Analysis of Performance Implications and Isolation Properties in Virtual Machine with Xen, KVM and LXC

Yang Liu, Ruimin Sun, Yuemin Li

Abstract

Our project will focus on the overall performance, performance isolation, and scalability of virtual machines running on three hypervisors – Xen, KVM and Linux Containers (LXC). These three examples each represents a larger class of virtualization technique namely full virtualization, paravirtualization and generic operating systems with additional isolation layers. We will focus on the CPU performance for each virtualization environment individually. With different controllers implemented, we successfully filled the gap of quantitative analysis lack in current benchmarks. In the benchmark we designed, CPU usage could be locked in a demanded level or be stressed to 100%. In order to intuitively compare the performance differences, we use Cbench to measure the latency and throughput of task Beacon – a famous Openflow controller. We will highlight the differences among these classes of virtualization systems as well as the importance of considering multiple categories of resource consumption when evaluating the performance isolation properties of a virtualization system. From this, developers could realize areas of improvement for their hypervisors or containers, and to help users make informed decisions about their choices.

Keywords: performance, isolation, Xen, KVM, LXC, Cbench, Beacon, benchmarks

I. Background

Nowadays, there are two kinds of virtualizations that are widely used. The hypervisor-based virtualization, in its most common form hosted virtualization, consists of a virtual machine monitor (VMM) on top of a host OS that provides a full abstraction of VM. In this case, each VM has its own operating system that executes completely isolated from the others. This allows, for instance, the execution of multiple different operating systems on a single host.

A lightweight alternative to the hypervisors is the container-based virtualization, also known as Operating System Level virtualization. This kind of virtualization partitions the physical machines resources, creating multiple isolated user-space instances. Figure 1 shows the difference between container-based and hypervisor-based virtualization. As is demonstrated, while hypervisor-based virtualization provides abstraction for full guest OS's (one per virtual machine), container-based virtualization works at the operation system level, providing abstractions directly for the guest processes. In practice, hypervisors work at the hardware abstraction level and containers at the system call/ABI layer.

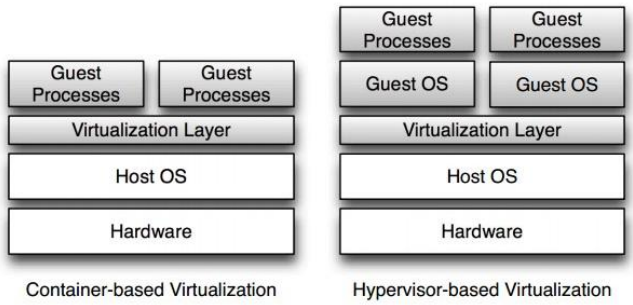

Fig. 1. Comparison of container-based and hypervisor-based virtualization.

For hypervisor-based virtualization, Xen and KVM are two typical representatives. Since its public release in 2003, Xen has been the subject of many performance comparisons. Its near-native performance and its use of para-virtualization makes it well known. KVM, however, relies on CPU support for virtualization and leverages existing Linux kernel infrastructure to provide an integrated hypervisor approach. This is opposed to Xen's stand-alone hypervisor approach. As KVM getting more and more mature, more performance testing and comparisons are being done with it.

In the following section, we will discuss how we come up with this idea of measuring CPU performance in quantitative level and what existing techniques such as stress could benefit us from the benchmark testing.

II. Motivation

With so many kinds of hypervisors and virtualization techniques, it is essential to choose the right one for our own needs. Therefore, in this paper, we will test the performance of KVM, Xen and LXC to see their strengths and weaknesses.

There are indeed lots of guys benchmarking on performance of CPU, memory, disk I/O, network I/O and so on. No one has ever try to give the influence of a misbehaving virtual machine on others among the three virtualization techniques we mentioned. Besides, no work has been done on a quantitative analysis on the misbehaving influence. For example, most of the CPU test benches make use of the intensive arithmetic operations with almost no IO operation. This approach could effectively increase the CPU usage to 100%. However, we argue that increase the CPU usage to 100% could somehow simulate a situation where highly CPU intensive programs are running. In real life, most of the workloads in most of the time are not the case. They will not stay high forever, but instead have certain patterns, for example compile kernel makes the kernel like a saw wave. So we argue that simulate such workload pattern in the neighbor virtual machine will be more preferable to evaluate the isolation of different virtualization techniques.

In the project, we purposed a new CPU test bench tool that is able to control the usage of the CPU. In addition to the intensive arithmetic operations, we add a controller, which is able to switch on or off the CPU intensive operations. For better evaluation, we made the controller pluggable so that the user is able to develop their own controller that is able to simulate their special pattern.

III. Controller Design

Intensive arithmetic operations could effectively increase the CPU usage but lack of the control mechanism makes it beyond the control. In the Operating System, the process as a task is scheduled by the kernel code but as a user level program we are not able to control the procedure of attaching or detaching our program to the CPU. However, on the other side, as a user level program we are still capable of sleeping actively and yielding the CPU resources to others by ourselves. Assume that in the OS, our CPU control program is the only dominating CPU consumption program then when the consumer goes to sleep, the CPU usage will be at quite a low rate. This is pattern is very similar to the Pulse Wide Modulation (PWM) used in voltage regulation. By switching voltage to the load with the appropriate duty cycle, the output will approximate a voltage at the desired level. Inspired by the PWM, we designed a CPU control test bench that differs from most of the CPU test bench.

We take advantage of the design in the voltage regulation and make use of the negative feedback to control the target. Figure 2 illustrates the architectural idea of our design.

i. Prototype of Controller

Four main modules contribute to the architecture are consumer source, executor, sampling and feedback controller. The consumer module intends to aggressively consume the CPU resources, like many of the stress test bench, it uses a lot of highly intensive arithmetic operations with few IO. An un-optimized prime finder is an ideal candidate for the consumer. A prime finder invokes tons of additions, subtractions, multiplications and divisions, which almost utilizes all arithmetic units of a CPU and can easily boot the CPU usage to 100%.

Executor is a module that function very similar to a switch circuit in the voltage regulator. It simply switches on and off according to the controller. The difference is that, in the executor, instead of detaching the whole process from the CPU like a kernel does, the thread actively sleep for certain time set by the controller and yield the CPU resource. In the real time test, this approach proves to be very effective, which will be discussed later in the report.

The sampling module is responsible to monitor the current CPU status. This is achieved by reading the processor file at /proc/stat. This file provides the information about the CPU status including CPU resources units consumed by user, nice, system as well as idle units since powered on. By comparing two read values, we are able to calculate the CPU usage during this period of time.

Feedback controller is one of the most important modules in the design. It controls the pattern of CPU usage. Many advanced algorithms can be used to

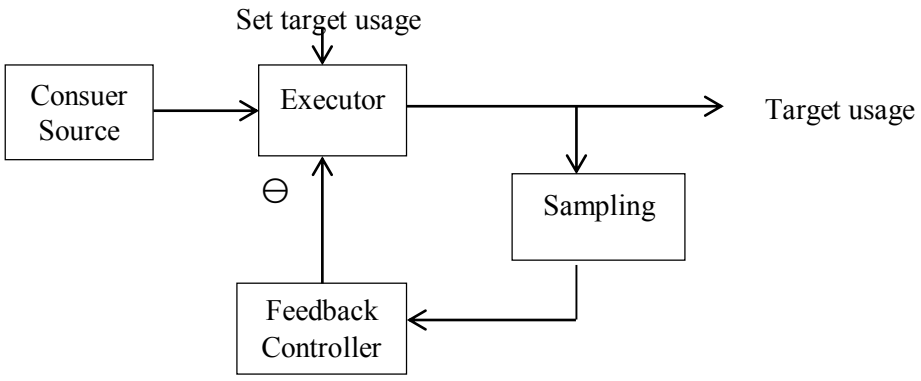

Fig. 2 The whole architecture of the CPU usage controller

develop the controller to shape different patterns and users' patterns vary one from the other. No filter controller is a panacea that can be used by everyone. In order to meet the requirement of the users', we make the controller as a plugin that can be developed by the users themselves. For our user case, which controls the CPU usage to certain level, we develop two controllers, a simple controller and Kalman Filter controller.

The simple controller is relatively intuitive. When the current CPU usage is higher than the target we set, it asks the consumer to sleep more time. On the other side, if the current usage is below the target we set, it reduces the sleep time of the consumer.

ii. Kalman Filter Controller

A Kalman Filter algorithm uses a series of measurements, with random variations and inaccuracies, observed over time, to produce estimates of unknown variables that tend to be more precise than those based on a single measurement alone.

There are two stages involved in this algorithm. During time update stage, the Kalman filter produces estimates of the current state variables and their uncertainties, which is $\hat{x}_k^-$ here. In the measurement update stage, the outcome of the next measurement (corrupted with some random noise) will be observed, and those estimates are updated given a weighted average K_k here. The higher certainty, the more weight being given to estimates. Because of the algorithm's recursive nature, it can run in real time using only the present input measurements and the previously calculated state and its uncertainty matrix. A and H are the coefficient of state model and output model respectively. Q and R are covariant of system model and measurement model respectively.

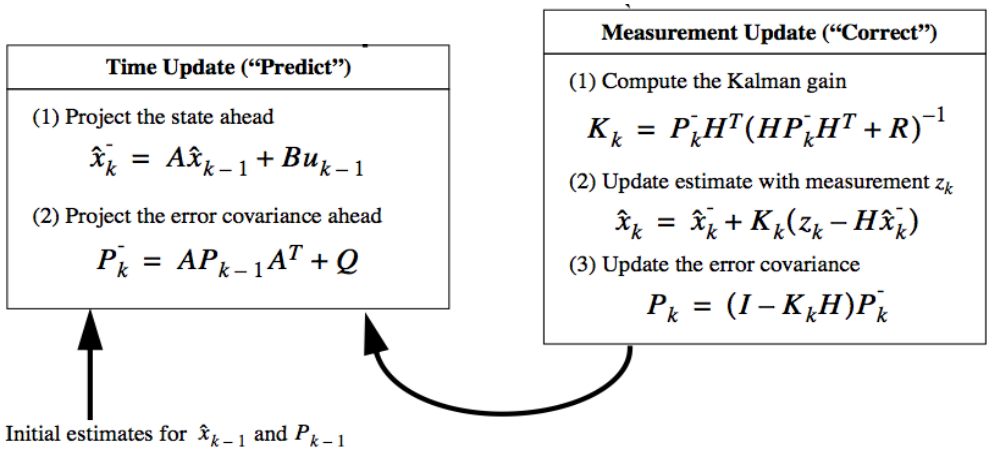

Fig. 3. The principle of Kalman Filter

In our case, the state is the current CPU usage of one guest operating system and the output we want is the sleep time to make the CPU usage stable at one level. We set up a target percentage level for CPU usage like 80% and then give the model of relationship between the sleep time S and the CPU usage U as follows. The real parameters we use will be listed in the evaluation part.

$$S = \begin{cases} 1 & if\ U \leq T \\ H \cdot (U - target) & if\ U > T \end{cases} \quad (1)$$

IV. Environment Setup

In this section, we mainly focus on the environment setup of the three kinds of virtualization techniques.

i. Xen Project and its Set-up

Xen Project is a Type 1 or "bare-metal" hypervisor, it runs directly on top of the physical machine rather than within an operating system.

Guest virtual machines are called "domains" running on Xen, and a special domain dom0 is responsible for controlling the hypervisor and starting other guest operating systems domUs. These other guest operating systems are "unprivileged" in the sense they cannot control the Xen hypervisor or start/stop other domains.

Xen supports 2 primary types of virtualization, para-virtualization and hardware virtual machine (HVM). Para-virtualization uses modified guest operating that enables them to be aware of being virtualized and as such don't require virtual "hardware" devices. They make special calls to Xen asking for allowance to access CPUs, storage and network resources. In our project, we use Xen because of its para-virtualization character to compare with full virtualization and OS level virtualization.

Installing Xen contains three parts, the Xen hypervisor installation, the network configuration and the guest operating system installation. In the following, we will first present the steps during our installing Xen hypervisor on Ubuntu 12.04.03.

1. *install a 64-bit hypervisor "xen-hypervisor-4.1-amd64"*
2. *Modify GRUB to default to booting Xen*
3. *Update GRUB and reboot*

After this, using "xm list" will show Domain-0 installed. You could also see the ID, memory size, number of virtual CPUs and current state of Domain-0. These information proves that you have installed Xen hypervisor successfully. Then we will go for the network configuration in several steps.

1. *Install bridge-utils*
2. *Use "brctl" to add bridge xenbr0*
3. *Use "brctl" to add interface between xenbr0 and eth0*
4. *Ip link set dev xenbr0 up*
5. *Set eth0 to manual and restart network manager*

With these steps, you could have your guest operating systems bridged to the network. Each of the guests could have a separate IP address just like a physical machine in the LAN. They could ping to the outside network and being connected by the outside network at the same time. Till here, we finished the preparation for installing guests. In the following steps, we will talk about configurations for the guest operating system using lvm2.

1. *Install lvm2*
2. *Create virtual disk for at least 2G size*
3. *Losetup the virtual disk to the /dev/loop1*
4. *pvcreate /dev/loop1*
5. *vgcreate myvolume-group1 /dev/loop1*
6. *lvcreate -n mylogical_volume1 myvolume-group1*
7. *get images from Ubuntu archive*
8. *Set up guest configuration in /etc/xen/Ubuntu.cfg*
9. *xm create -c /etc/xen/ubuntu.cfg*

ii. KVM and Set-up

KVM (for Kernel-based Virtual Machine) is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). KVM requires a linux kernel module to support full virtualization. The linux module consists of three files: kvm.ko, kvm_intel.ko (for Intel processors), kvm_amd.ko (for AMD processors). You can install these modules just like you install drivers for your video card. The good news is, from the 3.8.0-38-generic kernel version includes these kernel modules as part of the mainline kernel. It will depend on your distribution configuration whether these modules are actually in

the distribution kernel as a built-in or provided as modules (or possibly absent).

KVM also requires a modified QEMU although work is underway to get the required changes upstream.

iii. LXC and Set-up

We tested the isolation experiment on a lightweight container based virtualization technology LXC. The host here would be the Ubuntu 12.04.3 server, the kernel version is 3.8.0-29-generic, because of that fact that container shares the same operating system with the host, the four guests also use Ubuntu 12.04.3 operating system and has 3.8.0-29-generic kernel. By installing lxc package in the host, including its dependencies like cgroup-lite, lvm2 and debootstrap, one could start creating lxc-guest in its host server. We create 4 guset lxc containers in its host, named test, CN1, CN2, CN3. The test container is deployed as a stressed or misbehaved guest, and CN1 ~ 3 are assumed to be normal containers. The memory limitation size for each guest is 512M in our project, but one could easily adjust its memory limitation configurations by lxc-cgroup commands on the fly, or set specific memory configurations when create your container. Here we first deploy the default settings, and later change their configurations on demands. Similarly, for the CPU configuration of our guest containers, we also start with a default value, 4 processors and 1 core for each processor, but one could using cgroup command to assign specific processors for a specific container.

The testing environment is the same for those three kinds of virtualization techniques. Detailed system parameters are listed below.

Table. 1 System parameters for the experiments

Host OS	Ubuntu 12.04.3
Host Memory	4GB
Host Disk	20GB
Host VCPU	4
Host VT-x	Enabled
Guest OS	Ubuntu 12.04.3
Guest Memory	512MB
Guest Disk	3GB
Guest VCPU	4

V. Implementations

In this section, we will present the whole test implementation of our project. First, we will introduce the testing tools and benchmarks utilized in the project. Then an experiment plan containing four testing schemes will be presented.

i. Stress testing tool:

The 'stress' testing tool is used by administrators to evaluate the system performance or programmers to expose potential bugs when the system is under heavy load. In our project, we use it to stress our guest VMs' CPUs and memories. The 'stress' tool generate several works spinning on sqrt() arithmetic function to achieve a full CPU utilization, and/or multiple works spinning on malloc() and free() to put pressure on system memories. The allocation memory size could be configured by administrators. In this way, the memory stress could be assigned for each guest gradually, however, the 'stress' tool could only achieve a 100% CPU utilization for each processor, it cannot gradually stress the processor utilization on our demands.

ii. Beacon and Cbench:

Beacon is a fast, cross-platform, modular, Java-based OpenFlow controller that supports both event-based and threaded operation. It is the fastest openflow controller till now. Cbench (controller benchmarker) is a program for testing OpenFlow controllers by generating packet-in events for new flows. It emulates a bunch of switches which connect to a controller, send packet-in messages, and watch for flow-mods to get pushed down. If you think a change you are making may have a performance impact, it can be helpful to use Cbench to measure it.

Since Beacon requires considerable amount of CPU resources, and not as aggressive as prime finder that we mentioned above. Beacon also introduces some degree of IOs, which is very competent candidate as a test bench.

iii. Experiment plan

There are four tasks in our project with diverse experiment purpose and different configurations, as showed in the above table. In the first task, we expect to grasp the scheduling characteristics of

each virtualization technology by stressing each guest CPU, and then take a look at how these guest share four processors by monitoring each one's CPU usage with our CPU Monitor tool. The Second experiment would just provide a simple baseline for future experiments. Every host and guest is assigned with four processors with its host idle and one guest running a Beacon task and a Cbench task simultaneously. The latency result in Cbench here represent the guest performance when there is no misbehaving VM in the cluster. The third experiment task is what we focused on. In this part, each host and guest has four processors as before, one guest is stressed to represent the misbehaving VM in the cluster. For this guest, either one or two of its processors consume 100% CPU usage. The other three guests in this cluster are considered as normal behaved VMs, running Beacon and Cbench. To compare the isolation performance of Xen, KVM and LXC, we made a comparison between the Cbench latency result before (2nd Experiment) and after misbehaving VM appears. Finally, we implemented Beacon and Cbench in the host system, and compare the performance difference of host and guest VM.

Table. 2 Experiments Plan

	Purpose	Configuration	Host	Guests
1	Scheduling	Four Processor	CPU Monitor	One guest: Stress
2	Baseline	Four Processors	N/A	One guest: Beacon & Cbench
3	Isolation	Four Processors	CPU Monitor	One guest: stress; Other: Beacon & Cbench
4	Comparison Host & VMs	Four Processors	Beacon & Cbench	2nd experiment result

VI. Evaluations

In this section, we will demonstrate the performance evaluation of different tests we have carried on. The first part is the controllers introduced in the previous section. Then the CPU scheduling of multi-core in the host system will be analyzed with one of the guest setting to a specific CPU usage.

i. Controllers Evaluation

From Fig. 4, we can see the CPU usage measured from TOP command when running our CPU adjusting benchmark. The percentage level we set for the task is 70% and the simple controller shows minor fluctuation during the whole process. This shows that even the simplest controller we designed could provide an acceptable effect. In Fig.5, we simulate the rectangular wave with Kalman Filter controller. This controller is more suitable to low CPU usage adjusting. Here we set the CPU usage to 30% and even though a strong fluctuation is observed, the median CPU usage remains 30.084%, which is desperately close to our setting.

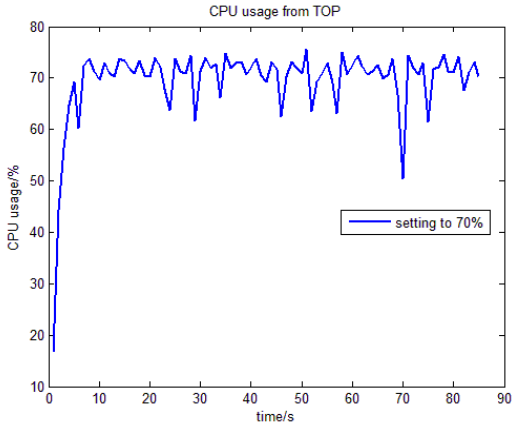

Fig. 4 CPU usage from TOP with simple controller

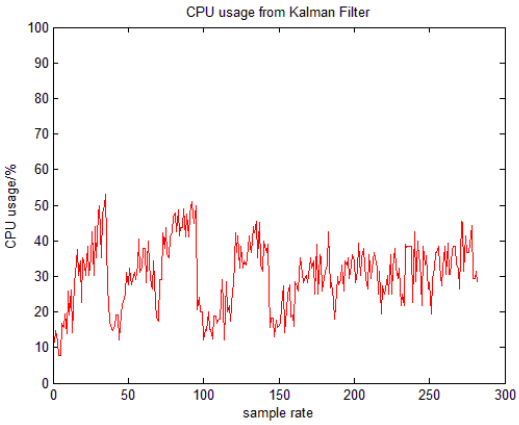

Fig. 5 CPU usage from Kalman Filter

ii. CPU Scheduling

We compare the scheduling performance between KVM and LXC using our CPU monitor in host system. One processor in one of the guest is stressed to 100% CPU usage, and we could grasp the host scheduling characteristic from the movement of the

full CPU usage plot among the 4 processors. For KVM, as is showed in fig. 6, this movement is quite frequent compared with LXC in fig.7, which means that KVM hypervisor did a much better job in the sense of scheduling. Neither of the processors would be stress for a long time. The workload is balanced among four processors. Note that the y-axis of fig () is in CPU4 is between 0 to 2, and if we broaden this rage to 100, the CPU4 is almost idle compared with others in LXC.

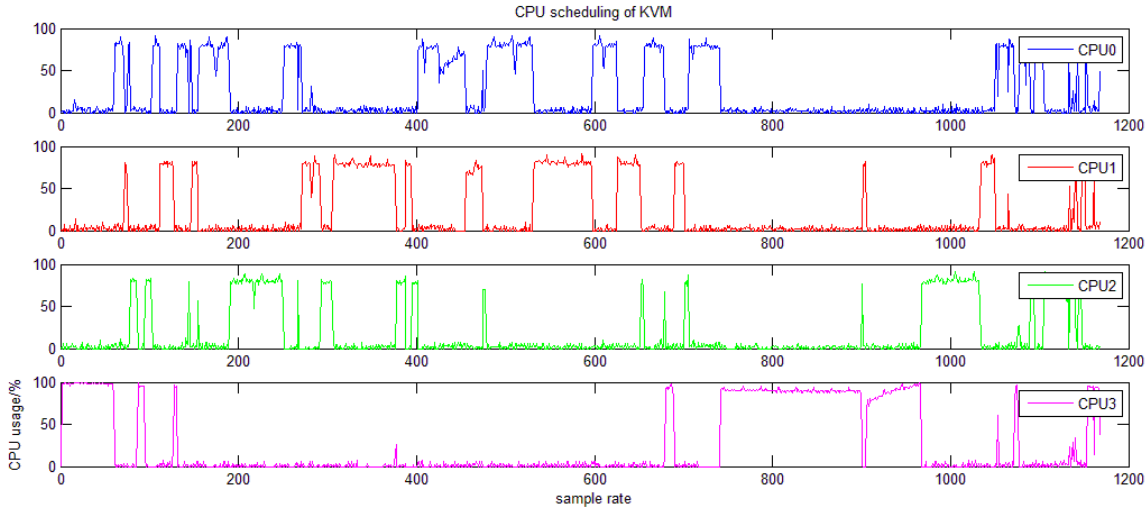

Fig. 6 The CPU scheduling in KVM host machine

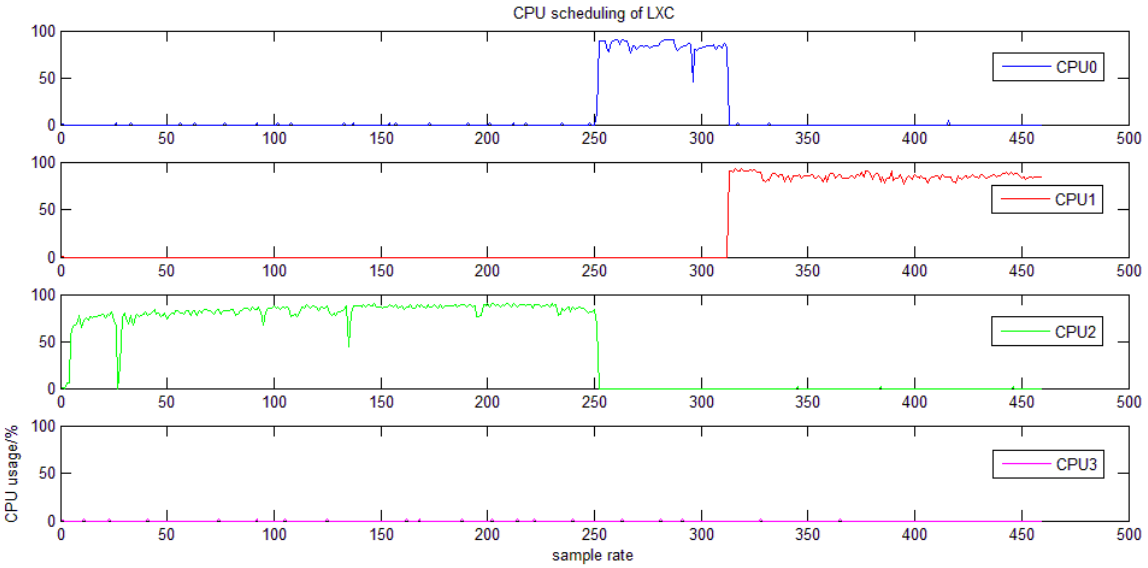

Fig. 7 The CPU scheduling in LXC host machine

iii. Bare-metal vs. virtualized

Start with the bare-metal comparison, we run some amount of tasks on the host. The performances on the three hosts are very similar. This is not surprising, as the underlying hard where of each host is identical. When virtualization layer is involved, we are able to see the difference even there are no neighbor guest noise. LXC outperforms the other two technologies when there's only one guest VM running on the virtualize layer. As LXC uses the technology to map the host processes to the guests. Our task in the LXC almost enjoys the native

access to resource with nearly zero performance penalties. The situations on the KVM and XEN are a little different. Both XEN and KVM need the involvement of the hypervisor in between of the resources and the guest VMs. Performance is jeopardized but only a very small amount.

Next we introduce more noise into the system. By adding stress behavior on the neighbor VMs, we want to see how our task performs in such a noisy environment. As the Figure. 8 show. Perform in Xen suffers a lot when one of the guest is polluted. Responses from the beacon controller dropped to half of its original performance. KVM and LXC on the other hand have better isolations. Although the pollution of other VM does have an impact on the task we are performing, the degradation is somehow negligible compare to the XEN.

We tried to add pollutions to the virtual machines one more time and stress two CPUs of the same neighbor VM. The performance of XEN deteriorates further, yielding to one third of its native performance. However, small amount of performance degradation happened to KVM and LXC.

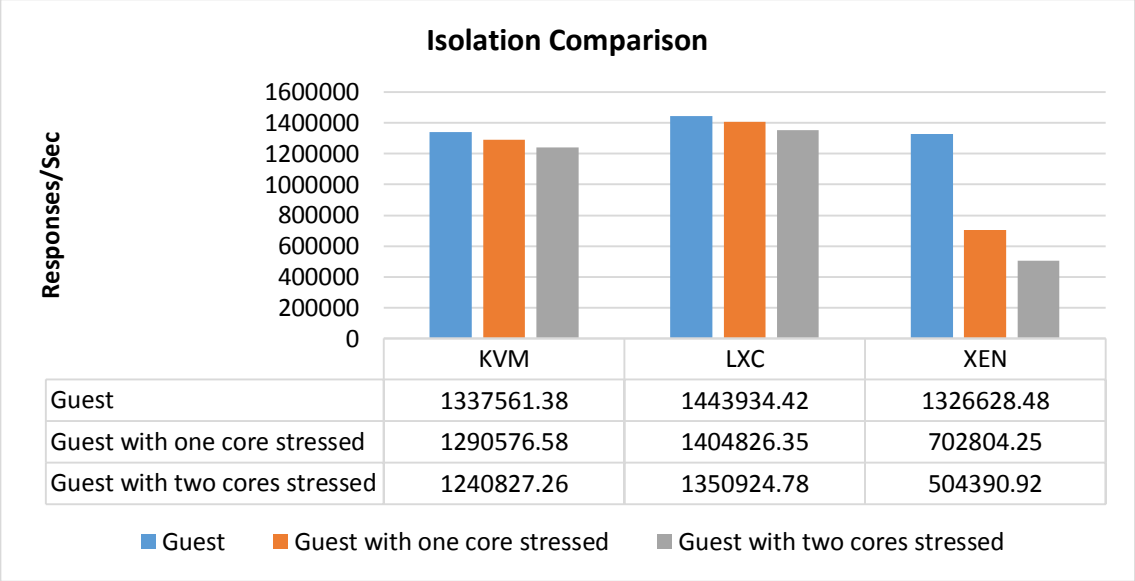

Fig. 8 Isolation Comparison of the three virtualization techniques

VII. Conclusions and Future work

In order to dip into it and see why KVM and LXC are so sustainable to the neighbor VM noise, we do another experiment on the KVM and LXC host to see the real time CPU usage when performing our tasks. The results show that the two stress cores of the neighbor are mapped to two different cores of the host physical CPU and the beacon task is mapped to another core of the host physical CPU. One of the most important similarities shared by both LXC and KVM is that they take advantage of Linux process scheduling algorithm. In KVM a QEMU process emulating the hardware is one process and in LXC every process of the guest VMs is actually a host process. Making use of the Linux processes scheduling algorithm make better utilize of all the available underlying physical resources. We believe that this is most important reason that the LXC and KVM are able to have very good isolation.

One should notice that although we successfully developed two CPU usage controllers in our project, we have not deployed them into our isolation test. This is because it is still hard for us to accurately control one guest CPU usage percentage while there are other workloads running on the other guest VMs in the whole cluster. Also, this controller is not stable enough yet. However, this control work could be quite useful because one could use this controller to simulate a specific workload like compiling kernel or intensive arithmetic calculations, and determine the isolation performance for a specific heavy workload or

misbehavior among different virtualization technologies.

VIII. Related Work

Virtualization has become a popular way to make more efficient use of server resources within both private data centers and public cloud platforms. While recent advances in CPU architectures and new virtualization techniques have reduced the performance cost of using virtualization, overheads still exist, particularly when multiple virtual machines are competing for resources.

In recent years, there have been lots of papers comparing the performance of different virtualization environments such as hardware-assisted virtualization settings in Hyper-V, KVM, vSphere and Xen [1]. Their results indicate that there is no perfect hypervisor, and that different workloads may be best suited for different hypervisors. They believe that the results of the study demonstrate the benefits of building highly heterogeneous data center and cloud environments that support a variety of virtualization and hardware platforms. While this has the potential to improve efficiency, it also will introduce number of new management challenges.

While in the area of resource isolation, though many people in both academia and industry had been working on it, it remains a hard problem for extending resource scheduling and fairness to all resources. For example, disk I/O, network for both incoming and outgoing, memory and CPU, etc. Even though several models could provide the necessary resource isolation, it may be more difficult to retrofit it into a general purpose OS that in a clearly defined virtualization layers.

Researchers have examined the performance degradation experienced when multiple VMs are running the same workload. This is an especially relevant metric when determining a systems' suitability for supporting commercial hosting environments – the target environment for some virtualization systems. In such an environment, a provider may allow multiple customers to administer virtual machines on the same physical host. Then how well do different virtualization systems protect VMs from misbehavior or resource hogging on other VMs? Issues regarding performance isolation [2, 3, 4, 14] emerged. It is natural for customers to want a certain guaranteed level of performance regardless of the actions taken by other VMs on the same physical host. [2] presented the results of running a variety of different misbehaving applications under three different virtualization environments VMware, Xen, and Solaris containers. VMware protects the well-behaved virtual machines under all stress tests, but sometimes shows a greater performance degradation for the misbehaving VM. Xen protects the well-behaved virtual machines for all stress tests except the disk I/O intensive one. For Solaris containers, the well-behaved VMs suffer the same fate as the misbehaving one for all tests. [4] talked about the LXC based performance isolation in HPC with Linux VServer, OpenVZ and Linux Containers (LXC). Their result showed that all container-based systems are not mature enough and the only resource that could be successfully isolated was CPU, in which LXC demonstrated to be the most suitable container-based system for HPC. In [14], they test KVM and LXC, and resulted in that KVM in combination with hardware support can provide better trade-offs between performance and isolation, but KVM has a slightly lower throughput.

For now, there has been no one comparing Xen, KVM and LXC together with all the six stressed tests in CPU intensive test, a memory intensive test, a disk intensive test, two network intensive tests (send and receive) and a fork bomb. Our project will focus on this and give a more detailed problem statement and potential approach in the following section.

The choice of benchmark is very important, we searched number of benchmarks and summarized each as followed.

VMmark [5] is a free tool that hardware vendors, virtualization software vendors and other organizations use to measure the performance and scalability of applications running in virtualized environments. VMmark uses workloads representative of those applications most often found in the data center, such as email servers, databases, etc.

SPECweb2009 [6] is the next-generation SPEC benchmark for evaluating web server performance. Its workloads include: Banking, which is a fully secure SSL-based workload; Ecommerce, which includes

both SSL and non-SSL requests; and Support, which is a non-SSL workload that includes large downloads. New with SPECweb2009 is the inclusion of a Power workload (based on the Ecommerce workload) and a performance/power metric.

Isolation Benchmark Suite (IBS) [7] is designed to quantify the degree to which a virtualization system limits the impact of a misbehaving virtual machine on other well-behaving virtual machines running on the same physical machine. This benchmark suite includes six different stress tests - a CPU intensive test, a memory intensive test, a fork bomb, a disk intensive test, and two network intensive tests (send and receive). The result can highlight the difference between different classes of virtualization systems as well as the importance of considering multiple categories of resource consumption.

Lmbench [8] is Suite of simple, portable benchmarks. It compares different UNIX systems performance in bandwidth, latency and miscellaneous.

As is stated before, there is no perfect hypervisor or container architecture and implementation, and that different workloads may be best suited for different hypervisors. In order to provide a detailed comparison on different resources control among different platform, we present this project on measuring the overall performance within native platform individually and performance isolation among different implementations of virtualization architectures.

Division of work:

Ruimin Sun will focus on the performance isolation measurement among all those three kinds of virtualization architectures. Yuemin Li is responsible of native environment overall performance testing. Yang Liu will summarize the performance and try to figure out some patching work for the hypervisors. All of us would work together to figure out the problems emerged during the experiment.

References

[1] Hwang, Jinho, Sai Zeng, and Timothy Wood. "A component-based performance comparison of four hypervisors." Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on. IEEE, 2013.

[2] Matthews, Jeanna Neefe, et al. "Quantifying the performance isolation properties of virtualization systems." Proceedings of the 2007 workshop on Experimental computer science. ACM, 2007.

[3] Deshane, Todd, et al. "Performance isolation of a misbehaving virtual machine with Xen, VMware and Solaris containers." submitted to USENIX (2006).

[4] Xavier, Miguel G., et al. "Performance evaluation of container-based virtualization for high performance computing environments." Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on. IEEE, 2013.

[5] VMmark: http://www.vmware.com/products/vmmark/

[6] SPECweb2009: https://www.spec.org/web2009/

[7] IBS: http://web2.clarkson.edu/class/cs644/isolation/

[8] Lmbench: http://www.bitmover.com/lmbench/

[9] Xen: https://help.ubuntu.com/community/Xen

[10] Xu, Yang, et al. "Performance evaluation of para-virtualization on modern mobile phone platform." Proceedings of the International Conference on Computer, Electrical, and Systems Science, and Engineering. 2010.

[11] Setting up LXC Containers Manually: https://www.suse.com/documentation/sles11/singlehtml/lxc_quickstart/lxc_quickstart.html#sec.lxc.setup.container

[12] LXC-HOW TO: http://lxc.teegra.net/#_this_howto

[13] Resource Control: http://docs.oracle.com/cd/E19044-01/sol.containers/817-1592/rmctrls-1/index.html

[14] Rathore M S, Hidell M, Sjödin P. KVM vs. LXC: comparing performance and isolation of hardware-assisted virtual routers[J]. American Journal of Networks and Communications, 2013, 2(4): 88-96

[15] Cbench: http://www.openflowhub.org/display/floodlightcontroller/Cbench

[16] Beacon: https://openflow.stanford.edu/display/Beacon/Releases

[17]Testing Beacon Controller with Cbench: https://openflow.stanford.edu/forums/topic/118-testing-beacon-controller-with-cbench/

Appendix

i. Simple CPU controller usage

This is the introduction to the simple CPU controller tool we developed in our project. It could be used either for single processor usage control or multi-processor monitor. Here are flags that can be used for different purpose.

-c --percentage is to set a specific percentage for your single processor usage. For instance, if you run ./cpu –c 70 in the directory of this controller, then the single processor usage in this system would be confined with 70%.

-m is to monitor the current processor usage regardless of the number of processor you have. If you run ./cpu –m in a four processor VM with one processor stressed like in our project, it would detect one CPU to be 100% used, and the others almost zero.

By default, the benchmark we designed use the simple controller, if you would like to use the Kalman Filter controller, please comment the sentences

"Simple_controller *ctrl = new Simple_controller();

Snake snake(ctrl); "

And comment out the sentences

" // KFilter *filter = new KFilter();

// Snake snake(filter); "

in main.cpp and make clean and remake the programs.