Faithful Virtualization on a Real-Time Operating System

Henning Schild **Adam Lackorzynski** **Alexander Warg**
Technische Universität Dresden
Department of Computer Science
Operating Systems Group
01062 Dresden
{hschild, adam, warg}@os.inf.tu-dresden.de

August 14, 2009

Abstract

The combination of a real-time executive and off-the-shelf time-sharing operating systems has the potential of providing both predictability and the comfort of a large application base. Isolation between the components is required to protect the real-time subsystem from a significant class of faults in the (ever-growing) time-sharing operating systems but also to protect real-time applications from each other. Recent commodity computer hardware significantly improved the ability of these machines to support faithful virtualization. Virtual machines provide the strong isolation required for security reasons. But questions regarding the temporal isolation remain open. In this paper we analyze how and to which degree recent x86 virtualization extensions influence the interrupt-response times of a real-time operating system hosting virtual machines.

1 Introduction

Reusing legacy time-sharing operating systems together with a real-time kernel is a common approach to get a large application base and predictability. Isolation in these systems ranges from shared-space systems, where real-time applications and the time-sharing operating system kernel reside in the most privileged mode, to separate-space systems, where only the real-time kernel runs in this mode, whereas all applications and the time-sharing operating system are deprivileged [10].

Separate-space systems have in common that paravirtualization is used to host the time-sharing operating system. Paravirtualization means that the time-sharing operating system is modified in a way that it can be executed next to the real-time kernel. The applicability of approaches based on this technique is limited by the need of source code access and by the considerable development and maintenance effort that the modifications require.

Recent commodity computer hardware has a significantly improved ability to support efficient virtualization because virtualization support was added to the architecture[5][9]. With faithful virtualization unmodified guest operating systems can be executed in virtual machines. Using such virtual machines to host time-sharing operating systems next to the real-time executive does not only decrease the development effort, it also allows for the use of closed source operating systems, opening access to an even wider range of applications. The isolation provided by this virtual machine technology is comparable to the strong isolation provided by separate-space systems.

The usage of hardware extensions for virtualization has implications on the responsiveness of the host operating system. In this paper we analyze the degree to which the interrupt-response times of a real-time operating system can be influenced by using these extensions.

We will proceed as follows: Section 2 will revisit the fundamentals, before we go into details of hardware-assisted virtualization in section 3. Our experiments are described in section 4.

2 Background

In this section we briefly introduce virtualization techniques, the x86 hardware extensions for virtualization, and L4/Fiasco, the microkernel we used for our experiments.

2.1 Virtualization Techniques

Virtualization is a term that is used in many different aspects of computer science and especially in the area of operating systems. In this paper we concentrate on virtual machines that enable running complete operating systems with all their applications on top of the host operating system [12]. We focus on CPU and memory virtualization for the x86 architecture. There are three fundamental techniques to implement virtual machines: emulation, paravirtualization, and faithful virtualization.

2.1.1 Emulation

Emulation is a technique that translates guest code into host code. This can be accomplished by translating single guest instructions one after another as the guest execution proceeds. To improve the performance of emulators, techniques like binary translation and caching can be applied.

One advantage of emulation is that it can be applied when the guest and the host instruction set architectures (ISAs) are not the same. However, for identical ISAs it is desirable to execute the guest code directly instead of using emulation. Compared to virtualization techniques where guest code is executed directly, the performance of emulation is poor. Optimized emulators like QEMU [7] run a guest system multiple times slower than executing the same code on bare hardware.

Still, emulation is a common technique in cases where the guest can or must not be given direct access to the hardware. Emulation is often used to provide peripheral devices for virtual machines.

2.1.2 Paravirtualization

Paravirtualization is a technique where the guest operating system is modified in a way that it may run on top of another operating system. Privileged instructions and code sequences are exchanged with calls to the underlying operating system environment.

After this modification, the guest code is executed directly on the host CPU, the performance achieved by paravirtualization is close to executing the guest on bare hardware [6] [8].

Paravirtualization requires access to the source code of the guest operating system. Therefore it is applicable only to guest operating systems where the source code is accessible. Another disadvantage of paravirtualization is a considerable development and maintenance effort.

2.1.3 Faithful Virtualization

Faithful virtualization is to construct a virtual machine that allows most of the guest code to execute directly on the CPU. When guest code cannot be executed, emulation is used for these specific parts. The basic technique for faithful virtualization is called *trap and emulate*. The guest code is executed in a processor mode where privileged instructions must not be executed. Instead of executing these instructions, the processor signals a fault to the host operating system. These faults are called traps. Instructions that cause a trap are then executed using emulation. After a short emulation phase direct execution resumes in the deprivileged processor mode.

On x86 processors *trap and emulate* cannot be applied easily [13]. Some instructions do not cause a trap when they are executed in a deprivileged mode, although they would need to in order to preserve the illusion of a real CPU. However, recent x86 CPUs have virtualization extensions to overcome this problem. Using faithful virtualization, a single implementation of a virtual machine monitor (VMM) is suitable for a series of unmodified guest operating systems. Faithful virtualization yields good performance because most of the guest's code is executed directly.

2.2 Hardware-Assisted Faithful Virtualization

Hardware extensions found in recent processors make it easier to implement faithful virtualization for the x86 architecture. The second generation of these hardware extensions did not only help to reduce the software complexity required for virtual machines, it also helped to increase virtual machine performance significantly by providing support for memory virtualization through nested page tables. Hardware-assisted virtualization has a series of advantages making it a promising technology. It is widely available in commodity hardware and yields good performance.

This section describes how hardware-assisted virtualization works. The provided details are required for the comprehension of the rest of this paper.

Two virtualization implementations are available, Intel VT[9] and AMD's SVM [5]. Because we conducted our experiments using SVM, we use AMD's notions throughout this document. However, the principles discussed in this section apply for Intel VT as well.

2.2.1 Processor Modes

The hardware extensions for virtualization allow executing privileged guest code directly on the CPU preserving the virtualization requirements. Two different modes of operation are supported by the processor.

The *host mode* is used for the host operating system. The *guest mode* is used to execute guest operating systems and their applications. Each mode supports all x86 privilege levels. Privileged instructions that access the CPU state in guest mode read or write the guest CPU state. Therefore guest operating system's code that expects to run on the highest privilege level may be executed at the level it expects. The isolation from the host is enforced by the hardware.

2.2.2 Control Flow

Before entering the guest mode, the host operating system needs to configure it. Therefore a control data structure for a virtual machine needs to be set up. This structure, the *VMCB*, contains the guest mode's CPU state and control information.

When guest code should be executed, the host operating system issues the `VMRUN` instruction to enter the guest mode. This instruction saves the host CPU state, loads the guest state provided by the *VMCB*, and begins execution in the guest mode. Interrupts triggered by host devices, faults caused by the guest code, or an attempt to execute certain instructions may lead to an automatic switch back to the host mode (`#VMEXIT`). Flags in the *VMCB* control which events or instructions exit the guest mode.

When switching back to the host mode, the guest CPU state is stored in the *VMCB* and the host state is automatically restored. After `#VMEXIT` has finished, the execution proceeds in the host mode until another switch to the guest mode is issued by the host.

2.3 The L4/Fiasco Microkernel

The L4/Fiasco microkernel is a small operating system kernel developed by our research group. In contrast to monolithic designs, microkernels aim at a minimal kernel. Functionality is only admitted into the kernel if it cannot be implemented at user-level without compromising security, or if a user-level implementation severely impacts performance. Components like device drivers, network protocol stacks, and filesystems are not part of the kernel, they are implemented on top of it. The reduced complexity at the kernel-level makes microkernel-based systems suitable for setups where strong security properties [14] and real-time capabilities [10] are required.

The L4/Fiasco microkernel provides a few basic mechanisms that can be used to construct complex systems. Virtualization allows us to reuse legacy software in our operating system. With L^{4}Linux [8] we are able execute Linux and unmodified Linux applications. The recently added support for hardware-assisted virtualization [11] enables us to reuse arbitrary unmodified x86 operating systems in a secure way, while reaching close to native performance and keeping the engineering effort low.

3 Hardware-Assisted Virtualization vs. Responsiveness

In this paper we are interested in the effects of hardware-assisted virtualization on the interrupt latency and therefore the real-time capabilities of the host operating system.

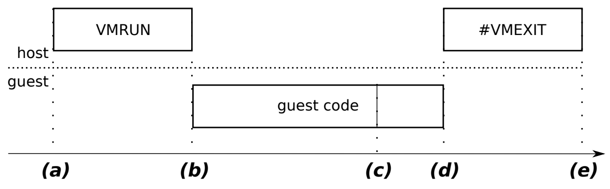

FIGURE 1: *Switching between processor modes*

The control flow described earlier is shown in figure 1. It highlights five points in time that are of particular interest.

(a) Execution of the `VMRUN` instruction begins

(b) CPU starts executing guest code

(c) Execution of final instruction before `#VMEXIT` begins

(d) Switching back to the host mode begins

(e) Host operating system resumes

Switching between the guest and the host mode needs certain amounts of time during which the processor cannot accept interrupts. In order for the host to regain control, `VMRUN` always needs to be followed by `#VMEXIT`. We identified two possible sequences that are candidates for having the most significant influence on the host's interrupt-response times.

An interrupt triggers right after point *(a)*. In this case no guest code is executed and `VMRUN` is directly followed by `#VMEXIT`. The duration of the context switch is at most $(b - a) + (e - d)$.

The interrupt triggers in the guest mode right after *(c)*. The processor finishes the execution of the currently running instruction until *(d)* and issues a `#VMEXIT`. The maximum duration of the context switch is $(e - c)$ for this sequence.

Which of the two sequences takes more time than the other depends on the duration of its components. The duration of `VMRUN` and `#VMEXIT` is one of these components. We need to answer the question how long they occupy the CPU and whether these times are constant or depend on the guest state. Another important component is the maximum duration of instructions that may be executed in the guest mode, leaving the processor in an uninterruptible state.

4 Experimental Results

In this section we qualify the observations of the previous section with experimental results. For the experiments we used an AMD Phenom^{TM} 9550 Quad–Core CPU on an ASUS M3A78–EM motherboard equipped with 2GB DDR2–800 RAM and a Samsung HD080HJ hard disk.

To analyze how using hardware-assisted virtualization increases the host's interrupt-response times we begin with a series of synthetic benchmarks, followed by application benchmarks that substantiate our results in more realistic scenarios. The application benchmarks where conducted using Linux with the `PREEMPT_RT` patch and L^{4}Linux running on the L4/Fiasco microkernel.

4.1 Measuring Interrupt Latency in L4/Fiasco

Most of the tests were executed on the L4/Fiasco microkernel. We developed a tool to measure the interrupt latency for an application on top of it.

This application sets up a periodic timer via the *High Precision Event Timer* (HPET) and attaches itself to the corresponding interrupt. The application then waits for the kernel to forward interrupts to it. When an interrupt triggers the microkernel schedules the application and delivers the interrupt. L4/Fiasco uses a scheduler based on fixed priorities. To make sure the test application is scheduled right after the interrupt arrives, the test program is configured to have the highest priority in the system.

This tool serves as our real-time application that handles external events, which are signaled via interrupts. The routine that handles the interrupts inside the application is used to measure the time it took to switch to this function. That is done by calculating the difference between the time the HPET was programmed to trigger an interrupt and the time this interrupt arrived in the user-level handler.

4.2 Analyzing Specific Mode Switching Scenarios

To answer the question of how long `VMRUN` and `#VMEXIT` take and whether the times depend on the guest state, we developed a simple virtual machine monitor. We used this application to set up various scenarios to measure their effect on the host's interrupt-response times.

With the help of this tool we found the duration of `VMRUN` and `#VMEXIT` to be dependant on various factors. Table 1 shows the maximum measured interrupt latencies for different test scenarios running on top of the L4/Fiasco microkernel. Some of them use virtualization with specific mode switching, others do not use hardware-assisted virtualization.

	setup	**virtualization**	**time [μs]**
1	busy loop	–	7.05
2	vm simple	X	8.38
3	vm expensive	X	9.57
4	vm triplefault	X	12.78
5	cache	–	10.41
6	cache vm	X	12.5

TABLE 1: *Interrupt latencies for experimental setups on the L4/Fiasco microkernel.*

The baseline for the latencies is defined by the first setup ("busy loop"). In this setup an endless loop is executed next to the latency measurement. It does not make use of L4/Fiasco's virtualization capabilities. Running the simple VMM in an endless loop where it keeps entering and leaving the guest mode ("vm simple"), the latency increases by 1.33 μs.

The setup "vm expensive" shows the results of the VMM that keeps entering and leaving the guest mode with a couple of conditions that increase the mode switch duration. In this scenario the VMM enters the guest mode injecting an exception and flushing the tagged TLBs. The guest's stack pointer is prepared in a way that the CPU needs to access two separate memory pages to store the exception return information. The *interrupt description table* (IDT) is set up, so that the injected exception causes another exception. Additionally it is aligned in a way that the CPU needs to access two distinct memory pages to read the entries required for the execution of the test. All these conditions increase the mode switch duration. Overall, this scenario has an interrupt latency that is 2.5 μs above "busy loop".

We stopped trying to increase the interrupt latency for the "vm expensive" scenario further after we came up with the "vm triplefault" setup. In the "vm triplefault" scenario the VMM keeps entering the guest mode with a condition that causes a triplefault in the guest. The VMM injects an exception that causes another exception, which causes a *shutdown* event [5] in the guest mode. In this setup no guest code is ever executed. After the *#VMEXIT* we never observed another exit reason than the *shutdown* event. Therefore we think this *VMRUN/#VMEXIT* combination is not interruptible. The interrupt latency for this setup is 12.78 μs, which is 5.73 μs above the baseline.

Like ordinary applications virtual machines influence the timing behaviour of the overall system through using the processor's caches. To show that, we developed another synthetic benchmark. Setup number 5 is an application that stresses the CPU's caches by writing to 32 MB contiguous memory. In the sixth setup ("cache vm") we executed the same code in the guest mode using our simple VMM. It can be seen that both setups increase the interrupt latency by using the caches. The increase that is caused by executing the code in a virtual machine is 2.1 μs, which is between the differences measured for the cheapest and the most expensive scenario.

From the results we conclude that the duration of the mode switches clearly depends on the guest state that is switched to. We measured increases from 1.33 up to 5.73 μs in host interrupt latency when SVM was used. Our results also confirm that virtual machines influence the timing through caches.

4.3 Long running Instructions

As described in section 3, we also need to analyze how long instructions in the guest mode may take to execute.

To find the durations of single instructions we used a manual provided by AMD [4]. It contains detailed information on the execution times of the individual instructions. Notably, there are instructions where the manual does not provide figures because their execution times are not fixed. All these instructions may be executed in the CPU's highest privilege level only. Therefore they do not pose a problem for real-time operating systems that execute applications in user-level. Developers can avoid using these long running instruction in the operating system kernel.

When hardware-assisted virtualization is used, such instructions can be executed in the highest privilege level of the guest mode. Because we want to be able to run legacy operating systems in the guest mode, we either need a mechanism to enforce that these instructions are not executed, or we will have to account for their duration in our worst case latency.

`WBINVD` is an example for one of these long running instructions. It writes data that was modified in the CPU's caches back to main memory. Its execution time depends on the amount of modified cache entries. For another experiment we used the virtual machine that writes to the contiguous memory region, which was used for our earlier cache experiments. We modified it to execute the `WBINVD` instruction after having written the 32 MB of memory.

The highest host interrupt latency we measured when executing this code in the guest mode was 643.66 μs. However, the *VMCB* can be used to control, which instructions must not be executed using instruction intercept flags. Instead of executing intercepted instructions, the processor issues a `#VMEXIT`. Intercepting the real-time critical instruction resulted in a worst case interrupt latency of 12.15 μs. This latency is close to the one measured for the "cache vm" setup that was modified.

The results of this experiment show that all real-time critical instructions should be intercepted if possible. SVM allows to intercept all long running instructions we found using the AMD manual. This means, the host can prevent the guest from executing them.

4.4 Interrupt Latency for L^{4}Linux and KVM-L4

After running the synthetic benchmarks we analyzed the effects of using faithful virtualization in a more complex setup. In these experiments we did not specifically try to trigger special cases for virtualization. Instead we used application benchmarks to put the system under load. We compared two setups:

The first one was L4/Fiasco with L^{4}Linux running on it. L^{4}Linux was executing a Linux kernel build and the `lmbench3` benchmark suite.

In the second setup we added virtual machines using *Kernel-based Virtual Machine* [3] for L^{4}Linux (KVM-L4). On the host, we ran the same benchmarks as in the first setup. As guests we used two virtual machines running Linux. One of them was running a Linux kernel build, the second virtual machine was executing a busy loop.

For the first scenario, where hardware-assisted virtualization was not involved, we measured a maximum latency of 29.05 μs. When we used hardware-assisted virtualization with KVM-L4 the maximum latency was 30.38 μs. Figure 2 shows the distribution of interrupt-response times for the two setups.

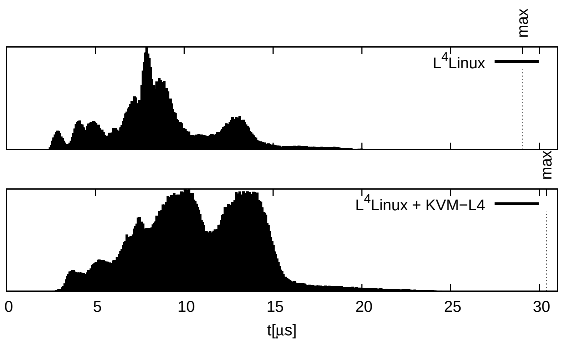

FIGURE 2: *Interrupt latency histograms for L^{4}Linux without and with KVM-L4*

The results of these benchmarks show that even in a complex scenario, the influence of using virtualization is within the bounds we found with the synthetic benchmarks. The difference of the individual maxima is 1.33 μs, which does not exceed the 5.73 μs increase we measured for the "vm triplefault" setup. The two maxima only show virtualization effects on the critical paths. They cannot be used to conclude that switching modes never took more than 1.33 μs or 5.73 μs in the setup using KVM-L4. But the histograms give confidence in the figures from the synthetic benchmarks. The shapes of the two graphs are similar. The one that displays the KVM-L4 setup is shifted to the right by 1.5 – 2 μs and contains more samples in the rightmost spike.

4.5 Linux with *PREEMPT_RT* patch

So far we only presented results from experiments using L4/Fiasco. But the figures should be applicable to other implementations as well. The mode switch durations are mainly implied by hardware, instead of software. To show that the measured latency penalty applies for other implementations as well, we conducted some experiments with Linux. For our measurements we used Linux 2.6.29.6 and applied the *PREEMPT_RT* patch set version rt23 [2]. To measure the latency we used the `cyclictest` application [1] developed by Thomas Gleixner.

Again, we set up two scenarios with the same load as in the L^{4}Linux setups. For the first scenario, which did not involve KVM, we measured a maximum latency of 52 μs. When we used hardware-assisted virtualization with KVM the maximum latency was 55 μs. The results of these experiments are given in figure 3. Again the graphs have a similar shape. Using hardware-assisted virtualization through KVM increases the latency by 3 μs. This result meets the expectations derived from the earlier experiments. The cost of using the hardware extension for virtualization applies for Linux/KVM as well. It is within the bounds we expected it to be.

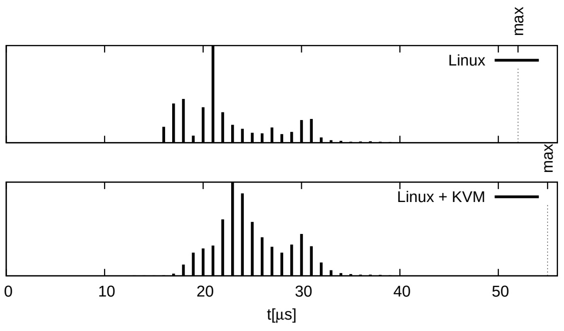

FIGURE 3: *Interrupt latency histograms for Linux with* PREEMPT_RT *patch, without and with KVM*

5 Conclusion

Hardware assisted faithful virtualization is an easy and efficient way of running legacy third party applications on top of a real-time operating system. We explained how it works and how virtual machines might influence the responsiveness of the host operating system. In our experiments, using the hardware extension SVM increased the interrupt latency of the real-time operating system by at most 5.73 μs on a 2.2 GHz machine. We also showed that virtual machines may cause a significantly bigger increase when they are not configured with attention to real-time. The host operating system needs to make sure that long running instructions cannot be executed in guest mode.

6 Acknowledgement

We would like to thank Björn Döbel, Marcus Völp, and Jean Wolter for their constructive criticism on the way to this version of the paper. Authors of this paper have been supported by the European Research Programme FP7.

References

[1] Cyclictest . `http://rt.wiki.kernel.org/index.php/Cyclictest`, 2009-06-17.

[2] CONFIG PREEMPT RT Patch . `http://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch`, 2009-07-29.

[3] Kernel Based Virtual Machine . `http://www.linux-kvm.org/`, 2009-08-11.

[4] Advanced Micro Devices. *AMD Athlon^{TM} Processor, x86 Code Optimization Guide*, publication no. 22007, rev. k edition, 2002.

[5] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, rev 3.14 edition, 2007.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003.

[7] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[8] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of μ-kernel-based systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 66–77, New York, NY, USA, 1997. ACM.

[9] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 3B: System Programming Guide, Part 2*, 253669-028us edition, 2008.

[10] F. Mehnert, M. Hohmuth, and H. Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 124–133, Austin, Texas, USA, Dec. 2002.

[11] M. Peter, H. Schild, A. Lackorzynski, and A. Warg. Virtual Machines Jailed: Virtualization in Systems with Small Trusted Computing Bases. In *VDTS '09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, pages 18–23, Nuremberg, Germany, 2009. ACM.

[12] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[13] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association.

[14] M. Roitzsch and H. Härtig. Ten Years of Research on L4-Based Real-Time Systems. In *Proceedings of the Eigth Real-Time Linux Workshop*, Lanzhou, China, 2006.