Implementation of a Purely Hardware-assisted VMM for x86 Architecture

Saidalavi Kalady *, Dileep P G, Krishanu Sikdar, Sreejith B S, Vinaya Surya, Ezudheen P †

Abstract—**Virtualization is a technique for efficient utilization of hardware resources. Virtual Machine Monitor (VMM) is a piece of software which facilitates hardware Virtualization. Software based Virtualization technologies encompass considerable instruction overhead. We have implemented a purely hardware-assisted VMM for x86 architecture on AMD Secure Virtual Machine (SVM) [1] to overcome the instruction overhead caused by software based Virtualization. The performance impact due to Virtualization is measured in terms of the CPU time consumed by certain critical sections of Virtualization specific code. A quantitative performance analysis using the purely Hardware-assisted Virtual Machine Monitor reveals that Hardware Virtualization at its current stage suffers from performance overhead, but improves considerably with hardware enhancement.**

Keywords: Hardware Virtualization, Virtual Machine Monitor, Performance Analysis, x86 architecture, AMD Secure Virtual Machine

1 Introduction

Virtual machine monitor (VMM) was first introduced as a software abstraction layer that could multiplex costly hardware units for multiple applications in the 1960s. Introduction of multitasking operating systems and inexpensive hardwares obscured Virtual machine monitor technology between 1980 and 1990 [7]. Beginning of twenty first century witnessed re-introduction of VMM as a solution for couple of problems like Hardware underutilization, information security and Power Consumption.

Virtual Machine Monitor (VMM) also known as Hypervisor, is a piece of software which is used to implement hardware Virtualization. VMM controls the concurrent execution of multiple operating systems on a single physical machine. It is a thin software layer that provides Virtual Machine (VM) abstractions. The abstraction resembles the real hardware to an extent that is sufficient to enable software written for the real machine to run without change in the VM [7]. The VMM is referred to as the Host and each Operating System (OS) that runs atop the VMM as Guest. VMM provides appearance of full control over a complete physical machine (processor, memory, and all peripheral devices) to every Guest OS.

Until the recent wave of processor Virtualization assistance from hardware manufacturers, x86 - the world's most popular architecture was hostile towards VMM technology. Techniques such as Full Virtualization and Paravirtualization resorted to complicated software workarounds to virtualize x86 machines, but not without the associated overhead [4]. Software based Virtualization technologies have considerable instruction overhead to virtualizes x86 machines. In 2005, hardware manufacturers introduced architectural extensions to support x86 Virtualization. Hardware-assisted Virtualization, though not fully mature at present, represents the future of Virtualization [6].

Section 2 describes the limitations of x86 architecture for Virtualization and architectural extensions to x86 which facilitates classical Virtualization on AMD Secure Virtual Machine (SVM) [1]. Section 3 describes design and implementation of a simple prototype of a purely Hardware-assisted Virtual Machine Monitor (HVMM). The performance impact due to Virtualization is measured in terms of the CPU time consumed by certain critical sections of Virtualization specific code. Section 4 describes a quantitative study on the performance impact associated with Hardware Virtualization.

2 VIRTUALIZATION SUPPORT FOR x86 ARCHITECTURE

Processors based on x86 architecture failed to meet classical Virtualization standards until major hardware vendors came up with the recent architectural extensions in support of Virtualization [5]. The respective technologies from Intel and AMD in this domain are Intel VT codenamed Vanderpool and AMD Secure Virtual Machine (SVM) codenamed Pacifica.

2.1 x86 Architectural Limitations

In its native form, the x86 architecture doesn't meet the Popek & Goldberg's formal requirements for Virtualiza-

*Saidalavi Kalady (e-mail: said@nitc.ac.in) is a faculty member in the Department of Computer Science and Engineering, National Institute of Technology Calicut, India.

†Dileep P G (e-mail: dileeppg.nitc@gmail.com), Krishanu Sikdar (e-mail: Krishanu.6@gmail.com), Sreejith B S (e-mail: sreejithbs.nitc@gmail.com), Vinaya Surya(e-mail: vinaya-surya@gmail.com), Ezudheen P (e-mail: ezudheen@gmail.com) are students in the Department of Computer Science and Engineering, National Institute of Technology Calicut, India.

tion [5]. In a virtualized environment, those instructions that require control to be handed over to the VMM are called *sensitive instructions*. Privileged instructions within a VM can be trapped to handover control to the VMM. If all the sensitive instructions are privileged, then the processor architecture is said to be virtualizable. There are 17 instructions in x86 which violate this basic requirement [8].

Moreover, the x86 architecture has privilege levels called Rings. The OS normally runs at ring level 0. So, to control a Guest OS within a VM, the VMM must run at a higher privilege level than the OS. But the highest privilege available is at level 0. Even in that case the Guest OS must execute at a level numerically greater than or equal to 1. But most proprietary Operating Systems are designed to run at level 0 or they will fail to operate. One solution to this problem is modifying the kernel of the Guest OS. But for some commercial Operating Systems it may violate their Licensing terms and conditions. In such cases implementing a virtualized environment requires expensive changes in Operating Systems and it might also cause software incompatibility issues [8].

Different software techniques have been in use for x86 Virtualization. These include Full Virtualization which completely emulates the underlying hardware and Paravirtualization which necessitates modifications to the OS kernel. Both these techniques requires expensive software techniques to overcome the inherent limitations of the underlying hardware.

The recent x86 hardware Virtualization extensions from Intel and AMD are the answer to many of the aforesaid problems. The two technologies are similar but incompatible in the sense that a VMM designed to work on one cannot automatically run on the other. We have chosen the AMD SVM for our study and analysis.

2.2 AMD SVM Architectural Extensions

The AMD SVM architecture is a set of hardware extensions to the x86 architecture specifically designed to enable effective implementation of Virtual Machine systems [1].

The AMD SVM is designed to provide quick mechanisms for *World Switch* between Guest Operating Systems and the Host VMM. *World Switch* refers to the operation of switching between the Host and the Guest [2].

Important features of the SVM include the ability to intercept selected instructions or events in the Guest, external access memory protection, assistance for interrupt handling, virtual interrupt support, a Guest/Host tagged Translation Look-aside Buffer (TLB), and Nested Paging to reduce Virtualization overhead [2].

AMD SVM introduces several new instructions and modifies several existing ones to facilitate simpler and yet robust implementations of Virtual Machine systems on the x86 architecture or more specifically the AMD64 architecture. The newly introduced instructions are VMRUN, VMLOAD, VMSAVE, VMMCALL, STGI, CLGI, SKINIT, and INVLPGA [1].

Another feature provided by the architecture is a new processor mode called Guest Mode entered through the VMRUN instruction. While executing in Guest Mode, there are subtle changes introduced in the behavior of some x86 instructions in order to facilitate Virtualization [1].

There is also a new memory resident data structure called Virtual Machine Control Block (VMCB) for each running Guest OS. The VMCB is divided into the Control area and the State area. Control area contains various control bits including the intercept vector with settings that determine what actions cause #VMEXIT (transfer of control from the Guest to Host). The CPU state for each Guest is saved in the state area. Also, Information about the intercepted event is put into the VMCB on #VMEXIT [2].

The SVM also includes Nested Paging facility to allow two levels of address translation in hardware, thus eliminating the need for the VMM to maintain the so called shadow page table structures that are involved in software Virtualization techniques [1].

With nested paging enabled, the processor applies two levels of address translation. A guest Page Table (gPT) maps Guest virtual address to Guest physical addresses located in Guest physical space. Each Guest also has a host Page Table (hPT) which maps Host virtual addresses to Host physical addresses located in Host physical space. Both Host and Guest levels have their own copy of the CR3 register, referred to as hCR3 and gCR3, respectively. The complete translation from Guest virtual to Host physical address is cached in the TLB and used on subsequent Guest accesses [1].

In addition, the TLB entries are tagged with an Address Space Identifier (ASID) distinguishing Host-space entries and different Guest-space entries from each other. The ASID tag specifies to which Virtual Machine, the corresponding memory page is assigned. This eliminates the need for TLB flushes each time a different Virtual Machine is scheduled [1].

3 HARDWARE-ASSISTED VIRTUAL MACHINE MONITOR

The HVMM which we use for performance analysis completely relies on the AMD SVM Hardware Virtualization support. It is based on an open source prototype VMM called Tiny Virtual Machine Monitor (TVMM) by Dr. Kenji Kaneda [10].

The TVMM is a simple VMM developed for the purpose of education and verification and has the following functionalities [10]. It performs the basic tasks necessary to initialize the AMD SVM processor extensions. It successfully creates a single VM and then boots a skeletal Guest OS within the VM. Our HVMM is an extended version of the TVMM with the following additional capabilities.

- It can handle multiple Guest Operating Systems.
- It creates multiple Virtual Machines each of which runs different Guest Operating Systems.
- It has the functionality to schedule the guests one after the other in a Round Robin fashion.

3.1 HVMM Design

The pseudo code contained in Table 1 and 2 describes the control flow behind the working of the HVMM.

3.2 HVMM Implementation

The HVMM is designed to run on AMD64 machines with the 64-bit Long mode and SVM extensions enabled. The bulk of the HVMM code is written in 'C' language. The startup code needs to be in assembly and is done using the GNU Assembler [3]. HVMM development environment is listed in Table 3. The object files from AS and GCC are linked together to form a single x86-64 Executable and Linkable Format (ELF) binary file 'hvmm' by the linker 'ld' using a separate custom linker script.

A disk image is created using the dd utility to emulate a real physical disk. The virtual disk created is mounted on a 64 bit Linux machine installed on AMD SVM supported hardware. The binary files of one or more guest kernels, HVMM along with GRUB boot files, and settings are copied to the disk. Then the disk is loaded in AMD SimNow and the kernel is run from within the HVMM.

We use simple skeletal 64-bit kernels in ELF format for our purpose [11]. The OS kernels complete the minimal initialization tasks and then simply loop infinitely printing dummy messages on to the screen. We can load any number of OS kernels provided as separate GRUB modules along with the HVMM kernel. The OS kernels are copied to disjoint locations in memory from within our HVMM. Then Virtual Machines are created for each OS which involves populating SVM specific data structures such as the VMCB to control the VM execution [12]. The VMM finally boots the kernels within the corresponding Virtual Machines in a Round Robin fashion with a specific time slice. Once an OS gains control, it continues to execute until certain specific operations, such as the completion of a time slice causes a #VMEXIT, forcing transfer of control back to the VMM. The OS can also transfer the control explicitly to the VMM using a special VMMCALL instruction.

4 PERFORMANCE ANALYSIS

In this section, we present a quantitative study of performance overheads associated with Hardware-assisted Vir-

Table 1: HVMM Pseudo Code

```
HVMM {
        INITIALIZE_SVM();
        HYPERVISOR_CORE();
}

INITIALIZE_SVM() {
        ENABLE_SVM();
        SETUP_HYPERVISOR();
}

ENABLE_SVM(){
        //Initialize SVM specific flags such as
EFER, SVME with appropriate value;
}

SETUP_HYPERVISOR(){
        //Allocate and setup basic VMM data
structures such as Host State Save Area
}

HYPERVISOR_CORE(){
        for(each Guest OS gos) {
                vm=VM_CREATE(gos);
                ADD_ACTIVE(vm);
        }
        while(1) {
                vm=GET_NEXT_VM();
                if(no active vm) break;
                VM_BOOT(vm);
                HANDLE_VMEXIT(VM);
        }
}

VM_CREATE(gos) {
        vm- >VMCB=SETUP_VMCB();
        LOAD_GUEST_IMAGE(gos);
        }

SETUP_VMCB(){
        ALLOCATE_VMCB();
        SETUP_CONTROL_AREA();
        SETUP_STATE_AREA();
        }

ALLOCATE_VMCB(){
        //Allocate a region of physically contigu-
ous, page-aligned memory
}

SETUP_CONTROL_AREA(){
        //Initialize the control area of the VMCB
with conditions that will control the guest OS ex-
ecution
}
```

Table 2: HVMM Pseudo Code (cont'd.)

```
SETUP_STATE_AREA(){
        //Initialize the state area of the VMCB
with the initial machine sate of the guest OS
}

LOAD_GUEST_IMAGE(gos){
        //Load the image of the guest OS gos into
memory
}

ADD_ACTIVE(vm){
        //Adds the Virtual Machine vm to the list
of VMs considered for Round Robin execution
}

GET_NEXT_VM(){
        //Return one from the active list of VMs
following a simple Round Robin rule
}

VM_BOOT(vm){
        //Transfer control to the Guest OS within
the vm using special SVM Instructions
}

HANDLE_VMEXIT(vm){
        //A naive #VMEXIT handle which just
displays the information associated with the
#VMEXIT
}
```

Table 3: HVMM Development Environment

CPU	AMD x2 3800+
Base OS	Open SUSE 10.3
Simulator	AMD SimNow V 4.4.2
Compiler	GCC 4.3
Assembler	GNU AS
Linker	GNU ld
Image Tool	dd
Bootloader	GRUB
Other	GNU Make utility

Table 4: HVMM Test Environment

CPU	AMD x2 Turion, AMD x2 3800+, 4400+
Base OS	Open SUSE 10.3 64 bit
Simulator	AMD SimNow V 4.4.2
Host	HVMM
Guests	64 bit skeletal OS kernel

Table 5: Comparisons of Switch Log values

Processor	Clock Speed	Switch lag
AMD x2 Turion	1.6 GHz	89 ms
AMD x2 3800+	2.0 GHz	79 ms
AMD x2 4400+	2.2 GHz	75 ms

tualization using the AMD SVM. The overhead is measured in terms of CPU time consumed by certain critical sections of HVMM code.

4.1 Experimental Setup

The HVMM is made to boot and run two Guest Operating Systems one after the other. It is possible to setup break points at critical points in the OS code with the help of the SimNow debugger [9]. We setup break points at the following two points (1) Immediately before the #VMEXIT in the Guest OS 1, and (2) Just before start of Guest OS 2.

The time at the two instances are noted as the value given by the Host seconds field in the SimNow main window. The difference between the two values gives the time taken to execute minimal VMM specific tasks before the next OS gets control. We call this value as "Switch Lag". Here, we neglect the time taken for specific #VMEXIT handling code and just consider the minimal time taken for a blind transfer of control back to the next Guest OS.

The test is performed on processors with three different clock speeds - AMD x2 Turion, AMD x2 3800+, and AMD x2 4400+ all of which have AMD SVM extensions and a comparison is made between the values of Switch Lag obtained. Table 4 shows the Test Environment

4.2 Results

Table 5 shows the switch lags corresponding to different processors having various clock speeds while running HVMM with identical virtual machines atop real machines.

Figure 1 demonstrates the variation of Switch Lag corresponding to different clock speeds while running HVMM with identical virtual machines atop real machines.

From Figure 1, It can be inferred that Switch Lag decreases considerably with ascending clock speeds. But the switch lag does not linearly decrease with increase in clock speed. This implies that increase in clock cycle and hardware enhancements will not reduce switch lag to an acceptable level (order of micro seconds). The VMM scheduler which enables the HVMM to execute multiple guest operating systems in a round robin fashion is obliged to set scheduling time quantum in the order of seconds. It results in lower response time for Guest operating systems running upon single threaded machines

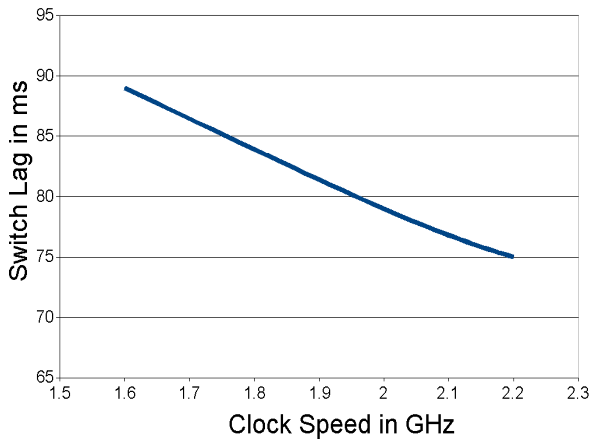

Figure 1: CPU Speed vs Switch Lag

5 Conclusions and Future Work

Comparison of running times with the HVMM running identical Virtual Machines atop real machines having differing processor speeds reveals that the performance overhead diminishes with increasing processing power. With evolving hardware enhancement techniques, the performance impact due to Virtualization is expected to come down. Hardware Virtualization is still at a nascent stage. The results obtained from the test with different processors show that there is still a long way to go in reducing the performance overhead associated with Hardware-assisted Virtualization.

A multithreaded implementation of VMM scheduler with hardware assisted thread scheduling will bring down the switch lag into order of microseconds. It will necessitate additional inter process communication between Host OS and Guest OS for exchanging Virtualization events among them. This will results in higher response time for Guest operating systems and more efficient CPU utilization.

References

[1] AMD, *AMD64 Virtualization Codenamed Pacifica Technology Secure Virtual Machine Architecture Reference Manual*, Publication No.33047 Revision 3.01, May 2005

[2] AMD, *AMD64 Architecture Programmer's Manual*,Volume 2:System Programming, Publication No.24593 Revision 3.13, July 2007.

[3] AMD, *AMD64 Architecture Programmer's Manual Volume 3:General-Purpose and System Instructions*,Publication No.24594 Revision 3.13, July 2007.

[4] Intel, "Intel Virtualization Technology," *Intel Technology Journal*, Volume 10, Issue 3, August 2006.

[5] POPEK, G. J., GOLDBERG, R. P., "Formal requirements for virtualizable third generation architectures," *ACM Communications,*, July 1974

[6] Adams, Keith; Agesen; Ole, "A Comparison of Software and Hardware Techniques for x86 Virtualization," *ACM International Conference on Architectural Support for Programming Languages and Operating Systems,*, 2006.

[7] Rosenblum, Mendel; Garfinkel, Tal. "Virtual machine monitors: current technology and future trends," *IEEE Computer*, volume 38, issue 5, May, 2005.

[8] J. Robin, C. Irvine, "Analysis of the Intel Pentiums Ability to Support a Secure Virtual Machine Monitor," *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[9] AMD, *AMD SimNow v4.4.2 Simulator Users Manual.*.

[10] http://web.yl.is.s.u-tokyo.ac.jp/ kaneda/tvmm/, Tiny Virtual Machine Monitor.

[11] http://osdever.net/ , An introduction to OS development.

[12] http://www.osdev.org/, Advanced OS development.