Heterogeneous Live Migration of Virtual Machines

Pengcheng Liu, Ziye Yang, Xiang Song, Yixun Zhou, Haibo Chen *, and Binyu Zang

Parallel Processing Institute
Fudan University

Abstract

An indispensable feature enabled by system virtualization is the live migration of virtual machines (VMs). By dynamically relocating an entire execution environment including the operating system from one machine to another, a networked system such as data-center and computing grid can benefit from load balancing, online maintenance and fault tolerance. Ideally, a virtual machine should be able to be migrated across any node with similar machine configuration and granted resources. However, as there are currently a number of VMM vendors, the heterogeneity of the underlying VMMs makes it hard to live migrate a VM originally running on one VMM to run on another VMM.

In this paper, we propose Vagrant, a live migration framework which bridges the heterogeneity among diverse VMM abstractions and implementations. Vagrant supports the live migration of VMs across heterogeneous VMMs. We have implemented a prototype, that supports live VM migration between the Xen VMM and KVM. Experimental results indicate that Vagrant achieves acceptable downtime and total migration time.

1 Introduction

System virtualization has become a disrupting force to computer systems and is likely to be the new foundation of system software [7]. Evidences include the wide and rapid adoption of system virtualization in various usage scenarios, including data-center, grid computing, and recent cloud computing. One of the key enabling features provided by system virtualization is the ability of live migrating virtual machines (VMs) [18, 6, 16, 19, 4] among computing nodes. By dynamically relocating an entire execution environment including the operating system from one machine to another, a networked system such as data-center and computing grid can benefit from load balancing, online maintenance and fault tolerance.

One spirit of system virtualization should be to bridge the heterogeneity of underlying resources. By providing a common abstraction for guest operating systems, virtualization is capable of hiding the heterogeneity of hardware resources. Hence, ideally a virtual machine should be able to live across any node with granted resources, despite the heterogeneity of the underlying virtual machine monitors (VMMs). Unfortunately, there are currently a number of different kinds of VMMs including VMware Workstation/ESX server [22, 20], Microsoft Virtual PC/Server [14, 15], Microsoft Hyper-V [13], KVM [12], VirtualBox [10], and Xen VMM [2]. Each VMM usually provides its own abstraction of the hardware resources, which are usually not identical. Worse even, different versions of the same VMM may also have some differences in abstracting resources. These obstacles make it hard or even impossible to allow a VM originally running on one VMM to run on another VMM.

On one hand, the battle among VMM vendors in market may not likely have an end and no one may completely defeat others. On the other hand, the scale of federated computing is now rapidly growing and the computing model will likely cover more resources. Examples include the Chinagrid computing systems [11], PlanetLab [5] and cloud computing [9]. The adoption of more resources will inevitably introduce resources managed using different virtualization software. It is thus reasonable to see lots of heterogeneous VMMs running simultaneously within one federated computing system. This heterogeneity essentially violates the spirit of virtualization and may cause resource management problems. Such a violation downgrades the benefits brought by virtualization, such as platform mobility, load balancing, utilization and fault tolerance. To still retain the benefits of system virtualization despite the heterogeneity of underlying VMMs, it is desirable to support the live migration of virtual machines across heterogeneous

*Corresponding author: hbchen@fudan.edu.cn

VMMs.

This paper describes the design and implementation of Vagrant, a VM migration scheme aiming at migrating virtual machines among computing nodes even if they are managed by heterogeneous VMMs. To render it practical, Vagrant is designed without mandating changes to the hosted operating systems and its applications. It also requires only minimal changes to the core VMM abstraction.

To support live migration among heterogeneous VMMs, the key issue is the semantic gap among different resource abstractions and migration protocols. First, each VMM provides its own abstraction of the underlying hardware resources. Second, the migration software for different VMMs usually has its own format of migration protocols. Third, there are currently several memory migrating algorithms, including stop-and-copy [17], pre-copy [6], push and pull. Moreover, there are also different manners in migrating memory pages, including batched [6] and one-by-one [12] transfer.

To bridge the heterogeneity, we make a detailed analysis of several common VMMs, including Xen VMM, KVM and VirtualBox. Based on the analysis, we propose an applicable framework, namely Vagrant, to support heterogeneous live VM migration. Vagrant has a common migration protocol and common virtual machine abstraction. It intercepts the migration control and data issued by the source VMM (or migration software) and transforms them into the Vagrant format. On the destination side, the data in Vagrant format is transformed into the format of the target VMM. For memory migration algorithms, Vagrant provides a pool of common algorithms in both VMMs and dynamically selects the migration algorithms according to the types of the communicating VMMs.

We have implemented a prototype system based on Xen VMM [2] and KVM [12], two prevalent open-source VMMs. We measure the performance of Vagrant in terms of migration time and application performance during migration. The results show that the downtime and total migration time in Vagrant is close to Xen and KVM, and the application performance during migration is also similar.

In summary, we made the following contribution in this paper:

- The proposal of heterogeneous migration of virtual machines.
- The design of Vagrant, which aims at bridging the heterogeneity of virtual machine abstraction and algorithmic difference of different VMMs.
- The implementation and evaluation of our prototype based on Xen and KVM.

The rest of the paper is structured as follows: Section 2 describes the challenges in heterogeneous live migration as well as some background information. Section 3 explains the design choices and architecture of Vagrant. Section 4 describes the implementation issues. Next, the performance evaluation is presented in section 5. Then, section 6 describes related work. Finally, this paper ends up with some concluding remarks and future work in section 7.

2 Background and Challenges

Comparing with homogeneous live VM migration, live migration across heterogeneous VMMs faces many challenges due to the heterogeneity of VM abstractions. This section describes the heterogeneity in abstracting CPU, memory and I/O devices in common virtualization software. As migration is usually controlled by migration tools lying on the hosting VMMs, for simplicity, we use VMM to denote all the software components participating in live VM migration.

2.1 Virtualization of Hardware Resources

2.1.1 CPU Virtualization

Different VMMs may have different ways to virtualize CPUs. Traditionally, each VMM has its own virtual CPU module, which provides the abstraction of CPU to guest OSes. For example, the virtual CPU state in Xen's HVM includes general purpose registers, control registers, debug registers, floating point registers, segment registers, model specific registers, time stamp counter and information for pending events. By contrast, the virtual CPU state in Xen's para-virtualization includes more information, such as virtual IDT and virtual TSS. KVM's virtual CPU abstraction is similar to Xen HVM. Even if the abstraction is simmilar, the data format of virtual CPU state transferred in live VM migration is different.

The supported features of virtual processors provided by each VMM may also be different even for the same hardware platform [8]. In x86 architecture, the *CPUID* instruction is used by both applications and operating systems to obtain the supported features of underline CPU. Some VMMs present only a generic virtual processor to their VMs by selectively masking bits in CPUID, while others export almost all CPU features to VMs. A typical application usually obtains the set of available CPU features by executing CPUID only once during its startup time. Therefore, live migrating a VM to a VMM which exports different CPU features may cause applications to act unexpectedly [1, 21].

2.1.2 Memory Virtualization

Currently, many existing OSes assume that they have a continuous address space. In a virtualized environment, VMM manages the whole physical memory and allocates memory for each VM. The memory pages allocated for each VM may not be continuous. A typical memory virtualization solution is adding an extra level of address translation:

from guest physical address to machine address(host physical address). The guest OS may run in two different modes: direct mode and shadow mode. In direct mode, the page tables maintained by the guest OS maps guest virtual addresses to machine addresses directly. Guest OS only has read access to page tables and special APIs are needed to do page table update. In shadow mode, VMM maintains a set of shadow page tables for each VM. Shadow page tables map guest physical addresses to machine physical addresses while guest page tables map guest virtual addresses to guest physical addresses. Para-virtualization can support both the two modes while full virtualization usually only support shadow mode. In live VM migration, the machine pages allocated for the VM may be different in the destination host. Therefore, if the VM runs in direct mode in the source host, the machine addresses in guest page tables must be transformed to guest physical addresses. Furthermore, even if two VMMs implement the same mode (e.g., shadow mode), the exposed interface to manage them are different. For example, Xen maintains a reference count and type count for each page and requires transferring this information during memory migration. By contrast, KVM uses a much simpler way to manage physical memory.

2.1.3 I/O Devices

The device model presented to guest OSes differs significantly in para-virtualization and full virtualization. In para-virtualization, as the source code of the guest OS can be modified, there is no need to use existing device drivers and VMM can choose to expose simple devices to guest OSes. For example, instead of providing a common abstraction of each specific device, such as SCSI device and IDE device, Xen's para-virtualization provides an abstract block device which supports only two simple operations: read and write a block. In comparison, full virtualization solutions, including binary translation and hardware-based full virtualization, have to emulate each I/O device. For example, VirtualBox and KVM rely on QEMU [3] to emulate each existing hardware device, such as RTL8139 and NE2000 network interfaces. In live VM migration, the state of existing virtual I/O devices is transferred to the target and then loaded into the device model of the VM. The heterogeneity of device models poses significant difficulty for heterogeneous live migration.

2.2 Memory Migration Algorithms

Although moving the contents of a VM's memory from one physical host to another is straightforward, different VMM vendors may use different migration algorithms. By halting the VM, and copying whole VM state to the destination, pure stop-and-copy [17] minimizes the total migration time but can lead to unacceptable downtime. Demanding migration resumes the VM early and relies on the destination VMM requesting memory pages from the source VMM on demand. While Demanding migration incurs much shorter downtime, it may produce a long total migration time. By transferring dirty memory pages iteratively to the target while the VM is running, pre-copy migration [6] balances the requirements for minimizing both downtime and total migration time and is used by many VMM vendors, including Xen and KVM. VMWare's VMotion [16] adopts an even more complex algorithm by combining pre-copy and demanding migration. After a period of memory pre-copy, it pauses the VM and sends non-memory state to the target after which the VM is resumed. The resumed VM runs by pulling remaining memory pages from the source machine. Different VMM vendors may also use different manners for transferring memory pages. For example, Xen VMM uses a batched transfer manner while KVM uses a one-by-one manner. Table 1 shows the migration algorithms of several mainstream VMMs.

Table 1: Migration Algorithms of Different VMMs

VMM	Algorithm	Transfer Manner
Xen	pre-copy	batched
KVM	pre-copy	one-by-one
VMware	pre-copy & demanding	unknown

2.3 Migration Protocols

In live VM migration, a set of control commands are transferred between the source and destination VMMs. One challenge posed by heterogeneous live migration is that each VMM issues a different set of control commands to start and control the migration process. For example, Xen VMM initiates migration by sending a string "receive" to the target. By contrast, KVM sends a special integer to start migration. The migration of VM's state can usually be divided into several stages, such as memory migration and CPU state migration. For each stage, VMMs may issue specific commands to mark the beginning and end of the stage. For example, Xen sends "0" to indicate that memory migration has been finished while KVM uses "1" to mark the end of pre-copy.

3 Design

To render it practical, we design Vagrant without mandating changes to the guest operating systems and VMMs. To achieve such a goal, vagrant is built within the privileged VM or the hosted operating system. In this section, we first illustrate the architecture and working flow of Vagrant. Then we describe how Vagrant bridges the heterogeneity among different VMMs.

3.1 Architecture of Vagrant

Figure 1 shows the architecture of Vagrant. Vagrant enables heterogeneous live VM migration using three compo-

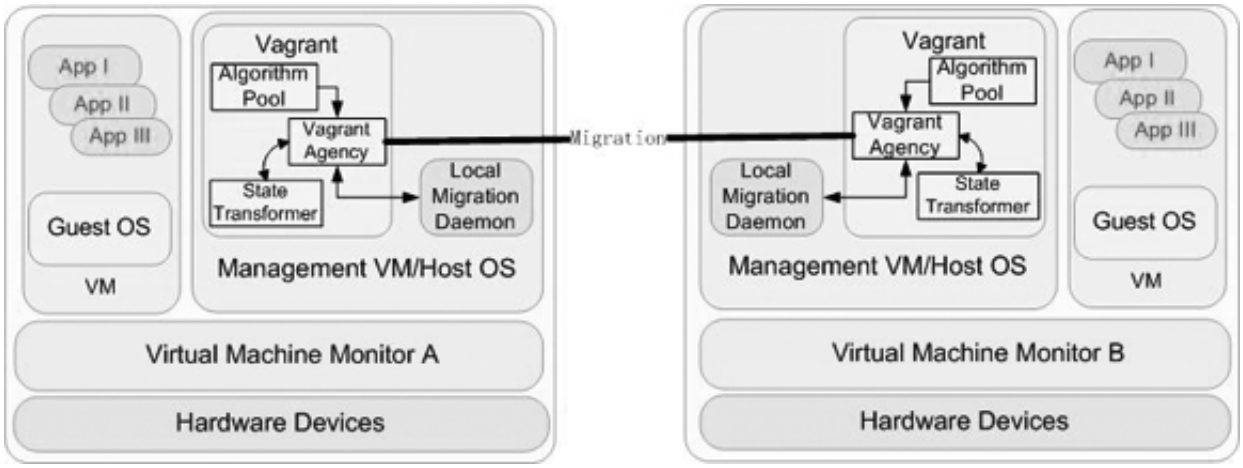

Figure 1: The heterogeneous migration framework. The Local Migration Daemon refers to the migration tool implemented in each VMM.

nents. The Vagrant Agency (VA) serves as a middle-man between two migrating VMM systems. It handles the negotiation between the source and destination VMM. The state transformer transforms VM state between Vagrant format and the specific VMM format. The algorithm pool implements several memory migration algorithms. A simplified working flow can be expressed as follows:

- VA at the source side intercepts the migration requests issued by the source VMM and establishes a connection with the destination VA.
- A memory migration algorithm is chosen from the algorithm pool and memory pages of the VM are transferred using the chosen algorithm.
- VA at the source side suspends VM and transforms VM non-memory state into Vagrant format by using the source side state transformer.
- VA at the source side transfers VM state to the target.
- VA at the destination side transforms the received VM state into the target VMM's format and forwards the data to the target VMM.
- VA at the source side closes the connection when the migration finishes.

3.2 Bridging the Heterogeneity

One precondition of heterogeneous live VM migration is the ABI supported by each VMM must be the same. It is impossible to live migrate a para-virtualized VM to a VMM which supports another para-virtual ABI or only supports full virtualization. However, paravirt_ops provides a common ABI for para-virtualization and is adopted by Xen, VMWare VMI and Lguest. This provides the potential for live migrating a para-virtualized VM between different VMMs.

Bridging the heterogeneity of hardware resources abstraction of different VMMs is the most important issue in Vagrant. Vagrant is required to accommodate mainly three kinds of virtual resources, including CPU, memory and I/O devices. For memory, traditional live migration requires that guest page tables should store mappings between guest virtual address and guest physical address. This is also an important issue in heterogeneous live VM migration. The following paragraphs discuss issues related to abstracting CPU and I/O devices.

CPU The virtual CPU module in each VMM is different, especially for VMMs using different virtualization solutions. However, not all virtual CPU state needs to be transferred in live migration. For example, the information of processor features maintained by the KVM virtual CPU module is not transferred to the target. Vagrant tends to extract the essential virtual CPU state from each VMM's abstraction. After a detailed analysis of several common VMMs, including Xen (both para-virtualization and HVM), KVM and VirtualBox, Vagrant proposes a common virtual CPU format which integrates all essential virtual CPU state needs to be transferred in heterogeneous live migration.

As each VMM may expose a different set of CPU features to guest OSes, live VM migration between different VMMs may cause applications to act unexpectedly. Live VM migration between different hardware platforms faces the same problem. Many VMM vendors solve this problem by only exposing a set of common processor features to the guest software, including guest OS and guest applications. To ensure the proper behavior of guest software after live migration, VMM has to control the return information of CPUID instruction executed by guest software. For hardware based virtualization, all the CPUID instruc-

tion executed in guest software can be intercepted easily by the VMM. However, for binary translation and para-virtualization, VMM needs the ability to control CPUID instruction executed in guest applications. The AMD-V ^{TM} extended migration technology [1] provides capabilities to VMMs for specifying the subset of processor features returned by CPUID instruction. In heterogeneous live migration, Vagrant requires that the destination VMM must have the ability to provide a compatible set of processor features. This is ensured by the migration protocol implemented in the VA.

I/O Devices Each VMM provides its own device models to guest OSes. We believe it is generally hard to bridge the heterogeneity between para-virtualized and emulated I/O device models. As hardware based virtualization should be the trend of virtualization, device emulation is now widely adopted by VMMs. So Vagrant mainly focuses on emulation-based virtual devices. Currently, many VMMs rely on an extra device emulation module, such as QEMU[3], to emulate each I/O device. The device emulation module runs as an independent process in the privileged domain or host OS. For each VM, there is an instance of the device models which waits for I/O events from the VM and dispatches it to the device emulation module. Device emulation provides the potential for bridging the heterogeneity of device models in different VMMs. As long as the emulated device type is the same, the virtual device state can be easily transformed and reloaded into another device model. On the source side, the state transformer receives the local I/O device state from the VA and transforms it into the Vagrant format. On the destination side, the received I/O device state of Vagrant format is transformed into target device emulator's format and loaded into the target device model.

Each VMM's device emulation module may support a different set of I/O devices. Live migrating a VM to a VMM which cannot support all the existing I/O devices will cause the VM to lose state or even crash. Similar to CPU state, Vagrant requires that destination VMM must have the capability to support all I/O devices existing in the migrated VM. Otherwise, it should be able to write a state transformer that seamlessly transfer state between two virtual devices.

3.3 Memory Migration Algorithms

In Vagrant, the algorithm pool provides different kinds of memory migration algorithms. During the first stage of migration, the source side VA negotiates with the destination side VA and chooses a compatible migration algorithm from the migration pool. The precondition is the migration pool on each side has implemented at least one compatible migration algorithm. If there are more than one candidates, the algorithm can be specified by the user or is chosen by VA randomly.

3.4 The Migration Protocol

Migration protocol is used to initiate and control the whole process of live migration. For each VMM supporting live migration, there is always a migration daemon listening on a port through which the source and destination VMMs can establish a connection. Once the connection is established, the source VMM begins migration by sending a set of control commands. Generally, the source VMM first sends a signature to ensure the destination VMM is compatible with the source. After that, a configuration file of the VM will be sent to the destination. On receiving this configuration file, the destination VMM checks whether it has enough physical resources to host such a VM. If it does, the destination does some preparation for receiving the state of the incoming VM, such as memory allocation, and issues commands to notify the source VMM to begin VM state transfer. The migration of VM's state can usually be divided into several stages, such as memory migration, CPU state migration and I/O device state migration. For each stage, special commands are issued to mark the beginning and end of the stage.

As VMM's migration protocol usually differs from each other, Vagrant proposes a common migration protocol and a set of migration commands to initiate and control the heterogeneous live VM migration. The VA intercepts all the migration commands issued by the local VMM and replaces them with the commands of Vagrant. On receiving a migration request, VA on the source side issues a special command to establish a connection with VA on the destination side. As the VMMs in pre- and post- migration environment may provide different capabilities, such as processor features, I/O devices, the source side VA negotiates with the destination VA to ensure the destination VMM can provide the same capabilities required by the migrated VM. If the negotiation succeeds, the source side VA chooses a proper migration algorithm from the algorithm pool and starts migration using the chosen migration algorithm.

4 Implementation

We have implemented a working prototype of Vagrant that supports live migration of VMs between Xen and KVM. Our current implementation is based on x86 architecture with VT-x extensions.

The following subsections discuss the specific implementation issues of Vagrant. We first present the migration protocol used by Vagrant. Then, we discuss the common abstraction of hardware resources in Vagrant. Finally, we present the implementation of the memory migration algorithm in Vagrant.

4.1 Migration Protocol

In live VM migration, a configuration of the VM is usually sent to destination VMM for resources reservation. The

Table 2: Part of Control Commands in Vagrant

Usage	Source	Destination
request	MIG_REQ	REQ_OK,REQ_FAIL
send config	MIG_CFG	CFG_OK,CFG_FAIL
begin pre-copy	MEM_BEG	no response
send cpu state	CPU_BEG	no response
send I/O devices	DEV_BEG	no response

configuration sent by Xen is very detailed, including memory size, disk image location, VM name and etc. In comparison, KVM only sends memory size of the VM to the destination. In heterogeneous live migration, a detailed information is required to be collected as it must be ensured that the destination VMM can provide the same capabilities as the source VMM, otherwise the migration may fail midway. Vagrant's format of configuration is based on Xen's format. Generally, it specifies the required memory size, general I/O devices and CPU features requested by the VM. It also includes a signature which shows the identity of the migrated VM. On receiving the configuration file, the VA on the destination side checks whether the local VMM can provide the required capabilities. The migration process proceeds only when the check of capabilities is successful.

Based on the analysis of the migration protocol of common VMMs, Vagrant uses a common set of control commands, to initiate and control the whole process of live migration. Table 2 shows an incomplete list of them. Some of the commands have corresponding response commands, while others do not have. For example, the command "MIG_REG", used by the VA to initiate live migration, may be responded by a "REQ_OK" or "REQ_FAIL" command. "REQ_OK" means the target VMM is ready for live migration while "REQ_FAIL" tells the migration can not proceed any more. The command "MEM_BEG" marks the beginning of memory migration. It does not have any response command and the source side vagrant agency begins sending memory pages immediately after issuing this command. During live migration, as the destination VMM may be unresponsive for various reasons, such as network problems, the VA sets a 3 seconds timeout for each command that needs a response.

4.2 Abstraction of Hardware Resources

4.2.1 CPU

In KVM and Xen, most CPU features are exposed to the guest software, such MMX and SSE. Some of the features can be configured, such as APIC and ACPI. In our implementation, we transfer the capability of the KVM and Xen to support a joint set of both VMMs' CPU capabilities. That is, a migration of VM will not downgrade its capability. The virtual CPU state of Vagrant includes the state of standard registers, control registers, segment registers, debug registers, floating point unit, sysenter registers, model specific registers, time stamp counter and pending events. The virtual CPU state of KVM and Xen VMM to be transferred in live migration is first transformed into Vagrant's format before migration and then transformed back after migration.

4.2.2 I/O Devices

In Vagrant, the migrated VM is configured with keyboard, mouse, rtc, PIC/APIC/IOAPIC, PIT, IDE disk, NIC, serial port and VGA. As most virtual I/O devices in KVM and Xen VMM are emulated by QEMU, the development effort is significantly reduced.

However, for the sake of performance, both KVM and Xen VMM have moved some virtual devices out of QEMU. While KVM has kernel level support for PIC/APIC/IOAPIC, Xen VMM moves more virtual devices into the VMM, such as PIC/APIC/IOAPIC and PIT. Therefore, in the source VMM, Vagrant merges the QEMU and VMM maintained device state and transforms it into the common format. In the destination, Vagrant transforms the device state into target device model's format.

4.3 Migration Algorithm

Currently, we only implemented a pre-copy based algorithm for the algorithm pool. The algorithm begins by copying all memory pages of the VM to the destination VMM. Then it iteratively copies the dirty pages in batched manner. Currently, the maximum batch size is 1024, which is the same as Xen's implementation. In each iteration, Vagrant first sends the batch size and page frame number of each page, then it sends the contents of the whole batch. During each iteration, Vagrant also determines whether the threshold has been reached. At this point, further pre-copy will make no progress. When the threshold is reached or the pre-copy is complete, Vagrant halts the VM and copies the remaining memory pages to the destination.

An important issue should be considered in memory migration is the tracking of dirty pages. In both KVM and Xen, there are two types of bitmaps recording the dirty pages in each iteration. While the dirty bitmap of normal pages is maintained by KVM kernel module or Xen VMM, the bitmap for pages used by DMA is maintained by QEMU. By merging the bitmaps maintained by VMM and QEMU, Vagrant differentiates the dirty pages from others at the start of each iteration.

5 Experimental Evaluation

In this section, we measure the performance of Vagrant to answer the following questions: (1) Whether VM migrations using Vagrant can have an acceptable total migration time and minimal downtime? (2) Whether applications during VM migration using Vagrant can still provide an acceptable level of performance? (3) Is Vagrant's performance

comparable to other migration frameworks such as Xen and KVM?

5.1 Experimental Setup

All tests were performed on a pair of PCs with 2.33GHz Intel Core(TM)2 Duo CPU with 2GB RAM, an RTL8169 Ethernet NIC, and a single 300GB 7200 RPM SATA disk. Another PC with a 3.0GHz Pentium IV with 1GB RAM, an Intel Pro 100/1000 Ethernet NIC and a single 250 GB 7200 RPM SATA disk was used as NFS server. The machines were connected via switched Gigabit Ethernet. The version of Xen VMM was 3.1.0(HVM). The KVM used was KVM-48 running on Linux 2.6.23. The migrated VM was provided with 128 MB of RAM and used a 5GB disk image with installation of Fedora Core 6. In all the evaluations, there was only one VM running on the source machine and there were no VMs running on the destination machine(We do not consider the privileged DOM of Xen as a VM).

5.2 Migration Time

The two metrics usually concerned in live VM migration are downtime and total migration time (end-to-end time). To gain a comparison results of Vagrant, four kinds of migrations were performed, including migrating a VM from KVM to KVM, Xen VMM to Xen VMM, KVM to Xen VMM and Xen VMM to KVM. All kinds of migrations were evaluated under different VM workloads, including idle, kernel-compile, memtest86 and apache *ab* benchmark.

Downtime In Vagrant, we divide migration downtime into normal downtime and state transfer time. Normal downtime consists with the downtime incurred in homogeneous live migration and usually consists of the time to pause the VM on the source, transfer remaining memory pages and device state to the target, and load the device state in the target. State transfer time consists of the time to transform source side device format into Vagrant device format and then into target device format. In our evaluation, the state transfer time is less than 1 millisecond.

Figure 2 shows the downtime of all four kinds of migrations on different workloads. The downtime of heterogeneous live migration is comparable to homogeneous live migration. For most workloads, the downtime is less than or close to 1 second. As shown in the results, the migrations between Xen VMMs incur a longer downtime than KVM migration. It is because during live migration, Xen spends several hundreds of milliseconds to pause the VM and saves the VM state in a file.

End-to-end Time End-to-end time consists of memory pre-copy time and downtime. For memory pre-copy, Xen VMM uses a batched manner while KVM uses a one-by-one manner. Vagrant's common algorithm behaves in a batched manner. Figure 3 shows that after we have changed the memory pre-copy manner, the migration between KVM and Xen outperforms the migration between KVMs for all

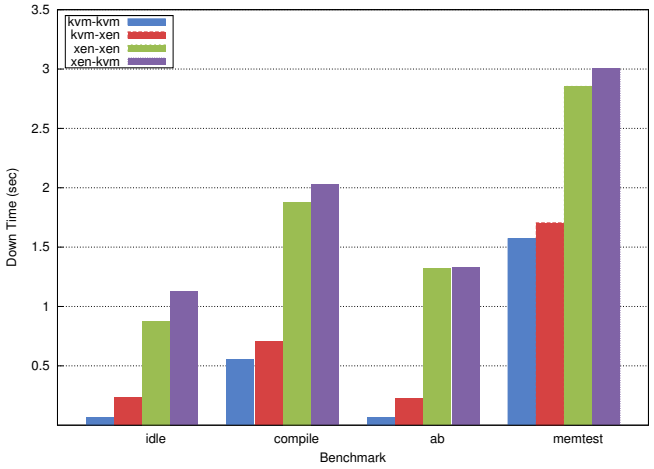

Figure 2: Migration downtime for various workloads.

workloads. The time of Xen-to-KVM migration is also shorter than Xen-to-Xen migration. This is because Xen-to-KVM migration requires smaller number of pre-copy iterations.

KVM outperforms Xen for all workloads except for memtest86. memetest86 dirties memory pages so quickly that several pre-copy iterations are required to migrate the loaded VM. Xen's batched memory transfer manner shows its advantage in this scenario. Nevertheless, Vagrant performs comparably to Xen VMM.

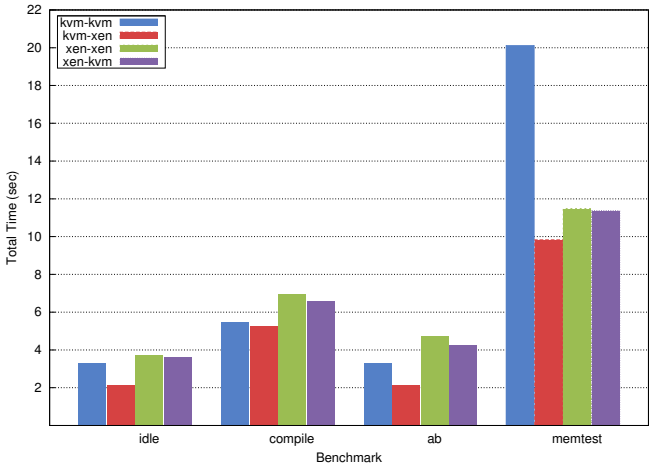

Figure 3: Total migration time for various workloads.

5.3 Migrating a Running Web Server

To examine the performance of Vagrant under heavy workloads, we evaluate the prototype by migrating a VM running an Apache 2.2.3 web server. The web server continuously serves a 256 KB file to a remote client. Figure 4 illustrates the throughput during the migration between

KVM and Xen. As shown in the figure, the total migration is rather small and the downtime is minimal. This experiment shows that Vagrant can successfully live migrate a heavily-loaded VM between KVM and Xen with very little performance impact.

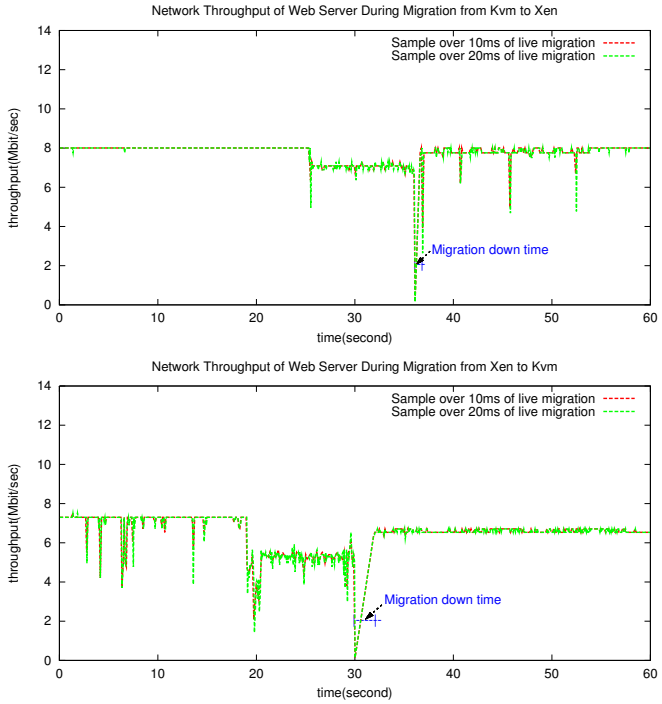

Figure 4: Performance of Apache during VM migrations between KVM and Xen.

6 Related Work

While live VM migration has been heavily studied in the past few years, our work differs from previous work in that it is the first framework to support live migration of virtual machines among heterogeneous VMMs. The following discussion only focuses on the most closest work to Vagrant.

Xen VMM [6] can dynamically relocate a VM among different hosts in a LAN. By pre-copying memory pages iteratively before suspending the VM, it achieves a relatively low downtime. Xen determines the end of the pre-copy phase based on the analysis of the *writable working set* behavior of several typical server workloads. After suspending the VM, it transfers CPU and I/O device states to the target host. To avoid disrupting active services through resource contention with the migrating OS, they use dynamic network rate-limiting to balance network contention against downtime.

KVM [12] also supports live VM migration within a LAN. Similar to Xen, it iteratively pre-copies the VM memory to the target in parallel with normal VM execution. However, the memory pages are transferred in an one-by-one manner while a batched manner is used in XEN live migration. The termination criteria of pre-copy is relatively simple: the number of copied memory pages is increasing in each iteration or thirty iterations have elapsed. VMware VMotion [16] enables live migration of VMs running on VMware ESX Server. The migration process involves 5 steps which are also similar to XEN live migration.

"VM Turntable" demonstrator [19] shows that VMs can be live migrated over long-haul networks with negligible downtime. The long-haul scenario poses new requirements for live migration. Due to the high round trip times (RTTs) experienced in a long-haul scenario, the migration process is required to pack and transfer more state than just memory pages, such as hard disk state. "VM Turntable" demonstrator does not solve the problem of migrating disk state, however, it demonstrated that the goal can be accomplished by relying on the snapshot capability of a Logical Volume Manager. Another requirement posed by long-haul live migration is that the VM can not rely on the usual routing infrastructure, the use of which would cause the VM to acquire an new IP address. To accomplish this, "VM Turntable" demonstrator uses dynamically configured IP tunnels to retain VM connectivity.

Bradford et al. [4] enables wide-area network (WAN) migration by not only transferring the in-memory state of a VM, but also transferring the local disk state. Each host maintains an immutable template disk image. Therefore, only the differences between the template and the customized disk image are transferred to the target host. As VM's IP address is changed in long-haul live migration, they maintain VM's ongoing network connections by forwarding all packets for the VM's old IP address to the target. In a short time, the migrated VM seems to have two IP addresses. The old one is used by existing network connections while the new one is used by new connections.

7 Conclusion and Future Work

We have presented Vagrant, a heterogeneous live VM migration framework that enables live VM migrations among different kinds of VMMs. Based on the study of heterogeneity of different VM abstractions and migration algorithms, we designed a common migration framework that provides general abstraction of VMs and migration protocols. We have also implemented a working prototype that supports the live migration of VMs between Xen VMM and KVM. Performance measurements indicate that the downtime and end-to-end time in Vagrant are comparable to homogeneous live VM migration.

In our future work, we plan to extend Vagrant in several ways. First, we are currently working to make vagrant more full-fledged by investigating its implementation on other virtual machines, such as Virtual Boxes and VMware. Second, as currently Vagrant relies on network file systems and

only support migration within LAN, we plan to extend it to support live migration in wide-area network. Third, we plan to add more desirable features such as QoS control to make Vagrant more flexible and robust.

References

[1] AMD. Live Migration with AMD-V ^{TM} Extended Migration Technology. http://whitepapers.theregister.co.uk/paper/view/375/.

[2] 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 *Proc. SOSP*, pages 164–177, 2003.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proc. Usenix, Freenix Track*, pages 41–46, 2005.

[4] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proc. VEE '07*, pages 169–179, 2007.

[5] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

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

[7] S. Crosby and D. Brown. The virtualization reality. *Queue*, 4(10):34–41, 2006.

[8] Y. Dong, S. Li, A. Mallick, J. Nakajima, K. Tian, X. Xu, F. Yang, and W. Yu. Extending Xen with Intel Virtualization Technology. *Intel Virtualization Technology*, 10, 2006.

[9] D. Gannon. Head in the clouds. *Nature*, 449, 2007.

[10] Innotek. Innotek virtualbox. http://www.virtualbox.org/.

[11] H. Jin. ChinaGrid: Making Grid Computing a Reality. In *Proc. ICADL 2004*, 2004.

[12] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Proc. Linux Symposium*, 2007.

[13] Microsoft. Microsoft Hyper-V. http://www.microsoft.com/virtualization/.

[14] Microsoft. Microsoft Virtual PC. http://www.microsoft.com/windows/products/winfamily/virtualpc/.

[15] Microsoft. Microsoft virtual server. www.microsoft.com/windowsserversystem/virtualserver.

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

[17] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proc. OSDI*, 2002.

[18] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proc. OSDI*, pages 377–390, 2002.

[19] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. de Laat, J. Mambretti, I. Monga, B. van Oudenaarde, S. Raghunath, and P. Y. Wang. Seamless live migration of virtual machines over the man/wan. *Future Gener. Comput. Syst.*, 22(8):901–907, 2006.

[20] VMware. Vmware esx server. http://www.vmware.com/products/esx/.

[21] VMware. Vmware vmotion and cpu compatibility. http://www.vmware.com/resources/techresources/1022.

[22] VMware. Vmware workstation. http://www.vmware.com/products/ws/.