Squirrel: Scatter Hoarding VM Image Contents on IaaS Compute Nodes

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

Ana Ion
Dept. of Computer Science
VU University Amsterdam
The Netherlands
a.ion@student.vu.nl

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

ABSTRACT

In IaaS clouds, virtual machines are booted on demand from user-provided disk images. Both the number of virtual machine images (VMIs) and their large size (GBs), challenge storage and network transfer solutions, and lead to perceivably slow VM startup times. In previous work, we proposed using small VMI caches ($O(100\,\mathrm{MB})$) that contain those parts of a VMI that are actually needed for booting. Here, we present Squirrel, a fully replicated storage architecture that exploits deduplication, compression, and snapshots from the ZFS file system, and lets us keep large quantities of VMI caches on all compute nodes of a data center with modest storage requirements. (Much like rodents cache precious food in many distributed places.) Our evaluation shows that we can store VMI caches for all 600+ community images of Windows Azure, worth 16.4 TB of raw data, within 10 GB of disk space and 60 MB of main memory on each compute node of our DAS-4 cluster. Extrapolation to several thousands of images predicts the scalability of our approach.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Storage hierarchies;
E.4 [**Coding and Information Theory**]: Data compaction and compression

General Terms

System Design, Experimentation

Keywords

VM Images, Caching, Deduplication, Compression

1. INTRODUCTION

With the advent of public Infrastructure-as-a-Service (IaaS) clouds, like Amazon EC2 or Windows Azure, the use of virtualized operating systems, "virtual machines", has gained widespread use. 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 [16, 21]. One important factor contributing to VM startup time is the transfer of the VM image (VMI) from a storage node, via the data center network, to the selected compute node [35].

The simplest VMI transfer technique copies the whole VMI, typically several GBs, to the selected compute node's disk, from where the VM will boot. State of the art is to use Copy-on-Write (CoW) images on the compute nodes, which means accessing the VMI over a network file system and reading only those parts that are needed at boot time, while directing write operations to a local CoW image.

In our recent work [34], we proposed to put a VMI cache in between VMI and CoW image, preferably on the compute node's local disk. As with CoW images, the compute node mounts the VMI and reads it on demand. The difference is that VMI caches are populated with the data read from the VMI in a Copy-on-Read (CoR) fashion. As soon as the VM is booted, the VMI cache contains the boot working set. The next time the compute node needs to boot from the same VMI, it finds a warmed-up cache and can boot the VM without further network transfers, both speeding up the VM boot process, and lowering the traffic pressure on the data center network.

While the VMI caching mechanism overcomes scalability problems of VM startup, it introduces a new set of challenges for avoiding cold caches. Traditional solutions to this problem include cache replacement policies (e.g. LRU [32]) as well as cache-aware VM scheduling. In this work, we take a radically different approach: We propose a fully replicated design, storing *all* VMI caches of a data center on *all* its compute nodes. We show that this is both possible and scalable. We term this approach *scatter hoarding* in analogy to the technique of creating a large number of small hoards (caches) by which squirrels store their food reserves.[1] We describe our implementation of such a design, named *Squirrel*, and thoroughly evaluate its important properties.

The contributions of this paper are as follows:

1. We study the effects of deduplication when combined with compression on both VMIs and VMI caches (Section 2). To the best of our knowledge, this is the first study of its kind. We show that smaller block sizes do

[1] `http://en.wikipedia.org/wiki/Hoarding_(animal_behavior)`

not necessarily yield better overall compression ratios. More importantly, we show that storing caches of *all* VMIs of a data center is feasible with modest storage requirements on the compute nodes.

2. Backed by these observations, we devise a new, fully replicated storage architecture aimed at storing the caches of *all* VMIs of a data center on *all* the compute nodes. We present Squirrel, our implementation of such a design, based on VMI caches and the ZFS file system [4] (Section 3).

3. We evaluate Squirrel using the set of the 600+ community images from Windows Azure[2] to show its desirable properties (Section 4).

Our evaluation shows that that VMI caches have higher cross-similarity than their associated VMIs. This is because VMI diversity mostly comes from installed software, while boot working set diversity (i.e. VMI caches) comes from only a few OS distributions. Better cross-similarity of VMI caches means that they add fewer hashes to a deduplicated storage on average, making their storage more scalable. For our Windows Azure dataset, worth 16.4 TB of raw data, Squirrel only requires 10 GB of disk space and 60 MB of main memory on each compute node of our DAS-4 cluster. Extrapolating these storage requirements shows that Squirrel can scale to thousands of VMI caches with modest disk and memory requirements on current or near-future compute node hardware. Further, we show that, with proper parameter tuning, booting from a deduplicated and compressed file system can be as fast as a normal file system, despite earlier reports [26, 44].

After our evaluation, we discuss related work on VMI storage and transfer in Section 5, and we conclude in Section 6.

2. BACKGROUND

There are essentially two main ideas behind the work presented in this paper: (1.) using VMI caches and (2.) applying deduplication combined with compression to these VMI caches. In Section 2.1, we briefly explain how VMI caches operate. In Section 2.2, we discuss the effects of deduplication and compression on VMI caches. We then hint at the possible effectiveness of compressing VMI caches when applying these techniques.

2.1 VMI cache chaining

Figure 1 (top diagram) shows the internal operation of CoW. A compute node mounts the base VMI and reads from it on demand. Write operations are saved on the CoW image, keeping the base VMI clean. CoW saves the need for copying the entire VMI before booting can begin. This significantly reduces both the VM startup time, and the load on network and storage servers. One drawback with CoW is, however, the need to transfer the boot working set every time a new VM starts up.

In our previous work [34], we showed that transferring the boot working set at scale results in scalability problems. More specifically, when starting one VM from a single VMI on many nodes, the data center network becomes the scalability bottleneck, and when starting VMs from different VMI sources, the storage nodes become the scalability bottleneck.

[2]More precisely, there were 607 images in November 2013.

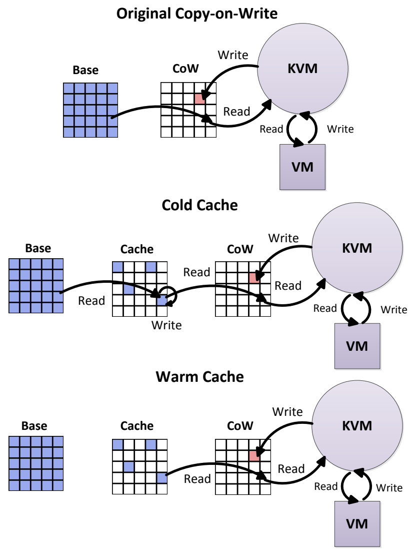

Figure 1: The introduction of VMI caches.

The former scenario is common for high-performance computing applications (e.g., parameter sweeps) as well as autoscaling systems (e.g., [13]). The latter is more common in large-scale, multi-user IaaS clouds with concurrent VM startups by different users.

To address these scalability problems, we introduced VMI caches, a mechanism that allows us to quickly boot a VM by storing the boot working set of a VMI on or near the compute node, avoiding the use of the data center network or the disks at the storage nodes at boot time. Typically, this boot working set is only a small fraction ($O(100\,\text{MB})$) of the original VMI, usually several GBs. The VMI caches are chained in between traditional CoW images and VMIs.

Figure 1 (middle) shows how VMI chaining operates with VMI caches, in contrast to the original CoW architecture. When a node boots from a particular VMI for the first time, the VMI cache is still empty ("cold"). With a cold cache, before handing the reads from the base VMI to the VM process, we write them to the VMI cache. This process is called copy-on-read (CoR), and we showed that it is possible to perform CoR with competitive performance compared to the original CoW architecture [34]. Once the caches are warm (shown at the bottom of Figure 1), the system does not need to read from the base VMI anymore during the boot process.

As mentioned, CoW images typically reside on compute nodes, and VMIs on storage nodes. In principle, VMI caches can reside on any storage medium in between CoW images and base VMIs. The candidates are the disks at compute nodes or the memory of storage nodes. While each candidate has its benefits, we showed in [34] that storing VMI caches on the disks at compute nodes is the preferable option for most scenarios. In this paper, we are looking at efficiently storing our VMI caches on the disks of the compute nodes.

2.2 Compression efficiency

In this section, we analyze the trade-offs involved in compressing the contents present in VMIs and VMI caches. Although the focus here is on VMI caches, the results with

VMIs are also provided for additional insight. In Section 4, we provide information on our VMI repository as well as the data set used in the figures presented in this section.

We use compression ratio as a metric for compression efficiency. If, for a given set of VMIs or VMI caches, the set of unique blocks is defined as U, and the set of nonzero blocks is defined as N, compression ratio for deduplication (i.e., the deduplication ratio) is defined in [12] as:

$$\text{Deduplication ratio} = \frac{|N|}{|U|},$$

where $||$ is the cardinality of a set. The compression ratio for content compression is defined as:

$$\text{Compression ratio} = \frac{\sum^{i \in U} \frac{size(compress(i))}{size(i)}}{|U|},$$

where *compress* is the compression routine (e.g., gzip).

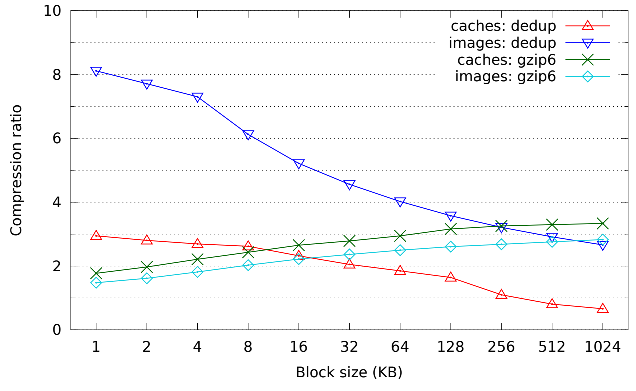

Figure 2: Compression ratio of VMIs and caches with dedup and gzip6

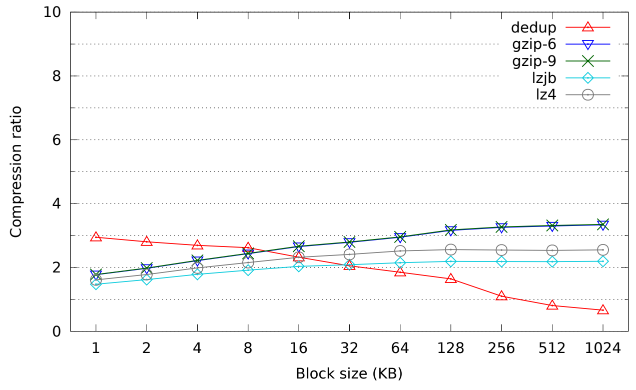

Figure 3: Compression ratio of VMI caches with different routines.

Figure 2 shows the compression ratio with deduplication and gzip6 compression. When decreasing the block size[3], we can see two conflicting trends: Deduplication ratio starts increasing, and gzip's compression ratio starts decreasing. As numerous VMI deduplication studies [18, 19, 26, 44] have pointed out, the reasons for higher deduplication ratio with smaller block sizes are (1.) the fact that small differences in otherwise similar larger blocks do not result in different hashes for the whole block, and that (2.) similar data with different alignments have a better chance of producing the same hash (i.e., deduplicating). Larger block sizes have better compression (with e.g., gzip) because the chance of finding similar duplicate strings within the block increases. We have also measured cache compression ratios for gzip9, lz4, and lzjb algorithms shown in Figure 3. gzip9 is compressing almost the same as gzip6 with higher CPU cost. lz4 and lzjb are faster compression algorithms than gzip6, but with lower compression ratios. As we will show in Section 4.2.3, since the extra CPU cycles for decompression do not lead to performance degradation, we decided to continue with gzip6. The results for VMIs are the same and omitted for the sake of brevity.

[3]In this paper, we study the trends when making block size smaller. Thus, it is natural to read the figures, that have block size on the horizontal axis, from right to left.

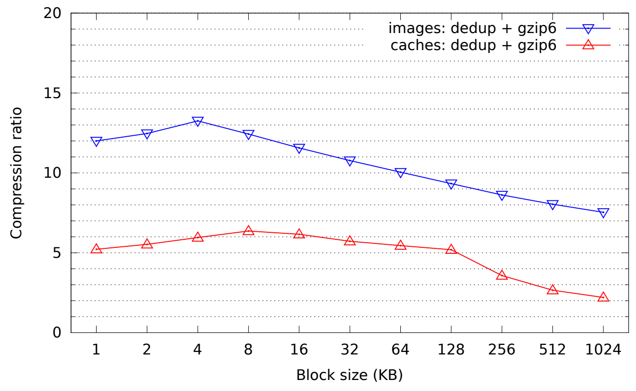

Figure 4: Combined compression ratio of VMIs and caches.

Because of the conflicting trends in compression efficiency for deduplication and compression, there exists an optimization point, after which lowering the block size will result in lower storage efficiency. We define combined compression ratio (CCR) as a metric that considers the effects of both compression techniques combined, and is calculated as:

$$\text{CCR} = \text{deduplication ratio} \times \text{compression ratio}$$

Figure 4 shows the CCRs for VMIs and caches. Despite common understanding, the smaller block sizes do not necessarily result in better compression ratios when considering both deduplication and compression. This is an important finding, not only because of storage efficiency, but also because smaller block sizes consume more memory for deduplication tables as we will show in Section 4.2.2. For VMIs, when reducing the block size, the CCR steadily increases up until 4 KB and then starts to decrease. For caches, the CCR does not show much improvement when reducing the block size after 128 KB, and starts to decrease after 8 KB. Note that while VMIs have better combined compression ratios than caches, we will show in Section 4.3 that storing them at compute nodes is neither efficient nor scalable.

2.3 Summary and discussion

We have highlighted the essential background work in this section. We briefly looked at our enabling mechanism, VMI caches, in Section 2.1. We then looked at compression opportunities for VMI caches in Section 2.2.

Table 1 shows the storage reduction as we apply different storage techniques to our VMI repository. With block size

Table 1: Attained storage efficiency with 128 KB block size.

Original	Nonzero	Caches (Nonzero)	Caches/CCR
16.4 TB	1.4 TB	78.5 GB	15.1 GB

of 128 KB (default in the ZFS file system), if the file system supports sparse files, the original 16.4 TB raw data reduces to 1.4 TB. Our VMI caches reduce that to 78.5 GB, and if compression and deduplication are applied, we get to a modest 15.1 GB of disk resources. This indicates that our claim of storing all VMI caches on compute nodes is feasible. It allows us to deploy fully replicated storage architecture for VMI caches. In Sections 4.2 and 4.3, we show that our approach is not only efficient, but also scalable to thousands of VMI caches.

3. SYSTEM ARCHITECTURE

In this section, we will first discuss the architecture of Squirrel, a fully replicated VMI caching system that is aimed at caching *all* VMIs on *all* compute nodes. We then describe how certain operations such as *register*, *boot*, and *deregister* are implemented using VMI caches, and the ZFS file system that provides us with inline compression and deduplication as well as snapshot versioning. Further, we will explain how VMI caches are propagated if some of the compute nodes are not online during the *register* operation.

3.1 Squirrel

Mainstream in today's IaaS cloud architectures are the following assumptions: (1.) There is an explicit separation between compute nodes and storage nodes. (2.) Compute nodes use all their resources to run VMs. (3.) Storage nodes provide a high performance, high volume, and fault-tolerant storage space for the running VMs on compute nodes.

Mainstream in current VMI distribution engines is the assumption that data transfer between storage nodes and compute nodes, or between compute nodes (i.e. peer-to-peer systems) is coming for free. Violating the second and third assumptions of cloud architectures, VMI transfers, an administration overhead, become a burden on storage nodes and/or consume network bandwidth available on compute nodes, making the running VMs prone to network SLA violations.

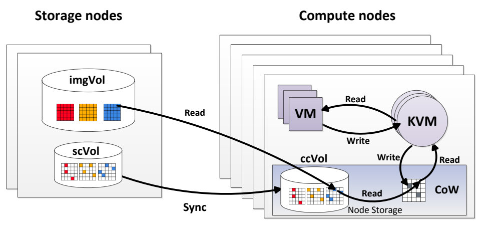

Figure 5: Squirrel architecture diagram.

Squirrel's architecture, depicted in Figure 5, is aimed at eradicating network I/O during VM startup for all VMIs of an IaaS cloud by using modest storage resources at compute nodes. We assume that there exists an off-the-shelf parallel file system that manages the storage nodes. In a typical scenario, the VMIs are stored on top of this parallel file system and are accessed by compute nodes during VM startup.

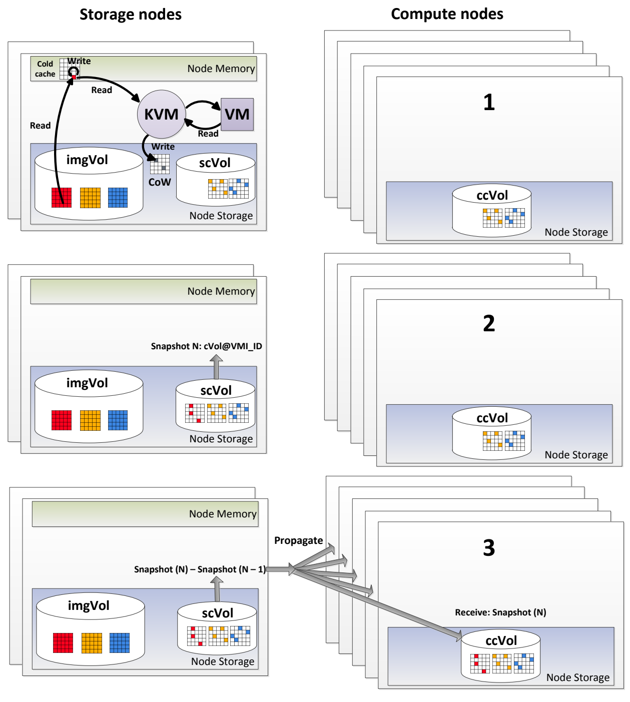

Figure 6: Squirrel VMI registration workflow.

Squirrel adds a cache volume, called *cVolume*, backed by the ZFS file system, next to the VMIs. The cVolume at the storage nodes, called *scVolume*, stores deduplicated and compressed VMI caches for all the VMIs currently registered to the system. At each compute node, Squirrel controls one cVolume, named *ccVolume*. In a stable state, the ccVolumes are in sync with the scVolume. We will explain how we exploit ZFS snapshots to bring the ccVolumes at lagging compute nodes (e.g. due to possible failures or down times) back in sync with the scVolume in Section 3.5.

We will now discuss how Squirrel implements the main VMI operations; namely *register*, *boot*, and *deregister*.

3.2 Register

We assume that the IaaS provider already has a mechanism for users to upload their VMIs to the storage nodes. For example, Amazon EC2 [2] provides **ec2_upload_bundle** [3] for this purpose, or OpenNebula [23] provides its users with the **oneimage create** command that uploads the VM image during the registration process.

Figure 6 shows Squirrel's workflow for image registration. With the VMI accessible, Squirrel first boots the VMI for the first time in one of the storage nodes to create the cache. In our previous work [34], we have shown that this takes no longer than a normal VM boot. In Section 4.2.3, we show that on average the VMs in our dataset boot in less than 20 seconds. Once the cache is created, Squirrel destroys the VM, and moves the cache from memory to the scVolume. Next, Squirrel creates a snapshot of the scVolume for this newly added VMI cache. ZFS snapshots are cheap in terms of storage as long as they do not reference data that no longer exists (i.e., deregistered VMIs). They are also cheap in terms of creation time since they are read-only. Finally, Squirrel propagates the VMI cache by sending the diff between this snapshot and the previous snapshot from the storage node to all online compute nodes. The diff is generated using ZFS incremental snapshot-send functionality. Transferring the diff (i.e., the new cache) from one node to many others is a common scenario in scalable data transfer, and has been extensively studied in the literature [8, 31, 38, 40]. With a simple IP multicast approach, transferring a diff of $O(100\,\text{MB})$, does not take more than a couple of seconds even on a commodity 1 GbE.

In total, the image registration workflow does not take more than a minute. Given that the actual VMI upload takes significantly longer, we consider Squirrel's registration workflow, which is an infrequent operation and not in the critical path for booting VMs, to be a modest price to pay for all the benefits that it delivers. These benefits are discussed in Section 4.

3.3 Boot

Booting a VM from a VMI in Squirrel is depicted in Figure 7. Squirrel chains an empty copy-on-write image on the local storage to the requested VMI cache on the ccVolume. The VMI cache on the ccVolume is chained to the original VMI, which is backed by the cloud storage and mounted on the compute node(s). During VM boot, all VM reads (of all possible VMIs) will be handled by the ccVolume. All VM writes will go to the copy-on-write image.

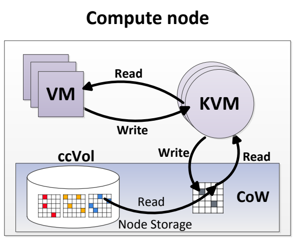

Figure 7: Booting a VM with Squirrel's ccVolume.

3.4 Deregister

VMI deregistration is as simple as deleting the original VMI and its associated cache on the scVolume. The more important background operation that keeps the cVolumes sustainable, while providing a window for offline propagation, is called garbage collection in Squirrel.

Squirrel implements garbage collection by keeping only the snapshots that are taken in the last n days, and the latest snapshot regardless of its creation time. n is a configurable variable that defines the offline propagation window, for when compute nodes miss a new cache, as well as the amount of time that dead references (i.e. unregistered VMI caches) remain in the system. Squirrel runs garbage collection as a daily cron job on all Squirrel cVolumes.

Note that Squirrel is not creating snapshots when deleting caches for the sake of simplicity. The information about unregistered VMIs will propagate to ccVolumes as soon as there is a new registered VMI (i.e., a new snapshot).

3.5 Offline propagation

With the increasing number of commodity servers in data centers, the likelihood of node failure increases. If a set of compute nodes are offline during the last stage of Squirrel's VMI registration or during deregistration, their ccVolumes become stale. To overcome this problem, each compute node, upon boot, queries for a diff between the latest snapshot available locally at its ccVolume, and the latest snapshot available at the scVolume. Two scenarios are likely to happen: (1.) The node has not been offline for more than n days. In this case, the incremental snapshot will succeed and the compute node receives the latest snapshot. (2.) The node has been offline for more than n days, or the node is a new addition to the compute node pool. In this case, the incremental snapshot fails, and Squirrel needs to replicate the entire scVolume. The second scenario however, does not happen often with a large enough n, and even if it does, as we show in Sections 2.2 and 4.3, the size of cVolumes never exceeds a few tens of GBs (same order as a single VMI).

One question that is likely to arise is why Squirrel uses snapshots to keep the ccVolumes in sync with scVolume, rather than simply using the rsync [37] utility for this purpose. The answer to this question is twofold. First, rsync is a many-to-one operation, and can easily create a bottleneck at the storage nodes. The ZFS incremental snapshot can be streamed using any conventional peer-to-peer or multicasting approach. Second, using rsync on-demand could potentially avoid the bottleneck, but it translates to VM booting delay for the first time. Further, with commodity networks (e.g., 1 GbE) booting on-demand at scale with a cold cache can introduce network bottlenecks as we have shown in [34].

4. EVALUATION

We evaluate different aspects of Squirrel's cVolumes in this section. Namely, in Section 4.2, we evaluate the storage requirements and effects on boot time for Squirrel's cVolumes, in Section 4.3, we evaluate the scalability of cVolumes to large numbers of VMI caches, and we analyze network transfers with Squirrel in Section 4.4.

For all the experiments of this section, we have used up to 68 standard nodes of the DAS-4/VU cluster [11]. Each standard DAS-4/VU node is equipped with dual quad-core Intel E5620 CPUs, running at 2.4 GHz, 24 GB of memory, and two Western Digital SATA 3.0 Gbps/7200 RPM/1 TB disks in software RAID-0 fashion. The nodes are connected using a commodity 1 Gb/s Ethernet and a premium QDR InfiniBand providing a theoretical peak of 32 Gb/s.

At the time of writing this paper, DAS-4/VU nodes are running CentOS 6.4 Linux. XFS is used as the local file system, and we have used a native ZFS installation [41] to run cVolumes as images on top of the XFS file system. It is possible to run ZFS directly on the disk(s), but we decided not to change the configuration of local disks on the nodes, as they are shared by many users.

We use the 600+ community images of Windows Azure as test case for our experiments. We provide detailed information about our VMI repository in Section 4.1. To generate the data for Figures 2, 3, 4, and 12, we submitted simple MapReduce jobs to Hadoop [5] running on a subset of the DAS-4/VU nodes for analyzing our data set. For Figures 8, 9, 10, and 13, we used the statistics of the ZFS file system for our analysis. The "real" data points in Figures 14, and 16 are also reported by ZFS.

4.1 Dataset information

The VMIs in our dataset are consisting of Linux-based operating systems, that are registered by the users of Win-

Table 2: OS diversity in Windows Azure and Amazon EC2.

OS distribution	Windows Azure	Amazon EC2
Ubuntu	579	5720
RedHat/CentOS	17	847
OpenSuse/Suse Ent.	5	8
Debian	3	30
Windows	0	531
Unidentified Linux	3	2654
Total	607	9871

dows Azure. Table 2 shows the number of VMIs for each OS found in our dataset and, for comparison, in Amazon EC2[4], the largest existing IaaS provider. The numbers reported by Amazon EC2 are for all the regions combined, so the numbers per region are likely to be smaller. We have provided extrapolations to thousands of VMI caches based on our dataset in Section 4.3.2.

The community VMIs of Windows Azure do not include Windows distributions, likely due to licensing reasons. If we had Windows in the mix, the boot working sets of different Windows distributions would have deduplicated with each other, thus adding a constant factor to Squirrel's storage requirements.

4.2 Cache volume efficiency

In this section, we analyze the effects of compressing and deduplicating the contents of VMIs and VMI caches in ZFS volumes. We first look at disk and memory requirements, and then we move on to booting performance from ccVolumes. In the end, we will summarize our findings.

4.2.1 Disk

To verify our findings with respect to combined compression ratio presented in Section 2.2, we stored once the VMI repository, and once the corresponding VMI caches in the ZFS file system and measured the respective disk consumption. Figure 8 shows the disk consumption for VMIs and caches with varying block size. To our surprise, when lowering block size, the point which results in worse CCR happens sooner than what we measured in Figure 4 (16 KB for images and 32 KB for caches). We suspected this is due to the fact that the deduplication table itself also needs to be written to the disk, resulting in even lower compression efficiency for small block sizes. Figure 9 measures this overhead with varying block size. The overhead of storing deduplication tables on the disk is indeed considerable with decreasing block size and this verifies our theory.

4.2.2 Memory

As discussed in Section 2, deduplication, albeit effective in compressing VMI contents, comes at the cost of the need to keep a deduplication table in memory for fast access to data blocks on disk. We have established in the previous section that smaller block sizes may lead to better compression ratios, but resulting in bigger deduplication tables. Figure 10 measures the amount of memory consumed by a cVolume with varying block size. For VMI caches, the memory consumption is below 100 MB for block size of 32 KB and

[4]These numbers were reported by Amazon EC2 in their web console in October 2013

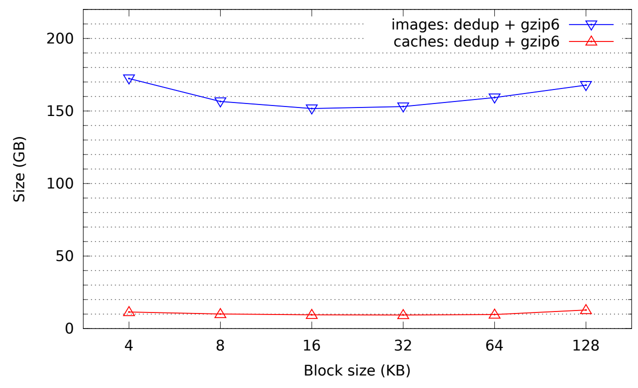

Figure 8: Disk consumption with deduplication and compression.

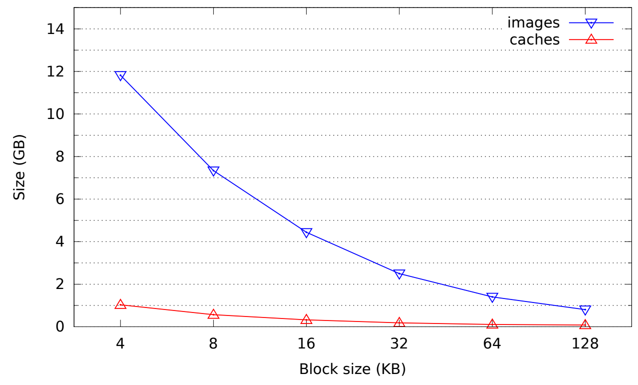

Figure 9: Deduplication table size on disk.

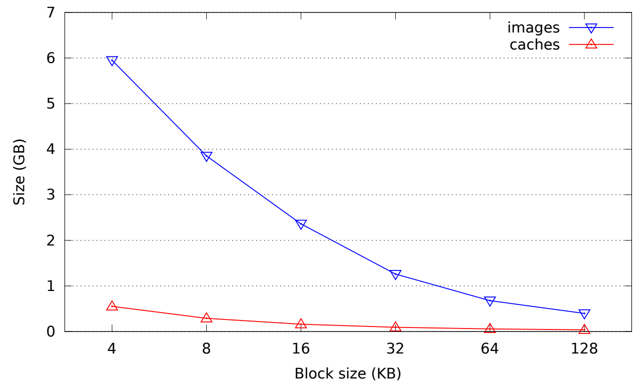

Figure 10: Memory consumption for deduplication tables.

above, and remains relatively small for smaller block sizes. For VMIs however, the memory consumption increases at an alarming rate when reducing the block size. We will explain this behavior further in Section 4.3.1.

4.2.3 Boot time

Deduplication generally slows down reading data from disk due to two reasons. First, each access needs a lookup in the deduplication table. Second, as shown in [14], as data is being deduplicated, adjacent data blocks will end up scattered on the disk. This effect results in random access patterns when reading a chunk in a file that consists of mul-

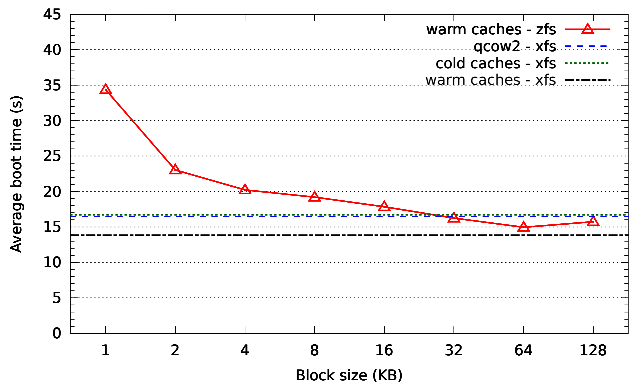

Figure 11: Performance of booting from deduplicated and compressed VMI caches.

tiple sequential data blocks, and conventional disks do not handle random accesses well.

As we reduce the block size, the deduplication table grows in size, resulting in slower lookups. Further, data blocks get even more scattered on the disk as the number of possible hashes increase. Figure 11 shows the effect of varying block size on VM boot performance from a warm cache. For this experiment, we have measured the average boot time of the VMs in our VMI repository when stored on cVolumes with different block sizes. For reference, we have also included the average boot time when booting: (1.) from the local file system, (2.) and creating the cold cache, (3.) from a warm cache stored on the local file system.

Booting from a warm cache increases the booting performance on average by 16% when compared to booting from the VMI stored on the local disk (baseline). To explain this speed up, we need to look at the operation of QCOW2 [22]. Without a VMI cache, each time a VM boots, there are a number of read requests from the CoW image to the VMI. These requests, that are in the form of *(offset, num_sectors)*, usually read from scattered locations around the disk depending on the file system layout of the root partition of the VMI (look at Figure 1 of [36]). Assuming the default cluster size of QCOW2 (64 KB or 128 sectors), when booting from a warm cache, the read requests from the CoW image to the cache image, although usually smaller than 128 sectors, translate to *(offset, 128)* due to the way QCOW2 operates. The Linux page cache, running on the host, caches these sectors. Incidentally, these cached sectors will be needed in a near future, mostly because these other sectors are also part of the boot working set as they are in the warm cache. The end result is a free boost in performance due to reading a large part of the boot working set from the page cache rather than the disk. One could say that Squirrel VMI caches are getting similar effects to prefetching [29] for free.

This boost in performance masks the deduplication and compression overhead for block sizes that are equal and bigger than 32 KB. Also, 128 KB cVolume boots slower than 64 KB cVolume despite the trend. The reason is most likely the default QCOW2 cluster size of 64 KB, which results in read sizes of 64 KB from the KVM process to the cVolume.

4.2.4 Summary

To summarize our findings, we showed in Section 2.2 that smaller block sizes than 64 KB do not yield considerable storage efficiency. In this section, we showed that both disk and memory requirements for the deduplication tables of cVolumes are fairly modest as long as the block size is not too small. We then measured that the cVolume with block size of 64 KB has the best booting performance. Even with deduplication and gzip compression, the average booting performance of Squirrel's cVolume is about 10% better than when the VMI is available *locally* on the compute node. To conclude, the block size of 64 KB is optimal for cVolumes. However, when necessary, we will report on other block sizes in the rest of this section.

4.3 Scalability

Scalability is the most important requirement for a system such as Squirrel that argues for storing *all* the VMI caches in its cVolumes. To demonstrate that Squirrel is scalable, we will first show in Section 4.3.1 that VMI caches share more similarity among each other, than their associated VMIs. We will then extrapolate the disk and memory consumption of cVolumes using standard extrapolation techniques in Section 4.3.2. At the end, we will iterate over the findings of this section.

4.3.1 Cross-similarity of caches

We form the theory that caches share a lot more contents with each other than their associated VMIs. VMI caches usually contain the operating system kernel, boot loader, and some standard services (i.e., a boot working set), whereas VMIs contain a significant amount of libraries and user-level software on top, that may or may not be used during a VM life time. The rationale behind our theory is that the boot working set does not differ much across many VMI caches, unlike the user-level software that shows more variability across different VMIs.

The reason behind better similarity in the boot working set, is related to the fact that VMIs, although different from each other, still are based on a certain number of distributions (e.g., Ubuntu, Debian, CentOS, etc.), and boot working sets of VMIs (i.e., VMI caches) from the same distributions are likely to show high similarity.

If our theory is correct, it shows the scalability of our cVolumes. To prove the theory, we have defined a new metric, called *cross-similarity.* Cross-similarity measures data block sharing across files and is defined as:

$$\text{Cross-similarity} = \frac{\sum^{i \in U} repetition_i}{\sum^{i \in I} |U_i|},$$

where U is the set of all unique blocks, I is the set of all VMIs, U_i is the set of unique blocks in VMI i, and *repetition* of a data block is defined as the number of times a data block appears across "different" files, or 0 if it has never been repeated. In extreme cases, cross-similarity is 1 if the images are the same, and 0 if they share no single data block.

Figure 12 measures the cross-similarity of VMIs and caches. This figure shows that:

1. VMIs do not exhibit a good similarity across each other, whereas caches show a strong cross-similarity, effectively proving the theory discussed earlier.

2. As a result, with high similarity, a new VMI cache, on average, introduces only a few hashes to cVolumes, making Squirrel's cVolumes scalable.

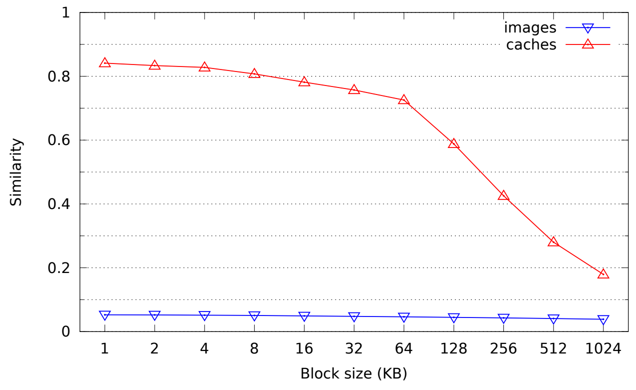

Figure 12: Cross-similarity of VMIs and caches.

3. With smaller block sizes, the similarity increases for caches, but not considerably with block sizes that are smaller than 64 K. This is yet another reason that argues in favor of choosing 64 KB block size for cVolumes.

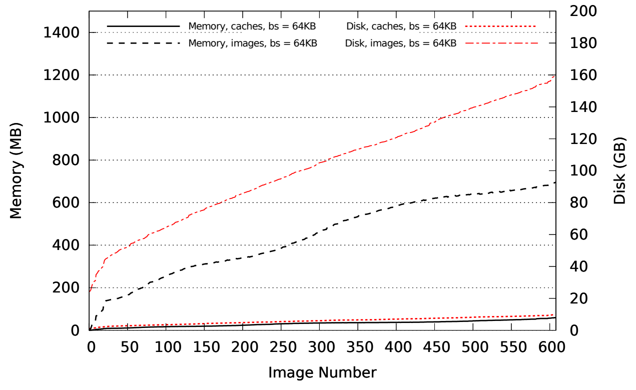

Figure 13: Resource consumption of ZFS when iteratively adding VMIs or caches.

To verify our theory in practice, we added VMI caches iteratively to a ZFS file system with 64 KB block size, and measured the memory and disk consumption when each VMI cache was added to the system. We repeated the same experiment with VMIs. Figure 13 shows the results for both memory and disk. The slopes for VMIs look much steeper than for caches. This means that each VMI is adding relatively more hashes to file system than its associated VMI cache. We use the resource consumption trends for VMI caches as a basis for the extrapolations in the following.

4.3.2 Extrapolations

To understand the extent of Squirrel's cVolume scalability, we need to extrapolate their resource consumption. We will use the disk consumption trend when adding new VMI caches as a basis for extrapolation. We then extend our extrapolation for memory as well.

Disk.

To extrapolate the disk consumption, we need to find a curve that fits the current data points well, and still has a relatively low error when predicting the future. We devised the following approach, common in machine learning, for finding the best fitting curve: (1.) We fed half of our data points to our curve fitting program [10], and asked for the two best non-polynomial fits as well as linear regression (the simplest curve). (2.) We measured root-mean-square error (RMSE) of the three candidate curves for *all* the data points. (3.) We then used the curve type with the lowest RMSE to find the best parameters that fit all the data points. (4.) The resulting curve is then used for extrapolation. The two functions with the best scores with the first half of our data points were Morgan-Mercer-Flodin (MMF) and Hoerl curves:

$$MMF(x) = \frac{a \times b + c \times x^d}{b + x^d}$$

$$hoerl(x) = a \times b^x \times x^c$$

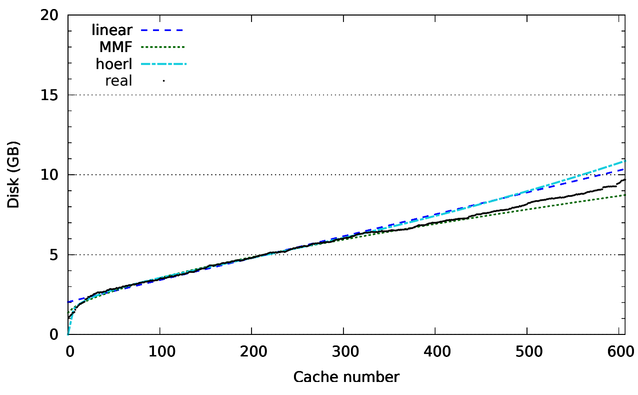

Figure 14: Disk consumption curve-fitting quality (BS = 64 KB).

Figure 14 shows the fitted curves when training with half of the data points. Visually, all curves seem to have a close estimate. MMF seems to be underestimating, while Hoerl and linear regression seem to be overestimating. To find the best fit, we need to calculate the RMSE with all our data points as explained earlier.

Table 3 shows the RMSE of the curves for various block sizes. According to the table, the linear regression has the lowest RMSE and is thus the winner.

Table 3: RMSE of various curves that estimate disk consumption.

Block size	Linear	MMF	Hoerl
128 KB	0.04	0.04	0.08
64 KB	0.03	0.04	0.04
32 KB	0.02	0.04	0.04
16 KB	0.02	0.05	0.03

With linear regression producing a good fit, this time we trained it with all our available data points. Figure 15 shows the resulting extrapolations for different block sizes. With the chosen 64 KB block size for cVolumes, we can store 1200+ caches in about 18 GB of disk space. The extrapolation in Figure 15 continues till 3000 VMI caches, but after 1200 (the vertical line in Figure 15) our curve fitting approach does not guarantee a small RMSE.

Memory.

We repeated the same exercise to extrapolate the memory consumption. MMF and Hoerl functions won good scores for

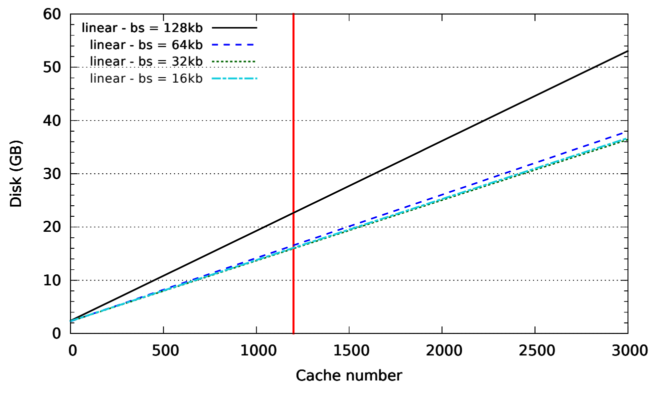

Figure 15: Extrapolation of disk consumption.

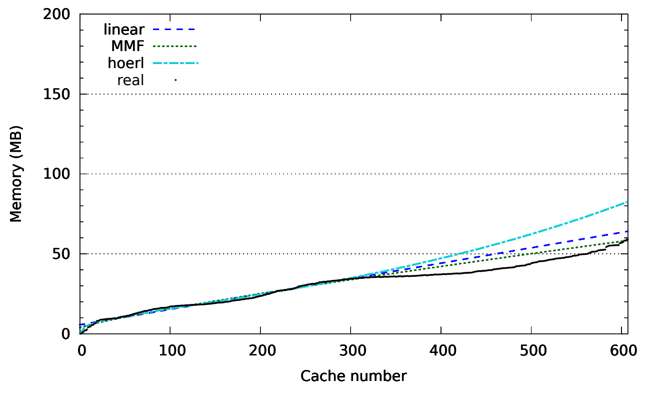

Figure 16: Memory consumption curve-fitting quality (BS = 64 KB).

memory consumption as well. Figure 16 shows the curves when trained with half of our data points. Visually, this time all three curves have a small overestimation, with MMF having the smallest. Looking at RMSEs in Table 4, except for 16 KB block size, MMF seems to estimate memory consumption the best, and specifically with 64 KB.

Table 4: RMSE of various curves that estimate memory consumption.

Block size	Linear	MMF	Hoerl
128 KB	0.21	0.14	0.15
64 KB	0.20	0.02	0.95
32 KB	0.44	0.26	1.03
16 KB	0.61	1.13	0.28

We then trained MMF with all our data points for memory consumption. Figure 17 shows the extrapolations with different block sizes. According to the extrapolation for 64 KB block size, Squirrel cVolumes only consume about 85 MB of memory for the deduplicating 1200+ VMI caches; a very modest requirement.

4.3.3 Summary

In Section 4.3.1, we have shown that VMI caches have a very good cross-similarity. This entails that adding new VMI caches to cVolumes does not result in too many new hashes on average, and makes our cVolumes scalable. We then extrapolated resource consumption for Squirrel's cVolumes beyond caching 1000 VMIs in Section 4.3.2, and showed

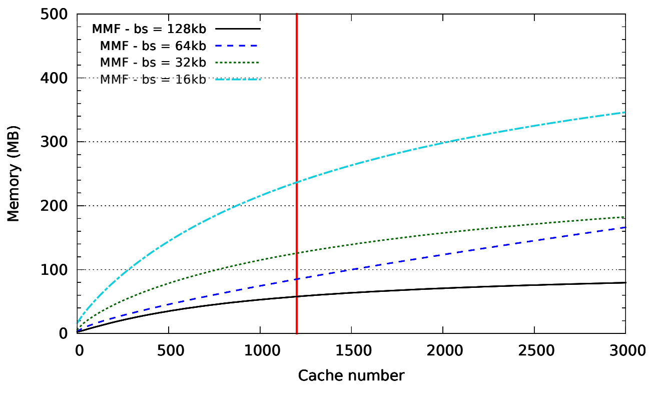

Figure 17: Extrapolation of memory consumption.

that cVolumes require modest disk and memory resources in such a scale, for example 18 GB disk and 85 MB memory for storing 1200+ VMI caches.

4.4 Network transfer size

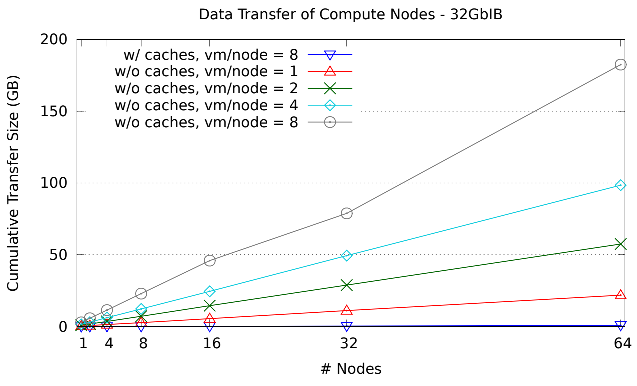

Figure 18: Network transfer size with scaling the number of nodes and the number of VMs per node.

In this section, we will look at the amount of network transfers of compute nodes when starting VMs at scale. We assigned 64 of DAS-4/VU nodes as compute nodes, and 4 nodes as storage nodes. On the storage nodes, we ran glusterfs, an off-the-shelf parallel file system. We configured glusterfs with two levels of striping and two levels of replication, to have a good random access performance (over four disks), and fault tolerance (tolerating one disk failure in each of the two groups). We measured the amount of network transfers at compute nodes once with Squirrel and once without any caching. Since Squirrel is the first system that caches all the bits that are needed for all VMI startups, the amount of network transfers of all previous VMI distribution systems will be in between the results shown.

Figure 18 shows the aggregated amount of network transfers for VM startup at compute nodes.[5] We are scaling the number of compute nodes, and the number of VMs per compute node. At each node, each VM is booting from a different VMI. Without Squirrel, the amount of network

[5]Results shown are for Infiniband. Results for 1 GbE are omitted for brevity as they are essentially the time despite a little smaller per-packet overhead.

transfers increase as there are more VMs starting up in the cluster. In the extreme case, with 512 VMs (64 nodes times 8 VMs), the aggregated transfer at compute nodes is about 180 GB, happening in a relatively short period of time, and disturbing any existing VM that is using the network. With Squirrel, however, compute nodes do not need to do any network I/O, even when executing 8 VMs on each node from different VMI sources.

4.5 Summary

We have evaluated the effects of placing Squirrel's VMI caches on all compute nodes of a data center using the set of 600+ community images from Windows Azure. All tests have shown a block size of 64 KB to be a sweet spot at which disk and memory consumption is acceptably low while boot times are not slowed down by using a compressed and deduplicated file system instead of a plain XFS volume. With block size of 64 KB, Squirrel's cVolumes only consume 10 GB of disk space, and 60 MB of memory for storing VMI caches for *all* Windows Azure community VMIs. Further, the average booting time over Squirrel's cVolumes, even though they are compressed and deduplicated, is reduced by 10%, compared to when the VMIs are available *locally* at compute nodes, due to the caching effect explained in Section 4.2.3.

We have extrapolated our results to larger numbers of VMI caches to be stored, and identified the need for 18 GB disk and 85 MB memory for storing 1214 VMI caches. Given current hardware, these amounts are very modest, if not to say negligible. We consider it likely that in the near to midterm future, even storing several thousand VMI caches per node can be sustained by available node hardware.

To complete our evaluation, we measured the network traffic necessary for concurrently booting up to 512 VMs on 64 nodes, comparing Squirrel to the absence of VMI caches. Here, Squirrel showed the need for zero network traffic, and hence no interference at all with other user network traffic.

5. RELATED WORK

We distinguish related work to Squirrel in three distinct, but overlapping categories: (1.) Systems that implement some sort of VMI caching or storage on the compute nodes. (2.) Systems that focus on scalable distribution of VMI contents. (3.) Systems that perform deduplication and/or compression for efficient VMI storage. We will discuss each category separately, and make comparisons to Squirrel or discuss how Squirrel complements an existing system.

5.1 Storing VMI contents on compute nodes

There are a number of systems that aim at minimizing the amount of network transfers by means of caching VMI contents. OpenStack's [17] Glance API server has the ability to cache VMIs. It is possible to run the Glance API server on many compute nodes to have cached VMIs on many locations. Compared to Squirrel, Glance API servers cannot cache many multi-GB VMIs.

The Liquid file system [44] is designed for VMI distribution and has an architecture similar to Squirrel. Liquid keeps a cache of deduplicated VMI contents on each compute node. The caches transfer data among each other and from storage nodes as necessary. Squirrel is different from Liquid in two important aspects: (1.) Squirrel's VMI caches are isolated from each other: The VMI blocks needed for starting one VM do not evict boot working set blocks of different VMIs. (2.) Squirrel's ccVolumes are fully replicated. They do not need to transfer data between each other over the network, reducing the variations in network bandwidth observed by other VMs.

VMThunder [42] is a VMI distribution engine that provides scalable, on-demand P2P streaming, and a per-VMI caching functionality similar to that of Squirrel. The evaluation in [42], however, does not consider the case with many VMIs, common in today's data centers. Compression mechanisms, absent in VMThunder, are necessary for scaling the caches to large numbers of VMIs.

Nicolae et al. [28] stripe VMI chunks across 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 [29] by means 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. BlobSeer [27], the file system that they have built upon, also keeps a cache of the most recently accessed chunks, very much like Liquid. Our comparison with Liquid holds here as well.

VDN [33] 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. Squirrel solves the problem that VDN is addressing by efficiently storing all VMI caches in its ccVolumes.

Ming et al. [43] 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 [7]. In contrast, Squirrel's VMI caches provide a clean caching abstraction at VMI-level. This makes it possible to perform resource accounting per VMI. Further, Squirrel's ccVolumes persistently store all the blocks needed for booting from all VMIs.

In general, Squirrel's cVolumes can be used as a replacement to improve the caching strategy used by the systems discussed in this section.

5.2 Scalable distribution of VMI contents

There is an increasing demand for scalable VM startup as cloud computing is being adopted for computationally demanding areas, such as high-performance computing. A number of studies have looked at scalable startup of VMs:

5.2.1 *Peer-to-peer approaches*

Peer-to-peer networking is a common technique for transferring a single VMI to many compute nodes [8, 31, 40]. 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. VMTorrent [36] combines on-demand access with peer-to-peer streaming to reduce this delay significantly.

While peer-to-peer transfer is a good match to scalable content transfer in slower networks, it uses substantial network resources to deliver the VMI to the compute nodes. Squirrel, in contrast, caches all data blocks of all VMIs needed for booting, and can save large amounts of network transfers as we have shown in Section 4.4.

LANTorrent [30], 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. LANTorrent, however, introduces startup delay, as it needs to transfer the multi-GB VMI through the network, most of which never gets accessed during the life of a VM. LANTorrent, however, is a good candidate to transfer our small VMI caches to Squirrel's ccVolumes during VMI registration.

5.2.2 *IP Multicasting*

IP multicasting is another approach for scalable data transfer from one-to-many nodes. Here, we will discuss related work that use multicast as a mechanism to deliver VMIs.

Schmidt et al. [38] use Unionfs, a stacked file system used for VMIs. 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 multicast, they achieve short startup delays. Squirrel only relies on multicast for offline transfer of VMI caches as part of VMI registration. Further, 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 if necessary.

Haizea [39] is a lease-management architecture that combines leasing VMs for batch execution. Haizea addresses the problem of scalable deployment by means of multicasting the complete VMI to the compute nodes and caching it there. Squirrel can be used alongside Haizea to reduce the deployment time significantly, and remove the burdon of cache management from Haizea.

Multicasting has been used for cloning the VM state as well. SnowFlock [20] can start many stateful worker VMs in less than 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 efficient, SnowFlock introduces change in all layers of the system, at the VMM, and the VM, down to the application. VMScatter [9] is a similar system, but less intrusive, and also less efficient in terms of scale.

5.3 Compressing VMI contents

Squirrel's cVolumes store VMI caches efficiently by exploiting deduplication and compression techniques in an off-the-shelf ZFS file system [4]. There is a strong body of research on deduplication of VMIs.

Jin et al. [19] performed a detailed study on the effectiveness of deduplicating VMIs. From their findings, the fact that fixed-size chunking works equally well (or sometimes even better) when compared to variable-sized chunking is relevant to our choice of using ZFS that employs fixed-size chunking. Their results have been independently verified over a large VMI repository in [18]. Jin et al. also report similar numbers to ours when looking at similarity between different VMIs. In Section 4.3, we showed that VMI caches show a much better similarity among each other than VMIs, a fact that makes Squirrel's cVolumes scalable.

LiveDFS [26] is a file system aimed at VMI deduplication. Compared with ZFS, LiveDFS' deduplication tables consume less memory, making LiveDFS suitable for deployment on commodity servers. Squirrel's VMI caches, however, have a high cross-similarity, and as a result they only need modest memory requirement for deduplication. We thus decided to use ZFS, which is of commercial quality.

Deduplication of disk content has been used to improve the effectiveness of the page cache. Content-based block caching [24] improves the overall system performance by reducing redundancy of the page cache, and avoiding some of the writes on disk that have the same content. Garces-Erice et al. [15] use a similar approach to improve the performance of the page cache in storage servers that serve IaaS compute nodes. Their approach makes sure that data blocks that are shared between many VMIs are less likely to be evicted from the page cache. In our previous work with VMI caches [34], we also made better use of storage servers' memory by storing VMI caches in a *ramfs*. With Squirrel, however, we are storing all the important blocks at compute nodes in order to completely eradicate the VM startup load on storage nodes.

VMFlock [1] is a cross-cloud co-migration system that transfers a set of VMIs, from one cloud to another. VMFlock uses deduplication opportunities among the set of VMIs to be migrated, as well as the VMIs already available in the destination cloud, to significantly reduce the amount of data that needs to be transferred between clouds. VMI caches are generally small ($O(100\,\text{MB})$), but due to their high cross-similarity, the information that each new one adds to Squirrel's cVolume (i.e., cVolume diff) is even smaller ($O(10\,\text{MB})$). Similar to VMFlock, we only need to transfer a cVolume diff, instead of the whole VMI cache.

Coriolis [6] is a deduplication management system that tries to efficiently cluster VMIs based on their similarity. The grouping of VMIs with similar content results in high deduplication ratio within a VMI cluster that can be stored on separate locations. Similar to Coriolis, Rangoli [25] tries to find similar files to maximize the amount space reclamation when migrating a set of files. The scalability of Squirrel's cVolumes gives us the luxury of storing all VMI caches, without worrying about space reclamation.

6. CONCLUSIONS

The promise of elastic cloud computing is instantaneous availability of virtual machines (VMs). In practice, however, users often have to wait several or even a few tens of minutes until they can actually use their requested VMs. An important factor of this delay is the actual VM boot process that is slowed down by the need to transfer bulky, multi-GB virtual machine images (VMIs) from storage nodes to the selected compute nodes.

In previous work [34], we proposed using a VMI cache on the compute nodes that contains the boot working set, removing the need for network transfers while booting VMs. We showed that VMI caches resolve scalability problems of VM startup. Whereas VMI caches work well, they need to be present and "warm" before a VM starts up.

This work is based on the observation that the many existing VMIs are mostly user customizations of only a few types of operating systems and OS distributions. We have shown that VMI caches (the boot working sets) have high cross-similarity among each other, hence lending themselves well for deduplication-based storage. Combined with compression, the storage of *all* VMI caches on compute nodes becomes possible. Thus, instead of studying cache replacement policies and/or cache-aware VM scheduling, we propose a fully replicated storage design for caching *all* VMIs of an IaaS cloud on *all* the compute nodes.

We have presented Squirrel, a concrete implementation of this design using VMI caches and the ZFS file system. Squirrel can store large amounts of VMI caches within a deduplicated and compressed file system, on the local disks of all compute nodes. We name this approach *scatter hoarding* after the rodent approach for creating many, small food caches.

Our evaluation using all 600+ community images from Windows Azure shows that Squirrel is able to store VMI caches for the overall 16.4 TB of VMIs within 10 GB of disk and 60 MB of main memory, on all compute nodes. We consider these requirements to be rather negligible on current hardware. We have then extrapolated these requirements to the storage needs for thousands of VMI caches and found confirmation that our approach indeed scales to such large numbers on current or near-future hardware.

To summarize, Squirrel completely removes the need for network transfers towards compute nodes when booting virtual machines, either from storage nodes or from other compute nodes. Hence, Squirrel enables large-scale, public IaaS clouds to provide dynamic VM startup purely within the time it takes to boot the virtual OS itself, which is typically tens of seconds, rather than within several minutes as it is common today. This advantage especially helps for dynamic scaling of (e.g., web) applications, helping to close the gap towards truly elastic computing infrastructures.

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, and HPDC's anonymous reviewers for providing valuable feedback on the earlier version of this paper.

7. REFERENCES

[1] S. Al-Kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu. VMFlock: Virtual Machine Co-migration for the Cloud. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 159–170, 2011.

[2] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`, 2006. [Online; accessed 22-01-2014].

[3] ec2_upload_bundle. `http://docs.aws.amazon.com/AWSEC2/latest/CommandLineReference/CLTRG-ami-upload-bundle.html`, 2006. [Online; accessed 22-01-2014].

[4] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. *The SNIA Software Developers' Conference*, 2008.

[5] D. Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[6] D. Campello, C. Crespo, A. Verma, R. Rangaswami, and P. Jayachandran. Coriolis: Scalable VM Clustering in Clouds. In *Presented as part of the 10th International Conference on Autonomic Computing*, pages 101–105, 2013.

[7] 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.

[8] 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.

[9] L. Cui, J. Li, B. Li, J. Huai, C. Ho, T. Wo, H. Al-Aqrabi, and L. Liu. VMScatter: Migrate Virtual Machines to Many Hosts. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 63–72, 2013.

[10] CurveExpert Professional. `http://www.curveexpert.net/products/curveexpert-professional`. [Online; accessed 24-01-2014].

[11] DAS-4 clusters. `http://www.cs.vu.nl/das4/clusters.shtml`. [Online; accessed 24-01-2014].

[12] M. Dutch. Understanding data deduplication ratios. *SNIA Data Management Forum*, 2008.

[13] H. Fernandez, G. Pierre, and T. Kielmann. Autoscaling Web Applications in Heterogeneous Cloud Infrastructures. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, Mar. 2014.

[14] L. Garces-Erice and S. Rooney. Scaling OS Streaming Through Minimizing Cache Redundancy. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops*, ICDCSW '11, pages 47–53, 2011.

[15] 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.

[16] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems*, 2010.

[17] K. Jackson. *OpenStack Cloud Computing Cookbook*. Packt Publishing, 2012.

[18] K. R. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei. An Empirical Analysis of Similarity in Virtual Machine Images. In *Proceedings of the Middleware 2011 Industry Track Workshop*, number 6 in Middleware '11, pages 6:1–6:6, 2011.

[19] K. Jin and E. L. Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *Proceedings of SYSTOR 2009: The Israeli*

Experimental Systems Conference, number 7 in SYSTOR '09, pages 7:1–7:12, 2009.

[20] 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.

[21] M. Mao and M. Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *5th International IEEE Conference on Cloud Computing*, CLOUD '12, pages 423–430, 2012.

[22] M. McLoughlin. The QCOW2 Image Format. `http://people.gnome.org/~markmc/qcow-image-format.html`, 2008. [Online; accessed 24-01-2014].

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

[24] 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.

[25] P. Nagesh and A. Kathpal. Rangoli: Space Management in Deduplication Environments. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, 2013.

[26] 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 *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '11, pages 81–100, 2011.

[27] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie. BlobSeer: Next-generation data management for large scale infrastructures. *Journal of Parallel and Distributed Computing*, 71(2):169–184, 2011.

[28] 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.

[29] 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.

[30] Nimbus Project. LANTorrent. `http://www.nimbusproject.org/docs/current/admin/reference.html#lantorrent`, 2010. [Online; accessed 27-01-2014].

[31] 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.

[32] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 297–306, 1993.

[33] 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.

[34] K. Razavi and T. Kielmann. Scalable Virtual Machine Deployment Using VM Image Caches. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, number 65 in SC '13, 2013.

[35] K. Razavi, L. M. Razorea, and T. Kielmann. Reducing VM Startup Time and Storage Costs by VM Image Content Consolidation. In *1st Workshop on Dependability and Interoperability In Heterogeneous Clouds*, Euro-Par 2013: Parallel Processing Workshops, 2013.

[36] 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.

[37] rsync. `http://rsync.samba.org`. [Online; accessed 22-01-2014].

[38] 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.

[39] B. Sotomayor, K. Keahey, and I. Foster. Combining Batch Execution and Leasing Using Virtual Machines. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 87–96, 2008.

[40] 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.

[41] ZFS on Linux. `http://zfsonlinux.org`. [Online; accessed 24-01-2014].

[42] Z. Zhang, Z. Li, K. Wu, D. Li, H. Li, Y. Peng, and X. Lu. VMThunder: Fast Provisioning of Large-Scale Virtual Machine Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 99, 2014.

[43] 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.

[44] X. Zhao, Y. Zhang, Y. Wu, K. Chen, J. Jiang, and K. Li. Liquid: A Scalable Deduplication File System for Virtual Machine Images. *IEEE Transaction on Parallel and Distributed Systems*, 2013.