Toward Automatically Deducing Key Device States for the Live Migration of Virtual Machines

Guodong Zhu*, Kang Li*, Yibin Liao*
*Department of Computer Science, University of Georgia
{guodong, kangli, liao}@cs.uga.edu

***Abstract*—The ability of migrating running virtual machines (VMs) in cloud environment provides significant benefits in dynamic resource load balancing and higher fault tolerance. The migration of a virtual machine consists of both the application/OS memory state migration and virtual hardware device state migration. Failures in the either of these two may lead to unpredictable behavior of the running VM. Previous researches have focused on the consistence of the application/OS memory state. In this paper, we inspect the migration of device states, which are also essential for the success of VM live migration. The current practice of defining device states is done by the developers of each virtual device and thus is prone to errors. We present an approach that automatically deduces what states are critical for a virtual device. Having the precise states defined is critical for the success of VM live migration.**

***Keywords*-Virtualization; Live Migration; Virtual Device;**

I. Introduction

Cloud computing revolutionizes the IT field by allowing users to access computing resource without physically own the hardware equipments. Virtualization is the underneath enabling technique, which provides the ability to launch, host, and migrate VMs in cloud environments on demand. Among all these cloud computing features, live migration of virtual machines provides significant benefit in resource management and fault tolerance.

Live migration of virtual machines consists of application/OS memory state migration and device state migration [4]. To make the live migration transparent to cloud users, the switching of VMs across physical hosts needs to be as fast as possible. To achieve fast migration, previous work [5], [9], [10] have proposed to use on demand memory copies. Once the virtual machine device states and a small OS footprint have been copied over, a VM can start to execute on the new host with additional application and OS memory copied over when needed.

Because migrating virtual device states is the very first step [5], it is critical to minimize the size of memory being copied in this round. Therefore, for all the memory related to those virtual hardware, only the key states (such as device I/O registers) that governs the upcoming behaviors of the devices need to be migrated. Using QEMU [2], a popular VM platform, as an example, its current practice is to have the virtual device developers specify what are the states need to be saved for the migration. These states are usually selected manually by inspecting the hardware specification and software driver implementation. However, this manual effort is error prone. If some important states are not included, a virtual device may not behave properly after the migration. Moreover, these problems are sometimes hard to detect. Missing device states does not always lead to an immediate crash of the VMs or the virtual devices, and errors could occur at a much later stage of VM execution.

We proposed a project to study the implementations of various VM platforms, especially their behaviors during live migration. The proposed efforts include developing environments to evaluate the virtual device implementations at the time of live migration, developing tools to analyze VM platforms and extracting critical device states, and scanning current VM platforms to detect implementation bugs.

This paper presents our initial effort of studying the virtual device behavior at the time of live migration. In this paper, we present the design of **VDSChecker**, a tool that can automatically check the completeness of the transferred device state during the live migration of VMs. By exploring the the difference in the behavior of the same virtual device between whether a live migration is involved during the execution, we can detect whether the migrated device state is sufficient for the virtual device to continue running as expected, and which device state is missing if not.

This paper makes the following contributions:

- We differentiated the requirement of device state migration from the application/OS memory migration, and exposed the potential problems of missing key state during live migration.
- We developed an environment that can mimic live migration of virtual machines in a program controlled way. With our environment, a tester can choose at which line of code a live migration event occurs.
- We proposed a solution that extracts the key states of a device by applying program analysis to virtual device implementations, which are essentially software programs. A software tool that does the automatic key device state extraction is current under development.

II. Virtual Devices and Key Device State

Although live migration can be applied on many full system virtualization platforms, our efforts focus on QEMU [2]

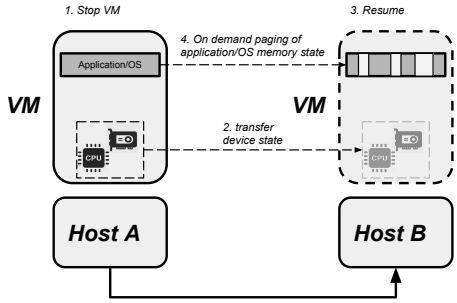

Figure 1. QEMU post copy live migration

virtual devices. We choose QEMU for two reasons. First QEMU is open source, meaning all the virtual device implementations are publicly available and easily accessible. Second, QEMU serves as the basis for many other widely used virtualization platforms such as KVM [8], XEN HVM [1], and VirtualBox [11].

A. QEMU and its Virtual Devices

QEMU is a generic CPU emulator and virtualizer that is able to emulate many different processor architectures including x86, SPARC, and ARM. As a full system virtualization platform, QEMU is able to emulate an elaborate set of peripheral devices including keyboards, mice, disk controllers, graphics cards, and network interface cards. The programs that emulate these devices are called *virtual devices*. These virtual devices are usually modeled after existing pieces of hardware. For example, the e1000 virtual device is modeled after Intel's E1000 gigabit network card.

The layout of a QEMU virtual device implementation shares many similarities with a device driver code. Most devices define a state structure, and register a set of interface functions to enable communication with the guest operating system or the environment through QEMU, these interface function then invoke transaction functions to handle the requests or the environment variables.

B. Live Migration of QEMU

Live migration includes device state migration and memory state migration. QEMU implemented post-copy live migration [6], as shown in Figure 1, with which the CPU and device state will be transferred before the memory state. During the migration, the virtual machine will be brought down on origin host when migrating the device state and continue running on destination host as soon as the CPU and device state be transferred, the memory state will be transferred to the destination host along the normal execution of the virtual machine.

C. Key Device State

When a virtual device gets migrated from one host to another, not all its variables need to be copied during the migration to ensure the correct execution. Only a subset of all the virtual device implementation's variables are needed and we call them ***key device states***, which govern a virtual device's upcoming behavior after migration.

Each QEMU virtual device implementation has a specific piece of code that defines what variables to save/load at the time of migration. These device state variables are defined in the *VMStateDescription* structure, which is manually specified by the developer. Figure 2 shows code excerpts for e1000 implementation of *VMStateDescription* structure.

This *VMStateDescription* data structure is supposed to capture all the key device state variables and only those variables. Containing variables that are not critical to the virtual device will affect the performance of the live migration, because the downtime of the live migration of QEMU is heavily dominated by the time spent on copying the device state [5]. Missing key device state variables from the *VMStateDescription* data structure is even worse, which case lead to misbehaviors and possible crashes of VM after migration.

Unfortunately, the current practice of QEMU virtual device implementation is that – this *VMStateDescription* structure is manually defined and filled in by the developer. For example, in the e1000 virtual device implementation in QEMU 1.7.1, there are more than 80 lines of code for defining what device state to transfer during the live migration. Figure 2 shows code excerpts for the e1000 implementation of this piece of code. Hypothetically, if the *rxbuf_size* state is not transferred during the live migration, the virtual device will not respond to any network packets or I/O requests before a reset request is sent to the device.

III. Design

We propose to design a software tool to automatically extract the key device states that must be copied at the beginning of a live migration. During a VM live migration, all the device states on the destination host will be set to initial value except the ones that are included in the *VMStateDescription* structure. Ideally, this *VMStateDescription* structure should contain all the key variables of a given virtual device implementation.

Our approach is to use program analysis techniques to study the behavior of these programs. The key observation is that, although mimicking complex hardware, virtual devices are actually software implementations. Therefore, the idea of automatically deducing the key device states is to analyze the virtual device implementation and detect which variables can not be reset to initial value during the execution without changing the program's behavior.

Although conceptually promising, we need to solve a couple of challenges in order to analyze virtual device

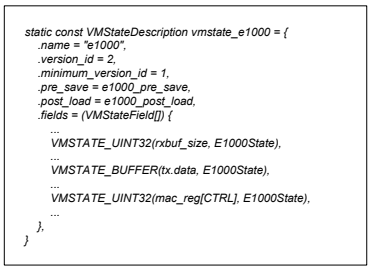

Figure 2. VMStateDescription structure defines the key device states need to be transferred during live migration

implementations and extract key device states. The major challenges are the followings:

1) *Not Standalone Programs* – virtual device implementation is not a standalone program, but existing program analysis techniques [3] prefer to have complete programs as inputs.
2) *Many Non-deterministic Inputs* – A virtual device program executes as a part of the virtual machine platform. The input to the program can vary, and any program analysis effort that tries to derive the key variables of the program needs to consider all possible inputs.
3) *Large Test Space for Migration* – A live migration can occur at many different execution points of a virtual machine. In terms of software implementation, a live migration could occur in between any two event handlers. A key state variable might only be needed under a special event sequence, and thus finding all key state variables needs to consider a large test space.

For each of the above challenges, we provide the following solutions (and some of them are still under development).

A. Virtual Device Extraction

The virtual devices by themselves are not executable, in order to extract a virtual device from QEMU implementation for the analysis purpose, a wrapper is needed to initialize the virtual device and steer its execution to explore the paths. With this wrapper we need to implement two main features:

1) Virtual Device Initialization: In order for the extracted virtual device to function properly, the device state need to be initialized, just like the device initialization process when starting a virtual machine in QEMU.

Because of the hierarchical model of QEMU hardware implementation, there are lots of references to the parent device objects in the device implementations, thus the initialization of one particular device requires its parent initialized first. So we need to recursively include the initialization of the parent devices before the device of interest can be properly initialized.

2) Steering the Execution of the Virtual Device: The virtual devices communicate with QEMU through its interface functions. When the device driver issues a request or there are I/O events related to the device, QEMU will invoke the corresponding interface functions to handle the task. We can steer the execution of the virtual device by invoking the interface functions in the wrapper.

B. Virtual Device Behavior Exploration

VDSChecker takes advantage of a software testing technique called symbolic execution [7] to solve the non-deterministic input problem, in particular, we use KLEE symbolic execution engine, which is capable of automatically generating test cases to enumerate the execution paths of the virtual devices [3]. With KLEE, we are able to achieve very high code coverage when testing a virtual device implementation and generate inputs that will lead to each of the paths. Also, by generating the path constraints that the inputs and variables have to satisfy for the virtual device to take a particular path, we are able to keep track of which device state under what condition may lead to errors of live migration.

C. Live Migration Simulation

In our approach, live migration is simulated by mimicking the the operations performed on the virtual devices during the migration, it consists of two parts:

- *Device State Save and Restore* – Save all the device states defined in *VMStateDescription* structure and load them to a newly initialized virtual device.
- *Massaging Functions* – QEMU introduced massaging functions, i.e. pre-save, pre-load and post-load functions, to solve the problems that there are situations when the key device state need to be copied from outside the virtual device data structure, our simulated migration also need to execute these functions to keep the device state correct.

We can address the problem of large test space for migration by enumerating all the combinations between two of the interface functions of a virtual device non-deterministically with simulated migration involved in between of the two interface function invocations.

Once the test cases are generated, we can compare the two sets of test cases, one with live migration involved during the execution of the virtual device and one without, and determine the completeness of the saved device states. The comparison consists of two steps:

1) Compare the two sets and find out all the test cases that exists in only one of the sets.
2) For each test case we get in the first step that belonged to the group of symbolic execution with live migration, match the most similar test case in the other group according to their path constraints. Here similar means

they share the longest path during the execution. In this way we are able to locate the exact statement that lead to the difference and thus find out the virtual device state involved in the statement.

IV. Preliminary Prototype and Known Challenges

A. Preliminary Result

With the solutions for the major challenges introduced in the previous section, we implemented the prototype of VDSChecker and the live migration simulation component. We tested it with an artificial test program in which we didn't save one of the key device states that need to be saved during the migration. The result showed that we are able to detect the differences in the behavior of the program and locate the device state that lead to the differences.

There are still ongoing work related to extracting virtual devices implementations that are complicated from QEMU, and we expect to finish the tool and test it with different classes of devices on QEMU.

B. Known Challenges and Future Work

In our approach, we simplified the exploration of the execution of virtual devices by extracting the their implementations from QEMU and treating them as standalone programs. However, this kind of approximation may change the behavior of the virtual device to some extent and thus lead to inaccuracy when observing the behavior of the virtual devices because of the interactions across different components in the VM or the function calls that are outside the scope of the virtual devices implementation. To solve this, one of our ongoing effort is trying to run the whole virtual machine with symbolic execution technique while only keep tracking of only the states of the virtual device we are interested in.

Another challenge in our work, which is brought in by the symbolic execution technique, is that starting from one interface function of a virtual device implementation, there may exist a huge number of execution paths, and many of the times, the number of paths increase exponentially. This makes it hard to enumerate all the possibilities and figure out the key device states by naive exploration. We can alleviate the problem by adding a loop bound and force exiting the loops after several number of iterations, however, with this approach we will definitely miss some of the cases that requires us to dig deep into the loops.

V. Conclusion

In this paper, we proposed a study on live VM migration and the need of derive key device states from virtual device implementations. We developed an environment that mimics live migration of virtual machines in a program controlled way and proposed a solution that extracts the key states of a device by applying program analysis techniques to virtual device implementations. This is still an ongoing work, and we expect to finish the tool and test it on QEMU implementations.

Acknowledgment

This work is partially funded by National Science Foundation (NSF) uner award No. 1319115 and a gift award from Intel Corp.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[3] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[4] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[5] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review*, 43(3):14–26, 2009.

[6] Takahiro Hirofuchi, Hidemoto Nakada, Satoshi Itoh, and Satoshi Sekiguchi. Enabling instantaneous relocation of virtual machines with a lightweight vmm extension. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 73–83. IEEE, 2010.

[7] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[8] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[9] Yingwei Luo, Binbin Zhang, Xiaolin Wang, Zhenlin Wang, Yifeng Sun, and Haogang Chen. Live and incremental whole-system migration of virtual machines using block-bitmap. In *Cluster Computing, 2008 IEEE International Conference on*, pages 99–106. IEEE, 2008.

[10] F.F. Moghaddam and M. Cheriet. Decreasing live virtual machine migration down-time using a memory page selection based on memory change pdf. In *Networking, Sensing and Control (ICNSC), 2010 International Conference on*, pages 355–359, April 2010.

[11] Jon Watson. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.