Discovering and Leveraging Content Similarity to Optimize Collective On-Demand Data Access to IaaS Cloud Storage

Bogdan Nicolae, Andrzej Kochut, Alexei Karve

Discovering and Leveraging Content Similarity to Optimize Collective On-Demand Data Access to IaaS Cloud Storage

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

Andrzej Kochut
IBM Research, USA
akochut@us.ibm.com

Alexei Karve
IBM Research, USA
karve@us.ibm.com

***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 on-demand read*, i.e., accessing the same VM image or dataset from a large number of VM instances concurrently. There are various techniques that avoid I/O contention to the storage service where the dataset is located without relying on pre-broadcast. Most such techniques employ peer-to-peer collaborative behavior where the VM instances exchange information about the content that was accessed during runtime, such that it is possible to fetch the missing data pieces directly from each other rather than the storage system. However, such techniques are often limited within a group that performs a collective read. In light of high data redundancy on large IaaS data centers and multiple users that simultaneously run VM instance groups that perform collective reads, an important opportunity arises: enabling unrelated VM instances belonging to different groups to collaborate and exchange common data in order to further reduce the I/O pressure on the storage system. This paper deals with the challenges posed by such a solution, which prompt the need for novel techniques to efficiently detect and leverage common data pieces across groups. To this end, we introduce a low-overhead fingerprint based approach that we evaluate and demonstrate to be efficient in practice for a representative scenario on dozens of nodes and a variety of group configurations.**

***Index Terms*—content similarity; deduplication; cloud storage; on-demand data access; collective I/O**

I. Introduction

One of the main features that has contributed to the growing popularity of Infrastructure-as-a-Service (IaaS) cloud computing is the elastic on-demand provisioning of resources: users can bring up a whole virtual cluster and reconfigure it dynamically with a simple click of a button. However, as the user interface grows simpler and the types of workloads diversify,achieving efficient on-demand VM provisioning is a non-trivial task.

A particularly difficult challenge in this context is the *collective on-demand read* pattern, i.e., provisioning a large number of inter-dependent VMs (e.g. part of the same virtual cluster running a large scale distributed application) that concurrently read (typically) a part of the content from the same VM (virtual machine) disk image (e.g., boot and launch applications) or from a large dataset (e.g., shared input data). This pattern is often encountered in the context of large-scale HPC (high performance computing) and data-intensive applications. Obviously, there is a need to minimize the provisioning time and guarantee scalability despite a growing number of VMs, otherwise users do not perceive IaaS as truly on-demand and lose interest, while at the same time cloud providers lose potential profit by not efficiently leveraging their computational resources.

Despite widespread need for scalable, high-performance solutions that handle the collective on-demand read pattern, IaaS cloud providers offer limited support in this regard. Most often, in an attempt to avoid any bottlenecks due to I/O contention to the storage service where the VM images and datasets are stored, it is very common to broadcast the full content to the local storage of the VM instances before allowing any read. However, 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, approaches that deliver content on-the-fly as needed in order to eliminate these two disadvantages saw increasing adoption, despite the added complexity of having to deal with the I/O contention to the storage service.

One major direction that addresses the problem of I/O contention for on-the-fly data delivery during collective reads is the use of peer-to-peer collaborative techniques. In this class of solutions, the VM instances are aware of each other's previously accessed data that is locally available and prefer to exchange the needed data among themselves rather than interact with the decoupled storage service, which risks the creation of bottlenecks due to I/O contention. Although related to pre-broadcast techniques (which are typically implemented as BitTorrent-like protocols), the focus in this context falls on how to detect and anticipate what content is actually needed during the runtime of a VM instance, in order to be able to prefetch it from the other VM instances as early as possible.

However, despite the success of such techniques to improve the performance and scalability of collective reads, most of the time they require foreknowledge about what VM instances are related and what dataset or VM image they share and read in a concurrent fashion. This is a significant limitation for large IaaS cloud datacenters where a large number of users share the infrastructure simultaneously, because there are multiple opportunities for VM instances to collaborate and exchange identical pieces of data even if they belong to different users for which the relationship between the VM instances, their access pattern and the data they are reading is unknown. This aspect is particularly important in light of several studies that confirm a large amount of redundancy among VM images, with the data duplication degree reported up to 94% [1], [2], [3].

In this paper, we focus precisely on this aspect. Our proposal envisions to organize the VM instances in a large universal group where they constantly exchange advertisements about the pieces of data (chunks) they read on-the-fly. Thus, the opportunity to exchange chunks in order to avoid I/O bandwidth contention goes beyond groups that form around individual collective reads, effectively enabling such groups that exist simultaneously to help each other out and further improve the overall scalability and performance. The key novelty in this context is how to efficiently detect and leverage such inter-group content similarity, considering the extra difficulty related to the lack of foreknowledge when operating such a large universal group.

We summarize our contributions as follows:

- We introduce a series of general principles that form a collective content exchange strategy optimized to handle the case when multiple VM instance groups simultaneously perform collective reads on potentially different datasets (Section III-A).
- We design a series of algorithms that materialize the aforementioned principles. We introduce low-overhead asynchronous techniques to identify and leverage *on-the-fly* content similarity between groups of VM instances that perform collective reads based on pre-calculated fingerprints (Sections III-B).
- We propose a hypervisor-transparent implementation as an independent FUSE module that implements our approach in userspace. Furthermore, we show how this FUSE module can be integrated in a typical IaaS architecture (Sections III-D and III-C).
- We experimentally evaluate the benefits of our approach on the Shamrock testbed by performing experiments on dozens of nodes for a representative scenario and a variety of group configurations (Section IV).

II. Related work

Content similarity detection is typically performed by means of deduplication, which is broadly classified into *static* and *content-defined*. Static approaches split the input data into equally sized chunks, which are then compared among each other (either byte-by-byte or, for increased performance, based on their hash values) in order to identify and eliminate duplicates. While simple and fast, static approaches suffer from misalignment issues (i.e insertions or deletions lead to the impossibility to detect duplicates). To deal with such misalignment issues, *content defined* approaches [4] were proposed. Essentially, they involve a sliding window over the data and that hashes the window content at each step using Rabin's fingerprinting method [5]. Many storage systems have adopted and refined deduplication techniques [6], [7], [8], [9].

Specifically in the context of IaaS clouds, deduplication techniques have proven effective at reducing the amount of space and network bandwidth necessary to store and transfer VM images. Several studies report significant reductions that can reach up to 94% [1], [2], [3]. This potential has been exploited in dedicated VM image repositories such as VMAR [10] and Squirrel [11] in order to store VM images (either fully or partially) in a deduplicated fashion. Utilizing content similarity in workloads from production storage systems is discussed in [12]. Several optimizations specifically targeting the performance of reading deduplicated data during on-demand accesses have also been recently proposed [13]. Furthermore, outside of storage, deduplication has demonstrated important benefits in the area of live migration [14], [15].

Techniques to fetch data from storage services to VM instances are broadly classified into *pre-broadcast* and *on-demand*. Pre-broadcast techniques use various scalable mechanisms (e.g., multi-cast [16], application level broadcast-trees [17] to peer-to-peer protocols [18], [19], [20]) to deliver a shared dataset from the storage service to multiple VM instances in advance, such that it can be used later without worrying about bottlenecks due to I/O bandwidth contention. However, on the downside, the broadcast can take a long time to finish and potentially delivers more content than is actually needed during runtime. On-demand techniques on the other hand eliminate both disadvantages at the cost of dealing with the I/O bandwidth contention during runtime. This approach is widely used in IaaS datacenters for virtual disk images using copy-on-write: a locally stored *QCOW2* image is instantiated from a shared backing image that is located remotely on the image store (e.g. NFS server). In an attempt to alleviate the I/O contention, various solutions ranging from decentralizing the storage (e.g. by using parallel file system [21], [22]) to using dedicated repositories [23]) and specialized prefetching techniques [24] have been proposed.

In a broader sense, collaborative caching has been explored in the MPI-IO context [25]. Our own previous work [26] explores how to improve collective reads to a shared virtual disk image by means of pushing accessed chunks among the members of the group, in an attempt to anticipate and avoid direct access to the storage service.

This paper focuses on exploiting content similarity *on-the-fly* in order to enable multiple VM instances, even if they belong to *different* dissemination groups, to collaborate, identify and exchange identical chunks of data in order to minimize the I/O pressure on the storage service under concurrency. To our best knowledge, we are the first to focus on this aspect in particular.

III. System Design

This section describes the design principles and algorithms behind our approach (Sections III-A and III-B), how to apply them in a cloud architecture (Section III-C) and finally how to efficiently implement them in practice (Section III-D).

Note that in the description of our approach, we focus on virtual disk images as the content accessed by collective reads. However, our approach can be easily adapted to handle any kind of generic unstructured dataset that can be represented as a sequence of bytes.

A. Design Principles

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 [27]. 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 appears to have already fetched and created a copy of all necessary content, however, the mirror gets populated with content only as needed during runtime. It 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 a remote source and copied locally (i.e., "mirrored"). Once all content is available locally, the read can proceed.

2) Collaborative Chunk Advertisement and Exchange: One of the major issues that plain copy-on-reference approaches face when dealing with collective reads is the I/O contention to the image store. This happens regardless of whether the store is centralized or decentralized, because the VM instances mostly follow the same I/O access pattern and thus access the same chunk simultaneously. Thus, it is important to develop techniques that reduce the I/O pressure on the image store in order to improve the performance and scalability of copy-on-reference. In this context, a natural idea is to let the mirrors collaborate and inform each other about what chunks they already posses, such that it is possible to directly fetch the chunks from other remote mirrors rather than the image store itself. To this end, we propose a peer-to-peer collaborative scheme where the mirrors advertise chunks to each other prefetch any missing chunk soon as an advertisement about it has been received, in anticipation of future read requests that will access that chunk. How to select what peer to talk to and when to decide if prefetching actually helps or just wastes network bandwidth (because no read request actually needs the chunks) is outside the scope of this work. We discuss some initial ideas related to the latter aspect in our previous work [26], where we propose a different prefetching mechanism based on pushes. With respect to the peers involved in the exchange, we opted for a circular double-linked list: we fix a predefined ordering of all mirrors in a ring and create links to the previous and next node in the ring. Thus, each peer has three potential sources from where it can fetch chunks: its neighbors in the ring and the original source of the VM image. To avoid I/O contention to the peer that advertised the chunk, we use 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) Fingerprint-Based Content-Aware Advertisements: When the VM instances need to access the same virtual disk concurrently, the process of advertising chunks to other peers is straightforward: it is enough to include information about the chunk offset in order to uniquely identify a chunk. Under such circumstances, this approach is optimal, because only a minimal information about the chunk needs to be included in the advertisement, which means advertisements spread fast and thus improve the chance that a chunk can be obtained from another peer rather than the original source. Thus, a naïve solution to deal with multiple concurrent groups of VM instances (where each group competes for a different virtual disk image) is to establish a peer-to-peer collaborative chunk advertisement and exchange scheme within each group. Again, such an approach would be optimal if the original virtual disk images are completely disjoint. However, as discussed in Section II, in practice there is a large degree of redundancy with respect to the content of virtual disk images, which means many chunks are identical despite belonging to different images and being located at different offsets. As a consequence, it is important to be able to extend the chunk advertisements and exchanges beyond the scope of a single group, especially since the potential to alleviate the I/O pressure on the image store grows with an increasing group size. Thus, a much better idea is to form a single group with all VM instances where they advertise and exchange chunks regardless of what virtual disk image they need to access. However, the problem of how to identify identical chunks is not as simple as exchanging chunk offsets anymore, because content-identical chunks might have different offsets in different images. Using a naïve solution that advertises full chunks is not feasible, as this would lead to an explosion of network traffic. To deal with this situation, we propose to use much smaller fingerprints that represent the content of the chunk and can be used for a quick comparison. Such an approach is often implemented using strong hash functions (e.g., SHA-1), for which it can be shown that the chance of collisions (i.e., obtaining the same fingerprint for two different chunks) is negligible in practice [9]. While the size of a fingerprint is significantly larger than the size of an offset (e.g., 8 bytes for a 64 bit value vs. 20 bytes for SHA-1), this increase in advertisement size does not significantly delay the propagation of advertisements (as discussed in Section IV-E).

4) Storage-Agnostic Chunk Pre-Hashing and Optimized Lookup: One of the major advantages of using a fingerprint-based advertisement scheme is the fact that duplicated content can be detected and exchanged *on-the-fly*. This aspect holds regardless of whether the image store has de-duplication support or not. If the image store does not support de-duplication (which is a common occurrence in production), then the benefits of our approach are obvious. However, even if the image store supports de-duplication and holds only unique chunks (as implemented by some related work discussed in Section II), this brings little benefit in our context, because the VM instances still need to access the same chunk multiple times if they are not aware of each other's content. On the other hand, in order to be effective, the fingerprint advertisement scheme needs to incur minimal overhead compared with a scheme that directly advertises offsets. To this end, we avoid the calculation of fingerprints during runtime in favor of a scheme that performs a pre-calculation: whenever a new virtual disk image is added to the image store, the fingerprints corresponding to each offset are calculated and stored in a separate map file. Thus, whenever a VM instance needs to mirror content locally, it first opens the map file and preloads all fingerprints in an optimized look-up table. Note that it is possible for the same chunk to appear multiple times in the same disk image. However, the data structures that enable efficient bi-directional lookup are optimized for bijections. To deal with this situation, we introduce a second uni-directional look-up table that associates each chunk offset to a unique parent offset that acts as a representative for the chunk content. Thus, whenever a read request is issued, the actual offsets are first converted into parent offsets, which in turn are associated to unique fingerprints whenever an interaction with other peers is needed.

B. Algorithms

In this section, we show how to implement the design principles discussed in the previous section using a series of algorithmic descriptions. To simplify the understanding, we assume a scenario where each mirror handles a *single* image. However, the algorithms can be easily extended in practice to implement a mirror that can share common chunks between multiple VM instances that are co-located on the same node.

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 four possible states (denoted $State$): $REMOTE$ (the chunk was not yet locally mirrored), $WAIT$ (the peer has asked another for the chunk and is waiting for the reply), $LOCAL$ (the chunk was successfully prefetched and mirrored locally, but was not yet read) and $READ$ (the chunk was needed by a read operation).

The READ operation is detailed in Algorithm 1. In a nutshell, it ensures that all chunks that cover the range $offset, size$ from $Mirror$ are locally available, after which it redirects the read request to the local mirror. More specifically, first the parent offset p of a chunk is calculated. This information is maintained in the $ParentOffset$ uni-directional data structure. Once p is known, any reference to the original chunk is equivalent to a reference to the chunk corresponding to p. Thus, if p 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 image store. Reverting to the image store is not immediate: the read request for the chunk is accumulated in the $Original$ 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 peer that advertises it (all advertisements for a chunk are accumulated in the $Source$ set). If such a peer (denoted $Peer$) exists, then READ picks the one that is the least loaded (i.e., it has a minimal number of pending requests it needs to answer to), fetches the chunk from it and finally advertises the chunk to the neighbors. The advertisement is based on the fingerprint of the chunk p, which is pre-loaded in the bi-directional look-up table $Fingerprint$. If no neighboring peer holds the chunk, then it reverts to the image store (i.e. the chunk is added to $Original$). In either case, the state of the chunk becomes $READ$. Once all chunks are processed, the ones scheduled to be read from the image store ($Original$ set) are finally fetched, mirrored locally and advertised 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.

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

```
1:  function READ($buffer, offset, size$)
2:      $Original \leftarrow \emptyset$
3:      for all $chunk \in Image$ such that $chunk \cap (offset, size) \neq \emptyset$ do in parallel
4:          $p \leftarrow ParentOffset[chunk]$
5:          if $State[p] = WAIT$ then
6:              wait until $State[p] = LOCAL$
7:              if timeout then
8:                  $Original \leftarrow Original \cup \{p\}$
9:              end if
10:             $State[p] \leftarrow READ$
11:         else if $State[p] = REMOTE$ then
12:             if $Source[p] \neq \emptyset$ then
13:                 select least loaded $Peer \in Source[p]$
14:                 fetch chunk $p$ from $Peer$ and mirror it
15:                 advertise $Fingerprint[p]$ to neighbors
16:             else
17:                 $Original \leftarrow Original \cup \{p\}$
18:             end if
19:             $State[p] \leftarrow READ$
20:         else if $State[p] = LOCAL$ then
21:             $State[p] \leftarrow READ$
22:         end if
23:     end for
24:     for all $p \in Original$ do in parallel
25:         fetch chunk $p$ from repository and mirror it locally
26:         advertise $Fingerprint[p]$ to neighbors
27:     end for
28:     return read $(offset, size)$ into $buffer$ from $Mirror$
29: end function
```

The collaborative chunk advertisement and exchange scheme is performed asynchronously by BACKGROUND_EXCHANGE, detailed in Algorithm 2. In a nutshell, it listens for advertisements about new chunks from all its neighbors and whenever it receives one, first it performs a reverse look-up in the $Fingerprint$ map to obtain the parent offset p for the chunk fingerprint referred to by msg. Then, it adds the originating $Peer$ to the corresponding

Algorithm 2 Collaborative chunk advertisement and exchange with other peers

```
procedure BACKGROUND_EXCHANGE
    while true do
        $msg \leftarrow$ listen for any message from any peer
        $hash \leftarrow$ extract chunk fingerprint from $msg$
        $p \leftarrow$ find parent $p$ for $hash$ in $Fingerprint$
        if $msg$ = fetch reply and $State[p] = WAIT$ then
            mirror chunk $p$ locally
            $State[p] \leftarrow LOCAL$
            advertise $Fingerprint[p]$ to neighbors
        end if
        if $msg$ = fetch request then
            send $chunk$ to requester
        end if
        if $msg = advertisement$ from $Peer$ then
            $Source[p] \leftarrow Source[p] \cup \{Peer\}$
            if $State[p] = REMOTE$ then
                $State[p] = WAIT$
                ask $Peer$ to send chunk $p$
            end if
        end if
    end while
end procedure
```

$Source[p]$ set. If the payload received from any peer is a request for a chunk, it will send the chunk as a reply if the chunk is actually mirrored 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 and the state will be updated to $LOCAL$. Furthermore, the chunk is advertised asynchronously to all neighbors.

Note that the local mirror also supports write operations: once a chunk is overwritten, it is invalidated and cannot be used in the collaborative chunk exchange any longer (i.e., it is not advertised anymore to other peers and any remote peer trying to fetch it will fail). However, this mode limits the potential to exchange chunks under write-intensive scenarios, which is why the mirror is typically used as read-only backing file for higher level copy-on-write layers (e.g., *QCOW2*).

C. 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 store* is the storage service responsible to hold the VM disk images that are accessed concurrently during collective reads. The only requirement for the image store 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 [23], [28]. In particular, solutions that de-duplicate VM image content [10] are well complemented by our approach, as discussed in Section III-A.

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 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 advertisement and exchange protocol described in Section III-A to pre-populate regions that are likely to be accessed in the future based on the collective access pattern.

D. Implementation

We implemented the mirroring module as file system in userspace on top of FUSE (File system in USerspacE). This has two advantages in our context: (1) it is transparent to the hypervisor (and thus portable); (2) it enables easy interface to the image store (regardless of the access interface) as well as easy prototyping of the collaborative chunk advertisement and exchange protocol, because all communication between the peers happens in userspace.

The collaborative chunk advertisement and exchange strategy runs in its own thread, which synchronizes with the main FUSE thread through the data structures presented in Section III-B. We rely on the Boost C++ collection of libraries for the implementation of high performance bi-directional maps and look-up tables used for the translation between parent offsets and fingerprints. The fingerprints themselves can be calculated using any strong hash function, however, for the purpose of this work we opted for *SHA-1*. Note that the mirroring module only uses the fingerprints and never calculates them (which is the responsibility of additional external tools or the image store itself if it has de-duplication support). We implemented our own external tool that pre-hashes a virtual disk image using a configurable chunk size based on the SHA-1 implementation provided by *OpenSSL*.

The communication between the mirroring modules is implemented on top of Boost ASIO, a high performance asynchronous event-driven library which is part of the standard Boost package. 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. Given the sensitivity of our approach to the timely dissemination of advertisements, especially when considering the extra overhead

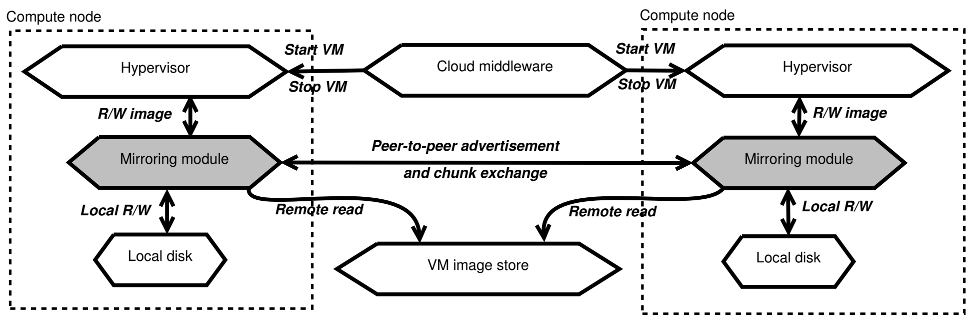


Fig. 1. Cloud architecture that integrates our approach (dark background)

of sending SHA-1 fingerprints, the choice of using UDP has much more benefits compared to the drawbacks of unreliability (which is quite low within a datacenter).

IV. Evaluation

This section evaluates the benefits of our approach experimentally based on a real-life scenario often encountered in practice.

A. *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* 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, whose backing image is stored on a NFS server that is accessible through the Gigabit Ethernet link. The format of the image is RAW and its total size is 4 GB. Each VM instance is booted from a locally-derived QCOW2 image. 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=writeback).

B. *Workload*

As a motivating scenario for our evaluation we choose a setting where users need to instantiate a large number of VMs and configure them on-the-fly using a shared repository. Often, such a repository includes software packages and configuration files. In order to save space, software packages are often compressed. Thus, a common operation is to read the compressed stream of bytes from the repository and unpack its content into the local storage of the VM instance.

The repository is abstracted as a read-only virtual disk that is formatted using the *ext4* file system to hold a series of compressed software packages. This virtual disk is mounted after boot and used to install all necessary components. Since the content of the repository is delivered on-the-fly only when needed (i.e. the virtual disk corresponding to the repository is not pre-copied), users typically consolidate all their software packages in a single large repository from which each VM instance can cherry-pick the needed components. Users prefer this approach to other on-demand solutions (e.g. setting up an extra VM instance as a FTP server), both because it is easier to setup (i.e., no extra overhead to configure a FTP server) and because it is more cost-effective (i.e., no extra dedicated VM needs to act as the repository).

TABLE I
VIRTUAL DISK REPOSITORY COMPOSITION

Package	Size	Repo-1	Repo-2
hadoop-2.5.1.tar.gz	142 MB		✓
jdk-8u25-linux-x64.tar.gz	154 MB		✓
Netw_DB2_Info_Ctr_V10.5.tar.gz	767 MB	✓	✓
TXSERIES_V8.1_EIMAGE.tar	387 MB		✓
WebSphere-C1FZ6ML.tar.gz	782 MB	✓	

However, due to the popularity of some software packages, the repositories compiled by the users often overlap to a large degree. To capture this aspect in our evaluation, we compiled two repositories as RAW virtual disks of 2 GB that include the content summarized in Table I. This content is representative of two users that run business analytics workloads at large scale using either Hadoop and DB2 or WebSphere and DB2. For the rest of this paper, we refer to the two repositories as *Repo-1* and *Repo-2*.

For the purpose of this work, both *Repo-1* and *Repo-2* are stored on the same NFS server where the OS image of the VM instances is stored. The experiments consist in booting a number of VM instances (up to 32, each on a dedicated physical node), a part of which mounts *Repo-1*, while the rest mounts *Repo-2*. This corresponds to two different users, each with its own repository, that need to simultaneously instantiate a set of VM instances. To emphasize the importance of detecting and leveraging content duplication during on-the-fly concurrent dissemination of data, we assume all VM instances run the same workload: unzip the common *Netw_DB2_Info_Ctr_V10.5.tar.gz* package, which is part of a DB2 on-the-fly installation.

Note that the nature of the data accessed by the workload (gzip archive) does not exhibit any internal redundancy. This is ideal in our context, because the lack of intra-repository duplication excludes from the measurements any benefits that could result from fetching unique chunks only once within the same group. Thus, the benefits observed in the experiments are exclusively the result of the ability to exchange the chunks between the groups. However, when intra-duplication is present, we anticipate even greater benefits for our approach.

C. Methodology

We compare three approaches throughout our evaluation:

a) Direct on-demand access: In this setting, the repository image residing on the NFS server is directly attached as a read-only snapshot to its corresponding VM instances. In this setting, the VM instances operate in complete isolation and read in parallel all necessary content directly from the NFS server, which greatly simplifies the setup, but creates a high degree of I/O contention. Such an approach is widely used in production and relevant as a baseline for comparison. We denote this approach nfs−direct.

b) Collaborative on-demand offset-identical chunk exchange: In this setting, the repository image residing on the NFS server is mirrored on-demand on the local storage of the node that hosts the VM instance. The mirror is aware of the other concurrently running VM instances that share the same repository image and advertises the accessed chunks to them. At the same time, it reacts to the advertisements received from other VM instances by prefetching any missing chunks from them in order to avoid accessing the NFS server if those missing chunks are needed in the future. Since the chunks refer to the same repository image, any exchange between the VM instances involves only a minimal amount of information about the chunk: it's offset. However, there are no exchanges between VM instances that do not share the same repository image. This approach aims to implement a solution that performs optimally in the case when all VM instances are aware of the information they are sharing. We denote this approach as collab−simple.

c) Collaborative on-demand content-aware chunk exchange: Similar to the previous setting, the repository image residing on the NFS server is mirrored on-demand on the local storage of the node that hosts the VM instance. However, the mirror implements our approach and is aware of any other VM instances, exchanging advertisements with them regardless whether they share the same repository image or not. In this case, the advertisements are larger compared to the previous case, because they include the SHA-1 fingerprint of the chunk as payload rather than the offset (20 bytes vs. 8 bytes). Furthermore, there is extra overhead needed to translate from the fingerprint to the chunk offset and back. However, other than these differences, every other aspect is identical to the previous setting. We denote this approach collab−dedup.

For both collab−simple and collab−dedup, the chunk size is fixed at 32 KB, which results in a total of 65536 chunks for both *Repo-1* and *Repo-2*, out of which 24559 are accessed by the workload. In the case of collab−dedup, all chunks belonging to *Repo-1* and *Repo-2* and pre-hashed using SHA-1 and indexed in a corresponding fingerprint file before the experiment begins.

These approaches are compared based on the following metrics:

Completion time: This is the total time required to run the workload. We report average values per VM instance, as well as extremes (i.e. time for all VM instances to complete), both of which are directly relevant for the user, because in many cases it is necessary to wait for all VM instances to finish the initialization step before they can be used. A low value indicates better overall performance.

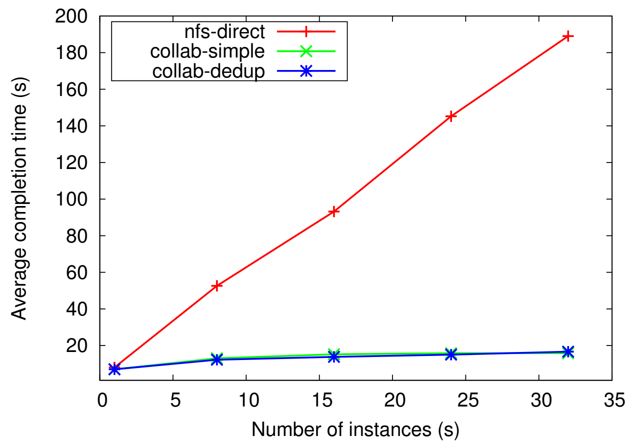
Source and amount of network traffic: This is the average received network traffic generated by on-demand reads of the package during the workload. It is broken down by source, indicating the amount read from the NFS server, the amount pre-fetched from other VM instances in anticipation of read requests and the amount fetched on-demand from other VM instances as a direct consequence of read requests. This information is important from the IaaS cloud provider perspective, because it shows how much the I/O pressure on the NFS server can be reduced thanks to collaborative chunk exchanges.

D. Scalability and performance

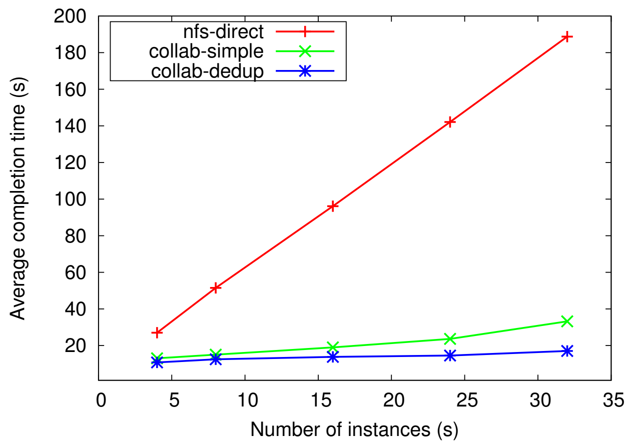
Our first series of experiments aims to study the weak scalability of all three approaches under a variable distribution of the VMs that access either *Repo-1* or *Repo-2*. To this end, we fix three representative configurations: (1) all VMs access only *Repo-1*; (2) 25% of the VMs access *Repo-1*, while the rest access *Repo-2*; and (3) half of the VMs access *Repo-1*, while the other half access *Repo-2*. For each configuration, we gradually increase the number of concurrent VMs that run the workload described in Section IV-B. To emphasize the I/O pressure on the NFS server due to contention, we start all VM instances simultaneously.

The average completion time for all three configurations is depicted in Figure 2(a), Figure 2(b) and Figure 2(c) respectively. As expected, with an increasing number of VMs, the average completion time is increasing, because more VMs compete for the same chunks. In the case of nfs−direct, the only source to fetch the chunks from is the NFS-server. Thus, all VM instances compete for the I/O bandwidth of the NFS server, regardless of the distribution between *Repo-1* and *Repo-2*. This is directly observable in the shape of the curves corresponding to nfs−direct in all three figures: there is an almost perfect overlap between them. At the extreme of 32 VM instances, the average completion time for nfs−direct is around 189s in all three cases, which amounts to a low throughput of around 4 MB/s.

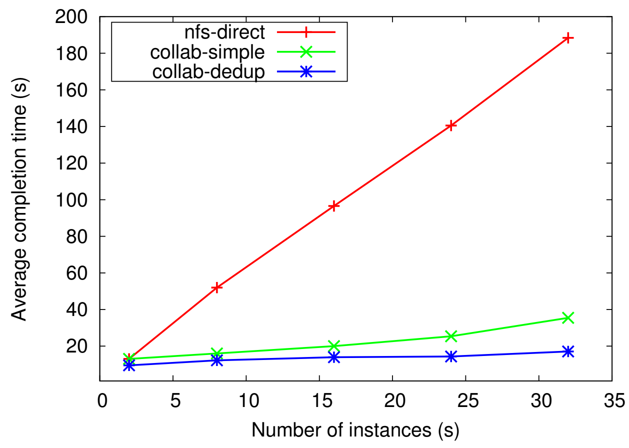
In the case of collab−simple and collab−dedup, the slope of the curves is more gentle, showing a major improvement in scalability. This is especially visible for the case when all VM instances access *Repo-1* (Figure 2(a)): the average completion time remains almost constant for both approaches with only little difference between them (15.85s for collab−simple and 16.66s for collab−dedup). Compared with nfs−direct at the extreme of 32 VM instances, the average completion time



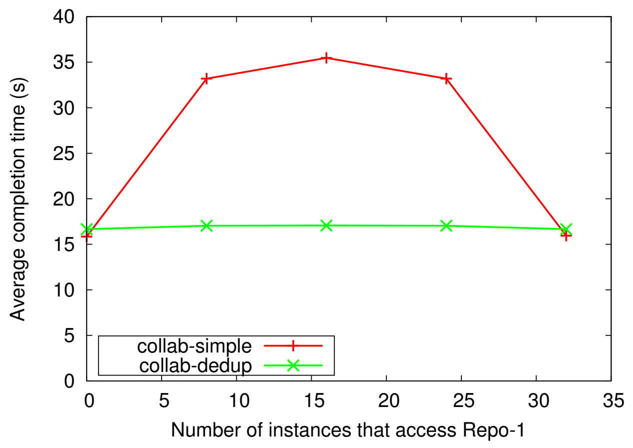
(a) Weak scalability when all VMs access *Repo-1*



(b) Weak scalability when 25% of VMs access *Repo-1* and 75% of VMs access *Repo-2*



(c) Weak scalability when 50% of VMs access *Repo-1* and 50% of VMs access *Repo-2*



(d) Performance of 32 VMs when gradually increasing the number of VMs that access *Repo-1*

Fig. 2. Study of scalability and performance for a set of concurrent VMs of which each is mounting one of two different virtual disk repositories but access identical overlapping content (unzip a compressed software package present in both repositories). The size of the compressed software package is 762 MB.

was reduced by more than 91% in both cases. Interesting to note is also the low overhead of our approach compared with collab−simple: exchanging more information about each chunk (fingerprint vs. offset) and translating offsets to fingerprints in both directions has a minimal performance impact (up to 4%) even when only a single group of VM instances that share the same repository is present (which favors the collab−simple approach).

However, when the VM instances access both *Repo-1* and *Repo-2*, a large gap between collab−simple and collab−dedup starts to become visible: at the extreme of 32 VM instances, collab−dedup is on the average 2.05x faster than in the case when 50% of the VMs access *Repo-1* (Figure 2(c)). These results are easy to understand: in the case of collab−simple each of the VM groups corresponding to *Repo-1* and *Repo-2* act in isolation and advertise the chunks only among themselves. Thus, each chunk is fetched at least twice from the NFS server by at least one of the members of each group. In the case of collab−dedup, all VMs act as a single group and advertise the same content-identical chunks among themselves, effectively halving the I/O pressure on the NFS server. When 25% of the VMs access *Repo-1* (Figure 2(b)), collab−dedup is 1.88x faster than collab−simple. This can be explained by the fact that collab−simple can propagate the chunks much better in the significantly larger group corresponding to *Repo-2*.

To verify this effect for symmetry, we ran two extra experiments: (1) 25% of the VMs access *Repo-2* and the rest *Repo-1*; and (2) all VMs access *Repo-2*. Armed with these two new experiments, we depict in Figure 2(d) a comparison between collab−dedup and collab−simple for a fixed number of 32 VM instances with a gradual increase in the number of VMs accessing *Repo-1* from 0 to 32. As expected, the maximal difference is when the two VM groups are equally sized, with little difference observable when all VMs belong to the same group.

Furthermore, as explained in Section IV-C, besides the aver-

TABLE II
ZOOM ON COMPLETION TIMES FOR 32 VM INSTANCES

Approach	# of *Repo-1*	Min	Avg.	Max
nfs−direct	8	131.0s	188.75s	221.0s
	16	129.0s	188.41s	221.0s
	32	112.0s	189.03s	221.0s
collab−simple	8	32.0s	33.18s	34.0s
	16	35.0s	35.47s	36.0s
	32	14.05s	15.85s	16.05s
collab−dedup	8	17.0s	17.03s	18.03s
	16	16.0s	17.07s	18.08s
	32	16.0s	16.66s	17.01s

age completion time per instance, an important aspect to study is also the extremes of the completion time (i.e. corresponding to the fastest and the slowest instance). To this end, we zoom on the completion times for the maximal deployment of 32 VM instances in Table II. As can be observed, in the case of collab−simple and collab−dedup, there is little variation between the average and the minimum or maximum: no more than 6% is observable. However, the same does not hold for nfs−direct: the slowest VM instance has a completion time that is almost twice as high as the fastest. This effect can be traced back to the high degree of contention, which causes read requests not to be served in a fair fashion, resulting in uneven delays. Thus, it can be noted that the collaborative chunk exchange strategy not only leads to a dramatic improvement in the scalability and performance, but also in terms of I/O fairness.

E. Network traffic

In this section we study the network traffic generated during the experiments presented in the previous section. We focus on the average amount of content received from remote sources in the case of 32 concurrent VM instances, broken down by source and type. More specifically, we are interested in understanding how much of the content is fetched from the NFS server and how much of the content is fetched from the other peers (in the case of collab−simple and collab−dedup). Furthermore, for the content fetched from the other peers (i.e. chunks of 32 KB), we make a distinction between the chunks that were advertised and pre-fetched before they were needed (peers−prefetch) and those chunks that were advertised but were fetched as a consequence of a read request (peers−get).

The results are depicted in Figure 3. All three approaches fetch the same total amount of information, with nfs−direct exclusively accessing the NFS server. However, in the case of collab−simple and collab−dedup, the majority of the chunks are obtained by a VM instance from other peers rather than the NFS server. This explains why a reduction of 91% in average completion time in the case when all VM instances access *Repo-1* is possible for both collab−simple and collab−dedup when compared with nfs−direct. Particularly interesting to note is the difference between collab−simple and collab−dedup: despite both accessing only *Repo-1*, collab−simple manages to prefetch a slightly larger amount of chunks than collab−dedup, because it receives the adver-

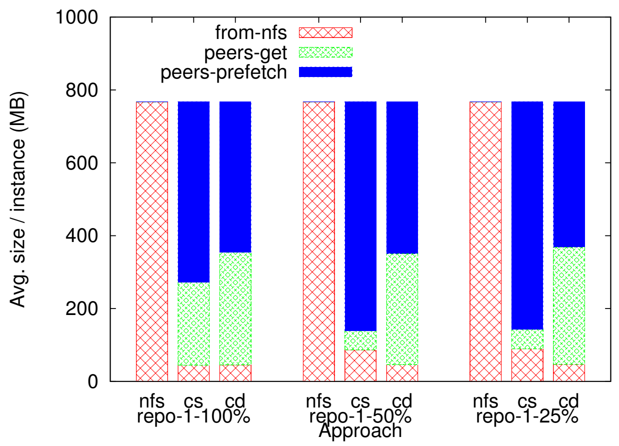


Fig. 3. Network traffic broken down by source and type for 32 VM instances and different group sizes corresponding to *Repo-1* and *Repo-2*. Abbreviations are used for the three approaches as follows: nfs stands for nfs−direct, cs stands for collab−simple and cd stands for collab−dedup.

tisements faster, which enables it to avoid waiting during read calls. However, even if collab−dedup needs to wait for more chunks to be fetched during read calls, the delay is minimal, because it still receives advertisements in time to be able to fetch the missing content from other peers, which limits the performance overhead to less than 4% (as shown in the previous section). This effect is directly observable when comparing the amount of chunks fetched from the NFS server: it remains the same in both cases.

However, when moving to the case in which the number of VM instances that access *Repo-1* is 50% and 25%, the interactions with the NFS server double for collab−simple when compared to collab−dedup. This confirms the speed-up of up to 2x for collab−dedup over collab−simple discussed in the previous section. Interesting to note in these two cases is the fact that collab−simple manages to prefetch an even higher amount of chunks than in the previous case, with only a small amount of chunks fetched on-demand from other peers. This is explained by the fact that the increasing I/O pressure on the NFS server introduces more latencies that can be exploited to finish more prefetch operations. However, this achievement is overshadowed by the need to access twice as many chunks from the NFS server. By contrast, collab−dedup maintains a constant proportion of accesses to the NFS server, which remains the same across all three cases.

V. CONCLUSIONS

In this paper we have proposed a novel approach to deal with unrelated groups of VM instances that perform collective on-demand read access patterns simultaneously. Our proposal is based on the idea of detecting and exchanging identical content on-the-fly both inside and outside of the groups, which reduces the overall I/O pressure on the cloud storage system to a higher degree than what techniques designed to deal with individual groups can achieve.

To demonstrate the benefits of our approach, we ran extensive experiments using a representative scenario for variable group compositions and multiple concurrency configurations. Our key findings are as follows: we observed a speedup of completion time up to 11x compared with a naïve solution that directly reads from the cloud storage for each VM instance individually. Also, we observed a speed-up of 1.88x-2x compared with collaborative schemes optimized for individual groups. The maximum speedup is reached when the groups are of equal size. Even when there is a single group, our approach exhibits a low performance overhead (4%) compared with an approach that is specifically optimized for a single group. Furthermore, we show excellent scalability both in terms of performance and consumption of the I/O bandwidth of the cloud storage (negligible growth when increasing the number of VM instances).

In future work we plan to explore several promising avenues. First, we did not explore what happens when the groups are not operating simultaneously and/or access common content at different times. How to leverage and anticipate such de-synchronizations can provide further potential for improvement. Second, our approach treats all chunks individually, both in terms of advertisements and exchanges. Thus, it would be interesting to understand and exploit correlations between chunks (e.g., what clusters of chunks go together and could be advertised and prefetched as a group).

REFERENCES

[1] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09. Haifa, Israel: ACM, 2009, pp. 7:1–7:12.

[2] K. R. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei, "An empirical analysis of similarity in virtual machine images," in *Middleware '11: Proceedings of the Middleware 2011 Industry Track Workshop*. Lisbon, Portugal: ACM, 2011, pp. 6:1–6:6.

[3] R. Schwarzkopf, M. Schmidt, M. Rüdiger, and B. Freisleben, "Efficient storage of virtual machine images," in *ScienceCloud '12: Proceedings of the 3rd Workshop on Scientific Cloud Computing Date*. Delft, The Netherlands: ACM, 2012, pp. 51–60.

[4] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, Oct. 2001.

[5] M. Rabin, "Fingerprinting by random polynomials," Center for Research in Computing Technology, Harvard University, Tech. Rep. TR-CSE-03-01, 1981.

[6] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*. San Jose, USA: USENIX Association, 2008, pp. 18:1–18:14.

[7] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: a scalable secondary storage," in *FAST '09: Proccedings of the 7th conference on File and storage technologies*. San Francisco, USA: USENIX Association, 2009, pp. 197–210.

[8] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *USENIXATC'11: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. Portland, USA: USENIX Association, 2011, pp. 25–39.

[9] B. Nicolae, "Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal," in *IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium*, Boston, USA, 2013, pp. 19–28.

[10] Z. Shen, Z. Zhang, A. Kochut, A. Karve, H. Chen, M. Kim, H. Lei, and N. Fuller, "Vmar: Optimizing i/o performance and resource utilization in the cloud," in *Middleware '13: Proceedings of the 14th ACM/IFIP/USENIX International Middleware Conference*, vol. 8275. Springer Berlin Heidelberg, 2013, pp. 183–203.

[11] K. Razavi, A. Ion, and T. Kielmann, "Squirrel: Scatter hoarding vm image contents on iaas compute nodes," in *HPDC '14: The 23rd ACM International Symposium on High-performance Parallel and Distributed Computing*. Vancouver, Canada: ACM, 2014, pp. 265–278.

[12] R. Koller and R. Raju, "I/o deduplication: Utilizing content similarity to improve i/o performance," in *FAST '10: Proceedings of the USENIX File and Storage Technologies*. USENIX Association, 2010, pp. 211–224.

[13] B. Mao, H. Jiang, S. Wu, Y. Fu, and L. Tian, "Read-performance optimization for deduplication-based storage systems in the cloud," *Trans. Storage*, vol. 10, no. 2, pp. 6:1–6:22, Mar. 2014.

[14] U. Deshpande, X. Wang, and K. Gopalan, "Live gang migration of virtual machines," in *HPDC '11: Proceedings of the 20th International Symposium on High Performance Distributed Computing*. San Jose, USA: ACM, 2011, pp. 135–146.

[15] S. Al-Kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu, "Vmflock: Virtual machine co-migration for the cloud," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. San Jose, USA: ACM, 2011, pp. 159–170.

[16] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb, "Fast, scalable disk imaging with frisbee," in *Proc. of the 2003 USENIX Annual Technical Conference*, San Antonio, USA, 2003, pp. 283–296.

[17] "SCPTsunami," http://code.google.com/p/scp-tsunami/.

[18] M. Schmidt, N. Fallenbeck, M. Smith, and B. 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*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 567–574.

[19] R. Wartel, T. Cass, B. Moreira, E. Roche, M. Guijarro, S. Goasguen, and U. 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*. Indianapolis, USA: IEEE Computer Society, 2010, pp. 112–117.

[20] J. Reich, O. Laadan, E. Brosh, A. Sherman, V. Misra, J. Nieh, and D. Rubenstein, "Vmtorrent: Scalable p2p virtual machine streaming," in *CoNEXT '12: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. Nice, France: ACM, 2012, pp. 289–300.

[21] F. Schmuck and R. 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: USENIX Association, 2002.

[22] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320.

[23] 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.

[24] B. Nicolae, F. Cappello, and G. Antoniu, "Optimizing multi-deployment on clouds by means of self-adaptive prefetching," in *Euro-Par '11: 17th International Euro-Par Conference on Parallel Processing*, Bordeaux, France, 2011, pp. 503–513.

[25] A. Nisar, W.-k. Liao, and A. Choudhary, "Scaling parallel i/o performance through i/o delegate and caching system," in *SC '08: Proceedings of the 20th ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, Austin, Texas, 2008, pp. 9:1–9:12.

[26] 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.

[27] M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemptable remote execution facilities for the v-system," in *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 1985, pp. 2–12.

[28] J. G. Hansen and E. Jul, "Scalable virtual machine storage using local disks," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 71–79, December 2010.