with two extremes: the case when prefetching is disabled completely (i.e. a threshold of 0) and the case when prefetching is performed regardless of the access pattern of the neighbors (i.e. an infinite threshold). The threshold is included in the notation: we refer to the first setting as ac−store−512 and the two extremes as ac−store−0 and ac−store−inf respectively.

These approaches are compared based on the following metrics:

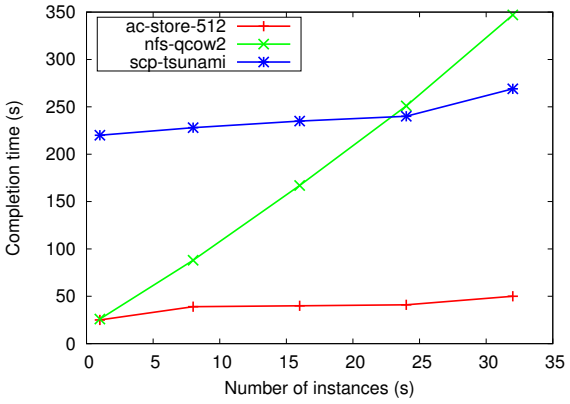
Completion time. This is the total time required to boot the VM instances and run the experiment to completion (i.e. until all VM instances have finished). In the case of scp−tsunami, it includes the pre-broadcast phase. This metric is relevant in order to understand how paying upfront for the overhead of pre-broadcast (as is the case of scp−tsunami) in order to avoid I/O bottlenecks during runtime on compares with the other on-demand approaches that avoid the pre-broadcast overhead at the expense of having to deal with concurrent I/O bottlenecks during runtime. A low value indicates better overall performance.

Sustained read throughput. This is the average read throughput as observed by the on-demand approaches for a given VM instance. This metric is irrelevant for scp−tsunami, because all content is locally available and cached before the experiment starts (for completeness, it stays around 1 GB/s, regardless of the number of VM instances). We use this metric as a means to compare the on-demand approaches with each other, both in terms of performance and scalability. Since we are dealing with multiple concurrent VM instances that share the same backing image, the sustained read throughput of each VM is expected to drop when increasing the number of concurrent VMs. However, not every VMs will experience the same sustained read throughput. Thus, an important part of our study is to understand the variance of sustained read throughput. For this reason, we measure and depict both the sustained read throughput of the fastest instance and slowest instance. In the notation, we append a corresponding suffix to mark the distinction between the two cases: max for the fastest and, respectively, min for the slowest. A high value for the sustained read throughput indicates better performance, while a small difference between min and max indicates a better fairness in terms of I/O allocation (i.e., a better interleaving of read requests that does not favor one VM instance over another).

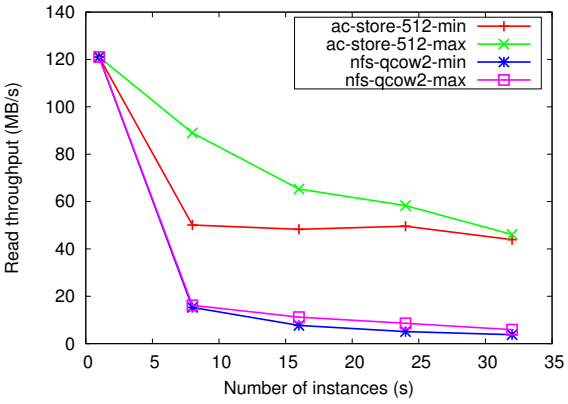
Average amount of chunk transfers per instance. This is the average size of the chunk transfers per VM instance performed by ac−store when grouped by access type. This metric is relevant in order to understand how each of the three settings (ac−store−512, ac−store−0 and ac−store−inf) interacts with the backing image and its neighbors under concurrency. The access type can be one of three options: from−source (which refers to the amount of data read on-demand directly from the backing image), peers−get (which refers to the amount of data fetched on-demand from the neighbors) and peers−prefetch (which refers to the amount of data pre-fetched from the neighbors before it was actually read). This metric is important from two perspectives: (1) to quantify the amount of data per VM instance that was not fetched from the backing file, which reveals to what degree the I/O bottlenecks due to concurrent access to the backing can be avoided; and (2) to understand how much unnecessary (i.e. chunks that were never read) data transfers are caused by prefetching compared to the ideal case (i.e. ac−store−0). A high value for (1) correlates with better sustained read throughput due to less I/O pressure on the backing file under concurrency and thus less I/O bottlenecks. A low value for (2) is desirable in order to avoid performance degradation, bottlenecks and extra costs due to unnecessary bandwidth utilization (as explained in Section 3.1.4).

4.3. Results

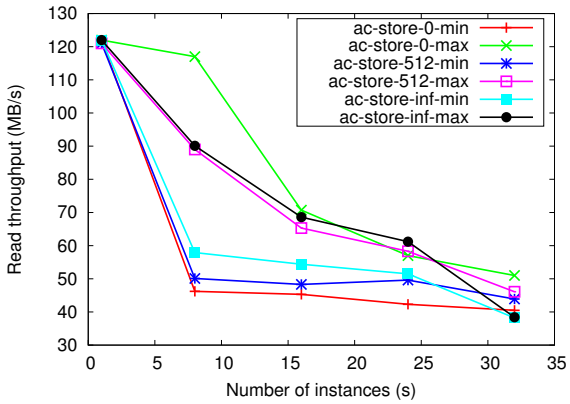
In our experiments, we focus on three collective read access patterns to virtual disks as exhibited by multi-deployments. First, we target reads of the same region (i.e., same set of chunks in the same order) when all VM instances start at the same time. This scenario is often encountered when a set of VM instances need to process the same input data concurrently (e.g., each VM scans the dataset for a query) VM instances that make up a distributed application (e.g., apply different algorithms concurrently on the same data). Second, we target reads of the same region when the VM instances do not necessarily start at the same time. This pattern is often encountered in applications that are conceptually similar to content delivery networks [30], i.e. they serve multiple queries in parallel by distributing them among a set of VM instances that act as a read-only cache. Third, we target concurrent reads of disjoint regions: all VM instances start at the same time, but read different parts of the virtual disk. This is a pattern often encountered



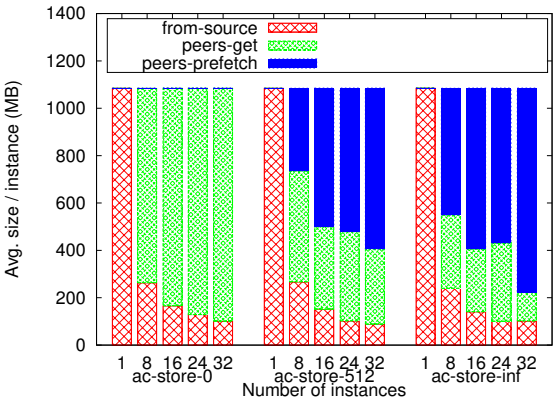
(a) Completion time for all VM instances (lower is better)



(b) Scalability of the read throughput compared with nfs−qcow2 (higher is better)



(c) Scalability of the read throughput for different prefetching strategies (higher is better)



(d) Average size of chunk transfers per VM instance grouped by source

Figure 2: Results for a scenario where all VM instances simultaneously read the same 1 GB large region of the virtual disk starting at offset 2^{30}. The reference repository is a centralized NFS server.

in embarrassingly parallel applications that split a large dataset among a set of workers, with each worker being responsible to read and process a different part of the dataset (e.g., the map phase of a Hadoop job). We denote the three patterns *Sync-Same*, *Shifted-Same* and *Sync-Disjoint* for the rest of this paper.

In all three cases, the experiments involve an increasing number of VM instances that run a micro-benchmark representative of the corresponding collective virtual disk access pattern mentioned above. For the purpose of this work, we use the dd tool to act as the micro-benchmark and issue the reads to the virtual disk. More specifically, after all VM instances finished booting, we run dd to read from the root virtual disk of each VM instance a predefined size S starting from offset O, where both S and O are expressed in MB: dd **if**=/dev/vda of=/dev/null bs=1M skip=O count=S. We start with one VM and gradually add 8 VMs until we reach the maximal number of 32 VMs.

4.3.1. Sync-Same: Read the same region at the same time

Our first series of experiments focuses on a collective read access pattern where all VM instances read the same region at the same time. The repository holding the backing image (i.e., the source of the data) is a centralized NFS server solution. We fix $O = S = 1024$, which corresponds a region that covers the second GB of the total of 4 GB. The results are depicted in Figure 2.

As can be observed, the completion time depicted in Figure 2(a) starts much higher for scp−tsunami compared

to the other two approaches: this is the effect of transferring the full content of the VM image before booting it, which dominates the completion time. At the other extreme, nfs−qcow2 starts with a very small completion time for a single VM instance, but suffers from poor scalability as the number of VM instances increases, up to the point where it performs slower than scp−tsunami for 32 VM instances. This effect can be traced back to the fact that the VM instances compete for the limited I/O bandwidth of the NFS server, which explains why the completion time is directly proportional with the number of VM instances. Our approach starts at the same level as nfs−qcow2, however thanks to our peer-to-peer dissemination strategy it maintains excellent scalability that even exhibits a smaller slope when compared with scp−tsunami, effectively achieving at the extreme of 32 instances a speed-up of more than 5x. This speed-up grows even higher to 7x when comparing ac−store−512 with nfs−qcow2.

To understand how on-demand accesses impact the performance perceived by the VM instances, we depict in Figure 2(b) the observed throughput of dd for both the fastest (max) and the slowest (min) VM instance. As mentioned in Section 4.2, an approach based on scp−tsunami sustains a much higher throughput because all content is already locally available and cached, at the expense of a costly pre-broadcast step. Thus, the results are not relevant in this context and were not depicted. As can be observed, ac−store−512 exhibits much better stability when compared with nfs−qcow2: at the extreme of 32 instances it is at least 10x faster, even when considering a comparison of its slowest instance with the fastest instance of nfs−qcow2. Furthermore, the difference between the fastest and slowest instance tends to stabilize below 5%, demonstrating a high degree of fairness among the VM instances. This result shows a significant improvement compared to the peer-to-peer push strategy described in our previous work [24] (where the average read throughput speedup was 20% compared with nfs−qcow2 and the difference between the slowest and fastest VM instance was very large at around 20x).

In terms of different prefetching strategies, Figure 2(c) shows a comparison of read throughput between ac−store−0, ac−store−512 and ac−store−inf. As can be observed, both ac−store−0 and ac−store−inf show a much larger difference between min and max when compared to ac−store−512 (almost 20% vs. 5%). These results show that both extremes of not using prefetching at all and, respectively, doing prefetching without any limitation negatively impact fairness either because of not fully making use of jitter, or, respectively, not leveraging temporal locality efficiently (i.e. prefetching "too far into the future" at the expense of more immediate requests). Furthermore, this also has a visible impact on the average sustained read throughput, as can be observed in Table 2.

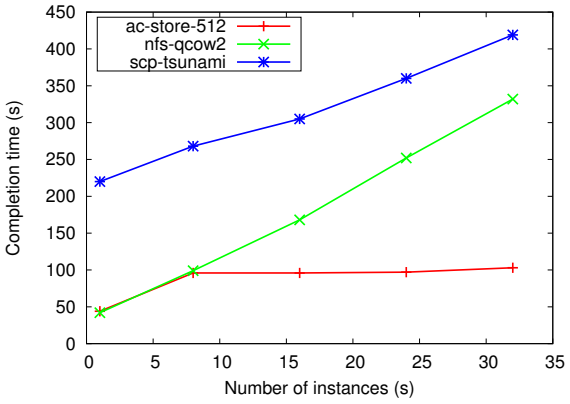
Table 2: Average sustained on-demand read throughput for 32 concurrent VM instances. The reference repository is a centralized NFS server.

Approach	Sync-Same	Shifted-Same	Sync-Disjoint
nfs-qcow2	4.4 MB/s	8.1 MB/s	4.8 MB/s
ac-store-0	42 MB/s	30 MB/s	4.5 MB/s
ac-store-512	44 MB/s	43 MB/s	4.7 MB/s
ac-store-inf	38 MB/s	159 MB/s	4.6 MB/s

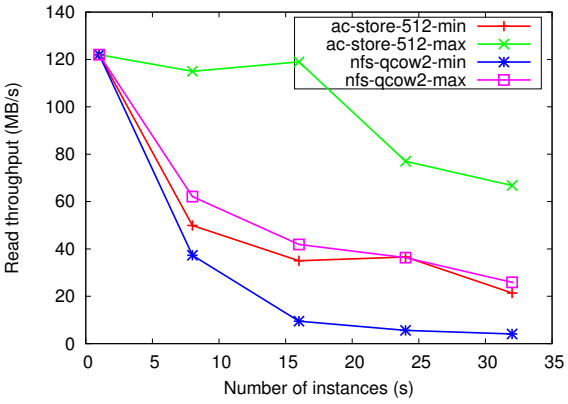
To understand where the chunks come from and how this impacts network traffic (as explained in Section 4.2), we depict in Figure 2(d) the average size of data transfers per VM instance grouped by source. As expected, in the case of a single VM instance, there are no neighbors available to get or prefetch chunks from. Thus, all chunks are read from the backing image. However, with increasing number of VM instances, the chance that a chunk can be obtained from a neighbor increases dramatically: from more than 70% for 8 VM instances up to more than 90% at 32 VM instances. Interesting to notice is that the prefetching strategy has negligible influence on interactions with the backing image. However, for the chunks that are obtained from neighbors rather than the backing image, it can be observed that jitter can be leveraged for prefetching for more than half of the chunks, both in the case of ac−store−512 and ac−store−inf. Furthermore, as expected, the proportion of on-demand chunks is smaller in the case of ac−store−inf when compared to ac−store−512. This excessive prefetching has negative impact on the fairness of the read throughput, as mentioned above.

4.3.2. Shifted-Same: Read the same region at slightly different time

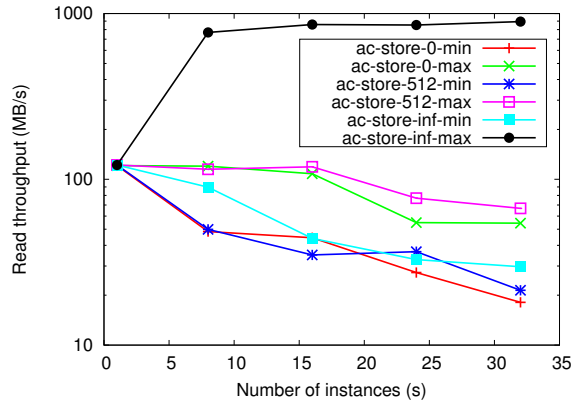
Our next series of experiments focuses on a collective virtual disk access pattern where all VM instances read the same region at different time. To this end, we fix the region $O = S = 1024$ as in the previous experiment. However, this time the VM instances do not start reading the region at the same time, but rather with a 5 second delay one from



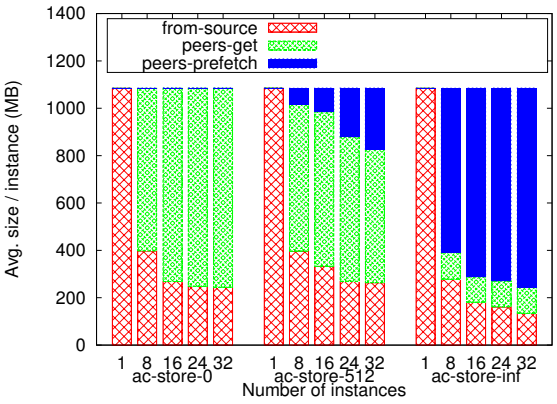
(a) Completion time for all VM instances (lower is better)



(b) Scalability of the read throughput compared with nfs−qcow2 (higher is better)



(c) Scalability of the read throughput for different prefetching strategies (higher is better)



(d) Average size of chunk transfers per VM instance grouped by source

Figure 3: Results for a scenario where the VM instances are started with a 5 second delay one from another. All VM instances read the same 1 GB large region of the virtual disk starting at offset 2^{30}. The reference repository is a centralized NFS server. Note the log-scale for the Y axis of Figure 3(c).

another (i.e., the first instance starts immediately, the second instance starts after 5 seconds, the third instance starts after 10 seconds, etc.). The results are depicted in Figure 3.

As can be observed in Figure 3(a), the delay between the VM instances is directly reflected in the completion time of scp−tsunami: compared to the previous scenario, for k VM instances the increase is roughly $5 \cdot k$ seconds. This is expected, since the absence of contention leaves the pre-broadcast, boot phase and read completion time unchanged, regardless of the collective read pattern. However, in the case of nfs−qcow2, the delay plays a major role in alleviating the I/O pressure on the NFS server: when compared to the previous scenario, the completion time is slightly smaller and consistently stays below scp−tsunami. In the case of ac−store−512, not only do the delays relieve the I/O pressure on the NFS server like in the case of nfs−qcow2, but they also increase the chance of successful (pre)fetches from the neighbors. Ultimately, this effect leads to much better performance and scalability for ac−store−512: the completion time remains almost constant starting from 8 VM instances and at the extreme of 32 VM instances, it is 3.5x times smaller compared to nfs−qcow2 and 4x smaller compared to scp−tsunami.

With respect to the sustained read throughput inside the VM instances, Figure 3(b) reveals a speedup of more than 3x when comparing the fastest instances, and, respectively, a speedup of more than 4x when comparing the slowest instances. Furthermore, the slowest instance of ac−store−512 is by almost 10% faster than the fastest instance of

nfs−qcow2. Also interesting to note is the larger difference between min and max for both approaches when compared with the previous scenario (Figure 2(b)), which is attributable to the delays. Overall, in terms of average sustained read throughput, ac−store−512 is more than 5x faster than nfs−qcow2, as can be observed in Table 2.

In terms of impact of prefetching strategy, Figure 3(c) reveals ac−store−0 as the worst performer, both in terms of min and max. This is understandable, considering that no prefetching is performed despite increased opportunity to do so due to the delays. In the case of ac−store−512, prefetching leads to a speedup of 18% for min and, 29% for max and an overall average speedup of 43% when compared to no prefetching. The best performer is ac−store−inf, with a speedup of 30% for min, 1752% for max and an average speedup of 530% when compared to no prefetching.

Figure 3(d) explains why a large gain for max in the case of ac−store−inf is possible: prefetching brings most of the required chunks locally before the read operations are issued. This is in contrast with ac−store−512, where the proportion of prefetches is much smaller compared to the proportion of on-demand fetches: the delays between the VM instances quickly lead to an accumulation of unmatched chunks beyond the threshold, which prevents further prefetching for most of the duration of the delays. Interesting to note is also the large gap between min and max in the case of ac−store−inf, which is expected because the first VM instance essentially acts as a seed for all other instances and needs to fetch the missing chunks directly from the backing image. However, this seeding effect does not eliminate the need of other VM instances to fetch missing chunks from the backing image: in the case of 32 VM instances, from−source shows an average size close to 200 MB for ac−store−inf, which amounts to a total of 6.4 GB and is well beyond the size read by a single VM instance (1.1 GB). This explains why the average throughput is much less than max. Note that this effect is also dependent on the delay: we wanted to emphasize a scenario where delays are small enough to exhibit runtime overlaps that cause concurrency issues and are non-trivial to study. Although we did not explicitly experiment with the case when such runtime overlaps are not present, we expect the seeding effect to fully take place under such circumstances: the first VM instance would read all chunks from the backing image (effectively behaving like in the case of a single isolated VM instance), while the rest would behave like in the case of max.

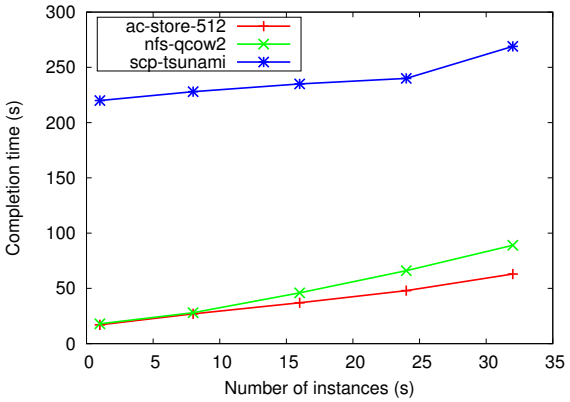
4.3.3. Sync-Disjoint: Read a different region at the same time

Our third series of experiments focuses on a collective virtual disk access pattern where all VM instances read a different region at the same time. More specifically, since the total size is 4 GB, we fix $S = 128$ and $O = k \cdot 128$, with $0 \leq k < 32$. The results are depicted in Figure 4.

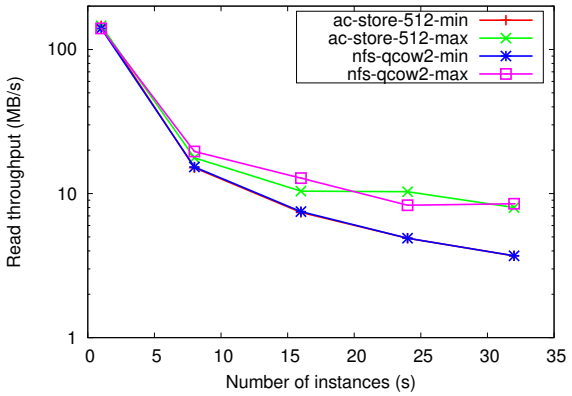
As can be observed in Figure 4(a), there is a negligible difference between the completion time for scp−tsunami and its corresponding completion time for the *Sync-Same* scenario. This can be explained by the fact that the read throughput during the experiments is high enough to make the latency of reading 128 MB instead of 1 GB negligible relative to the total completion time. A completely different situation is observable for nfs−qcow2: compared to the *Sync-Same* scenario, the completion time is much lower, because I/O contention to the NFS server is greatly reduced when all VM instances read a much smaller region. Note that the *Sync-Disjoint* pattern is the least favorable for ac−store−512, because no two VM instances access the same chunk. However, during the boot phase there is still a small set of chunks that is accessed by all VM instances, which enables exchanges of chunks to happen and ultimately leads to a reduction of completion time by 30% compared with nfs−qcow2 in the case of 32 VM instances. Compared with scp−tsunami, the reduction reaches 76% for 32 VM instances.

In terms of sustained read throughput, there is an almost perfect overlap between nfs−qcow2 and ac−store−512, both for min and max, as can be observed in Figure 4(b). This is expected, because in both cases the VM instances read the necessary chunks directly from the NFS server. Interesting to note is the difference between min and max: 3.7 MB/s vs. 8 MB/s, with an average of 4.7 MB/s. This shows that concurrent reads to a backing file stored on a NFS server can have significant fluctuations and are not served in a fair way.

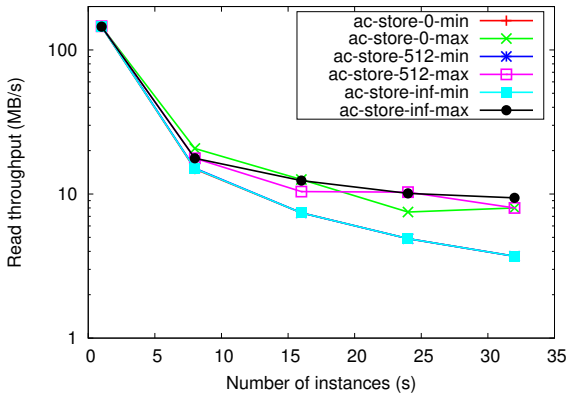
With respect to prefetching strategy, Figure 4 shows again negligible difference between all three approaches: ac−store−0, ac−store−512 and ac−store−inf. In all three cases, min and max is around 3.7 MB/s and 8.0 MB/s, with an average of 4.5 MB/s, 4.7 MB/s and 4.6 MB/s (as can be observed in Table 2). As can be noted, there is a slight difference in terms of average throughput, which can be explained by the fact that the VM instances can still benefit from chunk exchange during the boot phase. Since ac−store−0 is the worst performer, it can be concluded that prefetching has a little yet visible impact during the boot phase. However, later on, during the experiment itself prefetching does not bring any benefit and is only causing unnecessary performance overhead and bandwidth utilization, which explains the slightly less average throughput of ac−store−inf compared with ac−store−512.



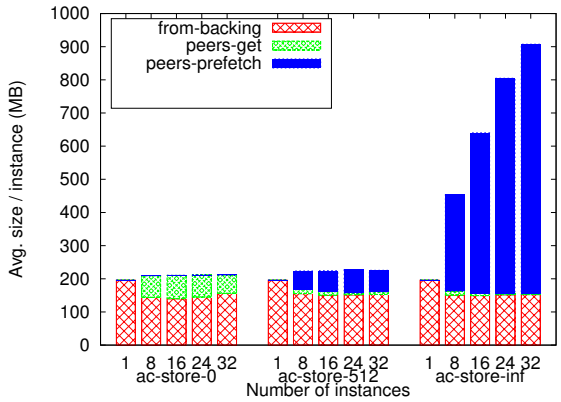
(a) Completion time for all VM instances (lower is better)



(b) Scalability of the read throughput compared with nfs−qcow2 (higher is better)



(c) Scalability of the read throughput for different prefetching strategies (higher is better)



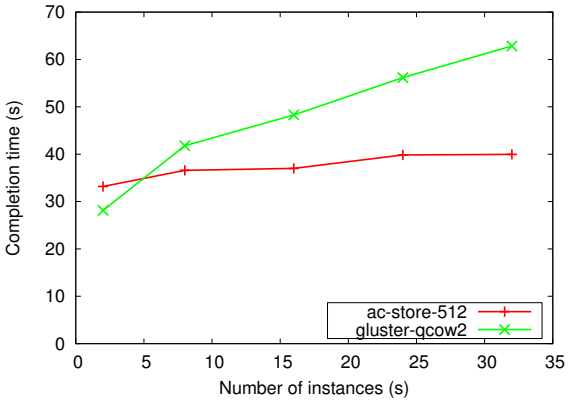
(d) Average size of chunk transfers per VM instance grouped by source

Figure 4: Results for a scenario where all VM instances are configured to read a different region of 128 MB such that there is no overlap between them. The reference repository is a centralized NFS server. Note the log-scale for the Y axis of the read throughput.

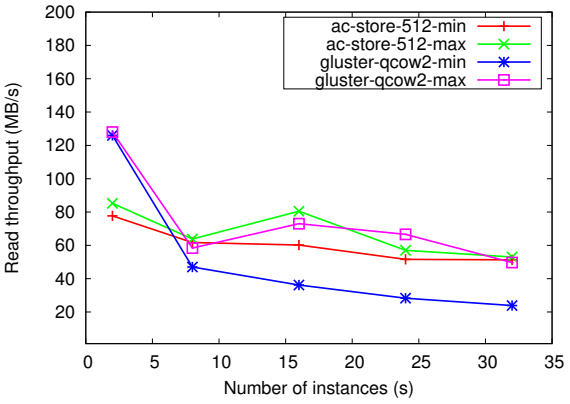
Since in this scenario all VM instances access different regions, prefetching will cause unnecessary bandwidth utilization. As mentioned in Section 4.2, studying this aspect is important. To this end, note in Figure 4(d) that the baseline where prefetching is not enabled (i.e., ac−store−0) shows an average read size per VM instance of around 200 MB, which includes the chunks read during the boot phase in addition to the 128 MB large region. This proportion is maintained almost unchanged when considering the source they were obtained from: only a small size over 128 MB is read directly from the source, while the rest of the chunks corresponding to the boot phase are mostly obtained from other VM instances (peers−get category). When using adaptive prefetching (ac−store−512), the distribution per VM instance is close to the case of ac−store−0 (amounts almost identical), showing a good adaptability potential demonstrated by the swift deactivation of prefetching once it becomes clear that it does not help. This is in sharp contrast to ac−store−inf, where prefetching generates large bandwidth utilization unnecessarily: on the average, 4.5x more chunks per VM instance are prefetched without ever being read.

4.3.4. Distributed repository deployed directly on the compute nodes

In the previous sections we have shown the benefits of our proposal using a centralized NFS server as the repository that stores the backing image. In a large datacenter and cloud infrastructure, a typical configuration would exhibit higher storage bandwidth (e.g. 10G Ethernet or higher) and a distributed repository that disperses the I/O load over a



(a) Completion time for all VM instances (lower is better)



(b) Scalability of the read throughput compared with gluster−qcow2 (higher is better)

Figure 5: Results for a scenario where all VM instances simultaneously read the same 1 GB large region of the virtual disk starting at offset 2^{30}. The reference repository is a distributed GlusterFS deployment.

number of storage nodes. Still, the number of compute nodes is typically orders of magnitude larger than the number of storage nodes, which results in an I/O behavior and bottlenecks that are similar to those observed at smaller scale using a centralized NFS repository. For this reason, in a typical large scale configuration, we expect to see similar benefits as shown in the previous sections.

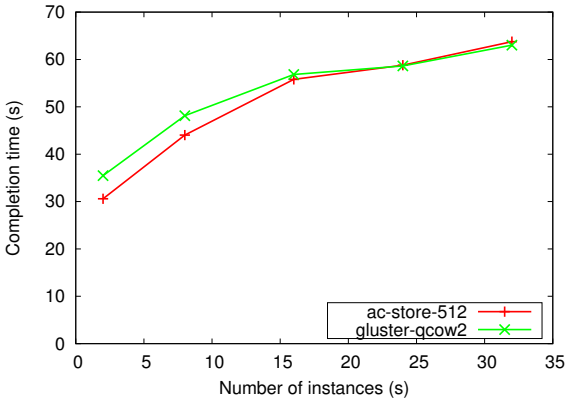
Nevertheless, it is possible to avoid the limitations of decoupled storage bandwidth contention by distributing the repository directly on the compute nodes (e.g., by leveraging local storage), which enables better scalability at the cost of potential interference with the application network traffic. For this reason, we include in this section an evaluation of our approach for the case when the repository is distributed at a scale that matches the number of VM instances.

To this end, we run experiments similar to those presented in Section 4.3.1 and Section 4.3.3. Specifically, we cover both *Sync-Same*: a favorable access pattern for which collaborative content exchange has benefits, and, respectively, *Sync-Disjoint*: an unfavorable access pattern for which collaborative content exchange introduces extra unnecessary overhead. However, this time the repository consists of a *GlusterFS* [10] deployment. More specifically, the number of GlusterFS servers matches the number of VM instances that access the backing image concurrently, while the backing image is striped on all GlusterFS servers without replication.

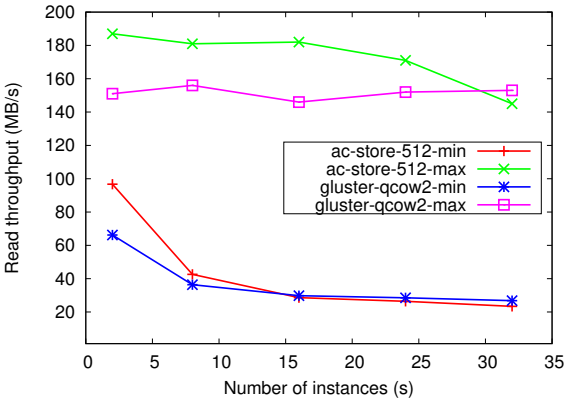
Table 3: Average sustained on-demand read throughput for 32 concurrent VM instances. The reference repository is a distributed GlusterFS deployment.

Approach	Sync-Same	Sync-Disjoint
gluster-qcow2	32.8 MB/s	50.1 MB/s
ac-store-512	53 MB/s	48.9 MB/s

With respect to the *Sync-Same* pattern, the completion time and scalability of the read throughput are depicted in Figure 5. As can be observed in Figure 5(b), the gluster−qcow2 baseline exhibits a large difference between the maximum and minimum sustained throughput for the VM instances, which increases at scale. Specifically, at 32 concurrent VM instances the maximum throughput is more than double the minimum. This result confirms the lack of I/O fairness also observed in the case of nfs−qcow2 and can be explained by the fact that a distributed repository has limited capability to alleviate the I/O contention effectively when the same chunks are accessed in the same order, because all VM instances want to read the same chunk (residing on a single storage node) simultaneously. On the other hand, our approach is effective at negating this effect thanks to the collaborative chunk exchanges. While the maximum throughput remains unchanged, the minimum throughput is very close to the maximum, which boost both the average throughput (listed in Table 3) and the overall completion time (Figure 5(a)). Specifically, the average sustained throughput is 60% higher for ac−store−512, while the completion time 40% lower.



(a) Completion time for all VM instances (lower is better)



(b) Scalability of the read throughput compared with gluster−qcow2 (higher is better)

Figure 6: Results for a scenario where all VM instances are configured to read a different region of 128 MB such that there is no overlap between them. The reference repository is a distributed GlusterFS deployment.

For the *Sync-Disjoint* pattern, the completion time and scalability of the read throughput are depicted in Figure 6. As can be observed in Figure 6(b), gluster−qcow2 starts of with a lower minimum sustained throughput compared with ac−store−512 but ends up under concurrency with a slightly higher one. With respect to the maximum throughput, it can be observed that both approaches can achieve higher than network bandwidth capacity. This happens because some regions will start with chunks that were already accessed before during the boot phase. The the maximum throughput of gluster−qcow2 is for most part lower than that of ac−store−512, because ac−store−512 managed to perform chunk exchanges between the VM instances as long as the unmatched threshold was below 512. Ultimately, the impact of this at the extreme of 32 instances is reflected for ac−store−512 in a decrease of average sustained throughput by 3% (Table 3) and a negligible increase in completion time (Figure 6(a)). Thus, it can be noted that the overhead of advertisements and collaborative exchanges is visible under unfavorable access patterns, however the negative impact is kept at a minimum thanks to the quick adaptation of our approach to the access pattern: it deactivates the chunk exchanges and leaves only very small advertisements on the network that interfere little with the repository accesses.

5. Conclusions

This work introduced a novel on-demand access technique to shared datasets under concurrency that fetches only the needed content on-the-fly, while avoiding I/O bottlenecks to the repository that holds the dataset thanks to collaborative chunk advertisement and exchanges between the VM instances. A key aspect in this context is how to leverage the advertisements in such way that it is possible to prefetch chunks before they are needed. To this end, we introduced a series of principles and algorithmic descriptions that emphasize adaptation to the access pattern in order to deliver high performance, scalability, fairness and low bandwidth utilization.

Our approach delivers high performance and scalability: for an increasing number of VM instances that concurrently access the same virtual disk it exhibits a completion time that is up to 7x faster compared with independent on-demand access to the repository and up to 5x faster compared to pre-broadcast. Furthermore, the observed average sustained read throughput is up to 10x higher when compared to independent on-demand access, with a high degree of I/O fairness that can go as low as 5% difference between the slowest and the fastest VM instance. Also, our approach is highly adaptive to the access pattern: it quickly recognizes unfavorable conditions where the VM instances access different parts of the virtual disk and the collaborative behavior is ineffective, leading to a near-optimal bandwidth utilization that is 4.5x lower than is achieved when prefetching is performed regardless of the access pattern. Furthermore, we have also shown that striping the virtual disk over a distributed repository can relieve the I/O pressure significantly for independent on-demand read access, however our approach is still able to boost the average read throughput by over 60% and the minimum throughput more than twice.

Thanks to these encouraging results, we plan to further investigate the potential benefits of collaborative chunk exchange. In particular, we did not explore how to choose and dynamically reconfigure the neighbor topology based on access pattern similarity, which is especially promising for access patterns that can be clustered. Furthermore, another direction is to explore how to extend the collaborative chunk advertisement and exchange strategy to work even when the mirrors are based on different original backing images. In this direction we reported several initial results [23]. Finally, the problem of transferring virtual disk blocks efficiently between hosts is also a key aspect of live migration [21]. Traditionally, live migrations involve a single source and destination. However, in a group of multiple related VM instances, the virtual disk blocks could be fetched from other VM instances besides the original source, which opens new opportunities for collaborative exchanges.

References

[1] File System in UserspacE (FUSE). http://fuse.sourceforge.net/.

[2] SCPTsunami. http://code.google.com/p/scp-tsunami/.

[3] The Boost C++ collection of libraries. http://www.boost.org/.

[4] Artur Andrzejak, Derrick Kondo, and Sangho Yi. Decision model for cloud computing under sla constraints. In *MASCOTS '10: Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 257–266, Washington, DC, USA, 2010. IEEE Computer Society.

[5] Netflix OSS: Asgard. https://github.com/Netflix/asgard.

[6] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the 2005 USENIX Annual Technical Conference*, pages 41–46, Anaheim, USA, 2005.

[7] Philip H. Carns, Walter B. Ligon, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[8] Han Chen, Minkyong Kim, Zhe Zhang, and Hui Lei. Empirical study of application runtime performance using on-demand streaming virtual disks in the cloud. In *MIDDLEWARE '12: Proceedings of the 13th ACM/IFIP/USENIX International Middleware Conference (Industrial Track)*, pages 5:1–5:6, Montreal, Canada, 2012.

[9] Adrian Cockcroft. Cloud native cost optimization. In *ICPE '15: The 6th ACM/SPEC International Conference on Performance Engineering*, pages 109–109, Austin, USA, 2015. ACM.

[10] Alex Davies and Alessandro Orsaria. Scale out with glusterfs. *Linux J.*, 2013(235), November 2013.

[11] Pradipta De, Manish Gupta, Manoj Soni, and Aditya Thatte. Caching vm instances for fast vm provisioning: a comparative evaluation. In *Euro-Par'12: Proceedings of the 18th international conference on Parallel Processing*, pages 325–336, Rhodes Island, Greece, 2012. Springer-Verlag.

[12] Marcel Gagné. Cooking with linux: still searching for the ultimate linux distro? *Linux J.*, (161):9, 2007.

[13] Jacob G. Hansen and Eric Jul. Scalable virtual machine storage using local disks. *SIGOPS Oper. Syst. Rev.*, 44:71–79, December 2010.

[14] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, scalable disk imaging with frisbee. In *Proc. of the 2003 USENIX Annual Technical Conference*, pages 283–296, San Antonio, USA, 2003.

[15] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 7:1–7:12, Haifa, Israel, 2009. ACM.

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

[17] Arnaud Legout, Nikitas Liogkas, Eddie Kohler, and Lixia Zhang. Clustering and sharing incentives in bittorrent systems. In *SIGMETRICS '07: Proceedings of the 2007 ACM International Conference on Measurement and Modeling of Computer Systems*, pages 301–312, San Diego, USA, 2007. ACM.

[18] Dave Levin, Katrina LaCurts, Neil Spring, and Bobby Bhattacharjee. Bittorrent is an auction: Analyzing and improving bittorrent's incentives. In *SIGCOMM '08: Proceedings of the ACM 2008 Conference on Data Communication*, pages 243–254, Seattle, USA, 2008. ACM.

[19] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[20] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel 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*, pages 147–158, San José, USA, 2011.

[21] Bogdan Nicolae and Franck Cappello. A hybrid local storage transfer scheme for live migration of I/O intensive workloads. In *HPDC '12: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 85–96, New York, USA, 2012. ACM Press.

[22] Bogdan Nicolae, Franck Cappello, and Gabriel Antoniu. Optimizing multi-deployment on clouds by means of self-adaptive prefetching. In *Euro-Par '11: 17th International Euro-Par Conference on Parallel Processing*, pages 503–513, Bordeaux, France, 2011.

[23] Bogdan Nicolae, Alexei Karve, and Andrzej Kochut. Discovering and Leveraging Content Similarity to Optimize Collective On-Demand Data Access to IaaS Cloud Storage. In *CCGrid'15: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Shenzhen, China, 2015.

[24] Bogdan Nicolae and Mustafa 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*, pages 305–316, Aachen, Germany, 2013.

[25] Kaveh Razavi and Thilo Kielmann. Scalable virtual machine deployment using vm image caches. In *SC '13: Proceedings of the 26th International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 65:1–65:12, Denver, USA, 2013. ACM.

[26] Joshua Reich, Oren Laadan, Eli Brosh, Alex Sherman, Vishal Misra, Jason Nieh, and Dan Rubenstein. Vmtorrent: Scalable p2p virtual machine streaming. In *CoNEXT '12: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, pages 289–300, Nice, France, 2012. ACM.

[27] Nicholas Aaron Robison and Thomas J. Hacker. Comparison of vm deployment methods for hpc education. In *RIIT '12: Proceedings of the 1st Annual conference on Research in Information Technology*, pages 43–48, Calgary, Canada, 2012. ACM.

[28] Matthias Schmidt, Niels Fallenbeck, Matthew Smith, and Bernd Freisleben. Efficient distribution of virtual machines for cloud computing. In *PDP '10: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 567–574, Washington, DC, USA, 2010. IEEE Computer Society.

[29] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2002. USENIX Association.

[30] Muhammad Zubair Shafiq, Alex X. Liu, and Amir R. Khakpour. Revisiting caching in content delivery networks. In *SIGMETRICS '14: The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, pages 567–568, Austin, USA, 2014. ACM.

[31] Zhiming Shen, Zhe Zhang, Andrzej Kochut, Alexei Karve, Han Chen, Minkyong Kim, Hui Lei, and Nicholas Fuller. Vmar: Optimizing i/o performance and resource utilization in the cloud. In *Middleware '13: The 15th International Middleware Conference*, pages 183–203, Beijing, China, 2013. Springer Berlin Heidelberg.

[32] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the v-system. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*, pages 2–12, New York, NY, USA, 1985. ACM.

[33] Romain Wartel, Tony Cass, Belmiro Moreira, Ewan Roche, Manuel Guijarro, Sebastien Goasguen, and Ulrich Schwickerath. Image distribution mechanisms in large scale cloud providers. In *CloudCom '10: Proceedings of the 2nd IEEE Second International Conference on Cloud Computing Technology and Science*, pages 112–117, Indianapolis, USA, 2010. IEEE Computer Society.

[34] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[35] Youhui Zhang, Yanhua Li, and Weimin Zheng. Automatic software deployment using user-level virtualization for cloud-computing. *Future Gener. Comput. Syst.*, 29(1):323–329, January 2013.