Leveraging Collaborative Content Exchange for On-Demand VM Multi-Deployments in IaaS Clouds

Bogdan Nicolae, Mustafa Rafique

Leveraging Collaborative Content Exchange for On-Demand VM Multi-Deployments in IaaS Clouds

Bogdan Nicolae and M. Mustafa Rafique

IBM Research, Dublin, Ireland
{bogdan.nicolae, mustafa.rafique}@ie.ibm.com

Abstract. A critical feature of IaaS cloud computing is the ability to deploy, boot and terminate large groups of inter-dependent VMs very quickly, which enables users to efficiently exploit the on-demand nature and elasticity of clouds even for large-scale deployments. A common pattern in this context is multi-deployment, i.e., using the same VM image template to instantiate a large number of VMs in parallel. A difficult trade-off arises in this context: access the content of the template on-demand but slowly due to I/O bottlenecks or pre-broadcast the full contents of the template on the local storage of the hosting nodes to avoid such bottlenecks. Unlike previous approaches that are biased towards either of the extremes, we propose a scheme that augments on-demand access through a collaborative scheme in which the VMs aim to leverage the similarity of access pattern in order to anticipate future accesses and exchange chunks between themselves in an attempt to reduce contention to the remote storage where the VM image template is stored. Large scale experiments show improvements in read throughput between 30%-40% compared to on-demand access schemes that perform in isolation.

1 Introduction

Infrastructure-as-a-Service (IaaS) cloud computing has matured over the years up to the point where it represents a potentially cost-effective solution with low entry barrier even for workloads that require huge amounts of computational resources, such as large scale scientific or data-intensive computations.

One of the main features that has contributed to the growing popularity of IaaS is the elastic on-demand provisioning of virtual machines (VMs). Users can bring up a whole virtual cluster and reconfigure it dynamically with a simple click of a button [1]. However, as the user interface grows simpler and the types of workloads diversify [2], achieving efficient on-demand VM provisioning is a non-trivial task.

A particularly difficult challenge in this context is the *multi-deployment* pattern, i.e., provisioning a large number of inter-dependent VMs concurrently from the same VM image template, which is often needed to deploy large-scale HPC and data-intensive applications. Obviously, there is a need 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. This issue is especially important in the context of spot instances [3]: users can bid for idle cloud

resources at lower than regular prices, however with the risk of their VM being terminated at any moment without notice when other users bid higher. Given such a context, long provisioning time is not only inconvenient for the user, but actually leads to loss of computational time and resources that could have been otherwise leveraged at a low price.

Despite widespread need for multi-deployments, little effort has been undertaken to improve their scalability. Current techniques often pre-copy the full VM image locally on the compute nodes before launching the VM instances, which can take in the order of tens of minutes or even hours [4], not counting the time to boot the guest operating system and deploy the application itself. Although on-demand techniques have matured (e.g., locally derived copy-on-write images [5] that use a remotely stored VM image template as a backing file) and they have been shown to generate little overhead on application performance compared to the case when a local copy is available [6], they saw comparatively little attention for multi-deployments due to the fact that they generate I/O contention to the repository where the VM image template is stored.

This paper contributes with a novel technique that aims to alleviate the aforementioned issue and improve the scalability of multi-deployments by enabling efficient decentralized on-demand access that avoids bottlenecks caused by competition to the repository. To achieve this, we leverage the fact that the VMs of the group typically have highly similar access patterns (e.g., during the boot phase they access the same chunks of the virtual disk in the same order [7]) in order to build a collaborative scheme where VMs exchange chunks between themselves in anticipation of expensive concurrent I/O accesses to the remote repository that would follow, which thus can be avoided. Our approach can dynamically adapt to the access pattern: it increases the rate of exchanges when remote accesses were successfully avoided, and backs off when the success rate starts dropping due to diverging access patterns. We summarize our contributions as follows:

- We introduce a collective content exchange scheme that optimizes the multi deployment pattern by enabling efficient sharing of virtual disk image templates in an on-demand fashion and show how to integrate this approach in a typical IaaS architecture (Sections 3.1 and 3.3).
- 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. This is functionally equivalent to the broadcast technique but completely removes the broadcast overhead (Section 3.4).
- We experimentally evaluate the benefits of our approach on the Grid5000 [8] testbed by performing multi-deployments on dozens of nodes (Section 4).

2 Related Work

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

Pre-broadcast techniques fully copy the VM disk image template 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 an 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 [9] and application level broadcast-trees [10] to peer-to-peer protocols [11, 4].

At the other extreme are on-demand access approaches: VMs are instantiated on-the-fly by keeping the remote virtual disk image template read-only and storing all modifications locally using copy-on-write, which is natively supported by many hypervisors [5]. While this eliminates the broadcast step, it introduces I/O competition between VM instances because they share a single disk content source. Obviously, a centralized repository generates the highest contention, but is still a very popular choice due to simplicity [12]. Using a decentralized storage solution (such as a parallel file system [13–15] or a dedicated repository [16]) reduces contention thanks to striping, but is only partially effective in our case, because the VM instances often access the same chunks in the same order. In our previous work [7], 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 multi-deployment is user-level virtualization [17]. The idea here is to use a minimally configured OS and virtual disk on top of which application packages and configuration files are applied on-the-fly during boot time. In this context, the same content propagation principles that apply at low level (i.e., virtual disk chunks) can be used for higher level contents (i.e., packages and configuration files).

Finally, there are several ways to complement multi-deployments with additional optimizations. A straightforward optimization is to use the local storage available on the compute nodes as a caching layer for VM images [18]. While this does not improve first-time deployments, given the dynamicity of the cloud, many VM images pass through the same compute node during its lifetime, effectively increasing the chance to avoid a first time accesses for certain members of the multi-deployment group. 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 [19] an effective tool to leverage local storage more efficiently.

Our own approach tries to leverage the best trade-off between VM disk content broadcast and on-demand access. Much like on-demand techniques we distribute only the needed content on-the-fly, but at the same time we avoid remote I/O contention and access latency by involving the VM instances in a collaborative chunk exchange protocol in a manner similar to peer-to-peer approaches. To the best of our knowledge, we are the first to explore this direction.

3 Our Approach

3.1 Design Principles

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 [20]. 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 remotely from the original source and copied (i.e., "mirrored") locally. 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.

Preventive Peer-to-Peer Content 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 VM instances to talk to each other and "help" each other out in order to reduce the pressure on the remote repository. Based on the observation that I/O contention leads to jitter [7] (i.e., slight differences in time when the same chunk is accessed), we propose to organize the VM instances in a peer-to-peer topology where each VM has a set of neighbors, with whom it "gossips" about the chunks that should be fetched on-demand. Based on this information, VMs are able to anticipate future trends in access pattern and obtain chunks from their neighbors before they are actually needed, effectively preventing costly remote accesses if the anticipation was successful.

Access Pattern Aware Content Exchange Throttling: Preventive peer-to-peer content exchange however is not without drawbacks. Although the performance overhead of exchanging chunks can be masked by decoupling it 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. Therefore, it is crucial to "focus the gossiping" 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 sending 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, nurturing its collaborations when it senses a common pattern, and backing off when it senses a divergence. To avoid the case when VMs converge again without lowering the amount of unmatched chunks accumulated in the past, one solution is to automatically eliminate unmatched chunks older than a predefined time window. For simplicity, we opted not to address this case for the purpose of this work.

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 preventive exchange scheme from on-demand access and perform it asynchronously in the background.

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

```
1: function READ(buffer, offset, size)
2:     for all chunk ∈ Image|chunk ∩ (offset, size) ≠ ∅ do
3:         if ChunkState[chunk] = REMOTE then
4:             fetch chunk from repository and mirror it locally
5:             if unmatched < THRESHOLD then
6:                 HintQueue ← HintQueue ∪ {chunk}
7:             end if
8:             ChunkState[chunk] ← READ
9:         else if ChunkState[chunk] = LOCAL then
10:            unmatched ← unmatched − 1
11:            ChunkState[chunk] ← READ
12:        end if
13:    end for
14:    return read (offset, size) into buffer from Mirror
15: end function
```

Each chunk of the virtual disk image can be in one of the four possible states (denoted $ChunkState$): $REMOTE$ (the chunk was not yet locally fetched), $LOCAL$ (the chunk was obtained through gossiping and is locally present, but was not yet needed), $READ$ (the chunk was requested by a read operation) and $WRITTEN$ (the chunk was overwritten either totally or partially).

The READ operation is detailed in Algorithm 1. In a nutshell, it determines all chunks that are missing locally and fetches them from the remote repository, after which it redirects the read request to the local mirror. If any chunk triggered on-demand access (i.e., was in the $REMOTE$ state), it is scheduled to be sent to the neighbors through $HintQueue$, which is then used by the preventive exchange. This happens only if the

number of unmatched chunks is lower than the threshold. On the other hand, if any chunk is already available locally thanks to preventive exchange, the number of unmatched chunks is decremented. In both cases, $ChunkState$ transitions into $READ$, in order to reflect the new state.

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

```
1: function WRITE(buffer, offset, size)
2:     for all chunk ∈ Image|chunk ∩ (offset, size) ≠ ∅ do
3:         if ChunkState[chunk] = REMOTE and chunk ⊄ (offset, size) then
4:             fetch chunk \ (offset, size) from repository and mirror it locally
5:         end if
6:         ChunkState[chunk] ← WRITTEN
7:     end for
8:     return write (offset, size) from buffer to Mirror
9: end function
```

The WRITE operation, depicted in Algorithm 2 simply needs to make sure that there are no missing chunks that are only partially overwritten and thus will generate gaps. If this is not the case, it fetches the missing content from the remote repository in order to fill those gaps. In either case, it marks all involved chunks as $WRITTEN$ and finally redirects the write to the mirror.

Algorithm 3 Asynchronous preventive chunk exchange with other peers

```
1: procedure BACKGROUND_EXCHANGE
2:     while true do
3:         if chunk received from neighbor and ChunkState[chunk] = REMOTE then
4:             mirror chunk locally
5:             ChunkState[chunk] ← LOCAL
6:             unmatched ← unmatched + 1
7:         end if
8:         if HintQueue ≠ ∅ then
9:             chunk ← POP_FRONT(HintQueue)
10:            if ChunkState[chunk] ≠ WRITTEN then
11:                send chunk to all neighbors
12:            end if
13:        end if
14:    end while
15: end procedure
```

Finally, the preventive chunk exchange scheme is performed asynchronously inside BACKGROUND_EXCHANGE, detailed in Algorithm 3. In a nutshell, it listens for gossips about new chunks from all its neighbors and whenever it receives one that corresponds to a chunk that is missing locally, it fetches that chunk and mirrors it locally,

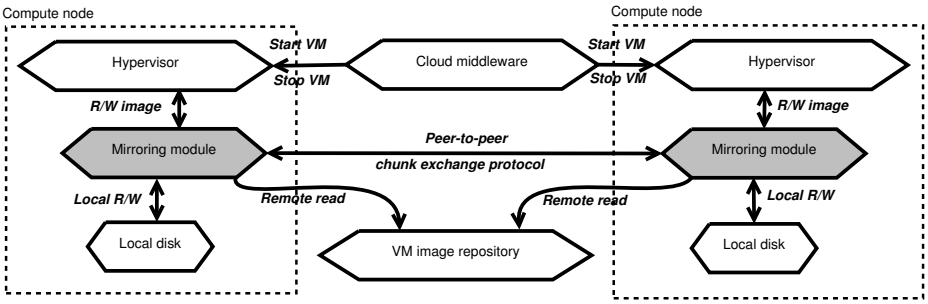


Fig. 1. Cloud architecture that integrates our approach (dark background)

incrementing the number of unmatched chunks. At the same time, if the READ operation enqueued any hints about on-demand chunks inside $HintQueue$ and the corresponding chunks were not overwritten in the mean time, then it informs all its neighbors about these new chunks.

Note that we opted for an optimistic scheme based on push, which improves latency compared to sending just the chunk id and waiting for a pull request. This technique favors the setting we explored in this work: small neighborhoods with a simple ring topology (See Section 4). However, with increasing neighborhood size it is more likely that a VM receives the same chunk from multiple sources, thus a push approach might unnecessarily waste bandwidth and create more overhead. Nevertheless, our algorithms require minimal changes to support a pull approach.

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 image templates used as the source of multi-deployments. 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 [16, 21].

The *cloud client* has direct access to the VM image repository and is allowed to upload and download VM images from it. Furthermore, the cloud client 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 [22]. 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.

To facilitate efficient on-demand access and copy-on-write support at kernel level, we map the remote VM disk image template into the memory of the host using the mmap system call. Since the kernel automatically manages memory page faulting and implicitly fetches any missing remote content, reads and writes are greatly optimized as they effectively translate to simple memory copy operations. Similarly, any chunks that were obtained through the preventive chunk exchange scheme can be mirrored locally again by simple memory copy operations.

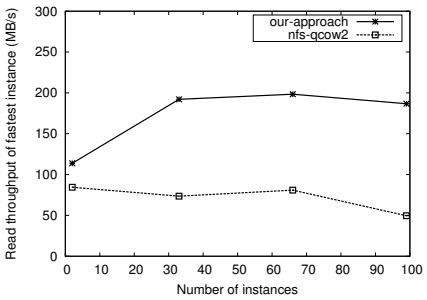
The preventive copy-on-write chunk exchange scheme runs in its own thread, which communicates with the main FUSE thread through the data structures presented in Section 3.2. The communication between the mirroring modules is implemented on top of Boost ASIO [23], a high performance asynchronous event-driven library which is part of the Boost C++ collection of libraries. Since the preventive peer-to-peer exchange scheme is not a pre-condition for correctness (i.e. our approach works even when no exchange is happening), we have opted for a lightweight solution that performs gossiping through UDP sockets. 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, as we can afford to occasionally lose hints about chunks.

4 Evaluation

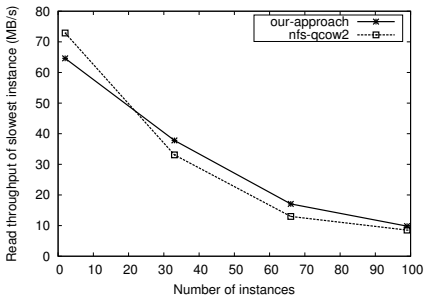
This section evaluates the scalability of our approach experimentally for a series of multi-deployment scenarios.

4.1 Experimental Setup

The experimental platform used to run our experiments is Grid'5000 [8]. For the purpose of this work, We reserved 100 nodes of the graphene cluster. The nodes are outfitted with x86_64 CPUs offering hardware support for virtualization, local disk storage of 277 GB (access speed $\simeq$55 MB/s) and 16 GB of RAM. The nodes are interconnected with Gigabit Ethernet (measured: 117.5 MB/s for TCP sockets with MTU = 1500 B with a latency of $\simeq$0.1 ms). The hypervisor running on all compute nodes is QEMU/KVM 1.2.0, while the operating system is a recent Debian Sid Linux distribution. For all experiments, a 4 GB raw disk image file based on the same Debian Sid distribution was used.



(a) Overall throughput of the fastest instance in the VM group (higher is better)



(b) Overall throughput of the slowest instance in the VM group (higher is better)

Fig. 2. Scalability of multi-deployments: throughput under concurrent on-demand access when reading 512 MB of the virtual disk using dd

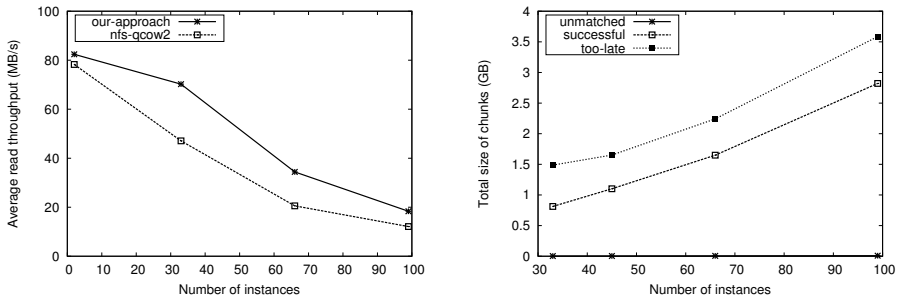
4.2 Methodology

We compare our approach (presented in Section 3.4) to the most widely used on-demand access technique in practice: local copy-on-write files that are derived from a shared backing file that is stored remotely. To enable copy-on-write, we rely on the *QCOW2* [5] image format, which is part of standard QEMU/KVM distribution. The backing file is shared through a NFS server. In order to deploy a VM instance, we create a fresh qcow2-derived file on the local disk of the compute node where the hypervisor is running and use this file as the VM disk image. For the rest of this paper, we denote this approach nfs−qcow2. Our approach uses the same backing file for local mirroring. With respect to the peer-to-peer topology, we opted for a ring: each mirroring module is linked to the mirroring module that is deployed on next compute node in a predefined ordering of all compute nodes. The $Threshold$ was fixed at 128, while the chunk size is fixed at 32 KB. We denote this setting our−approach.

The experiments consist in deploying an increasing number of VM instances concurrently, each on a dedicated compute node. Each VM instance boots and then reads the first 512 MB of its virtual disk (dd if=/dev/sda of=/dev/null count=1M). This simulates a read-intensive data access pattern (e.g. as sweeping through an input file) that is exhibited by all members of the multi-deployment in parallel and that generates high contention to the remote repository. We record the throughput of dd for every VM instance, as well as various other statistics gathered during the runtime of our−approach that relate to its internal workings.

4.3 Results

Figure 2 depicts the throughputs of the fastest and respectively the slowest VM instance for an increasing multi-deployment size. As can be observed, the fastest instance in the case of nfs−qcow2 experiences a significant drop in throughput with increasing number of VM instances (Figure 2(a)). This is expected because of increasing I/O pressure on



(a) Average throughput of the VM multi-deployment group as a whole (higher is better)

(b) Statistics about chunk accesses for our−approach

Fig. 3. Scalability of multi-deployments: average throughput and aggregated statistics under concurrent on-demand access when reading 512 MB of the virtual disk using dd

the NFS server. Our approach on the other hand achieves the exact opposite: not only does it start better (showing that preventive chunk exchange successfully avoids access to the NFS server even in a group of 2), but it also exhibits a dramatic increase in throughput as the number of VM instances is growing (showing that more VMs provide better chunk exchange opportunities). At the other extreme, even the slowest instance (Figure 2(b)) benefits from chunk exchange, albeit at lesser extent.

To put these results in perspective, Figure 3(a) depicts the average throughput achieved by the VM multi-deployment group as a whole. As expected, the increasing I/O pressure on the NFS server leads to a noticeable drop in both cases. However, when increasing the multi-deployment size beyond two, a stable gain of at least 30% more throughput is noticeable for our−approach when compared to nfs−qcow2.

To better understand the contribution of preventive chunk exchange to this result, we illustrate in Figure 3(b) the total size corresponding to how many chunks were unmatched (i.e., exchanged but never needed for a read), successful (i.e., exchanged and later contributed to avoid a remote NFS access) and too−late (i.e., exchanged but arrived too late to avoid a remote NFS access during an on-demand read). It can be observed that due to high similarity in the access pattern, almost all chunks are matched to a later read. This is observable by inspecting unmatched, which maintains a negligible level (less than $\simeq$10 MB) even for a large 100 node multi-deployment. Combined with a steady increasing trend for successful , this effectively explains the why the average throughput has a steady gain itself. However, it can also be noted that too−late is larger than successful , which shows that under high read pressure (as is our case) there is a high chance that a chunk is needed before it can be exchanged.

5 Conclusions

This paper introduced a novel multi-deployment technique based on augmented on-demand remote access to the VM disk image template. Being on-demand, it avoids an

expensive full pre-broadcast, while at the same time it pioneers the idea of exchanging chunks between multi-deployment members on-the-fly, in an effort to anticipate and prevent bottlenecks due to concurrent access to the remote repository where the VM disk image template is stored.

Our scheme is highly scalable, maintaining on the average a steady 30-40% improvement in read throughput compared to simple on-demand schemes in which the members of the multi-deployment are independent of each other. This is possible thanks to jitter between the VM instances, which enables the faster instances to effectively forward their chunks to slower instances in order to help them out. The results of this effect are dramatic: some instances become up to 4x faster compared to simple on-demand access.

Thanks to these encouraging results, we plan to further investigate the potential benefits of collaborative chunk exchange. In particular, we experienced a high number of chunks that arrived too late to be of use and thus an interesting direction to explore is how to avoid such chunks. Furthermore, as discussed in Section 3.2, we did not explore how to choose the optimal neighborhood size / topology and whether a pull scheme (i.e. push only chunk id as hint to others and prefetch chunk contents from others) might work better under the right circumstances. We plan to perform a deeper analysis in these areas.

Acknowledgments

The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative of the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/).

References

1. Amazon: Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/
2. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., Kozuch, M.A.: Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: SoCC '12: Proceedings of the 3rd ACM Symposium on Cloud Computing, San Jose, USA, ACM (2012) 7:1–7:13
3. Andrzejak, A., Kondo, D., Yi, S.: 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, Washington, DC, USA, IEEE Computer Society (2010) 257–266
4. Wartel, R., Cass, T., Moreira, B., Roche, E., Guijarro, M., Goasguen, S., Schwickerath, U.: 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) 112–117
5. Gagné, M.: Cooking with linux: still searching for the ultimate linux distro? Linux J. (161) (2007) 9
6. Chen, H., Kim, M., Zhang, Z., Lei, H.: 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), Montreal, Canada (2012) 5:1–5:6

7. Nicolae, B., Cappello, F., Antoniu, G.: 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) 503–513
8. Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Touche, I.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. Int. J. High Perform. Comput. Appl. **20** (November 2006) 481–494
9. Hibler, M., Stoller, L., Lepreau, J., Ricci, R., Barb, C.: Fast, scalable disk imaging with frisbee. In: Proc. of the 2003 USENIX Annual Technical Conference, San Antonio, USA (2003) 283–296
10. : SCPWave. http://code.google.com/p/scp-wave/
11. Schmidt, M., Fallenbeck, N., Smith, M., Freisleben, B.: 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) 567–574
12. Robison, N.A., Hacker, T.J.: Comparison of vm deployment methods for hpc education. In: RIIT '12: Proceedings of the 1st Annual conference on Research in Information Technology, Calgary, Canada, ACM (2012) 43–48
13. Carns, P.H., Ligon, W.B., Ross, R.B., Thakur, R.: Pvfs: A parallel file system for Linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, USENIX Association (2000) 317–327
14. Schmuck, F., Haskin, R.: 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)
15. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: 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) 307–320
16. Nicolae, B., Bresnahan, J., Keahey, K., Antoniu, G.: 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) 147–158
17. Zhang, Y., Li, Y., Zheng, W.: Automatic software deployment using user-level virtualization for cloud-computing. Future Gener. Comput. Syst. **29**(1) (January 2013) 323–329
18. De, P., Gupta, M., Soni, M., Thatte, A.: Caching vm instances for fast vm provisioning: a comparative evaluation. In: Euro-Par'12: Proceedings of the 18th international conference on Parallel Processing, Rhodes Island, Greece, Springer-Verlag (2012) 325–336
19. Jin, K., Miller, E.L.: The effectiveness of deduplication on virtual machine disk images. In: SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, Haifa, Israel, ACM (2009) 7:1–7:12
20. Theimer, M.M., Lantz, K.A., Cheriton, D.R.: 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) 2–12
21. Hansen, J.G., Jul, E.: Scalable virtual machine storage using local disks. SIGOPS Oper. Syst. Rev. **44** (December 2010) 71–79
22. : File System in UserspacE (FUSE). http://fuse.sourceforge.net/
23. : The Boost C++ collection of libraries. http://www.boost.org/