Prebaked µVMs: Scalable, Instant VM Startup for IaaS Clouds

Kaveh Razavi
VU University Amsterdam
k.razavi@vu.nl

Gerrit van der Kolk
VU University Amsterdam
g.vander.kolk@vu.nl

Thilo Kielmann
VU University Amsterdam
thilo.kielmann@vu.nl

***Abstract*—IaaS clouds promise instantaneously available resources to elastic applications. In practice, however, virtual machine (VM) startup times are in the order of several minutes, or at best, several tens of seconds, negatively impacting the elasticity of applications like Web servers that need to scale out to handle dynamically increasing load.**

VM startup time is strongly influenced by booting the VM's operating system. In this work, we propose using so-called *prebaked µVMs* to speed up VM startup. µVMs are snapshots of minimal VMs that can be quickly resumed and then configured to application needs by hot-plugging resources. To serve µVMs, we extend our VM boot cache service, Squirrel, allowing to store µVMs for large numbers of VM images on the hosts of a data center. Our experiments show that µVMs can start up in less than one second on a standard file system. Using 1000+ VM images from a production cloud, we show that the respective µVMs can be stored in a compressed and deduplicated file system within 50 GB storage per host, while starting up within 2–3 seconds on average.

I. 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 (VMs), according to the needs of an application or Web service. In practice, however, users face highly variable VM startup times in the range of several minutes, depending on the actual system load [1], [2].

Such long VM startup times have a strong negative impact on the elasticity of cloud applications. For example, a Web server facing spontaneously increasing load needs to scale out by adding hosts. If such added machines become available only after several minutes, the Web site might appear unreachable meanwhile, leading to unsatisfied clients and loss of revenue for the site operator. Alternatively, such a Web site would need to overprovision its cloud servers, causing latent costs of leasing additional, standby servers that were supposed to be avoided by using the cloud infrastructure in the first place. The question of how much to overprovision is also not straightforward to answer, making the design of autoscalers and capacity managers (e.g., [3], [4], [5], [6], [7]) more difficult. In this paper, we try to tackle the problem directly, and develop a solution to minimize the VM starting time.

Two important factors contribute to VM starting time: the transfer of the VM image from a storage node to the host, and the actual booting process of the VM. In previous work [8], we presented the Squirrel system of boot caches that stores the boot working sets for all VM images of a data center on all hosts. We have shown that Squirrel can scale up to storing thousands of boot caches by means of compression and deduplication. Using Squirrel, no VM image data needs to be transferred to hosts during VM startup, reducing typical VM boot times from several minutes to tens of seconds.

In this work, we address the other cause of slow VM startup: the VM boot process itself. Booting an operating system is a lengthy but repetitive computation. We propose to pre-boot a VM image in advance and to take a snapshot that can be resumed from whenever a user wishes to start one or more VMs from this image. The reason why VMs are usually booted on demand (rather than resumed) is that their operating system state reflects the actual resources available to the VM, like the CPU cores, the memory, or the disks. For each VM image and (virtual) machine type of a cloud provider, a VM snapshot would be needed, leading to a combinatorial explosion and to non-scalable storage requirements.

To avoid such combinatorial explosion, we propose to use *prebaked µVMs* instead. A µVM is a snapshot of a booted VM with minimal hardware resources, in our case a single CPU core and 512 MB of RAM. µVMs are relatively small, lowering storage requirements. We are using hardware hot-plugging for adding resources of larger machine types (CPU cores and memory) as part of the resume process, at the actual VM startup time.

The service for resuming VMs and hot-plugging resources is called the *VM bakery*. We have extended our Squirrel storage system in order to serve µVMs instead of boot cache images. Combining µVMs, Squirrel, and a carefully designed VM snapshot format, we can achieve instantaneous VM startup. Experiments with µVMs created from 1000+ community images of Windows Azure show that on average, we can start VMs in under a second using a standard file system, and within 2–3 seconds using a compressed file system consuming only 50 GB of storage space per host.

This paper is organized as follows. Section II provides background on the operating system boot process and how it can be improved for fast VM startup. Section III explains the fundamental properties of µVMs and the *VM bakery*. Section IV presents the actual implementation, and the integration with Squirrel. In Section V, we evaluate both VM startup times and storage scalability. Section VI discusses related work. Section VII concludes.

II. Booting virtual machines

Every time a tenant requests a VM, the provider needs to start the virtual machine monitor (VMM) with the requested resource parameters (i.e. number of cores, amount of memory, etc.) on an appropriate host. The VM then goes through various stages in the boot process, building the operating system's state *(OS state)*. Once the OS state is created, the VM becomes ready for running the tenant's applications.

Even when the VMs' boot working sets are local to the selected host, the boot process can take tens of seconds [9], reducing the potentials for elastic cloud applications. In this work, we propose to reuse the OS state. For doing so, we need to understand why providers usually need to boot VMs for every VM startup request.

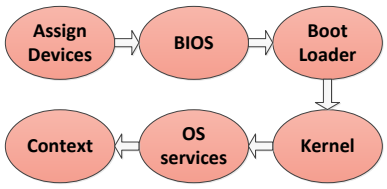


Fig. 1. Normal VM start up.

Figure 1 shows a rough sketch of the VM start up process after the host is selected. First, the VMM is invoked with the proper resources (CPU cores, memory), associated with the requested instance type (e.g. small, medium, large, etc.) to be handed to the VM. After the VM passes through BIOS that detects some of the available hardware, specially bootable storage, the appropriate bootloader starts. After a configurable timeout, the OS kernel is loaded into memory, and starts initialization of various devices such as cores, memory, disk drives, and so on. After that, the OS starts the configured services such as SSH (or RDP on Windows). Finally, in the last step, the OS executes a set of given commands that configure the VM to run the requested applications. The last step is typically referred to as contextualization. Contextualization makes the VM unique from other VMs that have started from the same base VM image.

Depending on the configuration of the VM, any of these steps can add to the total boot time. Among these steps, the only two that make a VM unique, are the initial device assignment and the contextualization. Contextualization is the last step, and takes little time for configuring the VM's network interface and starting the user applications. The important step that requires the providers to boot the VM "from scratch" is the actual device assignment that happens at the beginning.

Previous attempts for addressing this issue involve keeping VMs with different hardware resources running at all times, and contextualizing a match according to the VM image and the instance type of a user's VM startup request [10]. Unfortunately, this approach does not scale due to different possible instance types,[1] and the number of VM images that goes into the thousands. Other proposals suggest suspending the entire VM's state, including the state of the user application, and resuming it when necessary [11], [12], [13]. While these solutions work for a specific user, they do not scale for an entire cloud infrastructure since the VMs are already specialized for a particular user.

None of these solutions consider the fact that the OS state is initially the same regardless of the VM configuration. We propose a generic and scalable solution to capture and reuse this state.

[1]For example, in December 2014, Amazon EC2 provides more than 20 instance types.

III. µVMs and the VM bakery

We introduce µVM, a building block for reusable OS state in Section III-A. We then describe VM bakery in Section III-B, a service that runs on the hosts, and takes as input a µVM, and outputs a VM according to the requested user configuration. We describe the requirements for scalable host-side caching of µVMs in Section III-C, and we discuss the security implications of reusing OS state in Section III-D.

A. µVM

As discussed in Section II, the main reason that forces the providers to boot VMs for every single VM startup request is the initial device assignment. To relax this requirement, we propose starting the VM with *minimal* resources. In our experience, 512 MB of memory is more than enough to accommodate the initial state for both Linux and Windows OSes. Once the VM is booted, we take a snapshot of the VM. This snapshot contains the state of disk, memory, and devices at the moment when the VM is booted. We call this snapshot a µVM, and will explain in Section IV how this can be done in a reusable manner.

µVMs have some interesting properties:

1) µVMs can be arbitrarily resized to machine types with larger resources; only one µVM is necessary for every VM image registered at a provider.
2) µVMs are small in size, allowing for scalable host-side caching. With slight modification of the snapshot format, we achieve reasonable deduplication ratios, resulting in even more scalable host-side caching with slight increase in the µVMs' resume times.
3) Standard fast resume techniques can be applied to µVMs, resulting in fast VM startup times.

We exploit the first and second properties for scalable storage of µVMs at the hosts, and the third property for fast VM startup. An alternative approach would be to start with a larger VM and reducing its size by means of hot-unpluging. However, due to in-kernel data structures that are necessary for managing the extra (unused) memory, the storage of the memory snapshots would become inefficient.

B. VM Bakery

µVMs by themselves are not suited for running user applications as they lack resources. We hence need a mechanism for adding resources to µVMs.

Entire machine virtualization cleanly decouples hardware from the OS that runs in the virtual machine. While this decoupling provides isolation and consolidation possibilities exploited in clouds today, it also allows for more dynamic "physical" resource allocation to guest OSes, termed resource

hot-plugging and hot-unplugging. Realizing the resource management opportunities of resource hot-(un)plugging, VMMs and guest OSes are gradually implementing proper support for various resources such as cores, memory, and disks. The latest versions of QEMU/KVM (a popular open-source VMM) on one hand, and Linux, and Windows (among popular commodity guests), on the other, already support resource hot-plugging, though hot-unplugging is still under progress. (Hot-unplugging is not needed by µVMs and the VM bakery.)

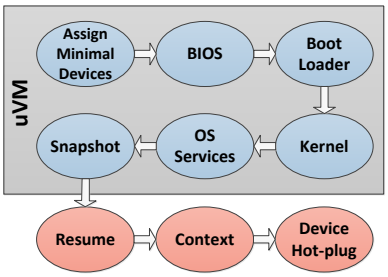


Fig. 2. VM Bakery: VM startup using µVM and resource hot-plugging.

We have implemented a service, called the *VM bakery*, that runs on the hosts, and employs resource hot-plugging to extend a µVM to the tenant-requested size. Figure 2 shows how VM startup looks like with VM bakery. The blue steps are precomputed and are what constitutes a µVM, and the red steps happen during the actual VM startup. Upon a VM startup request, the VM bakery resumes the µVM and contextualizes it. Immediately after contextualization, the VM bakery starts hot-plugging core and memory resources by interacting with the VMM process in control of the µVM. Effectively, the VM bakery moves the device assignment to the very end of the VM startup process, making it possible to reuse the precomputed OS state of µVMs. We perform contextualization *before* hot-plugging to make the VM available to the user as soon as possible. We explain how it is possible to contextualize a µVM in Section IV-C.

C. Host-side Caching of µVMs

For resuming a µVM, only the states of memory and devices are necessary. Hence, we need to consider the efficiency of storing these states for many µVMs. To investigate this, we created µVMs from our Windows Azure repository, consisting of more than 1000 Linux VMs of different flavours, using KVM's default *savevm* facilities [14]. The entire states of all µVMs amounted to 241 GB on an *ext4* partition, which is too large for host-side caching. To reduce this footprint, standard deduplication and compression techniques can be applied.

To understand the potential for deduplication, we stored the µVMs on deduplication-enabled ZFS volumes with varying block sizes and measured the deduplication ratios. The *default format* line in Figure 3 shows the result of the study. To our surprise, the results showed almost no opportunities for deduplication, even on page-granular block size (4 KB). The reason for this turned out to be the complex file format that KVM uses for storing each µVM state file. The default VM state file format contains different sections for different devices, and one of these sections, that is amounting to almost all of the state file, is the µVM's memory. Further, KVM

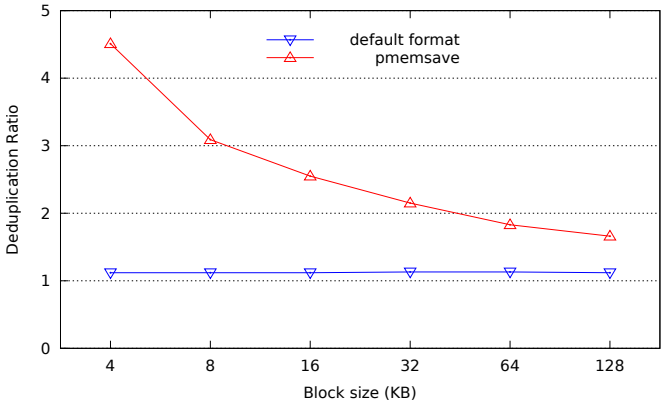


Fig. 3. Deduplication ratios of µVMs from Windows Azure repository with KVM's default state file format and the raw memory contents taken with *pmemsave*.

applies a basic compression routine that stores a single byte for pages that contain that specific byte (e.g., zero pages), and reflects this information in the state file [15]. These "meta-state" information and possible per-page misalignment of guest's physical memory, make low-cost fixed block-based deduplication ineffective.

To understand possible improvements to the deduplication ratio, we used QMP's *pmemsave* command to retrieve the µVMs' raw physical memory contents [16], bypassing KVM's complex snapshot file format. The gross size of these memory dumps is 512 GB (memory size of each µVM: 512 MB, multiplied by the number of µVMs), but simple sparse file support (not storing *zero* pages) of the *ext4* file system already reduces it to only 271 GB. In Figure 3, the *pmemsave* line shows that simplifying the µVM's memory state allows for better storage scalability due to improved deduplication possibilities when stored with varying block sizes in a deduplicating ZFS. We show how we have adapted KVM's snapshot file format according to these findings in Section V-C.

D. Security Considerations

Security is beyond the scope of this work. However, we briefly discuss the security implications of reusing the OS state.

Operating systems typically use Address Space Layout Randomization (ASLR) in combination with other techniques such as stack canaries, and non-executable stack and heap to protect against exploits that target vulnerable applications. By reusing the OS state, we may allow an attacker to defeat ASLR by starting VMs on the target provider to read the address space layout (ASL) of target processes. Since most OS kernels themselves do not employ ASLR, the only affected and important processes are OS services that run as part of the boot process such as *httpd* or *ssh*.

To restore ASLR to these important OS services, their ASLs should be re-randomized. A pragmatic approach involves restarting these processes after the VM starts. A more elegant approach is live layout re-randomization of these address spaces [17], [18]. We assume that such techniques will be used in combination with our µVMs.

IV. Implementation

Our implementation of µVMs allows providers to reuse the precomputed ("prebaked") OS state by several, concurrently started VMs during the VM startup process based on VM resume. To make this possible, the VMs created from the same µVM should not modify any µVM state, including the state of memory and devices, and the µVM's view of its virtual disk. Instead, their modifications should be local and visible only to themselves. We provide this property in a similar fashion to Linux process *fork*, using copy-on-write, both for memory and disk. The end result is *reusable* µVMs.

To allow for reusable memory state, as well as efficient host-side caching of µVMs, we needed to introduce changes to QEMU/KVM (Section IV-A). To allow for reusable µVM virtual disks, we used overlay images (Section IV-B). We then discuss the implementation of the VM bakery service (Section IV-C) that uses these µVMs for starting VMs, and how we modified Squirrel, an existing caching system for the efficient storage of µVMs (Section IV-D). Finally, we discuss some of the issues that we encountered during the implementation of this work (Section IV-E).

A. Supporting µVMs in QEMU/KVM

As we showed in Section III-C, raw storage of µVM memory is necessary for efficient and scalable host-side caching of µVMs. To allow µVM resume over raw µVM memory, we modified KVM's state file format to exclude the memory section. Instead, we provide a separate file containing the µVM's memory directly to KVM as a command line parameter at resume time.

The default resume mechanism in KVM reads the entire state file, including the VM's memory, before starting the VM (i.e., eager resume). µVMs are small, thus we can load their memory state in a small time amount. However, to improve the resume time further, we decided to use and modify an implementation of lazy resume [15]. The implementation leverages *mmap* [19] to provide a one-to-one mapping of the VM's raw memory file (stored on disk) in the QEMU/KVM's address space. Once the VM resumes, every access to its memory results in a page-fault. The Linux page-fault handler (on the host that runs KVM) then loads the memory page from the file, and resumes the execution of the VM. Using the raw memory file also satisfies our requirement for having the µVM's memory in a separate file.

We now discuss how we modified the lazy resume implementation to provide copy-on-write semantics for reusable µVMs. The small (O(1 MB)) device state is copied eagerly into the memory during µVM resume, and is hence reusable. The implementation of lazy resume in [15] needs to keep the VM's memory state file synchronized with the current VM's memory to provide support for snapshots. For this purpose, it uses a *shared memory mapping* (i.e., MAP_SHARED [19]). MAP_SHARED allows the changes to the VM's memory to be propagated to the memory state file on disk. The propagation happens eagerly with the *msync* [20] system call during the snapshot process, and also periodically by the Linux kernel in configurable system-wide intervals. This is undesirable for µVMs, since we intend to reuse µVMs memory states.

Fortunately, the alternative, *private memory mapping* (i.e., MAP_PRIVATE) provides us with a private copy-on-write mechanism that is not reflected on the underlying memory state file. Thus, using MAP_PRIVATE makes it possible to use µVMs by multiple VM's at any time without any concurrency issues. Unfortunately, in Linux, once a mapping is defined as private, it is not possible to propagate the updated pages to the underlying backing file. To address this limitation, we use MAP_SHARED/msync during µVM creation, and MAP_PRIVATE during reuse. The snapshot process invoked on VMs started from a µVM goes through the pages in the private mapping, and reflect them on a new memory state file.

B. µVM Storage Overlay

During its boot process, the OS writes data (e.g., logs, temporary files, etc.) to its booting storage device. We call these writes the *boot writing set* of a particular VM. A VM started from a µVM should have a consistent view of its storage device. In other words, we should protect the boot writing set from VMs that are started from a µVM. QCOW2 [21], QEMU's widely used copy-on-write (CoW) image format, provides support for overlays [22] among other things. We use this feature for creating a copy-on-write overlay for keeping the VM's boot writing set.

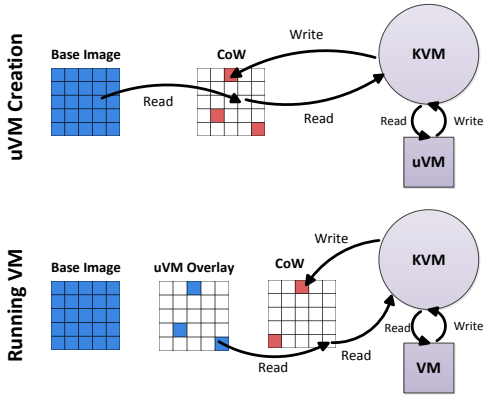


Fig. 4. Creation and reuse of µVM storage overlays.

Figure 4 shows how µVM overlays are created and reused. During µVM creation, we use a normal CoW image over the base VM image of the µVM. Once the µVM is booted, we take a snapshot. At this time, all the boot writing set of the µVM is in the CoW image. We call this CoW image the µVM overlay. Whenever there is a VM startup request, we create another CoW image, and chain it to the µVM overlay, and resume the µVM by passing the (last) CoW as the µVM's virtual disk. With this technique, the reads originating from the VM see the consistent view of the storage device at the time of the snapshot. Further, the writes originating from different VMs do not pollute the µVM overlay due to their copy-on-write nature.

We have previously shown that chaining overlays is a cheap operation in QCOW2 [9], as the small overlay metadata remains in the page-cache.

C. VM Bakery

Our VM bakery service takes as an input the states of devices and memory of a µVM and outputs a contextualized VM with the requested core and memory resources. VM bakery runs on the hosts and has access to a local cache containing the necessary states.

As soon as there is a VM startup request, VM bakery starts the KVM process, and resumes the desired µVM. At this point, the OS running in the VM needs to be contextualized. We implement contextualization in a way that does not require modifications to the guest OS. For this purpose, we decided to contextualize our µVMs using a hot-pluggable hard disk. In parallel to resuming the VM, we create a file with the required contextualization information, and attach it to the µVM. In Linux guests, we use a simple *udev rule* [23] that acts on the hot-plugging event of this special hard disk and contextualizes the µVM based on the information inside it. In Windows guests, we use the *Task Scheduler* [24] to register a "trigger" for hot-plugging this specific disk, and an "action" for contextualization.

Right after contextualization, depending on the requested instance type, VM bakery starts hot-plugging CPU cores, and a memory device with the necessary size through a QMP TCP connection to KVM. These devices are picked up, and initialized by the guest OSes immediately if support is available in their kernel. All recent versions of kernels found in common Linux distributions, and Windows Server editions support core and memory hot-plugging (also referred to as hot-add). While a contextualized µVM can already start executing user applications, we will show the hot-plugging times of new cores and memory of various OSes in Section V-A.

D. Host-side Caching of µVMs with Squirrel

We modified Squirrel [8], a host-side caching infrastructure, to be used by VM bakery. Squirrel is originally designed for scalable host-side caching of VM images. Hence, our modifications for caching µVMs instead were straightforward. We briefly describe the architecture of Squirrel, its VM image register operation, and then discuss the necessary changes for adding support for µVMs.

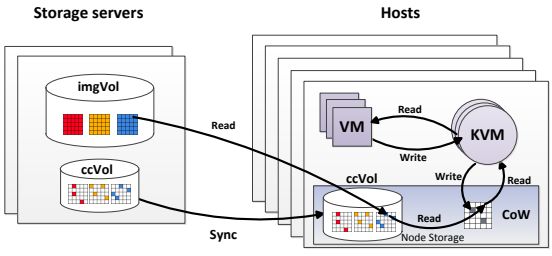


Fig. 5. Squirrel architecture diagram.

1) Architecture: Figure 5 shows the architecture of Squirrel. Squirrel keeps compressed cache volumes (ccVol), running ZFS [25], on the hosts. The volumes store the VMs' *boot reading sets*. Whenever VMs need to start up, they use their boot reading sets from Squirrel's cache volumes rather than reading remotely across the network. This allows for scalable startup of VMs without congesting the network and/or storage servers.

2) Squirrel's Register: Whenever a cloud user registers a VM image, Squirrel creates its boot reading set by booting a VM instance at a storage server, and efficiently replicates it to the VM hosts using ZFS incremental snapshots. A complete description of Squirrel can be found in [8].

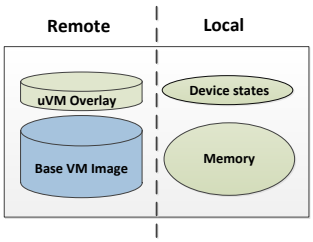


Fig. 6. The organization of a µVM from a host's point of view. The overlay as well as the devices and memory states are µVM-specific.

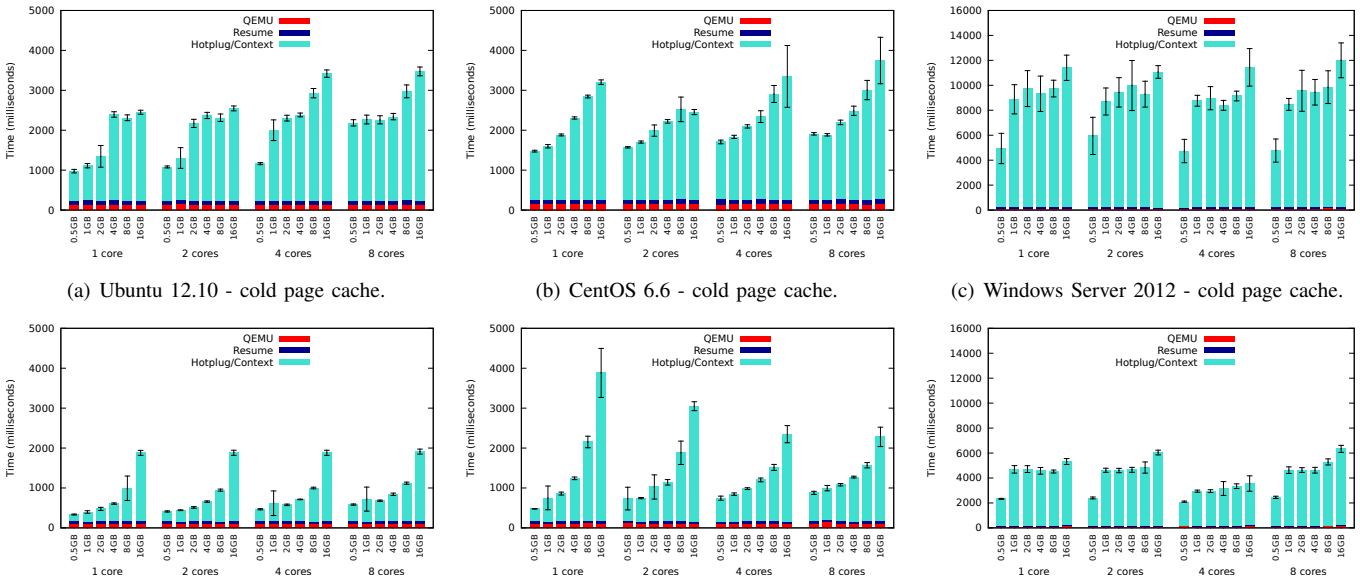
3) µVM Support: So far, our µVM constitutes the states of its memory and devices, and an storage overlay. Since only the states are actively read during µVM resume, we modified Squirrel to replicate these states rather than the boot reading sets. During Squirrel's register operation, we boot a µVM the same way as with the original Squirrel implementation, but instead of performing copy-on-read for creating the boot working sets, we perform copy-on-write for creating the µVM's overlay. Figure 6 summarizes the organization of µVMs from a host's point of view. We study the scalability of the cache volumes in Section V-C.

E. Encountered Issues

We describe some of the issues that we encountered during the implementation of µVM and VM bakery.

1) Lost events: One important issue that we needed to address was "lost" hot-plugging events. VM bakery needs to send hot-plugging events *after* KVM sets up lazy resume and starts the guest VM. Otherwise, if the guest VM is not running yet, these events will be lost. Hence, we needed to synchronize these events with KVM. Fortunately, we found out that KVM sends an end-of-resume event via QMP, so all we needed to do was making sure that we are connected to QMP before starting the µVM resume. For this purpose, we let the KVM process wait on the devices state file using the *inotify* [26] system call. Within the bakery service, after making a successful QMP connection, we create a symbolic link to the devices state file with the same path given to KVM's inotify. KVM picks up this event, and starts the resume. Using this simple mechanism, VM bakery always picks up the event that constitutes a running µVM, and starts hot-plugging the devices.

2) Sluggish (virtual) NIC: We initially implemented the mechanism that informs the µVM about the state change (resumed), by unplugging the NIC cable during µVM creation, and plugging it back right after resume. We then used a *udev* rule for picking up the context file via a configured link-local IP address. Unfortunately, the link state change notification to user-space could take up to tens of seconds after resume. Given



(a) Ubuntu 12.10 - cold page cache. (b) CentOS 6.6 - cold page cache. (c) Windows Server 2012 - cold page cache.

(d) Ubuntu 12.10 - warm page cache. (e) CentOS 6.6 - warm page cache. (f) Windows Server 2012 - warm page cache.

Fig. 7. VM startup time using µVMs of various OSes. The VMs can already start serving their applications as soon their µVMs resume (Leftmost line in each figure). Note the different vertical axis in the Windows Server experiments.

this observation, we decided to contextualize our µVMs using disk hot-plugging, like we discussed in Section IV-C.

3) No memory for new memory: The first version of our µVMs only had 256 MB of memory. We however noticed unstable kernel panics in Linux when hot-plugging memory. The reason turned out to be critical out-of-memory situations in the kernel, due to allocation of data structures needed for the initialization of the hot-plugged memory. We resolved the issue by moving to µVMs with 512 MB of memory.

V. Evaluation

Our evaluation of µVMs consists of three parts. First, we present a detailed analysis of VM startup times using three commodity operating systems, including device hot-plugging (Section V-A). Second, we compare startup times of µVMs and boot reading sets over a large repository of VM images using Squirrel (Section V-B). Third, we study scalability aspects of caching µVMs at the hosts (Section V-C).

For our experiments, we used compute nodes from the DAS-4/VU cluster [27]. The nodes are equipped with dual quad-core Intel E5620 CPUs (8 physical cores), running at 2.4 GHz, with 24 GB of memory, with two Western Digital SATA 3.0 Gbps/7200 RPM/1 TB disks in software RAID-0 fashion, and with a Crucial's C300 M4 256 GB SSD. For all these benchmarks, we used ZFS and ext4 natively [28] on the SSDs. Lazy resume is slow on HDDs due to their poor random access performance, but this can easily be improved by an standard technique that prefetches the µVM's resume working set ([11], [12], [13]). In this work, we are using SSDs for storing the µVMs.

Our test set consists of 1011 community images of Windows Azure, taken between April 2013 and June 2014 for our large-scale experiments in Section V-B and V-C. The repository with this set contains Linux images, mostly Ubuntu, but also CentOS, OpenSuse, Debian, and others. We have discussed the diversity of a smaller version of this repository containing 607 VM images in [8]. Most of these images had grub timeouts that would add tens of seconds to the startup times of their VMs. We removed these timeouts before running our experiments. These timeouts in grub (and the Windows bootloader) are designed to provide a mechanism to choose alternative OSes or boot options. This mechanism is, however, not helpful for cloud VMs that typically employ a single OS. It is worth mentioning that µVMs hide all boot-related timeouts automatically.

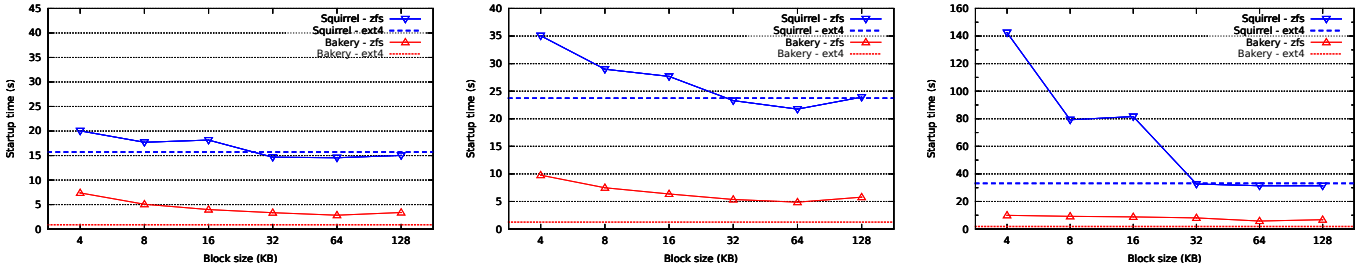
TABLE I. Booting time of various operating systems.

OS	Cold page cache	Warm page cache
Ubuntu 12.10	$\mu = 11.7$ s $\sigma = 0.5$	$\mu = 10.9$ s $\sigma = 0.5$
CentOS 6.6	$\mu = 21.9$ s $\sigma = 0.7$	$\mu = 20.9$ s $\sigma = 0.7$
Windows Server 2012	$\mu = 44.1$ s $\sigma = 2.1$	$\mu = 41.4$ s $\sigma = 2.7$

A. VM Startup Times

To evaluate VM startup improvements, we first measured the boot time of three different OSes over a cold and a warm page cache (repeated ten times) with their VM images available on the local ext4 drive. We expect the boot times of production runs to be in between these two extreme cases. Table I shows the results of this experiment, expressed by mean value μ and standard deviation σ. Note that the VM startup process is highly compute-bound and a warm page cache only slightly improves the VM startup time. We then used the same VM images to create µVMs, and measured the µVM startup times, contextualization, and the time it takes for guest OSes to initialize the new cores and memory.

Figure 7 shows the startup times using µVMs stored on the local ext4 drive, and hot-plugging up to 7 cores (total: 8 cores), and 15.5 GB (total: 16 GB) of memory. The leftmost bar in each graph shows a contextualized µVM that is making



(a) Average µVM startup time. (b) 99 percentile µVM startup time. (c) Maximum observed µVM startup time.

Fig. 8. µVM startup times over a gzip compressed and deduplicated ZFS volume measured over a cold page cache. Note the larger vertical axis in (c).

a network connection to state that it is live. Note that at this moment, the µVM can already start doing work and provides close to peak performance to the user applications ([15]). The results show that with a cold cache, µVMs start up 12x faster for Ubuntu, 15x faster for CentOS, and 9x faster for Windows Server. With a warm cache, µVMs start up 33x faster for Ubuntu, 43x faster for CentOS, and 18x faster for Windows Server. These results hint that an efficient deduplicated and/or compressed page cache (e.g., [29], [30], [31]) can have a significant performance impact on the µVM-based startup times.

The difference between the leftmost bar in each figure and the other bars in the same figure is the time it takes for *all* of the hot-plugged resources to become available. Since the initialization of memory by guest OSes is gradual, we do not expect user applications to consume memory faster than it takes to hot-plug it. The information on the final number of cores and amount of memory is made available as part of contextualization in case user applications need it. The conclusion of this experiment is that while there have been recent efforts to make core hot-plugging/unplugging fast [32], [33], hot-plugging memory seems to be expensive as well, and requires more immediate attention. We are planning to investigate this as part of our future work.

B. Startup Times of Compressed µVMs

µVMs are small; The average size of a µVM from our Windows Azure repository is only 275 MB. This means that a small-scale data center with a small number of VM images, can manage without compression to enjoy VM startup times similar to ones that we showed in the previous section. The same can be done for popular VM images of a larger data center. But for less popular, tenant-provided VM images, compression techniques are necessary for providing scalable host-side caching.

We defer the study of storage gains to the next section, and focus only on startup times here. We configured Squirrel's ZFS volumes with different block sizes (record size in ZFS terms), and for each block size, we registered all the images in our Windows Azure repository to Squirrel. We then measured the startup times of all the µVMs with a cold page cache.

Figure 8 shows the average, the 99 percentile, and the maximum VM startup times that we observed using the original Squirrel (boot caches), and with VM bakery. ZFS compression/deduplication affects the µVM startup more severely than the boot caches. This is most likely due to scattered page faults over the memory state file, resulting in unnecessarily larger read requests to ZFS. Going to smaller block sizes does not resolve this problem due to increased overhead of ZFS. A warm page cache and/or a file system tailored towards this type of workloads could improve the performance considerably.

Regardless of the negative impact of compression, µVMs consistently startup much faster than boot caches: On average with the optimal block size of 64 KB, µVMs start 5x faster, compared to using boot caches. The 99th percentile of µVMs is 4.5x times better, and finally the 100th percentile (i.e., maximum) of µVMs is 5.5x better.

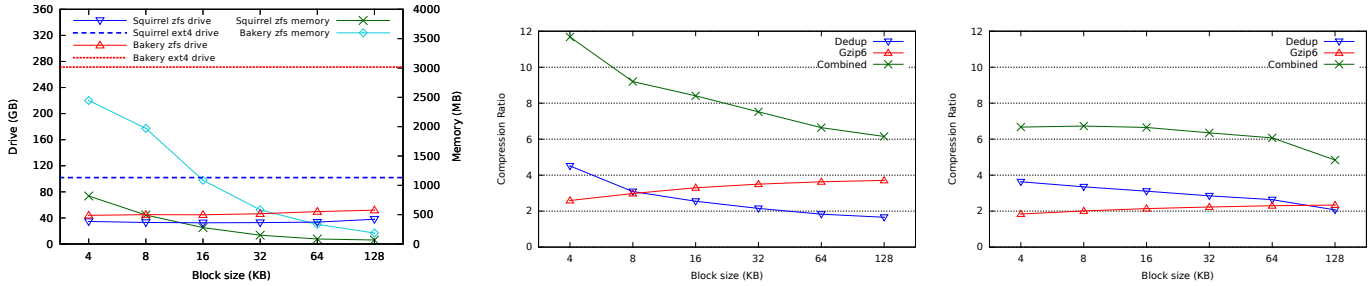
C. Storage Scalability

To measure the efficiency of deduplication and compression on cached µVMs with the modified state file format, we used the statistics reported from ZFS under different block sizes with its gzip6 compression and deduplication turned on. We then compare these numbers with the original Squirrel architecture that stores boot caches.

Our measurements are shown in Figure 9-a. µVMs, when stored in their raw format need about 2.7x more space than boot caches, however when compressed, at e.g. 64 KB block size, they only need 48% more space (49.6 GB vs. 33.5 GB for boot caches). This means that the compression techniques are more effective for µVMs than for boot caches. The compression pipeline first finds duplicates in the written blocks, and then compresses unique (i.e., deduplicated) blocks. *Combined compression ratio* is the multiplication of these two ratios that takes both into account.

Figure 9-b and Figure 9-c show the compression ratios for µVMs and boot caches. Comparing the figures, it is clear that µVMs tend to compress better, and as a result have a better combined compression ratio. The deduplication ratio of about 1.8x (at 64 KB) is still surprisingly high, given the fact that memory locations are much more volatile than disks (boot caches deduplication ratio is 2.6x at 64 KB). According to [34], and our own independent cross-similarity analysis [8], the duplications are mostly due to similar pages within the µVMs themselves. Regardless, this analysis shows that our modification to the µVM state file format improves the combined compression ratio from 3.98x to 6.64x.

While the combined compression ratio of µVMs increases significantly with smaller block sizes compared to boot caches, the overall compression efficiency does not increase. As we have shown previously in [8], This is due to the excessive



(a) Storage consumption of µVMs vs. boot caches.

(b) Compression ratios of µVMs.

(c) Compression ratios of boot caches.

Fig. 9. The effect of gzip compression and deduplication on µVMs vs. boot caches. µVMs show better compression efficiency with slightly higher overhead.

number of deduplication table entries that are present in the file system. Higher (potential) memory consumption of ZFS for deduplicating µVMs compared to boot caches is also due to this effect (Figure 9-a).

D. Summary

To summarize our findings, compared to normal booting, µVMs of three standard OSes on an ext4 file system startup between 9x and 15x faster with a cold page cache, and between 18x to 43x with a warm page cache. Compared to booting, using µVMs created from our Windows Azure repository and with a cold page cache, we observed on average 18x improvement with ext4, and 5x improvement with a deduplicated and compressed ZFS volume at 64 KB block size.

Our storage scalability analysis showed that our change to QEMU's state file improved the combined compression ratio from 3.98x to 6.64x, allowing us to cache µVMs of our entire Windows Azure repository, consisting of 1011 VM images, within 49.6 GB of drive storage and up to 342 MB of memory for deduplication table entries. In [8], we provided a scalability analysis of Squirrel caches for large amounts of VM images. There, we had found that the requirements on each host can be considered as modest with current and near-future host hardware. We can draw similar conclusions here, namely that the VM bakery will be able to store large amounts of µVMs on the hosts without prohibitive storage requirements.

VI. Related Work

In the following, we discuss work related to µVMs. We distinguish between the following three aspects that µVMs are touching upon:

1) *Dynamic Resource Allocation* deals with dynamic (de-)allocation of resources from the OS and VMM's point of view.
2) *Fast VM Startup* deals with fast and scalable techniques for provisioning VMs.
3) *Host-side Caching* deals with caching techniques for improving VM startup times.

A. Dynamic Resource Allocation to VMs

Commodity VMMs such as KVM [35] or Xen [36] support dynamic resizing of VM memory by means of a technique called ballooning. Cooperative guest OSes run a balloon driver that inflates and deallocates memory from the guest, or deflates to allocate more memory to the guest OSes. For our µVM design, we opted for memory hot-plugging instead for two reasons: 1) Hot-plugging does not require cooperative guests, an important requirement for IaaS clouds, and 2) with ballooning, the VM should be configured with the maximum possible memory *before* VM startup to allow for arbitrary resizing. This potentially makes caching of VMs with balooning less scalable than µVMs.

Core hot-plugging and hot-unplugging support in the OS kernel is increasingly becoming important due to their potential power saving and the approaching age of dark silicon [37]. Chameleon [32] is providing fast dynamic processors for the Linux kernel, and Zellweger et al. [33] provide a similar feature for a multikernel OS. These improvements can benefit the core hot-plugging time for µVMs. We showed in Section V-A, however, that memory hot-plugging takes significantly longer, and requires some research attention.

B. Fast VM Startup

Peer-to-peer networking is a common technique for transferring a single VMI to many compute nodes [38], [39], [40], [41]. The main issue so far has been the considerable delay of startup time in order of tens of minutes. This is because the complete VM image needs to be present before starting the VM. VMTorrent [42] combines on-demand access with peer-to-peer streaming to reduce this delay. Similarly, IP multicasting has been used extensively for scalable startup of VMs [43], [44]. All these approaches require booting of the guest VMs. µVMs are orders of magnitude faster when starting VMs, without sacrificing generality or requiring high-performance networks.

VMPlants [45] and similar systems (e.g., [46], [47]) try to optimize the size of VM images in order to reduce their transfer times during VM startup. While these systems are still beneficial for reducing the storage footprint of the VM images, µVMs are minimal snapshots of the memory state, and agnostic to the original size of the VM image. Further, µVMs are cached by Squirrel to eliminate data transfer during VM startup.

SnowFlock [48] 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. VMScatter [49] is a similar system, but less intrusive in modifying the guest OS kernels, but less efficient in terms of scale. Kaleidoscope [50] improves SnowFlock's on-demand paging by eagerly transferring the working set of the cloned VM. These systems, while efficient at starting stateful workers, require a live VM for cloning, and rely on significant multicast traffic during the cloning process. µVMs are designed for cloning of the OS state. They provide similar startup times, without requiring live VMs and high-performance networks. Further, µVMs rely on existing mechanisms only, not requiring any changes to the OS kernels. We have shown µVM's applicability and scalability using our large-scale VM repository.

De et al. [10] suggest keeping VMs with different hardware resources running at all times, and contextualizing a match according to the VM image and the instance type of a user's VM startup request. As discussed previously, this approach does not scale to IaaS clouds with many VM images and instance types. µVMs, in contrast, can start in mere seconds, hence providing the same benefits while being scalable.

Armstrong et al. [51] introduce *recontextualization* for reconfiguration and reusing of VMs to reduce the downtime of applications. Recontextualization can be used in conjunction with µVMs for compatible applications. µVMs, however, are more general, OS-agnostic, and can be started instantly as we showed in Section V.

DreamServer [13] and similar, stateful resume systems (e.g., [11], [12], [52]) rely on prefetching the resume working set for fast VM startup. While these techniques can improve µVM resume times further, they lack the generality and scalability that µVMs provide. We modified the open-source implementation of [15] for the lazy resume of our µVMs.

There has recently been an increased interest in running containers such as Docker [53] inside VMs (e.g., [54], [55]). One of the major benefits of such an approach is fast environment startup, in the order of seconds. Containers, however, have dependency on their hosting OS, due their process-based nature. This makes it difficult to migrate them, or start them without a compatible OS. In comparison, µVMs provide competitive startup times without these limitations. As an additional benefit, µVMs provide the full isolation of a VM.

Another source of overhead in VM startup is physical machine startup when there is no available physical node to host a new VM. Recent work on cloud schedulers [56], [57] improves this aspect by predicting the future number of VM startup/shutdown requests.

C. Host-side Caching

In this section, we look at caching techniques used to improve VM startup.

Zhao et al. [58] suggest that simply using NFS to transfer VM images is suboptimal. 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 [59].

The Liquid file system [60] and similar systems (e.g., [61], [62], [63]) are designed for scalable VM image distribution. These systems keep a cache of VM image contents (deduplicated or otherwise) on each compute node to improve VM startup times.

Squirrel's cVolumes [8] persistently cache all the blocks needed for starting µVMs instantly, by exploiting deduplication and compression techniques in an off-the-shelf ZFS file system [25]. None of these systems consider caching VM snapshots for scalable and instant VM startup like we proposed in this paper.

VII. Conclusions

Infrastructure-as-a-Service (IaaS) clouds promise instant creation of virtual machines for elastic applications. In practice, however, VM startup times range from several tens of seconds to several minutes. There are two major factors contributing to these long VM startup times, (1) the transfer of the VM image from a storage server to the host, and (2) the actual boot time of the virtual OS.

In previous work [8], we presented a host-side caching system, called Squirrel, that solves the VM image transfer problem by scalable storage of the boot reading sets for all VM images on the hosts directly. In this paper, we have addressed the OS boot time itself. Booting an OS is a lengthy but repetitive computation that we replace by much faster OS resume from precomputed snapshots. Such an OS snapshot needs to reflect the resources of the guest OS, like CPU cores and main memory. To avoid the combinatorial explosion caused by providing an OS snapshot per VM image and per machine type, we introduce prebaked µVMs, booted VMs with minimal resources (a single CPU core and 512 MB memory). At resume time, our *VM bakery* service that runs on the hosts is using hot-plugging of additional resources to match the requested machine types. Our evaluation shows that individual virtual machines can be resumed from µVMs in less than one second, when reading the µVM from an ext4 file system on a SSD device.

µVMs are small in size and lend themselves to scalable caching on the hosts of a data center. For scaling to large numbers of VM images, however, a storage layer with compression and deduplication is needed. We have extended our Squirrel system to store µVMs instead of VM image caches. For this purpose, we have analyzed the potential for compression and deduplication of snapshot data, and implemented the system accordingly. We can store µVMs for our repository of 1000+ VM images from Windows Azure in less that 50 GB drive space and 350 MB memory at each host, hence requiring only modest host resources in exchange for instant VM startup. Resuming from our compressed and deduplicated file system takes 2–3 seconds on average for our Azure VM image set.

With these results, we have demonstrated that instant VM startup is possible in IaaS clouds, allowing elastic applications like Web services to scale out according to changing workloads, while avoiding costly resource overprovisioning or elaborate workload prediction schemes.

ACKNOWLEDGMENTS

This work is partially funded by the Dutch public-private research community COMMIT/. The authors would like to thank Stefania Costache and Hanieh Bagheri for providing valuable feedback on an earlier version of this paper, and Kees Verstoep for his support on the DAS4 clusters.

REFERENCES

[1] 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.

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

[3] 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.

[4] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *11th IEEE/ACM International Conference on Grid Computing*, ser. GRID '10, 2010.

[5] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "AGILE: elastic distributed resource scaling for Infrastructure-as-a-Service," in *10th International Conference on Autonomic Computing (ICAS'13)*, 2013.

[6] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *IEEE Network Operations and Management Symposium (NOMS'12)*, 2012.

[7] P. Jamshidi, A. Ahmad, and C. Pahl, "Autonomic Resource Provisioning for Cloud-based Software," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '14, 2014.

[8] K. Razavi, A. Ion, and T. Kielmann, "Squirrel: Scatter Hoarding VM Image Contents on IaaS Compute Nodes," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14, 2014, pp. 265–278.

[9] 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*, ser. SC '13, no. 65, 2013.

[10] P. De, M. Gupta, M. Soni, and A. Thatte, "Caching vm instances for fast vm provisioning: A comparative evaluation," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par '12, 2012, pp. 325–336.

[11] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr, "Fast Restore of Checkpointed Memory Using Working Set Estimation," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '11, 2011, pp. 87–98.

[12] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite, "Optimizing VM Checkpointing for Restore Performance in VMware ESXi," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13, 2013.

[13] T. Knauth and C. Fetzer, "DreamServer: Truly On-Demand Cloud Services," in *Proceedings of the 7th ACM International Systems and Storage Conference*, ser. SYSTOR '14, 2014.

[14] "KVM Migration," http://www.linux-kvm.org/page/Migration, [Online; accessed 17-12-2014].

[15] T. Knauth and C. Fetzer, "Fast Virtual Machine Resume for Agile Cloud Services," in *Proceedings of the 3rd International Conference on Cloud and Green Computing*, ser. CGC '13, 2013.

[16] "QEMU Monitor Protocol Commands," https://github.com/qemu/qemu/blob/master/qmp-commands.hx, [Online; accessed 17-12-2014].

[17] C. Curtsinger and E. D. Berger, "STABILIZER: Statistically Sound Performance Evaluation," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013, pp. 219–228.

[18] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization," in *Proceedings of the 21st USENIX Conference on Security Symposium (Security '12)*, 2012.

[19] "mmap(2) - Linux man page," http://linux.die.net/man/2/mmap, [Online; accessed 17-12-2014].

[20] "msync(2) - Linux man page," http://linux.die.net/man/2/msync, [Online; accessed 17-12-2014].

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

[22] "QCOW2 backing files and overlays," https://kashyapc.fedorapeople.org/virt/lc-2012/snapshots-handout.html, [Online; accessed 17-12-2014].

[23] "udev Linux dynamic device management," https://www.kernel.org/pub/linux/utils/kernel/hotplug/udev/udev.html, [Online; accessed 17-12-2014].

[24] "Task Scheduler," http://technet.microsoft.com/en-us/library/cc721871.aspx, [Online; accessed 17-12-2014].

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

[26] "inotify(7) - Linux man page," http://linux.die.net/man/7/inotify, [Online; accessed 17-12-2014].

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

[28] "ZFS on Linux," http://zfsonlinux.org, [Online; accessed 17-12-2014].

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

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

[31] R. de Castro, A. Lago, and M. Silva, "Adaptive compressed caching: design and implementation," in *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, 2003.

[32] S. Panneerselvam and M. M. Swift, "Chameleon: Operating System Support for Dynamic Processors," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, 2012, pp. 99–110.

[33] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe, "Decoupling Cores, Kernels, and Operating Systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI' 14, 2014, pp. 17–31.

[34] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman, "An Empirical Study of Memory Sharing in Virtual Machines," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC '12, 2012.

[35] "QEMU's Kernel Virtual Machine," http://wiki.qemu.org/KVM, [Online; accessed 17-12-2014].

[36] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, 2003.

[37] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, 2011.

[38] 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*, ser. ICDCSW '09, 2009, pp. 324–329.

[39] 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*, ser. SIGUCCS '08, 2008, pp. 287–290.

[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*, ser. CloudCom '10, 2010, pp. 112–117.

[41] Nimbus Project, "LANTorrent," http://www.nimbusproject.org/docs/current/admin/reference.html#lantorrent, 2010, [Online; accessed 17-12-2014].

[42] 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*, ser. CoNEXT '12, 2012, pp. 289–300.

[43] 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)*, ser. PDP '10, 2010, pp. 567–574.

[44] 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*, ser. HPDC '08, 2008, pp. 87–96.

[45] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo, "VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing," in *2004 ACM/IEEE Conference on Supercomputing*, ser. SC '04, 2004.

[46] 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*, ser. Euro-Par 2013: Parallel Processing Workshops, 2013.

[47] C. Quinton, R. Rouvoy, and L. Duchien, "Leveraging feature models to configure virtual appliances," in *2nd International Workshop on Cloud Computing Platforms*, ser. CloudCP '12, 2012.

[48] 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*, ser. EuroSys '09, 2009, pp. 1–12.

[49] 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*, ser. VEE '13, 2013, pp. 63–72.

[50] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara, "Kaleidoscope: Cloud Micro-elasticity via VM State Coloring," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11, 2011.

[51] D. Armstrong, D. Espling, J. Tordsson, K. Djemame, and E. Elmroth, "Runtime Virtual Machine Recontextualization for Clouds," in *Proceedings of the 18th International Conference on Parallel Processing Workshops*, ser. Euro-Par '12, 2013.

[52] J. Zhu, Z. Jiang, and Z. Xiao, "Twinkle: A fast resource provisioning mechanism for internet services," in *Proceedings of the 2011 IEEE INFOCOM*, 2011.

[53] "Docker," https://www.docker.com/, [Online; accessed 11-04-2015].

[54] "Heroku," https://www.heroku.com/, [Online; accessed 11-04-2015].

[55] K. Razavi, A. Ion, G. Tato, K. Jeong, R. Figueiredo, G. Pierre, and T. Kielmann, "Kangaroo: A Tenant-Centric Software-Defined Cloud Infrastructure," in *Proceedings of the 3rd IEEE International Conference on Cloud Engineering*, ser. IC2E '15, 2015.

[56] A. Pucher, C. Krintz, and R. Wolski, "Using trustworthy simulation to engineer cloud schedulers," in *IEEE International Conference on Cloud Engineering*, Mar. 2015.

[57] R. Wolski and J. Brevik, "QPRED: Using Quantile Predictions to Improve Power Usage for Private Clouds," Computer Science Department of the University of California, Santa Barbara, Santa Barbara, CA 93106, Tech. Rep. UCSB-CS-2014-06, September 2014.

[58] 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*, ser. HPDC '04, 2004, pp. 202–211.

[59] 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*, ser. HiPC '07, 2007, pp. 21–34.

[60] 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.

[61] 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*, vol. 99, 2014.

[62] 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)*, 2011, pp. 147–158.

[63] 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*, ser. INFOCOM '10, 2012, pp. 181–189.