Rethink the Virtual Machine Template

Kun Wang, Jia Rao, Cheng-Zhong Xu

Department of Electrical & Computer Engineering
Wayne State University, Detroit, Michigan 48202
{kwang,jrao,czxu}@wayne.edu

Abstract

Server virtualization technology facilitates the creation of an elastic computing infrastructure on demand. There are cloud applications like server-based computing and virtual desktop that concern startup latency and require impromptu requests for VM creation in a real-time manner. Conventional template-based VM creation is a time consuming process and lacks flexibility for the deployment of statefull VMs. In this paper, we present an abstraction of VM substrate to represent generic VM instances in miniature. Unlike templates that are stored as an image file in disk, VM substrates are docked in memory in a designated VM pool. They can be activated into statefull VMs without machine booting and application initialization. The abstraction leverages an arrange of techniques, including VM miniaturization, generalization, clone and migration, storage copy-on-write, and on-the-fly resource configuration, for rapid deployment of VMs and VM clusters on demand. We implement a prototype on a Xen platform and show that a server with typical configuration of TB disk and GB memory can accommodate more substrates in memory than templates in disk and statefull VMs can be created from the same or different substrates and deployed on to the same or different physical hosts in a cluster without causing any configuration conflicts. Experimental results show that general purpose VMs or a VM cluster for parallel computing can be deployed in a few seconds. We demonstrate the usage of VM substrates in a mobile gaming application.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design,Distributed Systems

General Terms Management, Measurement, Performance

Keywords Virtual machine deployment, Data center, Virtual machine template, Cloud computing

1. Introduction

Cloud computing in its original form offers virtualized resources, and infrastructure in general, as a service over the Internet. A key requirement is resource provisioning on-demand in a real-time manner. In the model of infrastructure-as-a-service, applications are often run in virtual machines (VMs) and their performance relies on effective management of the VMs in the whole life-cycle from creation, deployment, execution, to termination. Because of the nature of on-demand computing, VM startup latency is a crucial performance factor in application responsiveness, in particular for those that interactive, impromptu, and short-lived computing [10].

An example of such applications is server-based computing (SBC) [12], in which resource-constrained client applications offload compute- or data-intensive tasks to VMs running in a data center, e.g., through computation offloading or wrapping mobile OS to VMs running in the cloud can significantly extends the computing capability of mobile devices as well as saves the scarce battery resource. [3]. In such case, the VMs may need to be created and deployed on the fly during the execution time of the applications. Another example is virtual desktop infrastructure (VDI) [25], in which clients would launch their VMs associated with their personalized working environments and data on a remote client device upon request. In addition, in virtualized parallel computing, the size of a VM cluster varies with the workload which requires new VMs worker can be created instantaneously. Startup latency is pivotal to the success of all these cloud computing usage cases.

VM creation from scratch requires to create a virtual hard drive image, configure virtualized resources, install OS and initialize application services. This process would take tens of minutes. To reduce the startup latency, in practice, public IaaS providers like Amazon Web Services provide users an option to create VMs from template. A VM template [20, 26] is a reusable image created from a clean VM and stored in disk as a file. Although a VM can be created by booting from a template in tens of second, the template become non-reusable by others. VM cloning from a template would retain the reusability of the template but at the cost of expensive disk copy of large image files. In either approach, there is no time-efficient way to create multiple VMs simultaneously from the sample template, although such parallel deployment is crucial to parallel computing and server clustering.

There were recent studies on reducing the startup latency and supporting parallel deployment of homogeneous VMs; see Potemkin [27] and Snowflock [11] for examples. Potemkin proposed a delta virtualization technique for flash VM cloning. It relies a copy-on-write optimization technique to have multiple VMs share memory pages as much as possible. Snowflock proposed a process-fork like API to fork VMs for parallel processing during the execution of a program. The VMs created inherit the software stack from their parent VMs and can not exist without the presence of their parents.

In this paper, we propose an abstraction of VM substrate as an alternative to VM template for rapid deployment and parallel deployment of VMs. VMs created from substrates have the same life cycle as template-based VMs and the VMs are of independent by origin and can be deployed across different physical hosts. Unlike templates that are stateless and stored in disk as an image file, substrates is a generic VM instance in miniature that docked in memory of a designated machine in an inactive state. They can be present with or without application footprints and ready to be

VEE'11, March 9–11, 2011, Newport Beach, California, USA.

Template Size	2G	5G	10G	20G
cp(local disk)	36.06s	58.75s	547.45s	1228.69s
cp(nfs)	46.16s	78.21s	640.28s	1412.42s
scp	43.31s	114.66s	749.97s	1589.35s
dd(single disk)	3.07s	45.55s	195.71s	515.17s

Table 1. Cost of creating VM from templates.

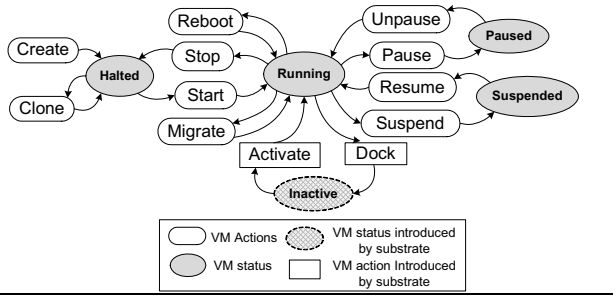


Figure 1. VM State Transition.

powered on upon request. Creation of VMs from substrates saves time from time-consuming disk-based booting and deployment. The substrate mechanism leverages an array of techniques, including VM miniaturization, generalization, clone and migration, page copy-on-write, and on-the-fly resource configuration, to save memory space, generalize substrate usages, and resolve resource configuration conflicts on VMs to be created. The mechanism facilitates parallel VM deployment via multicast.

We have implemented a prototype on a Xen/Linux server cluster and tested the system in two scenarios: on-demand deployment of VMs for cloud-assisted gaming and parallel deployment of heterogeneous VM clusters like LAMP (Linux/Apache/MySQL/PHP). Experimental results showed the mechanism capable of creating VMs in subsecond, while retaining the flexibility of VM resource configuration. The experiment results also show that the substrate mechanism makes it possible to deploy a VM cluster in a few second or a speedup of more than 50 times in comparison with default VM deployment from template.

The rest of this paper is organized as follows. Section 2 gives background information about VM lifecycle and in particular the cost sources of VM creation. Section 3 presents the concept of substrate, substrate pool, and revised VM life cycle due to the use of substrate. Section 4 presents implementation details and results from micro-benchmark testing of key implementation issues. Evaluation results of the system as a whole are discussed in Section 5. Section 6 discusses related work. Section 7 concludes this paper with remarks about future work.

2. Background

Deployment of a VM in a data center involves a number of steps: (1) VM creation with virtual hard disk; (2) Installation of OS images and applications; (3) Deployment with configuration (networking, etc) on selected host/cluster; (4) VM startup.

New VMs can be created either from scratch or from template. As the process of VM creation from scratch takes tens of minutes, it is rarely used in cloud. On the other hand, deploying a VM from templates, which removes the process of OS and software installation, is widely used in practice. VM creation from templates involves two steps: (1) create a copy of the template's virtual disk image and (2) customize the VM configuration as needed. Configuration customization includes parameter settings for boot option, host name and network. VMs can be created from templates through either cloning or conversion. VM templates are usually created for a specific purpose such as a web server or a database server. Once booted, the VM which originates from a template can be further extended by deploying more applications or run-time libraries. In the following, we first discuss the cost of VM creation and then examine the state transition of a VM. Next, we present the challenges of fast VM deployment.

2.1 Cost of VM creation

The cost of template-based VM creation comes from different sources. First, depending on the storage environment and VM template image size, the cost of VM disk duplication varies. In order to support VM live migration [4], VM disk images are usually stored in centralized storage servers. NFS and iSCSI are two popular choices for the deployment of VM virtual disks. In either case, the duplication of the template's disk image is necessary for a new VM creation. Table 1 shows the cost of disk duplication with different disk sizes and different methods. Regardless the underlying storage organization and duplication methods, the cost increases significantly with the VM disk size. A 5GB VM disk requires more than one minute to be copied. The latency incurred by disk duplication is not acceptable to interactive applications. Besides, according to the table, to clone a new VM from a template on remote host (*scp*) takes tens of seconds or even a few minutes, consuming a significant amount of network bandwidth in the data center. Note that although create a blank disk image on local disks (*dd*) takes less time, but deploying root filesystem takes even more time than directly duplicate a disk VM with root filesystem as a whole. Second, the booting process of a VM includes booting the kernel and starting default services. Kernel booting usually takes sub-seconds while starting different services is both error-prone and costly. The general purpose OS installation activates many services by default. RightScale [22] templates and Oracle VM templates [20] disable most of the application unrelated services to minimize the cost. Third, traditional JeOS templates are usually extended by installing more applications to generate new application specific templates. The cost of maintaining various VM templates increases with the diversity of application oriented templates. All these costs together makes template based VM creation impractical for interactive applications.

2.2 VM state transition

Starting from a template, a VM experiences multiple states in its life cycle. Figure 1 shows state transition diagram for a VM. Each VM is initially halted after being created from scratch or cloned from templates. Although each halted VM is a static instance only consuming disk space, it still can be edited or customized by installing new applications or changing the associated configuration. A VM is changed to a running state when it is started and A VM can be paused or suspended on local host or migrated to another host. We added one additional state and two new actions to the conventional VM state diagram [20]. A new substrate is generated from a running VM through docking. Docking can be done by converting or checkpointing. Converting puts the running VM to an inactive state, while checkpointing keeps the VM running. The tradeoff between these two solutions are discussed in substrate design section. Note that an inactive state is different from a halted state. An inactive VM consumes memory and maintains running status, but a halted VM only consumes disk space.

2.3 Challenges of rapid VM deployment

Rapid VM deployment calls for minimal costs in each step of VM creation. However, as discussed above, virtual disk image duplication is time-consuming. It leads to a large startup latency.

Moreover, if multiple VMs need to be created at the same time, disk duplication is the key impediment to fast VM deployment. In addition, the automatic resource reconfiguration of new VMs is also challenging, especially in a heterogeneous virtualized cluster of VMs with interactive applications.

Stateless VM creation has limited usage cases due to the fact that it creates brand new VM every time without preserving runtime environment or intermediate result. A brand new VM with necessary applications pre-installed is how the general VM template is used. This is insufficient for many of the cloud applications like parallel computing or mobile computation offloading. Thus the fast creation of statefull VMs is necessary.

Rapid VM deployment also requires that the creating process should be transparent to users and applications. Because creating a new VM always takes time, in the cases of user interactive applications or other request-driven VM creation, startup latency caused by creating a new VM must be small enough so as to make the creating process transparent to application. If the cost of creating process is negligible, from applications' perspective, VMs are always ready for use.

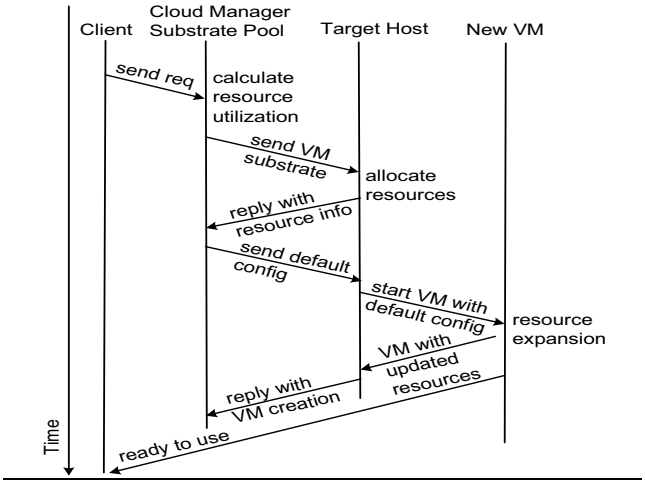


Figure 2. Create VM from substrate.

3. Design of VM substrate

Modern applications and libraries consume a considerable amount of disk space, which makes the size of templates usually large. To address these limitations, a few questions need to be answered. First, can image file be stored in memory instead of disk? Although, solid-state-disk(SSD) attempts to increase the efficiency of data transfer between disk and memory, it is still not fast enough to meet the requirement of duplicating disk image on demand. Moreover, the size of traditional templates can easily go beyond the limitation of the memory of a server or a common SSD. Thus it is impractical to maintain templates in memory and only a limited number of templates can be saved on SSD. Second, is it possible to avoid the booting process while still maintaining previous running states when starting a VM? An AMI [6] or oracle VM contains a minimal Linux installation with only essential Linux services, leaving the installation of additional applications to package management tools. Thus the images are much smaller than default Linux OS installation. However, it is a brand new OS with only a limited number of services installed. Third, is it possible to deploy a VM in a real-time manner? Real-time VM deployment allows VMs to be created on demand and only be activated when in use. In the remaining section, we elaborate the design of VM substrate and compare VM substrate with alternative approaches.

3.1 VM substrate and pool

The design of VM substrate aims to leverage existing virtualization techniques to provide an agile cloud computing environment which allows users to create VMs or VM clusters on demand. A VM substrate is a static reusable instance that can be duplicated or reactivated for later use. VM substrates are categorized into three types. Public substrates contain minimal clean JeOS and generic configuration. Restricted substrates are the extensions of public substrates with specific applications and run-time environment. Alternatively, private substrates include users' personal data which can only be reused under strict sharing policy. These types of substrates are designed for different use cases, but they follow the same docking and reactivating process.

Saving the running states of VMs into in-memory VM substrates has many advantages over having VMs always run in full capacity. If a VM in full capacity is paused or suspended to the local machine, the resulted memory footprint which contains the VM's running state is usually quite large, in proportion to the VM's original capacity. If the saved state is stored in local machine's memory, the restarting of the paused/suspended VM is instant but at a cost of wasted memory resources which can be otherwise used by other running VMs. If the state is saved on local hard disk, the time required to resume the VM is unacceptable. For example, it takes approximately 40 second to restore a VM with 2 GB memory from a 7200RPM SATA disk. In VM substrates, we first trim the VM to its minimal capacity (minimal CPU and memory, detached block and network devices) that preserve essential running states, and then temporarily dock the VM to memory other than disk. After the final compression, the resulted memory footprint which usually in a size of tens of megabytes is transferred and consolidated to a dedicated substrate pool. Upon resuming, the corresponding VM's substrate is activated by expanding to its real capacity. The restoration latency is comparable to the local in-memory restore but with a much less memory cost on each local host.

A substrate pool is a centralized repository where all the substrates are maintained. Unlike traditional VM template pool, The substrate pool stays mainly in memory and the backup substrates are stored on disk. The size of a substrate pool is dynamically reconfigurable without affecting the existing substrates. Our preliminary experiment results show that a substrate with minimal programming environment can be as small as 16MB. With the similar substrate, we successfully hosted several hundreds of substrates on a physical machine with a 4GB memory. However, the sizes of the substrates depend on the running status of the hosted application. In order to maintain a statefull substrate with manageable cost, we aim to embed only necessary data into a substrate.

VM substrate is proposed to be an alternative effective VM administration solution not only applicable to instant parallel workers creation, but also applicable to standalone VM deployment, also taking the reusability and scalability into consideration. Different from VM Descriptors proposed by Snowflock [11], VM substrate doesn't have heavy dependancy on any parent VM and has many varieties. VM Descriptors contain only the minimal critical metadata needed to start execution and use *Memory-On-Demand* mechanism lazily fetch portions of VM sate over network as it is accessed. In contrast, VM substrates are static VM abstraction resides in a pool in memory. Activation, resource expansion and remapping are the typical three steps to create a new VM from a substrate. In theory, it is possible to maintain a pool of template parent VM and then fork child VMs on demand. However, this solution can hardly get rid of the limitation of dependancy and hard to meet the requirement of VM creation for long standing services. Moreover, due to the size of the parent VM, the cost of maintaining template parent VMs is much higher than maintaining a substrate pool. In addition,

if the application is CPU intensive and requires minimal updates to disk or the intermediate results can be discarded, an alternative way to VM fork is to start multiple VMs on different hosts with the same disk image located on a centralized server. But this solution has very limited usage cases.

The abstraction of VM substrate introduces two VM state transfer actions in the life-cycle of a VM which are docking and activating. A VM substrate is constructed by docking a running VM maintaining applications' running status. There are two ways of docking: intrusive converting of a running VM and live checkpointing of a VM. In contrast, VM substrate activating includes dispatching substrate, launching substrate and reconfiguring substrate's resources. A new VM is created after the activating process.

3.2 VM clone from substrate

We employ four steps to address the challenges in on-the-fly VM creation. First, VM miniaturization and generalization. Before generating new VM substrate, the parent VM is shrunk to a miniature state. A VM substrate has minimal memory footprint, single vCPU core, detached network interface and reference to virtual disk. Since the memory size is a major factor of the final size of a VM substrate, the memory size needs to be shrunk to the greatest degree through either intrusive shrinking or live checkpointing. In either case, the data in the system cache is synchronized to disk first. Through predictive calculation, we reconfigure VM's memory to a size that only contains data necessary for the restoration. VM configuration generalization assures the VM specific configuration of public or restricted VM. Configurations such as host name, networking parameters are reset to the default value. The resources of a private VM substrate is minimized while still maintaining its original configuration. Second, raw VM substrate is generated right after the VM's resource shrinking. A snapshot of the minimal running VM is created and stored in local memory. Third, raw substrates are compressed to be the final VM substrates before they are moved to a substrate pool. Compression reduces the substrates to a size as small as tens of megabytes which can be transfered over WAN. Fourth, the minimal VM substrate on local memory is transfered to a centralized pool. Figure 3 illustrates the steps of docking a running VM to a substrate.

When a substrate is selected to create a new VM, as shown in Figure 2, it is duplicated to other physical hosts simultaneously via multicast. Each physical host then decompresses the VM substrate and activates it from memory. Through reconfiguration, newly created VMs on each host will be allocated more memory and vCPU resources depending on application needs. New network interface with predefined parameters is attached to the VM and the configuration takes effect immediately. Depending on the type of a substrate, root disk is remapped and user's personal disk partitions can be attached to the VM.

3.3 VM substrate generation

Converting a VM to a substrate starts with reconfiguring a running VM's resource to minimal memory footprint and vCPU number, detaching the network card and saving the disk states. The initial VM from which a substrate is constructed can be a VM template or any VM with applications running. Intrusive conversion can be initiated in the application level by administrators whenever the VM has no scheduled work and is ready to be docked.

A VM substrate can also be created through live checkpointing in system level. VM checkpointing has been widely used for various purpose like high availability [5], VM migration [2, 4, 16, 28],fault-tolerant [15] or debugging [8]. We also leverage checkpointing to create VM substrates without interrupting the running services. Most of existing VM level checkpointing techniques tend to save the entire running states(*cpu,memory,disk*) in a core dump

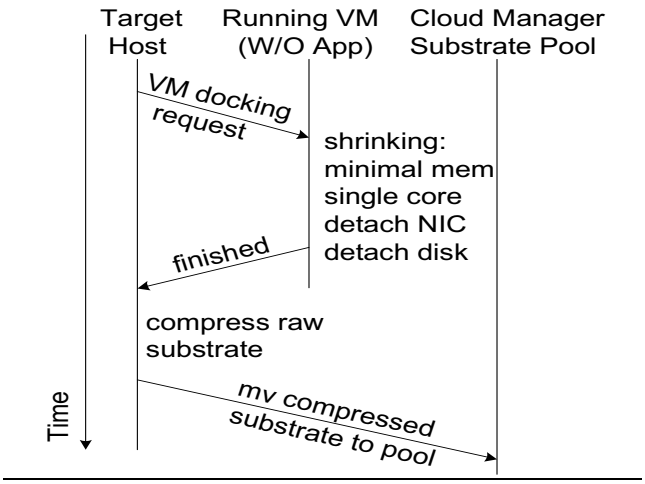


Figure 3. VM dock to substrate.

where the resulted checkpoint size is the VM's memory size, and the checkpointing time is closely related to memory page dirty rate. We employ two techniques to ensure that a VM can be correctly restored from a substrate and the size of the resulted substrate is minimized. First is selective memory checkpointing, through which only reusable memory pages are saved to substrates, discarding the reconstructable or zero pages. Selective checkpointing memory is able to reduce the size of raw substrates considerably. Second is the generalization of VM configuration, which set all VM specific resource identifiers like*vmid* or *uuid* to default values in a VM substrate. By using these two techniques, a checkpointing substrate of existing running VM instance can be created any time without conflicting with the original VM.

Compared with intrusive conversion, live checkpointing is able to create VM substrate without interrupting user applications, but it requires the modification of the VMM for selective memory dumping. In contrast, application level substrate conversion is independent on the underlying VMM. It only requires that virtual hardware resources of a guest VM can be configured dynamically without a restart.

3.4 VM fork

We note that VM fork has been recently proved to be an efficient way to clone a parent VM to multiple copies swiftly[11, 27]. Similar to process level fork, VM fork allows a child VM to inherit all the states originated from its parent VM prior to forking, enabling creating statefull computing instance rapidly. However, different from process fork, VM fork is capable of creating VM clones across a set of physical hosts. It can also work in a parallel manner where a single API call launches multiple VMs. Each child VM has its own independent copy of resources and runs independently from the parent VM. Once forked, and the changes made to each cloned VM are maintained separately. We analyze the advantages and the disadvantages of VM fork and compare it with VM dock and reactivate in the remaining of this section.

VM fork is capable of creating transient VMs whose virtual resources are discarded once they exit. The intermediate states or values generated by the applications in a child VM are lost unless being explicitly synchronized to the parent VM. Due to the characteristic of a fork operation, VM fork has a few limitations. First, VM fork is applicable to computation intensive applications with limited or disposable intermediate results. Existing VM fork leverages disk Copy-On-Write(COW) techniques to offer each child VM a COW slice of disk and all the disk updates or intermediate values

are preserved on the COW disk. The child VMs share the running environment of the parent VM and the coordination between the parent and the children is mainly limited to computation. In the case of IO intensive applications, each child VM needs to make changes to their own disks which are actually COW slices. When the tasks in children VMs finish the updates on each child may need to be synchronized back to the parent. The integration of the updated data to the base disk incurs significant cost. It is challenging to achieve consistent synchronization once several VMs changed the same data. Second, sharing the same base disk partition between parent and children VMs limits the scalability of VM multiplexing. With IO intensive applications, the disk bandwidth of the base partition can easily become the performance bottleneck. Although *multicast* can be used to render memory pages concurrently to all the children VMs and memory page prefetching can possibly speed up on-demand paging, VMMs like Xen only grants the privileged domain direct access to the devices and does not allow the guest domains to access them directly [7, 19]. If the number of child VMs that request missing pages is large, the parent VM would receive a considerably amount of page requests from network interface. The parent VM can possibly become a hot-spot. Third, current VM fork implementation remains at application level focusing on parallel applications which need to re-spawn additional temporary workers. However, VM fork is not ideally suitable for deploying longstanding independent VMs at cloud administration level. Server applications such as web hosting and database warehousing usually run in loose coupled virtual clusters with minimal correlation. Such applications often require persistent data storage for each virtual node. Another drawback of the VM fork mechanism is its inability to create a heterogeneous VM cluster at a time. The VM substrate approach proposed here tries to create a cluster of heterogeneous VMs in a real time manner.

4. Implementation

We have implemented our VM substrate pool mechanism on the Xen platform. Xen is capable of running two leading approaches for virtualization: para-virtualization(PV) and full virtualization(FV). FV is designed to provide total abstraction of the underlying physical system, in which guest OS or applications are not aware of the virtualized environment. However, it incurs much performance overhead and can not be reconfigured on the fly without reboot of the VM. In contrast, PV presents each VM an abstraction of the hardware and requires modification of OS, allowing near-native performance. The memory size and the number of vCPUs of a PV guest VM can be reconfigured without restarting the VM. Thus, we select PV VMs in our prototype implementation. Our implementation includes modifications to the hypervisor, the `libxc` library, and the `xend` management daemon. In the remaining of this section, we elaborate the implementation details and compare them with alternative approaches. We also present micro-benchmark results to show the feasibility and effectiveness of the VM substrate.

4.1 Resource shrinking and expanding

vCPU: vCPUs are what the guest sees as CPUs on which the guest OS schedules applications processes or thread. The final size of a VM substrate is not affected by the number of vCPU configured in a VM. In order to make each substrate be more generic and with minimal resources, each VM substrate has an default configuration of a single vCPU core. In practice, vCPUs are usually pined to specific physical CPUs for predictable performance. VM substrate is designed to be a generic mechanism that does not assume any physical host information. Thus, CPU affinity information is not maintained in the substrate. In a heterogeneous cluster, a VM substrate with a single vCPU is able to be deployed on any physical machine. Since Xen VMM does not allow the actual vCPU number to exceed the maximal number of vCPU specified in the guest's configuration file, we set the default maximal number of vCPU to be the total number of physical CPU cores for each substrate. Any newly created VM initially has single CPU core by default. More vCPUs can be allocated at a step of one vCPU.

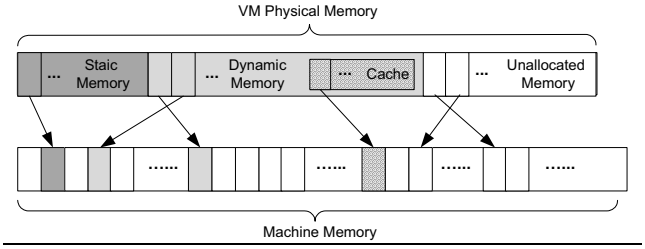


Figure 4. VM's memory footprint.

Memory: Xen VMM is responsible for managing the allocation of physical memory to guest domains and maintaining a triple indirection model(*virtual memory, pseudo physical memory and machine memory*). Each VM runs in an illusory flat, continuous address space. Xen reserves the top 64M of the virtual address space for every domain. The remaining physical memory is available for allocation at a granularity of one physical page. Xen maintains a globally readable mapping table between PFN(Pseudo-physical Frame Number) and MFN(Machine Frame Number). The OS running in a VM maintains the mapping between virtual memory and pseudo physical memory. As shown in Figure 4, each VM's physical memory is part of the machine memory and can be divided to several parts including used pages and unallocated free memory. The used pages can be further divided into static memory pages and dynamic memory pages. The later one also includes disk cache. Note that although the used pages are not available for reallocation, it is still possible that some of those pages are zero pages either because they are set to zero by programs or they are used as heap initialized by compiler. Traditional VM save `xen save` writes the VM's entire memory including zero pages, cache pages and free pages to a checkpoint file. Including free and zero pages in the checkpoint file is likely to be a waste because those pages store no information of the checkpointed states. In order to minimize the size of a VM substrate, we only keep the reusable and minimal memory footprint while still maintaining the integrity of a VM's state.

A VM substrate which excludes free pages does not harm the correctness of VM when it is relaunched because those free pages can be easily reconstructed by manipulating the mapping table of MFN and PFN. Zero pages are still included in a VM substrate for the following reasons: First, there is no more efficient way to extract zero pages other than doing a bit by bit comparision. The cost rises as the size of VM memory increases. Second, each VM substrate is compressed before going to a substrate pool, the compression algorithm is capable of compressing the zero pages with a large compression ratio which reduces the size of the substrates considerably. Note that disk cache is used for performance optimization where recently accessed data can be retrieved from memory without incurring disk IO. Before creating a public or a restricted VM substrate, disk cached pages are synchronized to the disk which yeilds more free pages and the final substrate size can be further reduced.

Most of existing Linux distributions enable many optional services by default even for a base installation. Rightscale[22] uses bash scripts to disable those optional services before building a template. In addition to kicking off disk cache pages, we also release part of the memory occupied by killing user applications that

are not relavent to the main purpose of the substrate. For example, in a substrate dedicated for web hosting applications, optional services like *sendmail, nfs* can be removed. We customize the application level services before docking a VM.

Memory ballooning is used by VMMs like Xen to achieve memory over-commitment. It provides the ability for the sum of the physical memory allocated to all active domains to exceed the total actually physically available memory on the system. Recent dynamic memory balancing work [29] proposed mathematical models to forecast memory needs and dynamically adjust the memory for VMs. The objective of these two memory adjustment approaches is to improve memory utilization. The later one also considers applications' throughput and performance. It is possible to instrument Xen to track memory accesses with each VM through the use of shadow page table. Shadow page tables are enabled during Xen's VM migration to determine which pages are dirtied during the migration. However, trapping each memory access results in a significant application slowdown and is only acceptable during migration [4, 23].

After a new VM is created from a VM substrate, it will start running at the initial state with minimal memory. It later expands to a larger size according to the setting in the configuration file. Each VM has a maximum and current memory size. Current memory size can be adjusted up to the maximum size. We configure the maximum memory of each substrate to be the physical memory size. The total memory size is extended dynamically. We implement an application level memory shrinking mechanism which is used to convert a VM to a substrate based on simple speculation in our prototype. We use the Linux /proc interface (in particular `/proc/meminfo`) to analyze the memory usage. Before docking a VM, we first kick all the cached data back to disk and consider the remaining memory size being actively used. Then we determine the minimal amount of memory the VM needs by adding a safe margin preventing Out-of-Memory crashes when the VM is restarted from substrate. The VM is set to the resulted memory size. The memory footprint of a guest VM will directly influence the final size of the VM substrate. The effect will be evaluated at the end of this subsection.

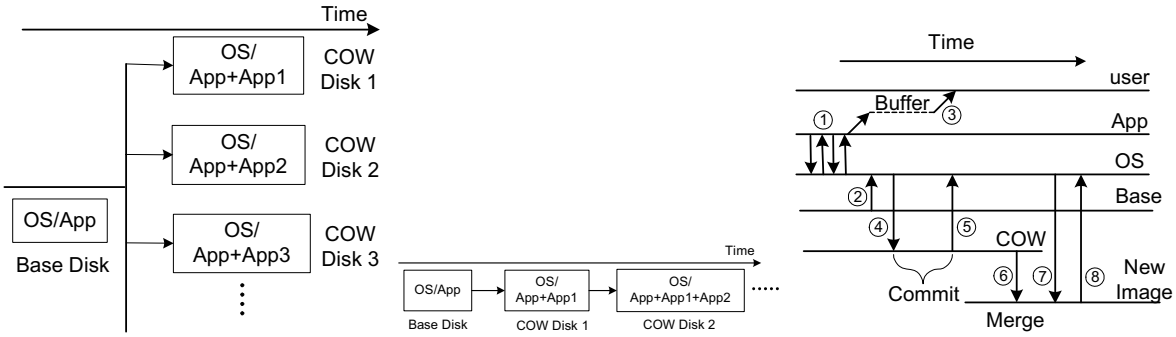
Network: The privileged domain in Xen VMM implements the network interface driver and all other guest domains access the driver via virtual device abstractions. Each domain is attached one or more virtual interfaces. Due to the fact that virtual interfaces are not necessary for booting a VM, their configurations can be postponed until rest of the guest OS ready to work. Conventional migration keeps network connection status by maintaining all protocol states and keeping IP addresses and MAC addresses in a record. Existing solutions used to manage network configuration during migration are to generate an unsolicited ARP reply form the migrated host, which lets the switch and other hosts know that the MAC is connected a new port [4]. However, even if the switch is configured not to block ARP broadcast, conflicts still exist if multiple VMs are created from the same substrate because all the network configurations of the new VMs are originated from the same substrate. In order to avoid the conflicts, We detached the network interface before docking a VM and VMs created from substrates do not have network interfaces initially.

The network parameters are configured when a new network interface is attached to a VM. In our prototype implementation, we also developed a mechanism to isolate the network in order to prevent interference between unrelated VMs. First, the networking mode (NAT,bridge or routing) can be dynamically configured with an interface in a physical host. Besides, the IP, MAC addresses and even the network mode can be determined within a physical host and transfered to guests as parameters. We implemented guest network configuration mechanism based on `Xenstore` to provide agile and immediate configurations. Depending on the purpose of newly created VMs. Especially when a virtual cluster is created, they are deployed with private network addresses and only guests within the same subnet are visible to each other.

A VM substrate is the snapshot of an original VM, and the memory and process running status are preserved in the substrate. This may result in some conflicts if new VMs are created based on one VM substrate because they share the same running environment. It is possible that multiple processes in different VMs may need to connect to the same socket or open the same file. In our prototype implementation, docking VM can be done at administrative level when one phase of computation is finished or before the application starts to run. Another solution is to create a substrate directly from a running template.

Disk: Disk image files are commonly used as virtual disks by guest VMs. Because the disk image files, which are usually in a size of tens of Gigabytes, stores the application specific data, costly disk duplication is often unavoidable if new VMs are to be created. Existing template-based VM creation simply distributes the virtual disk image in a copy-and-paste manner to reconstruct the same VM without reinstalling OS or applications. Thus any two VMs from the same template are independent from each other, guaranteeing the isolation of VMs. However, the time spent on copying virtual disks is unacceptable provided that the disk size is usually large. Disk copy-on-write is often used to avoid unnecessary disk space waste. Multiple COW slices can share the same read-only base image file and all the updates are directed to those COW slices. Wide-area VM migration used disk COW to transfer VM disk state over low bandwidth and high-latency links [9, 23]. To reduce the startup latency of new VMs, disk COW is also used recently by Snowflock [11] and Potemkin [27] to generate temporary disk slices for newly created VMs.

There are two different types of disk COW. Frist, a *blocktap* driver combined with a *qcow* slice, which is supported by Xen VMM. Second, LVM supports creating writable snapshots of logical volumes quickly and each snapshot can be used as a COW disk slice by guest VMs. However, both of these two approaches have their limitations. Traditional *qcow* based COW has a limit on the total number of slices created and also has to make the trade-off between the size of the COW disk and the depth of the COW disk hierarchy. Deeper hierarchy leads to bigger image files. Figure 5(a) and Figure 5(b) illustrate two typical ways to create a COW disk partition. The linear COW approach in Figure 5(b) applies incremental COW slices onto existing disk partitions. The existing disk partition can be an initial base partition or a partition already having COW slices on it. The vertical hierarchy as shown in Figure 5(a) dedicates a VM to a single purpose with fewer applications installed, thus it is able to limit the resulted partition size to a certain extent. In order to avoid the high The root COW disk is the initial image file and usually installed with the JeOS, then multiple child COW disks are created afterwards with each taking the previously created root COW as its parent and install with different kind of application. Due to the IO scheduling of virtualized disks, more COW slices result in higher dependency and the more degradation of the performance in either mode. Moreover, it is very challenging to merge multiple COW slices to the base image because the order of updating disk file is usually not preserved. On the other hand, LVM snapshots usually apprear as a physical partition and requires using tools like ATA over ethernet(AoE) or iSCSI[14] to export COW slices when VMs need to be deployed across multiple hosts. Each new slice requires an update to the running AoE or iSCSI service to export a new disk partition. In addition, only recent LVM version supports merging a COW back to the base and it also needs to use the latest Linux kernel. In conclusion, disk COW slice is only applicable to temporary VM creation.



(a) Vertical COW (b) Linear COW (c) IO sync

Figure 5. Cow disk hierarchy and IO flow.

Xen disk block device supports split driver model and the VMM provides a mechanism for device discovery and data movement between domains. The device drivers are split across Domain 0 and guest domains which are also called back-end and front-end respectively. Domain 0 is responsible for supporting hardware, running back-end devices drivers and providing the administrative interface to Xen. This VM disk model allows that a VM's disk can be reconfigured. We leverage COW techniques for substrate-based VM deployment with some modifications to the existing COW mechanism. The objective of real time VM creation are two folds. First, in the long run, VM substrate-based VM creation should guarantee the correctness and should generate consistent application result compared to the VMs created from templates. Second, from the users' perspective, a VM can be created on the fly in a real time manner with small latency. Inspired by [17], We create a temporary COW slices and remap it to a newly created VM from substrate, giving users near realtime responses to the VM creation requests. The temporary COW slices work as the root partitions in order to speed up the booting process. At the same time, we duplicate the base image in the background. Once duplication of the base image finishes, instead of merging COW slice back to the original base, we merge the COW slice to the duplication of the base, removing the dependencies between the parent's base image and children's COW slices. We changed existing *qcow* to work as a buffer of disk updates supporting dynamically merging to any duplicated copy of its original base. Thus, the time-consuming disk duplication can be hidden as a background job. An externally synchronous file system has been proposed by Edmund et al. [17] to amortize modifications across a single commit where only external output will trigger file modifications to be committed. Similarly, our COW slice can be regarded as the buffer of modifications, the commit will be triggered when the duplication of base image is done. Figure 5(c) shows the synchronization of disk IOs when a new VM is created from a substrate. Each VM is assigned a COW slice initially, but will have its own independent disk partition in the long run. Step 1 groups multiple modifications before committing the changes to the disk. Step 2 and step 4 represent retrieving data from the base image and the COW slice respectively. Step 3 and Step 5 show that disk changes are synchronized to the COW slice. When a request of creating a new VM is received by a cloud manager, the duplication of the base image file is started as a background job. Other than synchronizing the COW slice to original base image, we synchronize the changes to new base image which is shown in step 6. After merging COW slices to the new base image. VM starts to read and write data directly from and to the new image as shown in step 7 and step 8. After step 8, the VM creation process finishes and the VM works just as the VMs created from a static templates. Note that the VMs created from substrates are online whenever the COW slices are ready (step 1), which gives almost real time responses to users' requests. In practice, the intermediate COW slices turn to be very small after merging, thus can be discarded with minimal cost. The original base image still remains reusable.

Evaluation. To understand the impact of shrinking degree on generating VM substrates and reactivating substrates, we shrunk a VM's memory from different sizes. We experimented with various memory sizes from 128M to 2GB and verified the time spent on preparing raw VM substrates and the time reactivating them. All the cached data was synchronized back to disk before docking. As shown in Figure 6, the sizes of a raw substrate are slightly larger than the memory footprint. If VM's memory can be shrunk to around 128M, the docking or reactivating can be done within 0.875 seconds. In our test, a VM with some applications like Webserver, MySQL database or program development environment installed could further be compressed, leading to a final VM substrate as small as 16MB.

4.2 Substrate multicast and compression

In our prototype implementation, we use multicast to dispatch VM substrates in parallel to other physical hosts. Traditional point-to-point communication has the drawback of inefficiency if a substrate needs to be sent to multiple hosts simultaneously. The transferring of VM substrates consumes considerable network bandwidth. In order to make sure that all the VM substrates are only transfered within the data center, we set the time-to-live (TTL) value of all multicast packets to be 1. Since the size of VM substrates can be as small as 16MB, the multicast packets can be encapsulated into the payload of TCP packets and can be sent quickly to another node in a LAN environment. Due to the small footprint of the substrates, our current implementation can alse be extended to a WAN environment connecting different data centers.

The raw VM substrates are compressed before moved to a pool. The objective of the compression is to make each VM substrate as small as possible. In our prototype implementation, we used the gzip algorithm to compress raw VM substrates. In order to reduce the cost of compression, the compression is done in memory and the resulted compressed substrates are also stored in memory temporarily before they are moved to the substrate pool. In our experiment, a VM with a development environment installed leads to a size of 16MB after compression. Compression of a substrate is more costly than decompression. Decompression usually takes less than half of the time than compression. The small cost incurred by decompression further speed up the launching process of a VM from the substrate pool.

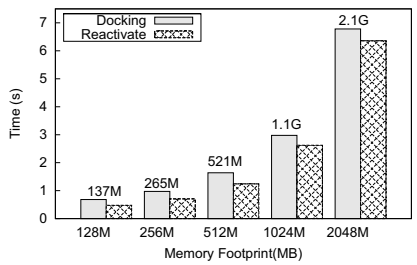


Figure 6. Impact of shrinking degree.

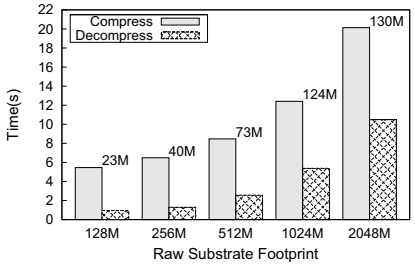


Figure 7. Compression cost.

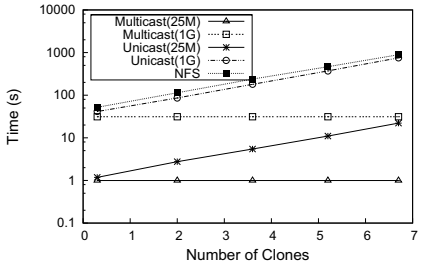


Figure 8. Effectiveness of multicast.

Evaluation. To evaluate the effectiveness of multicast, we compared the time spent on deploying multiple VMs from the same VM substrate. Figure 8 shows the strength of multicast, especially when the number of clones increases. In this experiment, we sent two different substrates with sizes of 25MB and 1GB to different physical hosts in order to create a group of new VMs. As shown in Figure 8, multicasting a 25MB substrate to different hosts took less than 1 second while sending the 1GB substrate took around 30 seconds. These are two extreme cases. In the more general case, VM's memory should be able to be shrunk to between 128MB and 1GB, most VMs with barely application environment installed could be shrunk to less than 200MB memory. Thus, the total time on multicast is in the magnitude of several seconds. On the other hand, in the case of unicast without using multicast, the total time of sending the substrates to the others would increase with the number of required clones. Figure 8 also plots the time of propagating the VM substrate by duplicating the saved state in a networked file system (NFS).

We also evaluated the cost of compression and decompression on the VM startup latency. We compared the time spent on conversion between raw substrate and final substrate when the size of raw substrate varied from 128MB to 2048MB. As shown in Figure 7, compression is more costly than decompression. The final size of each raw substrate is shown in the figure. For a raw substrate of 256M, which is the size for a typical VM after selective memory dumping, the decompression only took about one second. The startup laterncy incurred by the decompression algorithm does not significantly affect the users' experiences. Although compression is time consuming especially for large size of raw substrates, the compression is usually done before docking to prepare new VM substrate for future use which does not affect VMs' startup.

5. Evaluation

In this section, we examine the overhead and design a set of experiments to verify the effectiveness of VM substrate. We begin by examining the overhead of using a substrate to create new VMs, and then go on to explore one typical usage case of offloading mobile computation to a cloud environment. At the end of this section, we compare the cost of launching a VM with different methods.

The machines used in the experiments consist of a server dedicated to the VM pool and a client machine. All the experiments were conducted in a LAN environment connected by a Gigabit Ethernet switch. The physical hosts for the VM pool is a Dell PowerEdge 1950 server with two quad-core Intel Xeon CPU and 8GB memory. The client machine is a PC with dual CPU cores. We used Xen version 3.4.1 as our virtualization platform. Both dom0 and the guest VMs were running CentOS Linux 5.3 with kernel 2.6.18.

5.1 Overhead

We began our evaluation by examining the overhead of VM substrate. We study the latency of preparing a VM or VM cluster on demand. Figure 9 draws the time needed to create different number of VMs through VM substrate pool. In this experiment, we prepared several different VM substrates for each type of applications. Whenever a new VM is needed, in order to minimize the time spent on preparing virtual disks, we created a new VM using a temporary COW slice. The root partition of each VM is 4GB and the partition which is used to store the modification is set to 1GB.

In this experiment, we created different numbers of VMs from the same VM substrate and evaluated the absolute cost. The memory size of a raw substrate in this experiment was shrunk to 118MB, leading to the final compressed substrate of 16MB. This is the smallest size we can achieve with minimal installation of the guest OS and necessary running environment. We intend to answer the following questions in this experiment: (a) What is the optimal speedup VM substrate can achieve? (b) Where is the time spent on VM creation? (c) What is the scalability of the VM substrate approach?

Figure 9 shows the time for creating new VMs on demand from the VM substrate pool. The time is broken down into four parts: preparing the disk, multicasting substrate over local network, decompressing VM substrate, and activating VM. From this figure, we can see that the total time of creating a single VM from substrate is as small as 2.5 seconds. This time does not contain the time to generate VM substrates. It assumes that the substrate is always available in the pool. This figure shows VM substrate pool is capable of providing prompt response to laterncy sensitive VM creation requests. When the number of VMs to be created increases, the total latency of the VM creation does not increases significantly. This is due to the use of multicast, which does not incur proportional overhead when the scale increases. Similarly, the cost of transferring substrates to more than one physical host is almost the same as transferring to a single host. However, the cost of disk creation increases with the number of VMs. Note that the absolute creation time for a single disk is less than a second, given enough storage bandwidth, the disk creation part is not the limiting factor of the scalability of our VM substrate approach.

5.2 Case Study: Mobile Application Offloading

In this experiment, we analyze the effectiveness of VM substrate from the standpoint of cloud users. We implemented a usage case to create new VM on demand to offload the computation from mobile devices. We selected a mobile version of the chess game as the source of computation to be offloaded. It is representative because the user interface is simple and lightweight but the backend computation of the piece movement is computation intensive for mobile devices like smart phones. The chess game allows human players to play with a computer. We implemented an AI component on the server side to calculate each move for the computer side. This AI component is not only CPU intensive, but also consumes considerable amount of memory to record the game status of both players. For the computer side, It can make decisions for current

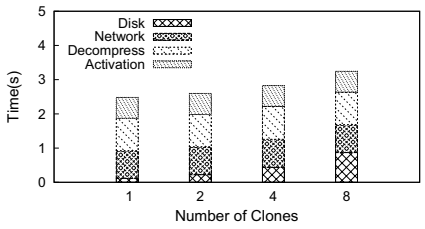


Figure 9. Time breakdown of VM creation.

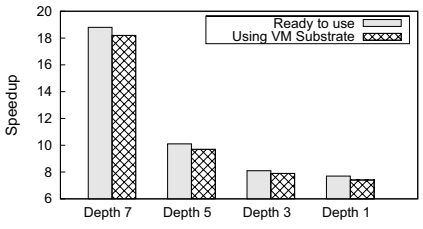


Figure 10. Delay time of offloading checker.

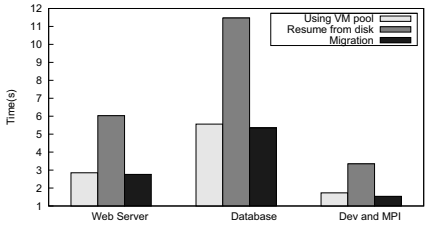


Figure 11. Startup time comparison.

move based on different number of steps looking into the future. More computation and resources are required if the computer side looks into the future with a large number of steps. Mobile devices are limited in terms of memory, computation power and energy reserves. If the mobile device is running an AI opponent which is configured with large resource requirement, the user would have to wait noticeable amounts of time for the opponent to make its move. The battery of the mobile device can also burn out quickly. In fact, if the computer side is configured to be intelligent enough, some early mobile phones may not even be able to run the chess game successfully. One solution is to offload the computation to a VM in cloud. Although offloading itself has some limitations because it requires computation of an application can be separated from the ches game itself and be executed on a remote VM server, it is still a viable workaround for many applications with simple frontend graphics interface and relatively loosely coupled backend. All the local mobile device needs to do is to collect and display the results. In our experiment, the chess game is one of the applications that can take advantage of on-demand prompt VM creation. Because the human player may have non-deterministic think time between two moves, we choose to dock the offloading VM after it finishes the computation for one move. When the human player initiate another move, the docked VM is restarted upon the human's move. In this setting, the latency incurred by VM startup from substrate directly affect human players' experience.

We compare the performance speedup of the chess game when the computation is offloaded to a VM with full capacity (i.e. "ready to use") with the performance when the computation is offloaded to a newly created VM from substrates (i.e. "using VM substrate"). The baseline performance was obtained from a cell phone running the android OS which has a 1Ghz CPU and 512MB memory. The speedup was calculated as the ratio of the processing time on the VM server over the time on the phone. Figure 10 shows very similar speedup between offloading to on-demand VMs from substrates and offloading to static VMs. It suggests that on-demand creation of VMs from substrate incurs mininal latency and thus can provide instant responses to users' request.

5.3 Performance Comparison

In practice, there are a few different options to start a new VM. These options includes suspend and resume [24], migrating VM from other hosts [4], creating VM from scratch and our VM creation from VM substrate Among these options, creating VM from scratch involves the whole OS installation process and takes a significant amount of time, which is not considered for comparison.

In this experiment, we created three new VMs containing a web server, a database server and a VM with development environment respectively in the above three different ways. Figure 11 draws the the startup time of these methods. The startup time is the time between the creation , migration or resume request is received and the time the VM is ready. A VM is considered ready when it is responsive to user's other request like launching a program. Technically, it is when all the virtual CPUs are back online, memory is ballooned back and network interfaces are attached. As shown in Figure 11, VM creation from substrate is almost as fast as VM migration. Note that VM migration need to maintain the VM running in its full capacity, which consumes a significant amount of resources limiting the scalability. In contrast, VM substrate maintains a large pool of substrates with mininal footprints. In our testbed with 8GB memory, we were able to host as many as 230 substrates. As expected, the suspend and resume approach incurred considerable startup time because the resume process needs to load a large state file from hard disk.

6. Related work

VM templates are widely used to create new VMs in the majority of system virtualization platforms. Through preparing reusable templates, which are usually configured to include a standardized set of hardware and software configuration settings, the efficiency of deploying VM infrastructure could be significantly increased due to the fact that many repetitive installation and configuration tasks are avoided. A base VM template contains the essentials of server image so called Just-Enough-OS(JeOS) and the base template can be extended by installing software application(s) in order to generate new template. VM templates[20] can be either converted to virtual machines and powered on without deploying them. The conversion will either turn the original template into VMs which means the template doesn't exist anymore or clone the templates to VMs through replication which involves time consuming disk copy. Moreover, starting a new VM created from a VM template needs error prone booting process.

The Amazon Elastic Compute Cloud (EC2) [6] is a widely used cloud computing platform. EC2 allows users to create an Amazon Machine Image (AMI) containing their applications, libraries, data and associated configuration settings or use pre-configured, template images to get up and running immediately. Amazon's EC2 claims to instantiate multiple VMs in "minutes" is still not enough to meet requirement of some real time VM creation requests. RightScale [22] also provides scripts to create and configure a basic VM from scratch. Although the installation and configuration are done automatically, it is often not applicable to on-demand VM creation due to the time consuming installation.

Some recent research work explores the idea of process fork to VM level where a running VM spawns child VMs that are clones of itself. The Potemkin project [27] realized a VM fork scheme that creates lightweight VMs from a static template locally within a single machine. Through aggressive memory sharing and COW techniques, Potemkin allows quick VM forking by deferring the duplication of memory pages until the contents of pages actually differ between VMs. It can support potentially hundreds of short-lived VMs on physical honeyfarm servers. However, Potemkin does not have the flexibility to create multiple VMs onto different hosts and does not offer runtime statefull cloning. Snowflock [11] extends the concept of VM fork in a distributed manner, enabling cloning a VM into multiple statefull replicas running in a cluster of

machines. Snowflock leverages the same COW technique used by Potemkin and takes advantage of the high correlation of the children VM, providing a immutable image of the parent VM and a demand-paging mechanism to let children retrieve missing pages. Similar to process fork, VM fork is able to efficiently share parent's resources and swiftly create interim VM clones that run simultaneously in a real time manner. However, current VM fork implementations do not aim to deploy longstanding independent VMs. VM substrate is different from Potemkin or Snowflock in their purposes. Potemkin and Snowflock aim to provide on-demand virtual clusters with "identical" and "temporary" VM children forked from a single parent. VM substrate's objective is to preserve and restore customized user working space (VM's with different running states) with minimal cost. The VMs in question are heterogeneous and not necessarily belong to the same user.

The idea of a pool structure is widely used in the design of computer systems. Most of early works focused on thread and process level pools [1, 13, 18, 21], or processor level pool [30]. The popular Apache web server [1] uses a thread pool to handle incoming request, but there is no resource reconfiguration for each thread. Iran Pyarali et al. [21] proposed an optimization to improve the quality of thread pools in real-time systems. They described the key patterns underlying common strategies for implementing RT-CORBA thread pools and evaluated each thread pool strategy from various aspects. In [13], Ling et al. characterized several system resource costs associated with thread pool size and analytically determined the optimal thread pool size to maximize the expected gain of using a thread and minimize the overhead of run-time memory allocation and deallocation while creating and destroying a thread. In [30], the authors proposed a class of scheduling algorithms based on a processor level pool which is used to organize and manage a large number of processors to improve performance.

7. Future Work and Conclusions

In this section, we briefly discuss a number of directions that we intend to explore in the future to improve and extend our VM substrate framework. As we have discussed in the previous sections, VM substrate based VM deployment is able to deploy diverse VM within seconds. The idea is preliminary and we plan to further investigate the following areas.

VM streaming. Our current implementation decompress the VM substrate to get the raw substrate and then start new VMs from the raw substrate. Although from Figure 7, we can see that decompression takes less time than compression, it is still costly to decompress the substrate when the memory footprint is large. Thus, a mechanism that allows a VM to boot while the decompression is in process will further reduce the startup latency.

Dynamically linked storage. Because VMs' resources such as vCPU number, memory size and network bandwidth are configurable, it makes the charge of VM resources in pay-as-you-go manner possible. However, storage is not so easily reconfigured as other resources. First, the change of disk size can not take effect without reboot even when LVM is used. Second, running VM's root disk is unable to be altered. Both of these two factors affect the agility of deploying VMs. On the other hand, if each VM can use dynamically linked storage, the actual physical disk partition can be dynamically changed.

Improved memory metering. As discussed in the previous sections, memory footprint is closely related to the final size of VM substrate. The smaller the memory footprint, the smaller the substrate. Our current implementation leverages the `proc` interface under Linux to get the memory utilization. Only the used memory pages need to be dumped in the VM substrate. Identification of unused memory pages or calculation of the memory utilization of a running VM is not trivial. Different from free pages, unused pages refer to those that once touched but not actively being accessed by the system. It can be calculated as the total memory minus the system working set. One possible direction is to integrate more accurate memory metering in VMM level.

In closing, we introduce the primitive of retrofitting VM deployment by using VM substrate and present the design, implementation, and evaluation of a novel approach to manage VMs in agile virtualized environment. Our VM substrate-based VM shrinking and expansion management allows VM creating, reconfiguration in a way that is transparent to users and enables the instantiation of statefull VMs or VM clusters with sub-seconds latency. Our VM pool architecture is effective in reducing the latency of preparing new VMs and increasing the reusability of VM substrates. It incurs small overhead on the creation of a single or a cluster of VMs. Experiment results on the computation offloading from mobile devices show that the pool of VM substrates is able to provide instantaneous response to user request in an interactive job.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments. This work was supported in part by U.S. NSF grants CNS-0702488, CRI-0708232, CNS-0914330, and CCF-1016966.

References

[1] Apache thread pool. http://commons.apache.org/sandbox/threadpool.

[2] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE*, 2007.

[3] B.-G. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *HotOS*, 2009.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[5] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *NSDI*, 2008.

[6] EC2. http://aws.amazon.com/ec2.

[7] S. Govindan, J. Choi, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: Communication-aware cpu management in consolidated xen-based hosting platforms. Jan 2009.

[8] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. pages 1–15, 2005.

[9] H. A. Lagar-Cavilla, N. Tolia, E. de Lara, M. Satyanarayanan, and D. O'Hallaron. Interactive resource-intensive applications made easy. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 143–163, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[10] H. A. Lagar-Cavilla, J. Whitney, A. Scannell, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Impromptu clusters for near-interactive cloud-based services. Technical Report CSRG-TR578, Department of Computer Science, University of Toronto, 2008.

[11] H. A. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *Eurosys*, 2009.

[12] F. Li and J. Nieh. Optimal linear interpolation coding for server-based computing. In *ICC*, 2002.

[13] Y. Ling, T. Mullen, and X. Lin. Analysis of optimal thread pool size. *SIGOPS Operating System Review*, 2000.

[14] K. Z. Meth and J. Satran. Design of the iscsi protocol. In *MSS*, 2003.

[15] A. B. Nagarajan and F. Mueller. Proactive fault tolerance for hpc with xen virtualization. In *In Proceedings of the 21st Annual International Conference on Supercomputing (ICS'07*, pages 23–32. ACM Press, 2007.

[16] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference*, 2005.

[17] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *In Proc. OSDI*, pages 1–14, 2006.

[18] S. Oaks and H. Wong. *Java Threads*. O'Reilly Media, Inc., 2004.

[19] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-796-4. doi: http://doi.acm.org/10.1145/1346256.1346258.

[20] Oracle VM Templates. http://www.oracle.com/technology/products/vm/templates/index.html.

[21] I. Pyarali, M. Spivak, R. Cytron, and D. C. Schmidt. Evaluating and optimizing thread pool strategies for real-time corba. In *LCTES*, 2001.

[22] RightScale VM Templates. http://blog.rightscale.com/2010/03/22/rightscale-servertemplates-explained.

[23] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Operating System Review*, 2002.

[24] M. Satyanarayanan, B. Gilbert, M. Toups, N. Tolia, A. Surie, D. R. O'Hallaron, A. Wolbach, J. Harkes, A. Perrig, D. J. Farber, M. A. Kozuch, C. J. Helfrich, P. Nath, and H. A. Lagar-Cavilla. Pervasive personal computing in an internet suspend/resume system. In *IEEE Internet Computing*, 2007.

[25] Virtual desktop infrastructure. http://www.vmware.com/pdf/virtual-desktop-infrastructure-wp.pdf.

[26] VMware. http://www.vmware.com/pdf/vc_2_templates_usage_best_practices_wp.pdf.

[27] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *SOSP*, 2005.

[28] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in hpc environments. In *SC*, 2008.

[29] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *VEE*, 2009.

[30] S. Zhou and T. Brecht. Processor-pool-based scheduling for large-scale numa multiprocessors. In *SIGMETRICS*, 1991.