Scalable Virtual Machine Deployment Using VM Image Caches

Kaveh Razavi
Dept. of Computer Science
VU University Amsterdam
The Netherlands
k.razavi@vu.nl

Thilo Kielmann
Dept. of Computer Science
VU University Amsterdam
The Netherlands
thilo.kielmann@vu.nl

ABSTRACT

In IaaS clouds, VM startup times are frequently perceived as slow, negatively impacting both dynamic scaling of web applications and the startup of high-performance computing applications consisting of many VM nodes. A significant part of the startup time is due to the large transfers of VM image content from a storage node to the actual compute nodes, even when copy-on-write schemes are used. We have observed that only a tiny part of the VM image is needed for the VM to be able to start up. Based on this observation, we propose using small caches for VM images to overcome the VM startup bottlenecks. We have implemented such caches as an extension to KVM/QEMU. Our evaluation with up to 64 VMs shows that using our caches reduces the time needed for simultaneous VM startups to the one of a single VM.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Storage hierarchies;
C.4 [**Performance of systems**]: Design studies

Keywords

Infrastructure-as-a-Service, Scalability

1. INTRODUCTION

With the advent of public Infrastructure-as-a-Service (IaaS) clouds like Amazon EC2 or Rackspace, the use of virtualized operating systems, "virtual machines," has gained widespread use. Also in privately owned computing environments such as compute clusters, the use of virtual machines (VMs) is gaining popularity due to its benefits like elastic machine allocation, user-controlled software installations, and the possibility to reduce energy footprints by consolidating multiple VMs onto a single physical machine.

The promise of elastic computing is instantaneous creation of virtual machines, according to the needs of an application or web service. In practice, however, users face VM startup times of several minutes, along with high variability, depending on the actual system load. Two major factors are contributing to VM startup times: the resource selection process by the cloud middleware (Amazon EC2, OpenNebula, OpenStack, etc.), and the actual VM boot time, including the transfer of the VM image (VMI) to the selected compute node.

While we have noticed that there is room for improvement in the resource selection process, e.g. in our own OpenNebula deployment, that problem is beyond the scope of this paper. Here, we focus on reducing the transfer time for VMIs from the storage node to the compute nodes. In particular, we study the scalability of VMI transfers with respect to simultaneous VM startups, from a single VMI or from many VMIs.

Our initial study shows that state-of-the-art, on-demand transfers ("copy-on-write") cannot sustain performance for 10 or more simultaneous image transfers when using a 1Gb Ethernet connection to the storage node. When using 32Gb InfiniBand (IB) instead, simultaneous startup of up to 64 machines (the size of our cluster) can be done in constant time, as long as all machines boot from the same VMI. When increasing the number of VMIs used, the storage node itself becomes the bottleneck, and startup times rise linearly with the number of images.

While investigating this problem, we have observed that during the boot process, virtual machines actually read only a small fraction (we have seen up to 200 MB) of the total VMI, typically sized at several GB. Based on this observation, we propose to use VMI caches to mask the actual transfer bottlenecks. Depending on the location of the bottleneck, VMI caches can be placed either on the disks of the compute nodes (when the network is the bottleneck, e.g. with 1Gb Ethernet), or in main memory of the storage node (when disk access is the bottleneck, e.g. with an IB network).

We have implemented our VMI caches as an extension to KVM/QEMU. (A similar extension could be implemented for Xen as well.) As such, our caching scheme is independent of the cloud middleware in use and could be deployed on a wide range of cloud infrastructures. We have evaluated our caching scheme on our DAS-4 cluster [6] at VU Amsterdam. Our results show that using our caches, with either network, reduces the time needed for (up to 64) simultaneous VM startups to the time needed for booting a single VM.

This paper is organized as follows. In Section 2, we demonstrate the limited scalability of on-demand VMI transfers. In Section 3, we present the design and in Section 4 the implementation of VMI caches to overcome these limitations.

SC13 November 17-21, 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2503210.2503274

The results of our experimental evaluation are shown in Section 5. Based on the results of our evaluation, we recommend cache placement strategies in Section 6. Section 7 discusses related work; in Section 8 we conclude.

2. SCALABILITY OF ON-DEMAND TRANSFERS

The simplest way of deploying a VMI on a compute node is to copy the VMI onto the compute node before booting the VM from it. As VMIs typically comprise one or more GB of data, this approach obviously is slow and easily consumes large amounts of network bandwidth in between the storage node and the compute nodes.

The current state-of-the-art is to reduce the amount of data transfer to those blocks of the VMI that are actually need during the boot process, called on-demand transfers. With on-demand transfers, VM writes go to a second, copy-on-write (CoW) image. VM reads, if not already in the CoW image, come from the original VMI (base), accessed through a remote file-system like NFS. In this scheme, the base VMI is read-only and can be shared simultaneously by an arbitrary number of nodes.

QCOW2 [11] is an example image format that supports CoW images. The base image can be of any supported format. The read and write granularity of QCOW2 is defined by QCOW2's cluster size with the default value of 64KB. Figure 1 shows the operation of QCOW2's CoW mechanism. QEMU's implementation of QCOW2 is used by KVM and partly Xen, two commonly used virtual machine monitors (VMMs) in public and private IaaS clouds.

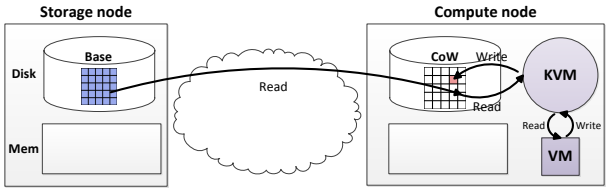

Figure 1: Copy-on-write with on-demand transfers in action. The VM writes go to a local *CoW* image, and reads are fetched from a *Base* image over a remote file-system like NFS.

Using QCOW2 with a remote base image is a good example of CoW with on-demand transfers. This approach significantly reduces the booting delay and the pressure on the network by voiding the need for the complete VMI transfer in the beginning. On-demand transfers, however, can have scalability problems of their own, which we are going to discuss in the remainder of this section.

2.1 Single VMI

In a single-VMI scenario, the content of one VMI needs to be transfered to many compute nodes on demand. This is a common case either for popular VMIs in public clouds, or for high-performance computations with many worker nodes of the same type, as with parameter sweep applications [19].

Figure 2 shows the booting time of a CentOS Linux VM on compute nodes[1]. When there are more than eight concurrent boots, the booting time increases linearly with the number of nodes in the case of a 1GbE network, suggesting that the network is becoming the bottleneck. This result already shows the need for some sort of efficient caching of the VMI at compute nodes. In the case of a 32Gb IB network, the booting time remains constant, suggesting that booting these CentOS VMs is not saturating the network.

[1]Machine details of the experiments presented in this section are explained along with our evaluation in Section 5.

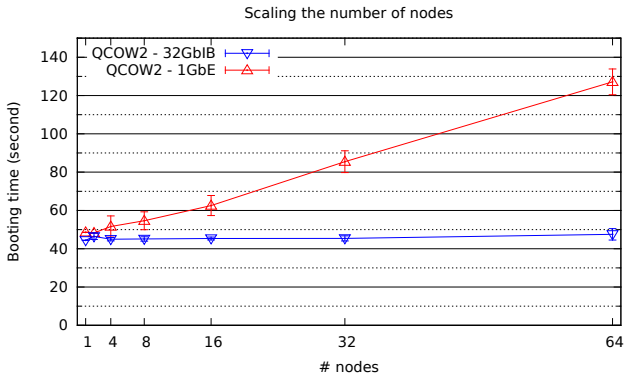

Figure 2: Booting time of a CentOS Linux VM on many compute nodes simultaneously using a single VMI. The reads are fetched from a remote base image and the writes go to a local CoW image.

2.2 Many VMIs

In this scenario, many VMIs need to be transferred to many compute nodes. This is a common case for public IaaS clouds, where many users may boot different VMIs simultaneously.

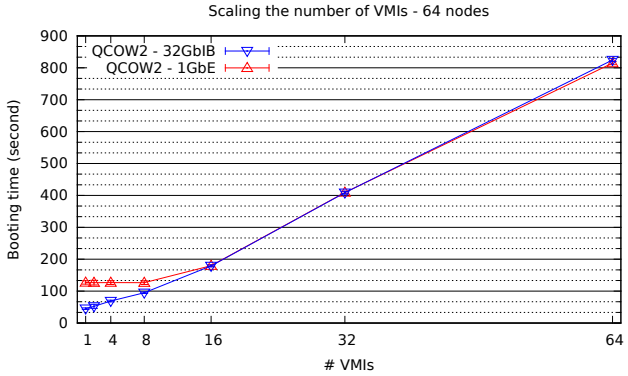

Figure 3: Booting time of a CentOS Linux VM on many compute nodes simultaneously using different number of VMIs. The reads are fetched from the assigned remote base image and the writes go to a local CoW image.

Figure 3 shows the booting time of 64 CentOS Linux VMs, scaling the number of VMIs used. (For this test, we have created 64 identical but independent copies of the CentOS VMI.) Regardless of the network speed, as the VMs use more independent VMIs, the booting time increases significantly. This is due to the disk queueing delay at the storage node, suggesting the use of VMI caches in between storage-node disk and compute-node memory.

Two possible locations for VMI caches are the storage node's memory or the compute nodes' disks. We will investigate both in the next section. Before doing so, we verify the feasibility of such caches by analyzing their required sizes.

2.3 VMI boot working set size

The basic idea for using VMI caches is to have those parts of the VMI in the cache that are required for booting the VM, leaving accesses to the remaining VMI parts for the actual service or application runtime, a period with much weaker performance requirements towards the VMI storage node. For this purpose, we have measured the amount of data that is read from the base image for three different VMIs. The results are in Table 1, and suggest that with a modestly sized cache, it is possible to boot many VMs while avoiding the potential bottlenecks described earlier in this section. From these values we can conclude that a VMI cache entry would need to have in the order of 250 MB (providing some margin). This size is small enough to build caches for multiple VMIs, either on the disks of the compute nodes or on the main memory of the storage node. In Section 3, we present our design of such a caching mechanism.

Table 1: Read working set size of various VMIs for booting the VM.

VMI	Size of unique reads
CentOS 6.3	85.2 MB
Debian 6.0.7	24.9 MB
Windows Server 2012	195.8 MB

3. VMI CACHES

We will now present the design of our VMI caching scheme. We begin by summarizing the underlying, fundamental requirements.

The first requirement for the cache is being a *VMI* itself. A VMI can then be created/stored on any desired medium (i.e., disk, memory) at any desired location (i.e., storage node, compute node). This also means that the cache is *standalone*; a VM can start booting using it. In the case of missing data however, the cache should be able to recurse to the base image.

The second requirement is support for *quota*. If the caching medium is a scarce resource like memory, a quota makes sure that it is not overly used by the cache. Further, it provides a fine-grained resource accounting of the cache per-VMI.

The third requirement is *immutability* with respect to the base image. An immutable cache, once created, can be reused many times in the future as long as the base image remains unchanged.

3.1 VMI chaining

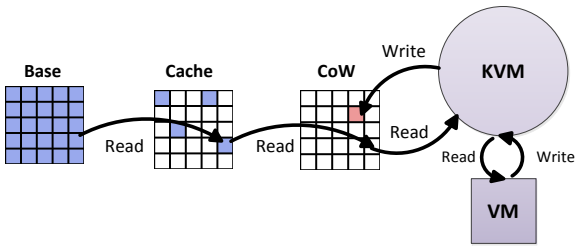

Figure 4: The new architecture with a VMI cache in between the base and CoW images.

To support these requirements, we have come up with an intermediate image between the base image and the CoW image, called the VMI cache.

Figure 4 shows how the image chain looks like with a VMI cache. The VMI cache is a *VMI* by definition, and with enough data blocks, a VM can boot using it. Since the cache image is separate from the CoW image, it is possible to enforce *quota* on it, satisfying the second requirement. To make the cache *immutable* with respect to the base image, we only write the data that comes from the base image into the cache. (All writes coming from the VM itself go to the CoW image.)

3.2 Cache creation

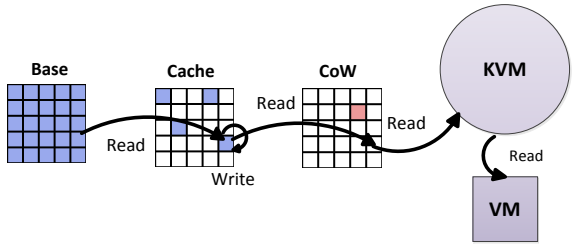

Figure 5: The process creating the cache. Every read from the base image incurs an additional write to the cache.

We now describe how we populate the cache with data. The first time a VM boots, an empty cache is created. Figure 5 shows the process of warming the cache. Every read that is fetched from the base image is also copied into the cache (copy-on-read or CoR). The first n blocks of data are stored in the cache until the quota is reached or the VM does not need any more data to be fetched from the base image. This CoR caching strategy ensures that the blocks that are needed for the booting process will be available in the cache.

The VMI caches can be created in variety of ways. The system can boot a sample VM upon a new VMI registration to create the cache. It is also possible to create the cache lazily when a VM is booted from a VMI for the first time.

3.3 Caching medium

Another design decision is the caching medium. Since the cache is a VMI by itself, it is possible to store it at compute or storage nodes. The quota allows a fine-grained control over how much resources should be dedicated to the cache image.

In the case of a slower network, caching on the compute nodes is an interesting option to reduce the load on the network. The small size of these cache images, makes it possible to store many of them within a modest amount of disk resources at compute nodes.

Another interesting medium is the storage node's memory to save on the limited performance of its disk(s). The read requests coming from different VMs are mostly random in nature and rotational disks do not handle this well. Memory (or a solid-state drive) provides a much better performance under a random access workload, but is scarcer in terms of capacity. The small size of our VMI caches can use this capacity effectively under the VM boot workload. Furthermore, it is possible to store many of these cache images on

the storage node's memory. This is not possible with normal VMIs due to the fact that their size is usually in the order of gigabytes. In Section 6, we will compare various cache placement strategies based on the results of our evaluation.

3.4 Cache-aware cloud scheduler

We discuss design considerations for a cache-aware cloud scheduler. Cloud schedulers are designed with various orthogonal goals. As an example, OpenNebula [12], an off-the-shelf cloud stack, has the following options for its scheduler:

- *Packing*: tries to minimize the number of nodes in-use by packing the VMs in the same host.
- *Striping*: tries to allocate VMs to nodes in a striping fashion in order to provide maximum available resources for VMs.
- *Load-aware mapping*: tries to allocate VMs to nodes with less load in order to provide maximum available resources for VMs.

One of the goals of a cache-aware scheduler should be allocation of VMs to nodes with an existing warm cache. This heuristic can be used in conjunction with any of the above desired strategies. One of the other tasks of a cache-aware scheduler should be the eviction of VMI caches whenever the allocated cache space is full for a new VMI cache. This can be a policy such as LRU at the node or cloud level. Further discussion on this topic is out of the scope of this paper and is left for future work.

4. IMPLEMENTATION

We have implemented the VMI caches as an extension to the QCOW2 block driver of QEMU. Before explaining our solution, we first take a look at the QCOW2 image format and block drivers in QEMU.

4.1 QCOW2 image format

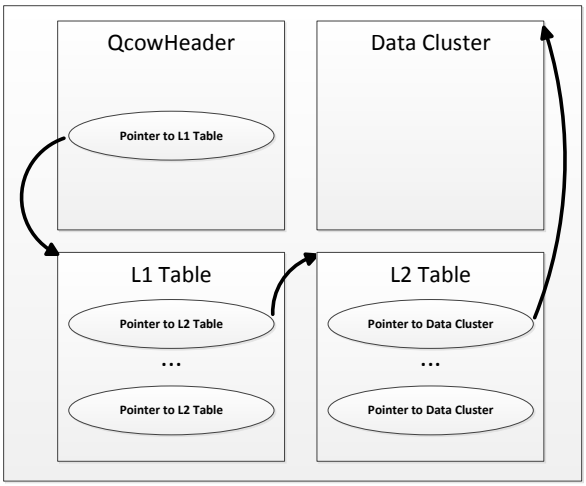

Figure 6: A simplified QCOW2 image structure. The choice of *cluster size* decides the size of L1-table and the number of data cluster pointers in L2-tables.

A QCOW2 image is a self-contained file with a number of meta-data structures followed by the actual data in clusters. The meta-data structures hold information about image parameters (e.g. size) and help the translation of virtual block addresses (VBAs) to physical block addresses (PBAs).

Figure 6 shows a simplified structure of a QCOW2 image. The first meta-data structure that is found in a QCOW2 file is the *QCowHeader*. Among other fields, it includes the cluster size, the image size, the backing file (if any) and an offset to the L1-table. QCOW2 uses a two-level look-up system using a level 1 (L1) and level 2 tables (L2). For a look-up operation, first the high n bits of the 64-bit VBA is used as an offset into a L1-table to find the corresponding L2-table. Then the next high m bits is used as an offset within the L2-table to find the corresponding cluster offset within the image file. The rest of the bits (cluster bits or d) are used as an offset within the cluster. L2-tables also occupy one cluster. This means that given the cluster size, it is easy to calculate n and m. For example, with the default cluster size of 64KB (18 bits):

$d = 18$ bits
$m = 18 - 3$ (address size) $= 15$ bits
$n = 64 - (18 + 15) = 31$ bits

The actual size of the L1-table depends on the number of L2-tables. The number of L2-tables depends on the image and cluster size. More details on the QCOW2 format can be found in [11].

4.2 Block drivers in QEMU

Like any other block driver, QCOW2 needs to implement certain functions to be exported as a block driver in QEMU. The most relevant of these functions are *create*, *open*, *close*, *read*, and *write*. These functions are then used in two applications that come with the QEMU/KVM suite: *qemu-img* and *qemu-kvm*.

qemu-img is used for creating and/or manipulating virtualized images. As an example, when creating a new QCOW2 image, *qemu-img* is invoked with the relevant parameters such as the image size or in the case of QCOW2, the cluster size or the path to the backing file (if any). This information is then passed to the create function of the QCOW2 driver to prepare the requested image file.

qemu-kvm provides virtualization and device emulation. One of the emulated devices is the disk controller. When a VM is running under *qemu-kvm*, all its read and write requests to the disk controller are handled by the relevant block driver. In this case, the VM is completely unaware of the underlying block driver functionalities.

Once the caching mechanism is included in the QCOW2 block driver, *qemu-kvm* will use it seamlessly. *qemu-img*, however, must be invoked with the relevant arguments for creating and/or manipulating the cache images. We have explained this further in Section 4.4.

4.3 Cache extension

To support cache images, we needed to add two more fields to the QCowHeader of the QCOW2 image. These new 8-byte fields define the quota and the current size of the cache. It was not possible to re-use the size field of the QCowHeader since it has to be the same as the base image's. This is because the CoW image can in theory have the same size as the base image and we decided not to propagate the differentiation between the CoW and the cache image

throughout the source.

We now describe our modifications to QCOW2 functions to support VMI caches:

create: If the quota passed to the *create* function is not zero, it is assumed that the new image will be used as a cache. The *create* function then stores the quota and the current size of the cache (= size of the header and initial tables) as part of a new extension to the QCowHeader in the image file. The implementation of these new fields as an extension is to ensure backward compatibility with normal QCOW2 images.

open: When opening a QCOW2 image, it is checked against our new caching extension. If the extension is detected, then the two size fields are read into QEMU's QCOW2 main data-structure and the image is treated as a cache image.

read: When we get a read on the cache image, two scenarios are possible. Either the data exists in the cache (warm cache), or the data needs to be fetched from the base image (cold cache). In the first scenario, the data is read from the cache image and returned to image requesting it (CoW image). In the second scenario, we recurse to the base image for the data. Once the data is available, before returning it to the CoW image, we write it to the cache. It could be that we get a space error when trying to write to the cache due to full quota. In this case, we stop writing to the cache for the future cold reads.

write: In our design, described in Section 3, the cache image is protected from writes coming from the VM. The only writes that the cache image gets are for warming it up (with data from the base image). Whenever we see a write to the cache, we check whether there is enough space left by looking at the quota and the currently used size. If there is enough space, we write the data to the cache and update the currently used size. If not, we return with a space error that is handled at the read function described above.

close: When closing a QCOW2 image, if the cache quota field is present (i.e. a cache image), the (new) current size of the cache is written back to the image file.

Other than these modifications to the QCOW2 block driver, we also had to change the permission flags of QEMU when opening an image. The default flag for the backing images is read-only and the cache image is used as a backing image for the CoW image. The cache image, however, needs write permission at least at its creation time. It is not known at the opening time whether an image is a cache image or a base image. To address this problem, we first open the backing image with read *and* write permissions, and then if we detect that the image is not a cache image, we re-open the image with read-only permission.

Since the QCOW2 driver already has the concept of recursion for its CoW feature, our introduction of cache images required minimal changes to QEMU's QCOW2 block driver. A complete patch against the original QEMU/KVM modifies about a hundred and fifty lines of code. With this design, we achieve both backward compatibility with QCOW2 and massive code reuse.

4.4 Chaining cache images with qemu-img

With normal QCOW2 operation, first a CoW image is created using *qemu-img*. The base image is given to *qemu-img* as CoW image's backing file. After that, a VM is started with *qemu-kvm* pointing to the CoW image as its booting disk.

With the cache images, there is another step involved when creating the cache image. First, *qemu-img* is invoked with a cache quota and pointing to the base image as its backing file. This step creates a cache image. Second, *qemu-img* is invoked with no cache quota and pointing to the cache image as its backing file. This step creates a CoW image. Now the VM can be started with CoW image as its booting disk. With a warm cache, there is obviously no need to invoke *qemu-img* for creating the cache.

One of the benefits of our approach, as we just discussed, is its simplicity for chaining cache disks. This makes it ideal for integration with any cloud stack with an already existing support for QCOW2.

5. EVALUATION

We have conducted an extensive experimental evaluation of the VMI caching mechanism. We used the DAS4/VU [6] cluster as our evaluation testbed. Each standard DAS4/VU node is equipped with dual-quad-core Intel E5620 CPUs, running at 2.4GHz, 24GB of memory and two Western Digital SATA 3.0-Gbps/7200-RPM/1-TB in software RAID-0 fashion. The nodes are connected using a commodity 1Gb/s Ethernet and a premium Quad Data Rate (QDR) InfiniBand providing a theoretical peak of 32Gb/s.

Our experiments use up to 65 of these nodes, one of them acting as **storage node**, and up to 64 other nodes as **compute nodes**. The storage node runs an off-the-shelf NFS-server; the compute nodes mount the NFS location. We have tuned the NFS *rwsize* to 64KB (the default cluster size of QCOW2), as the default NFS *rwsize* of 1MB does not match well with the small-sized read requests during boot time. This *rwsize* has been used for all experiments. Besides, we use the Linux *tmpfs* and *tmpfs exports* for backing (remote) files by memory when necessary.

For all the experiments described below, we have used a default installation of CentOS 6.3 as our VMI. The other images mentioned (Debian Linux and Windows Server) have only been used for estimating their cache size requirements.

We are mostly interested in the boot time of virtual machines. We measure the boot time as the time from invoking KVM for starting the VM until the VM connects back (automatically) to a given port as soon as it has completed its boot process.

5.1 Cache creation

We study the performance of cache creation, the effect of cache quota and the reduction on the storage node transfers with a warm cache. All these experiments use one storage node and one compute node. The base image is on a NFS export on the storage node and the cache is created at the compute node. We use the 1GbE network in these experiments. (The results are similar for the 32Gb InfiniBand and are omitted for brevity.)

Figure 8 shows booting times with increasing cache quota, hence controlling the amount of data that can be stored in a cache image. The boot times with a warm cache are roughly the same as with the original QCOW2 mechanism, as expected. With a cold cache, however, writing into the cache file during boot time significantly slows down the boot process, due to delays from slow, synchronous writes to the cache image. To circumvent this problem, we *create* the cache in memory, such that the cache write operations do not

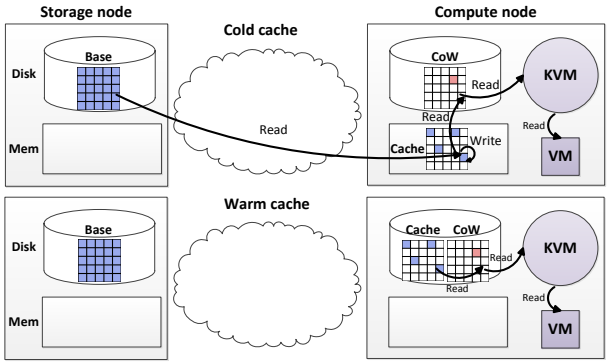

Figure 7: Caching the VM image on the compute node. The cache is created on the memory of the compute node to avoid slowing the VM down due to expensive writes. With a warm cache, there is no need to go to the network anymore.

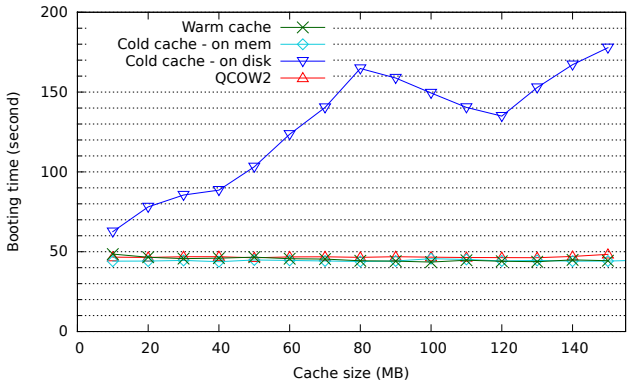

Figure 8: Cache creation overhead with increasing cache quota.

delay the reads from the booting VM. This reduces the cache creation overhead to a negligible amount. When creating the cache in memory, the cache still needs to be written to the disk. We delay this actual write to the moment after the VM has been shut down, taking it out of the critical path for booting. Due to the small size of the cache, the transfer to the disk takes less than one second.

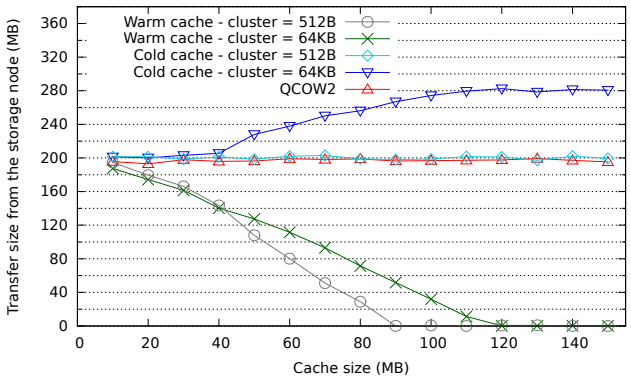

Figure 9: Observed traffic at the storage node with increasing cache quota.

Figure 9 shows the observed traffic at the storage node. With a warm cache, we see a smaller traffic with a bigger cache quota. An interesting observation is that a cold cache with the default QCOW2 cluster size of 64KB, is causing more traffic than the original QCOW2. Investigating further revealed that this is because small writes to the cache need to fetch more data from the base image to meet the cluster granularity. Reducing the cache cluster size to the minimum of 512 bytes (sector size), circumvented a potentially unscalable cold cache.

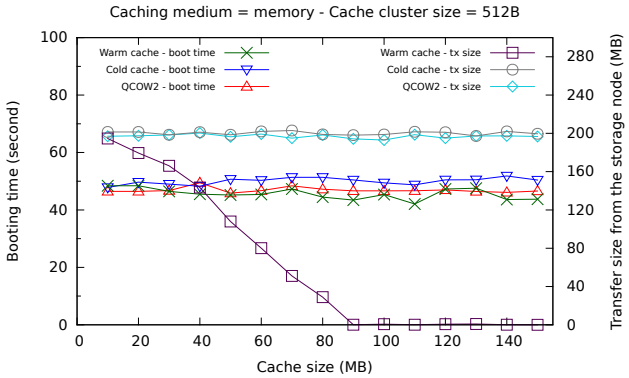

Figure 10: Final arrangement for cache creation.

Figure 10 shows the observed performance (boot time and data transfer size) with the cold and warm cache when the cluster size is set to 512 bytes. The results show that with the careful choice of the cache cluster size and placement of the cold cache on memory, it is possible to make cache creation scalable with near-zero overhead. The same arrangement, also shown in Figure 7, is used with the scalability benchmarks in the rest of this section.

According to our discussion on the QCOW2 image format in Section 4.1, a smaller cluster size will result in more frequent lookups and more L2-table entries for the cache image. As shown in Figure 10, the frequency of lookups does not affect the booting time since most reads during boot are small and need a lookup anyway. For a cache quota of 200 MB, only 3.1 MB is necessary for L2-tables. Thus, we believe the smaller cluster size for the cache image is justified.

5.2 Cache quota

Table 2: Cache quota necessary for various VMIs

VMI	Warm cache size
CentOS 6.3	93 MB
Windows Server 2012	201 MB
Debian 6.0.7	40 MB

Table 2 shows the necessary cache quota for various VMIs with cache cluster size of 512 bytes. From Figure 10 it is clear that CentOS 6.3 needs a cache size of about 90MB. The created cache image has about the same size on the file-system. A Debian that is taken from the services image of an open-source PaaS [23], creates a cache image of 40MB. A Windows Server needs a substantially bigger cache image for the boot workload. The numbers in Table 2 are slightly bigger than the read working set sizes shown in Table 1. The difference is caused by the meta data added by QCOW2 at various locations of the VMI file.

5.3 Scaling

The micro benchmarks presented so far provide us with a good understanding of the individual caching behavior. Now, we continue to our primary concern of this work: scalability.

5.3.1 Scaling nodes

As shown in Section 2, on-demand transfers have a scalability problem with commodity networks when booting one VMI over many compute nodes. We show that this can be resolved by our proposed cache images. In this experiment, 64 compute nodes start a VM from the same VMI simultaneously.

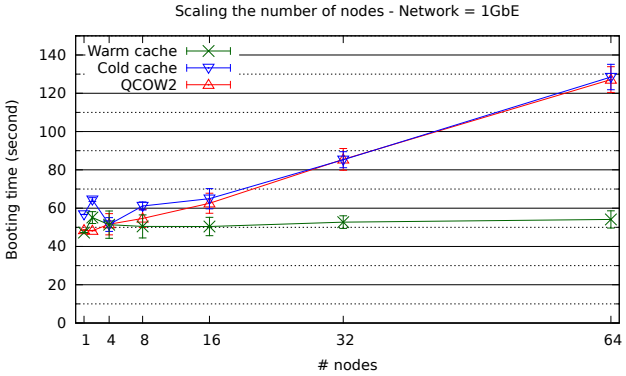

Figure 11: Caching a single VMI image at compute nodes over a 1GbE.

Figure 11 shows the average booting time of the VMs. With a cold cache, it takes about the same time to boot the VMs as the original QCOW2. With a warm cache, the booting time over many compute nodes is similar to that of a single VM. These results suggest that the VMI caches are effective in resolving the network bottleneck.

In a real-life scenario, we do not expect that all the nodes start from a cold or a warm cache. Depending on the cloud node scheduler, it can be that some of the nodes start from the cold cache and some from a warm cache. A cache-aware scheduler should always prefer the nodes with a warm cache. Studying a cache-aware node scheduler is left for future work. Regardless of the node allocations, the nodes with a warm cache contribute to reducing the network load on the storage node(s). (We do not present quantitative results for such mixed scenarios, however, in this paper.)

5.3.2 Scaling VMIs

Networks are getting much faster than they used to be and disk-based storage is not catching up. We have shown in Section 2 that the disks at the storage node become a severe scalability bottleneck when many VMs are booted from many VMIs simultaneously. In this set of experiments, we show how VMI caches can help addressing this problem. In all the points in the graphs, 64 nodes are booting VMs while they share various number of VMIs. The caching setup is the same as in Figure 7, where compute nodes store VMI caches on their local disk.

Figure 12 shows the effect of caching the VMIs at the disk of the compute nodes. For the 1GbE network, with a single VMI, the difference between warm caches and QCOW2 is

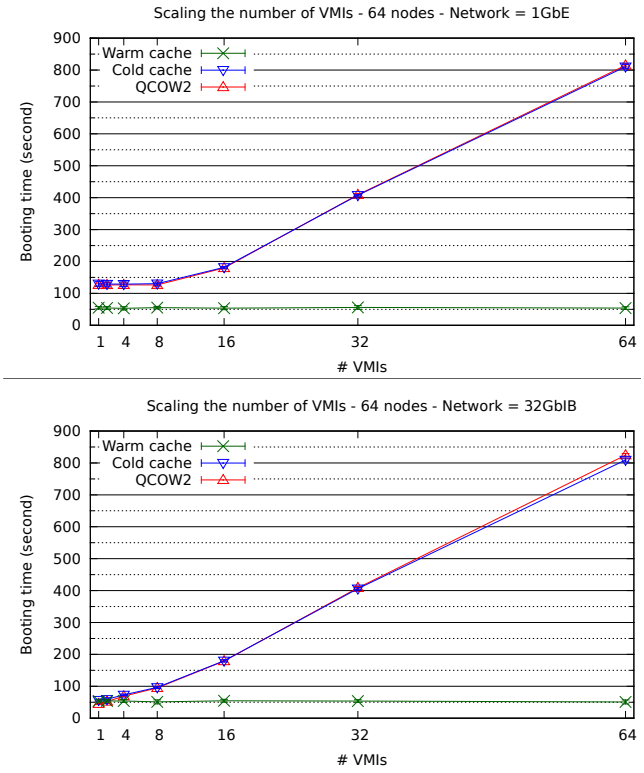

Figure 12: Caching many VMIs at the compute nodes' disk over the two different networks.

the cost of the network bottleneck. This is the same bottleneck that we observed in Figure 11 at 64 nodes. Starting from 16 VMIs, the storage node's disk is becoming the primary source of the scalability bottleneck. The VMI caching at the compute nodes avoids both scalability bottlenecks at the network and at the storage node's disk. For the 32GbIB network, the caching avoids the scalability bottleneck of the storage node's disk. Since the network is not a scalability bottleneck in this scenario, the difference in booting time with more than a single VMI is only due to the bottleneck at the storage node's disks.

Since the storage nodes' disks are proving to be a more severe scalability problem than that of the network, another attractive caching strategy is caching over the memory of the storage node. Figure 13 shows a possible setup where the VMI caches are created on the compute nodes and then transferred back to the storage node's memory before being used as a warm cache.

In this set of experiments, we have added the time of cache transfers to the booting time with the cold cache to reflect on the fact that the cache image transfers are now a necessary part of the system. When VMIs are shared between VMs, only one of the VMs creates and transfers the cache back to the storage node while other VMs just proceed with normal QCOW2.

Figure 14 shows the results over the two networks. In the case of the 1GbE network, this caching strategy does not solve the network scalability bottleneck, but it does solve that of the storage nodes' disk. With the cold cache, the booting delay is slightly higher with 64 nodes, due to the transfer time. In the case of the 32GbIB network, the only

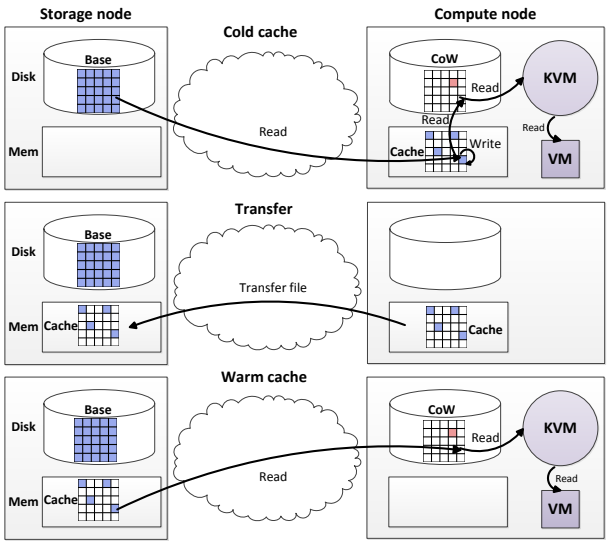

Figure 13: Caching the VM image in the memory of storage node. The cache is created in the memory of the compute node and then transferred to the memory of storage node. With a warm cache in memory, there is no need to go to the storage disk anymore.

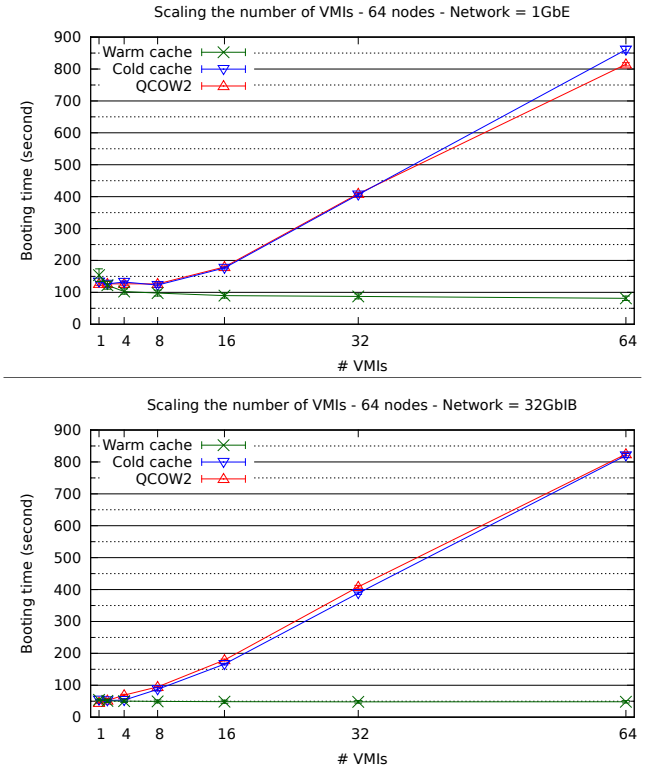

Figure 14: Caching many VMI on the storage node's memory over the two different networks.

scalability bottleneck is resolved without any overhead.

With VMI caches on the storage node's memory, there needs to be a mechanism that decides on the eviction policy from the cache pool. Strategies similar to the ones discussed in Section 3.4 can be applied here.

In the next section, we further discuss cache placement considerations based on the evaluation of this section.

6. CACHE PLACEMENT

In the previous section, we showed that VMI caches can be placed either on the compute nodes' disks or on the storage node's memory.

There are certain advantages for placing the caches on the storage node's memory, compared to compute nodes' disk:

- The compute nodes do not need to reserve any disk space for the VMI caches.
- There are fewer security concerns, regarding the content of the VMIs cached at any time at any compute node.
- The storage node's memory is efficiently used for the task at hand; transferring blocks of VMI data.
- A cache-aware scheduler can treat all compute nodes equally as the VMI cache is centrally available at the storage node.

Thus, in scenarios where the network is fast enough to handle on-demand transfers of many simultaneous VM startups, using solely the storage node's memory for placing the VMI caches is the superior solution.

The only remaining problem is the question whether a certain network is able to handle such a workload. If this is not the case, caching on the compute node's disk is one possibility. Caching only on the compute node's disk, however, still leaves the possibility that the storage nodes' disks become a bottleneck in a multi-VMI scenario. To address this, we recommend using caches at both storage and compute nodes. Algorithm 1 uses chaining to the cache at the proper location, or creates one if necessary.

Algorithm 1: Chaining to a proper cache VMI

Input: Compute node C, Storage node S, VMI $Base$
Output: A VMI to be chained to a CoW image
if $Cache_{Base}$ *exists in* C **then**
  return $Cache_{Base}$
if $Cache_{Base}$ *exists in* S **then**
  if $Cache_{Base}$ *is on disk* **then**
    Copy $Base_{cache}$ to tmpfs
  Create $NewCache_{Base}$ on C
  Chain $NewCache_{Base}$ to $Cache_{base}$
  return $NewCache_{Base}$
Create $Cache_{Base}$ on C
Chain $Cache_{Base}$ to $Base$
Copy $Cache_{Base}$ to S on VM shutdown
return $Cache_{Base}$

We assume that there is a cache eviction policy, as described in Section 3.4, that removes caches in case the reserved cache space is full. Algorithm 1 prefers chaining to a local cache (if it exists) to avoid the network as much as possible. In case the cache does not exist at the compute node, it tries to create one while chaining to another cache at storage node's memory, avoiding the storage node's disks.

Since, with a fast network, random access on remote memory can be faster than on local disk, a VM might boot faster

from the storage node's memory. This might conflict with Algorithm 1. We have investigated the severeness of this effect with our machine setup from Section 5, using a CentOS VMI. Our results show at most 1 % difference in startup times between a cache on the compute node's disk, compared to the storage's memory. We hence consider the difference to be negligible.

7. BACKGROUND AND RELATED WORK

We distinguish different research work related to VMI caches in four overlapping categories:

1. *Efficient VMI transfer* deals with the problem of moving VMIs from storage nodes to compute nodes.
2. *Efficient VM migration* deals with efficient migration of VMs along with their state from a node to another. The goal is reduction in downtime to improve user-perceived experience.
3. *Caching storage data* is on the mature field of caching data in storage layers. We focus on the relevant work to VMIs.
4. *Virtualized disk performance* discusses the advances in virtualized storage that are directly applied to VMIs.

Below, we discuss each category separately. Whenever appropriate, we make comparisons to our VMI caches or discuss how our solution complements a different work.

7.1 Efficient VMI transfer

There are two different scenarios in which efficient VMI transfer becomes a primary concern. In the first scenario, a single VMI needs to be transferred to many physical nodes. In the second scenario, many VMIs need to be transferred to many physical nodes concurrently and start executing on behalf of different users. Different techniques have been adopted for these two different scenarios.

7.1.1 Single VMI

Peer-to-peer networking is a common technique for transferring a single VMI to many compute nodes [4, 18, 27]. The main issue so far has been the considerable delay of startup time in order of tens of minutes. This is because the complete VMI needs to be present before starting the VM. A recent work by Reich et al. [24] combines on-demand access with peer-to-peer streaming to reduce this delay significantly.

While peer-to-peer transfer is a good match to this problem in slower networks, it uses substantial network resources to deliver the VMI to the compute nodes. In a shared environment, this can become problematic for other users of the cloud. VMI caches are orthogonal to these peer-to-peer techniques. The small cache size makes it possible to store many of them simultaneously on the compute nodes, effectively reducing the total bandwidth used in the peer-to-peer transfers. Further, the networks are becoming faster, and as we discussed in Section 2, the workload of on-demand transfers is easily handled by premium networks, voiding the need for peer-to-peer networking in near future.

LANTorrent [17], from the Nimbus project, combines simultaneous VMI requests and builds a pipeline for streaming complete VMIs from the storage node to all requesting compute nodes. This is very adequate for applications or services starting up with many VMs at the same time. For small, private clouds, where all nodes are connected to a single network switch, this chaining maximizes the throughput. Our VMI caches serve this case too, but also work well in other situations, like an already running service requesting one or a few more VMs when scaling out, while the cloud infrastructure is under stress by other users. The main advantage of caching and reading VMIs on demand comes from the fact that only small parts of a VMI are needed while booting the machine. The small number of remote reads that happen after the initial booting, while slower, are transient and largely being offset by the gain in the booting time. Amazon reports similar behavior for their EBS-backed images.[2]

SnowFlock [10] can start many stateful worker VMs under one second. It introduces *VMFork* and *VM descriptor* primitives that fork child VMs that are in the same state as the parent VM when they start. SnowFlock achieves good performance by multicasting the requested data to all workers and uses a set of avoidance heuristics at child VMs to reduce the amount of memory traffic from the parent to the children. While SnowFlock is a good solution to the single VMI scenario, it uses substantial multicast traffic for delivery. Furthermore, almost all layers from the application up to the VMM require substantial changes. We envision a possibility of building a system with memory caches, with similar ideas to VMI caches, to be used for efficient stateful VM creation without much changes to the stack.

7.1.2 Many VMIs

Schmidt et al. [25] use Unionfs, a stacked file-system, to incrementally prepare the VMI. The base VMI that contains a big chunk of the final VMI, remains constant among different VMIs. By caching the base VMI on the compute nodes and transferring the rest using a sophisticated transfer method like multicast, they achieve startup delays that would not have been possible otherwise. There are two distinct ways that VMI caches can be beneficial for this method. First, a much smaller cache image can be created out of the base VMI and be stored on the compute nodes instead of the base VMI itself. This effectively reduces storage resources of the compute nodes that are used for caching. Second, the read-only nature of VMI caches can relax the requirement for a stacked file-system. VMI caches can be created for any type of image in any state. The user-customized part can be transferred in the form of a copy-on-write image to the compute nodes that recurses to the cache image when necessary.

VDN [22] is a network hierarchy-aware system for transferring VMIs to compute nodes. Each VMI is divided into chunks and each compute node has a cache for these chunks. When booting a new VM, the compute node fetches the VMI chunks that it lacks in its local cache from its peers in a network topology-aware fashion. Again, VMI caches can help in such situations because of their smaller size. Further, VMI caches have an implicit knowledge about future accesses when they are being created. This can help prioritizing fetching of different chunks.

Nicolae et al. [15] stripe VMIs' chunks into the disks of many compute nodes. During VM boot, if a chunk is required and missing, it is fetched from a peer that has that chunk. They further improve their approach in [16] by means

[2]http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ComponentsAMIs.html

of prefetching the chunks required for VM startup. The access-pattern knowledge used in prefetching is retrieved from the peers that boot a little faster. In most scenarios, only a small number of data blocks is required for a VM to boot from multi-GB VMIs. Striping our VMI caches with exactly these blocks can help saving disk resources at compute nodes.

7.2 Efficient VM migration

While VM migration is a different problem than that of VMI transfer, both share many of the same obstacles due to the requirement for sizable transfer of data.

Internet Suspend/Resume [9] is one of the earliest systems that looks at the possibility of using a VM with explicit state transfer over a distributed file system to achieve seamless migration of personal user desktops over space and time. The idea of using very small files instead of huge transfers in the beginning to hide the network latency is very similar to today's on-demand transfer technologies.

Clark et al. [5] introduce a low-overhead mechanism for live migration of virtual machines. They use techniques such as pre-copying of VM state and the subsequent transfer of the memory write working size to the destination node. Hines and Gopalan [8] report on improvements with post-copying of VM state on a faster network while using two optimizations: 1.) self-ballooning of guests to release free memory pages back to the VMM and 2.) pre-paging [20] to intelligently push future pages to the migrated guest. The ideas presented in that research work can be effective for caching the VM state, a future direction for our VMI caches.

7.3 Caching storage data

The collective [3] uses a VM appliance to ease up the administration of many users over the network. VM images get cached locally at the client's host and methods like copy-on-write and prefetching are employed to improve the user experience. In the collective, the VM appliance does not change often and only a small number of VMI versions are available and stored locally, voiding the need for the selective caching technique that we have used in our implementation of VMI caches to make it suitable for a cloud environment.

Dm-cache [1] partitions a fast storage device (i.e., SSD) between many concurrently running, co-hosted VMs to improve the observed I/O performance. Dm-cache can improve booting performance if many VMs boot from the same VMI. There is, however, a considerable performance penalty when VMs are booting from a cold cache.

Content-based block caching [13] is a caching method that takes into account the content of data blocks to improve the efficacy and efficiency of the cache by means of deduplication and silent writes. Since VMIs created from the same operating system distribution share content, this method can be deployed to reduce the effective size of cache images of different VMIs on the compute nodes even further. The performance implications on the booting time with content-based caching and data deduplication is a subject for future research.

Patterson et al. [21] show that by using application provided disclosures, one can parallelize the otherwise sequential disk access by prefetching the blocks that are going to be accessed in future. Disclosures also help to balance prefetching against caching. A possible optimization for VMI caches is prefetching based on disclosures to speedup VM booting. A fundamental difference is that the disclosures of the cache images can be inferred automatically at their creation time. Our preliminary experience with prefetching, however, showed no substantial benefit. For example, in the CentOS case, the VM only waits 17% of its total boot time on reads and prefetching can only mask that.

7.4 Virtualized disk performance

Ming et al. [28] suggest that simply using NFS to transfer VMIs is sub-optimal. By adding a module to NFS to cache a number of NFS requests at the compute nodes or a proxy, they improve the VM booting process with a warm cache. They further improve the performance of the virtual disk by doing copy-on-write in an NFS proxy that is running inside the VM [2]. In contrast, VMI caches provide a clean caching abstraction at VMI-level. This makes it possible to do resource accounting per VMI. It would have been difficult otherwise to decide, for example, which NFS requests should remain in the memory cache of the storage node (or the cache proxy), or the disk at compute nodes.

FVD [26] is a new image format that makes the choice of different features and their implementations orthogonal while providing high performance. While there are many interesting independent features, copy-on-read and copy-on-write are co-dependent, making the FVD image format unsuitable for persistent read caching. Our implementation of the VMI caches is providing an immutable cache which is backward compatible with QCOW2.

8. CONCLUSIONS

Virtualized operating systems (VMs) have recently become popular due to their use in IaaS cloud environments, as well as in privately-owned compute clusters. The promise of elastic computing is instantaneous creation of such virtual machines, according to the needs of scalable web services or high-performance applications. In practice, however, virtual machine startup times typically take several minutes, facing strong variability depending on the actual system load.

Virtual machine startup delays come from two sources. One is the time that the cloud middleware (EC2, OpenStack, OpenNebula, etc.) needs for selecting a suitable physical machine. Those delays are beyond the scope of this paper. Here, we have dealt with the other major source for startup delays, the time for transferring the virtual machine image (VMI) from a storage node to the selected compute nodes.

The state-of-the-art technique transfers the VMI during the actual boot process in an on-demand fashion, a technique known as *copy-on-write*. We have investigated copy-on-write for simultaneously booting up to 64 VMs and discovered serious slowdowns when using a 1Gb Ethernet between the storage node and the compute nodes. Using a 32Gb InfiniBand connection, copy-on-write only scales when the VMs read from the same VMI. Booting 64 VMs from more than one VMI simultaneously, the storage node itself becomes a bottleneck, and copy-on-write slows down significantly.

The key observation from our tests is that VMs actually read only small fractions of the huge (multi-GB) VMI during the boot process, with ≈200 MB being the biggest size we have observed (from a Windows Server 2012 image). This small size allows to create VMI caches that, once warmed up, can significantly reduce the amount of network traffic for booting a VM. We have implemented VMI caches as an

extension to QCOW2, QEMU's implementation of copy-on-write. Our VMI caches go logically in between the CoW image and the base image on the storage node. Physically, caches can be placed either on the disks of the compute nodes or in main memory of the storage node. We have proposed a simple algorithm that chains VMI caches on both possible locations, combining the benefits of both approaches.

Our performance evaluation shows that using warm caches reduces the startup time of up to 64 VMs to roughly the startup time of a single VM, independent of how many different VMIs are used for booting. Also, the initial creation of a cache image requires only negligible runtime overhead, compared to booting a VM with the vanilla copy-on-write mechanism. Placing the VMI caches in the main memory of the storage node is the most elegant solution, combining high performance with minimal administrative overhead and the absence of storage requirements on the compute nodes. Only with slow network interconnects (like the 1Gb Ethernet), placing the VMI caches on the compute node disks performs somewhat better as they avoid bulk network transfers altogether.

Our results have shown that using VMI caches significantly speeds up the time needed for concurrently booting virtual machines. The next step of our work is to integrate this scheme into the cloud scheduler, like the OpenNebula package on our DAS-4 system. This includes hints for the node selection process for using VMI caches on the compute nodes themselves. Another interesting line of work is to apply our caching scheme to memory snapshots of already booted virtual machines, starting from which instead of the VM image could improve the VM starting time even further. Finally, we think it is worthwhile to investigate data compression and deduplication techniques that have been developed for VMI storage (e.g. [7, 14]) in the context of VMI caches to gain even more storage efficacy.

Acknowledgments

This work is partially funded by the FP7 Programme of the European Commission in the context of the Contrail project under Grant Agreement FP7-ICT-257438, and by the Dutch public-private research community COMMIT/. The authors would like to thank Kees Verstoep for providing excellent support on the DAS-4 clusters.

9. REFERENCES

[1] D. Arteaga, M. Zhao, P. V. Riezen, and L. Zwart. Dynamic Block-level Cache Management for Cloud Computing Systems. Poster, 10th USENIX conference on File and Storage Technologies, FAST'12, 2012.

[2] V. Chadha and R. J. Figueiredo. ROW-FS: a user-level virtualized redirect-on-write distributed file system for wide area applications. In *Proceedings of the 14th international conference on High performance computing*, HiPC '07, pages 21–34, 2007.

[3] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The collective: a cache-based system management architecture. In *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI '05, pages 259–272, 2005.

[4] Z. Chen, Y. Zhao, X. Miao, Y. Chen, and Q. Wang. Rapid Provisioning of Cloud Infrastructure Leveraging Peer-to-Peer Networks. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems Workshops*, ICDCSW '09, pages 324–329, 2009.

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI '05, pages 273–286, 2005.

[6] DAS-4 clusters. `http://www.cs.vu.nl/das4/clusters.shtml`. [Online; accessed 22-April-2013].

[7] L. Garces-Erice and S. Rooney. Scaling OS Streaming through Minimizing Cache Redundancy. In *31st International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 47–53, 2011.

[8] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 51–60, 2009.

[9] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, WMCSA '02, 2002.

[10] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 1–12, 2009.

[11] M. McLoughlin. The QCOW2 Image Format. `http://people.gnome.org/~markmc/qcow-image-format.html`, 2008. [Online; accessed 16-April-2013].

[12] D. Milojičić, I. Llorente, and R. S. Montero. OpenNebula: A Cloud Management Tool. *IEEE Internet Computing*, 15(2):11–14, 2011.

[13] C. B. Morrey and D. Grunwald. Content-Based Block Caching. In *23rd IEEE, 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '06, 2006.

[14] C.-H. Ng, M. Ma, T.-Y. Wong, P. P. C. Lee, and J. C. S. Lui. Live Deduplication Storage of Virtual Machine Images in an Open-Source Cloud. In *Proc. Middleware 2011*, number 7049 in LNCS, pages 81–100, 2011.

[15] B. Nicolae, J. Bresnahan, K. Keahey, and G. Antoniu. Going Back and Forth: Efficient Multideployment and Multisnapshotting on Clouds. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC '11)*, pages 147–158, 2011.

[16] B. Nicolae, F. Cappello, and G. Antoniu. Optimizing multi-deployment on clouds by means of self-adaptive prefetching. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par '11, pages 503–513, 2011.

[17] Nimbus Project. LANTorrent. `http://www.nimbusproject.org/docs/current/admin/reference.html#lantorrent`, 2010. [Online; accessed 4-August-2013].

[18] C. M. O'Donnell. Using BitTorrent to distribute virtual machine images for classes. In *Proceedings of the 36th annual ACM SIGUCCS fall conference: moving mountains, blazing trails*, SIGUCCS '08, pages 287–290, 2008.

[19] A.-M. Oprescu and T. Kielmann. Bag-of-Tasks Scheduling under Budget Constraints. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CloudCom '10, pages 351–359, 2010.

[20] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: a system for migrating computing environments. *SIGOPS Operating Systems Review*, 36(SI):361–376, 2002.

[21] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 79–95, 1995.

[22] C. Peng, M. Kim, Z. Zhang, and H. Lei. VDN: Virtual machine image distribution network for cloud data centers. In *29th Conference on Computer Communications*, INFOCOM '10, pages 181–189, 2012.

[23] G. Pierre and C. Stratan. ConPaaS: a Platform for Hosting Elastic Cloud Applications. *IEEE Internet Computing*, 16(5):88–92, 2012.

[24] J. Reich, O. Laadan, E. Brosh, A. Sherman, V. Misra, J. Nieh, and D. Rubenstein. VMTorrent: scalable P2P virtual machine streaming. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 289–300, 2012.

[25] M. Schmidt, N. Fallenbeck, M. Smith, and B. Freisleben. Efficient Distribution of Virtual Machines for Cloud Computing. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, PDP '10, pages 567–574, 2010.

[26] C. Tang. FVD: a high-performance virtual machine image format for cloud. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC '11, pages 18–18, 2011.

[27] R. Wartel, T. Cass, B. Moreira, E. Roche, M. Guijarro, S. Goasguen, and U. Schwickerath. Image Distribution Mechanisms in Large Scale Cloud Providers. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CloudCom '10, pages 112–117, 2010.

[28] M. Zhao, J. Zhang, and R. Figueiredo. Distributed File System Support for Virtual Machines in Grid Computing. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, HPDC '04, pages 202–211, 2004.