Volume 4, No. 4, April 2013
Journal of Global Research in Computer Science
ISSN-2229-371X
REVIEW ARTICLE
Available Online at www.jgrcs.info

PERFORMANCE OPTIMIZATION UNDER A VIRTUALIZED ENVIRONMENT

Argha Roy[*1]
*[1] M.Tech in CSE, Netaji Subhash Engineering College, Kolkata, West Bengal, India
arghacse@gmail.com[1]

Abstract: Virtualization technology has found a renewed interest owing to the need for cost-efficient operations, better manageability and increased availability of systems. The increased use of physical resources introduces new bottlenecks in performance. This paper proposes methods trying to eliminate these bottlenecks and achieve close to native performance for various guest OS. Here we try to address the following two performance aspects –1. Additional I/O bottlenecks introduced due to virtualization 2. LIVE MIGRARTION support for each virtual machine to ensure maximum uptime

Keywords: virtualization, hypervisor, device emulation, IOVM, live migration

INTRODUCTION

In computer terminologies, virtualization refers to a process of decoupling the software from the physical hardware so that the same piece of hardware can be shared by multiple operating systems in a secured and managed manner. Here multiple operating machines run in what is known as *virtual machines* (VM), each with its own specifications. A *virtual machine monitor* (VMM) is used to manage and securely run these virtual machines. Virtualization can thus allow us to allocate portions of a hardware resource as per requirements to each virtual machine. Current generation processors have evolved to a position where they have much more processing power than what a single OS can utilize at a given instant. Studies have revealed that on an average, *less than 15 %* [1] of the actual processing power gets utilized under a single OS per machine scenario. Currently companies support their ever increasing requirement for business services by buying multiple physical boxes. Compounding this cost is the added overhead of an inflexible computing infrastructure, if the demand changes, the system requirements also changes.

Enter *virtualization*. In its latest form, virtualization makes use a *hypervisor* to achieve decoupling and a lot of research has been going on this front lately [2]. In the discussion that follows, we shall be using the terms VMM and hypervisor interchangeably. Thus now the same physical box can be utilized for providing multiple services which would have previously required two or more physical machines. Additional benefits include *scalability*, *reduced power consumptions*, *lower hardware support costs*, and the same level of *isolation* as different physical boxes and easier management through consolidation.

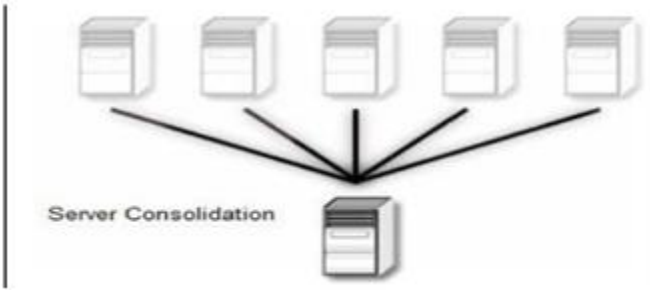

Figure: 1

There are various approaches for virtualization, each with its own advantages and disadvantages. However, running multiple operating systems in virtual machines introduces new challenges which affect performance under a virtualized environment. There is a notable degradation in I/O performance due to multiple context switches [3]. Further running multiple virtual machines on the same physical box makes all of them dependent on the same physical box for dependability. A secure and feasible method needs to be present to do appropriate hardware address re-mapping and enable migration from one box to another in case of hardware failures, without adversely affecting performance during the hardware downtime.

CURRENT SITUATION

The *x86*architecture is the architecture of choice for most servers and desktops, given its popularity and wide range of applications built around it. With respect to virtualization, we focus on the protection levels offered in various OSes and of course, on the instruction set itself. There are a bunch of privileged (or kernel mode) instructions and many unprivileged (or user mode) instructions in the x86 instruction set. On the OS side, we have a *hierarchical protection* security model being implemented in the form of RINGs. A program running in RING 0 has access to the entire instruction set (including the privileged one). User mode applications run in the less privileged RING 3.

However, most virtualization software's (except Para virtualization) along with its virtual machines run in user mode at RING 3. Running software (here guest OS) at a privilege level other than what it was originally designed for is called ring aliasing [4]. Now we have a problem (called ring compression [4]) when a guest OS issues a privileged instruction at RING 3. Thus the x86 instruction set is not a fully virtualizable one as it violates the EQUIVALENCE requirement [5] of POPEK and GOLDBERG virtualization requirements. A variety of methods are adopted for vitalizing the entire instruction set of x86, which ranges from binary patching to modifying the guest OS itself. In binary patching, the VMM "traps" the non- virtualizable instructions [6] from the guest OS and does dynamic runtime patching so that the guest OS is abstracted from the

fact that it's running under a virtualized environment. This method consumes a significant amount of processing power, given the multiple number of context switches that takes place inside the processor. In the second approach, called Para-virtualization, the guest OS itself is modified to make it virtualization "aware" and let it know that it's sharing the hardware platform with others. Latest hardware support in the form of *Intel VT* [4] or *AMD Pacifica* [7] technology addresses the problem of executing privileged instructions at RING 3. These techniques provide a privileged RING 1 (called VMX root mode in Intel [4]) for guest OS and hypervisor to run in. However, these technologies provide no support for I/O of the VMs.

The largest number of context switches takes place during I/O instructions as these are the most common type of instructions issued by the guest OSes. A single context switch in a Pentium 4 (P4 2.8 GHz) costs 995 ns (= 2598 CPU cycles) + software context switch delays![8] So we shall be focusing on I/O instructions more intensely and propose methods for reducing the number of context switches that occur during issuing of these instructions and thus improve on the amount of time processor spends processing instructions rather than sitting idle doing switching. Before we proceed further, we need to understand the different types of virtualization. Here we shall only give a top level idea of the various approaches and propose ideas to increase overall performance. The position and mode of operation of the hypervisor depends on type of implementation adopted. Each of these methods has their own way of vitalizing the platform hardware so as to enable hardware sharing among guests.

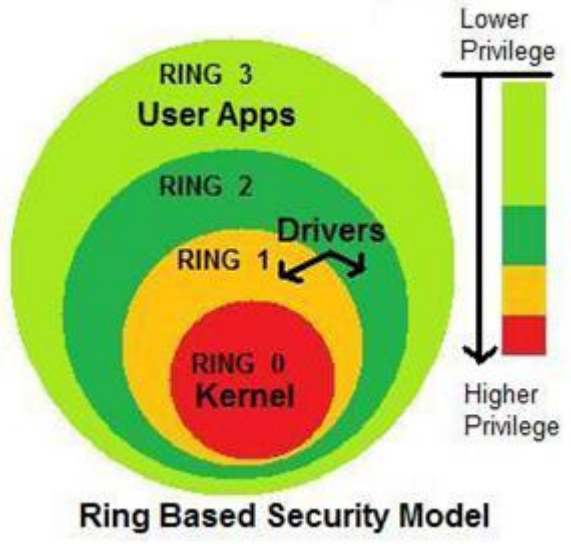

Figure: 2

PARA-VIRTUALIZATION

In this approach, the guest OS is modified, preferably at the kernel level, to directly interact with the VMM for various operations, including I/O and interrupt handling. Here the mode of operation of the guest OS is itself modified so as to bring about tighter coupling between the guest and the hypervisor. This makes Linux and other open source OS ideal candidates for Para virtualization.

OS-HOSTED VIRTUALIZATION

In this mode of virtualization, the hypervisor runs on top of an operating system which acts as the host, for other guest operating systems. The biggest advantage of this type of virtualization is that the hypervisor can leverage the existing drivers of the host OS for its operations.

VIRTUALIZING I/O HARDWARE

Every operating system needs some dedicated hardware to run upon. However in virtualization, we only have a fixed number of instances of a given hardware and many guest OSes run on top on this fixed hardware. Thus we need to ensure that no two guest's issues conflicting commands and also that results of an operation are properly routed back to the guest from whom a given command was issued, so as to enable sharing of hardware. Thus all hardware, starting from the processor to the physical hard disk must be virtualized for virtualization to function properly. Varieties of methods are used for the same and are discussed below.

COMPLETE EMULATION

Emulation refers to implementation of the complete hardware in term of software. Emulation offers us the advantage of complete portability of the guest- the guest is totally unaware of the underlying physical hardware. The guest only knows of the emulated device being presented to it. Also, since emulation presents an OS with an exact interface of some existing hardware device, the guest OS is not affected if the underlying hardware changes altogether. However, emulation as a method of I/O virtualization suffers from poor performance because of the tremendous overheads of emulating a complete hardware device in software.

PARA-VIRTUALIZATION APPROACH

As already discussed, in Para-virtualization, the guest OS is made "virtualization aware" by modifying the operating system to be virtualized. Although this method promises lesser code traversal path from guest to actual hardware, the primary limitation is the non-availability of Para-virtualized drivers for an acceptable range of hardware platforms. Further, complete para-virtualization optimization cannot be carried out on proprietary OS like Windows.

DIRECT ASSIGNMENT

In this approach, a given hardware is assigned directly to a VM. The hardware is owned and controlled by the VM and not by the hypervisor or the service OS. *The biggest advantage of this approach is that native performance can be achieved using the VM's native driver for the given hardware.* This also reduces the size of the VMM which no longer has to include the device driver within it. However, in direct assignment approach, the VMM can only assign as many devices that are present in the platform physically. Also, in the absence of proper hypervisor support for direct assignment, this method too fails to reach its optimal performance. Thus direct assignment offers us relatively better performance, however as pointed out, we cannot directly use this method for virtualization.

OUR OBJECTIVES

From the above discussion we conclude that the while Para virtualization may be a good solution for vitalizing applications which needs higher throughput, its application is limited due to the limited number of Para virtualized drivers available in the market. Even writing new drivers compatible with newer firmware of the same hardware is a challenging task, leave alone optimizing it. Additionally, there is the problem of executing a very large amount of codes in RING 0 .On the other side, in OS hosted hypervisor, we note that there is a substantial performance penalty while carrying out full device emulation, to achieve portability across a wide spectrum of hardware devices. In the following section, we propose a hypervisor design, keeping the following requirements in our mind –

a) The system should be scalable, performance oriented and fault tolerant.
b) Improve security by running less code in privileged RING 0 i.e. to maintain a lower trusted computing base (TCB).
c) Felicitate migration of guest OS running on a given physical box to another in case of hardware failures on a given box.

We now move on to propose a hypervisor design which tries to comply with our above stated requirements.

THE HYPERVISOR DESIGN

A major emerging trend among hypervisor designers is to decompose the hypervisor. The current concept of a centralized driver domain within the hypervisor is in question here. The primary problem with this design is that any optimization of the centralized driver domain for a particular device may not satisfy the conflicting needs of other devices being maintained by the hypervisor, given the different usage patterns of different hardware components. Thus if we move the required operation onto a separate domain and optimize that domain, better performance can be achieved. Following this trend, we move from a *monolithic general purpose* hypervisor to a thin privileged "*micro-hypervisor*" to be run in RING 0 on top of the platform hardware *along with the host OS* and some of the other subsystems and services of the VMM being run in a separate VM that are *de- privileged at RING 3*. In the process we get to ensure *lower TCB* at RING 0. Now these de-privileged components of the VMM which are to be run in a virtual machine specially optimized for its job, becomes our center of attention.

We can now have a small, lightweight VM specifically designed and optimized for a specific job of the hypervisor. Since system memory is nowadays relatively plentiful, running these small VMs will not be as taxing, given the performance benefits they offer. We note here that the guest OSes and these small VMs are now to be treated together as one entity, though the two execute as separate processes. This is so as the guest needs these small VMs to get vital functions done through them, functions which are no longer been provided directly by the hypervisor. Of the various functions that are to be "out-sourced" from the hypervisor to these lightweight VMs, our center of attention turns to I/O as a broad function which includes both the disk as well as network I/O. We focus primarily on I/O as its one of the biggest bottlenecks affecting the performance of a virtualized guest OS.

IOVMs

An IOVM is a highly flexible, lightweight guest OS dedicated to and optimized for the virtualization of a certain device over which I/O operations can take place. Through IOVMs, we try to move I/O virtualization work out of the hypervisor or the service OS into a dedicated driver domain.

IOVM FUNCTIONING

The splitting up of the drivers into a frontend and backend might seem to increase the path an instruction has to travel from a guest OS to a physical I/O device, but this splitting up enables us to employ already researched and practically implemented stack optimization techniques to deliver better performance than what we would have achieved without the IOVM. Thus we may also label IOVMs as software based solution to the problem of direct assignment of hardware.

HOW IT WORKS

We shall be using existing technology of live migration here, and show how IOVMs are well adaptable in these situations as well. We shall see how the design of having an IOVM frontend and backend facilitates LIVE MIGRATION. In the event of a hardware device failure, first the agent notifies the hypervisor about the same and stacks up instructions temporarily. The agent next updates the multiplexer with consultation of the hypervisor and reloads appropriate driver modules of another I/O device on some other box in the network. At the same time a temporary IOVM in the target machine is prepared by linking its interface with the network. On the faulty machine, the multiplexer is updated to redirect all traffic over the network. Thus now we have a backend driver of some other device although the I/O takes over the network temporarily. Once this stable condition is established, network bandwidth reservation is requested; following which I/O operations are temporarily suspended again and all requests are queued up in the stack and copying of the entire IOVM and its associated VMs to the target box's memory (RAM) starts. The preloading of drivers of target machine ensures immediate resumption of work as soon as VMs with its associated IOVMs are transferred to the target box. Once copying is over, the target box completely takes over execution of the VMs. We note that this method requires close communication of the hypervisors on both the source and target machines, details of which will differ according to the mode of implementation adopted. Thus we see that IOVMs also offers us a scalable and reliable support for live migration of virtual machines. Thus the multiplexing split driver design of IOVMs helps facilitate live migration, while abstracting guest OSes from hardware failures, making them fault tolerant.

CONCLUSION

Virtualization technology is all set to revolutionize the way we deploy and maintain servers, offering unmatched scalability and savings. This paper proposes techniques for

using IOVMs for scalable and high performance I/O operations under a virtualized environment thereby allowing guest OSes to deliver their close to native performance. The paper also shows how IOVMs, by virtue of their structure, assists in live migration, thereby ensuring maximum uptime for a virtual machine.

REFERENCES

[1] Inside Xen 3.0 – A XenSource Whitepaper PDF, p2 http://xen.xensource.com/files/xensource_wp2.pdf

[2] The IBM research hypervisor project – for generic ideas and goal determinations - http://www.research.ibm.com/hypervisor/

[3] Intel Technology Journal – virtualization technology for directed I/O: http://www.intel.com/technology/itj/2006/v10i3/2-io/1-abstract.htm

[4] Intel VT – Hardware support for efficient processor utilization: http://www.intel.com/technology/itj/2006/v10i3/1-hardware/1-abstract.htm

[5] "Formal requirements for virtualizable third generation architectures" by GJ Popek, RP Goldberg – Communications of the ACM (July 1974)

[6] VMware and CPU Virtualization Technology: http://download3.vmware.com/vmworld/2005/pac346.pdf

[7] AMD Pacifica overview: http://www.theinquirer.net/en/inquirer/news/2005/06/06/amds-pacifica-revealed-in-full

[8] Performance reports at OsDev.org - http://www.osdev.org/osfaq2/index.php/Context%20Switching

[9] Xen/IOMMUs – Breaking new grounds: http://www.xensource.com/files/xs0106_xen_iommu.pdf

VMotion Features, VMWare:

[10] Performance reports at OsDev.org - http://www.osdev.org/osfaq2/index.php/Context%20Switching

[11] Xen/IOMMUs – Breaking new grounds: http://www.xensource.com/files/xs0106_xen_iommu.pdf

[12] VMotion Features, VMWare: http://www.vmware.com/products/vi/vc/vmotion.html

Short Bio Data for the Author

Argha Roy received his B.Tech (Computer Science & Engineering) degree from West Bengal University of Technology, Kolkata in 2011 and pursuing his M.Tech degree from same University. He has presented more than 6 papers in National and International Conferences also he has many publications in refereed journals.His areas of interests are Cloud Computing , Networking and Automata Theory .