CSE 506 Operating Systems A Lightweight Hypervisor and its Latency Measurement

Abhishek Revadekar, Harshit Barot, Kushaal Rana
Stony Brook University, Stony Brook, NY, USA

Abstract

This report summarizes the development of a custom lightweight hypervisor using C and shell scripting, aiming to improve virtualization overhead performance on Linux platforms. We also tested the latency for running the guest code on the custom hypervisor.

The hypervisor was built for the Intel VT-x technology. Furthermore, we checked the hypervisor performance for multiple guest tasks running on different levels of virtualization.

ACM Reference Format:
Abhishek Revadekar, Harshit Barot, Kushaal Rana. 2023. CSE 506 Operating Systems A Lightweight Hypervisor and its Latency Measurement. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Hypervisors play a critical role in virtualization technology by enabling multiple operating systems to run concurrently on a single physical host. The hypervisor acts as an intermediary between the physical hardware and the virtual machines, allowing multiple operating systems to share a single set of physical resources such as CPU, memory, and storage. The hypervisor provides the necessary virtualization layer to abstract the physical hardware and present it to each virtual machine as a virtualized set of resources.

There are two types of hypervisors: Type 1 and Type 2. Type 1 hypervisors, also known as bare-metal hypervisors, run directly on the host machine's hardware and provide the highest level of performance and security. Type 2 hypervisors run on top of an existing operating system and are typically used for desktop virtualization and testing environments.

In this project, we developed a lightweight, Type 2 hypervisor using the C language and our focus was to find potential regions for hypervisor efficiency, aiming to develop a custom lightweight hypervisor. We also test the performance drops in the nested virtualization abilities.

The cornerstone of this research was to measure and analyze VM execution latencies in diverse guest tasks. By leveraging this data, we can precisely identify the performance bottlenecks and proposed promising optimization techniques.

2 Overview of Intel Virtualization Technology (Intel VT-x)

VMX is a technology that allows a computer's processor to create and manage multiple virtual machines. It does this by using a set of special instructions that are built into the processor. These CPU instructions provide a way for the hypervisor (which manages the virtual machines) to control and monitor the virtual machines.

VMX main function is that it helps to create and manage virtual machines. These instructions allow the hypervisor, which is responsible for managing virtual machines, to allocate memory, storage, and CPU resources for the virtual machine. Additionally, VMX instructions help control how the virtual machine interacts with the physical hardware of the system.

Another role of VMX is to maintain the security of the virtualizaed environment and their isolation. This is done by providing hardware support for virtualization. VMX enables the hypervisor to monitor and restrict access to physical resources such as memory and I/O devices, preventing virtual machines from accessing resources that they shouldn't.

Nested virtualization is another feature provided by VMX, which allows virtual machines to host other virtual machines within them. This capability enables the creation of highly complex virtualized environments, such as cloud computing platforms, where multiple layers of virtualization are utilized to manage and control resources.

The VMX operations are classified into two types: VMX root operation and VMX non-root operation. The VMM typically runs in VMX root operation while the guest software runs in VMX non-root operation. The transitions between these two operations are known as VMX transitions. There are two types of VMX transitions, the first type being the transitions into VMX non-root operation called VM entries,

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

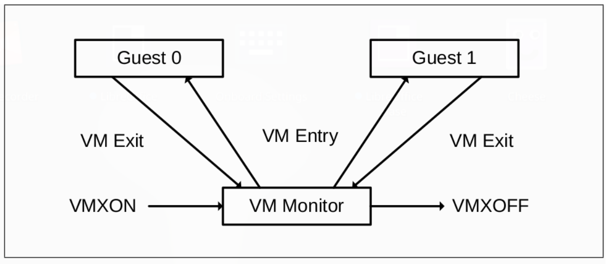

Figure 23-1. Interaction of a Virtual-Machine Monitor and Guests

Figure 1. VMX activity life cycle as given in the Intel's developer manual

and the second type being the transitions from VMX non-root operation to VMX root operation called VM exits.

When the processor is in VMX root operation, its behavior is similar to that outside of VMX operation. However, there are some key differences, such as the availability of a new set of instructions called VMX instructions, and limitations on the values that can be loaded into specific control registers.

When a processor operates in VMX non-root mode, its behavior is modified and limited to enable virtualization. Specific instructions and events trigger VM exits to the VMM instead of their usual behavior. This limits the functionality of software in VMX non-root mode and allows the VMM to maintain control of processor resources. The new VMCALL instruction is an example of such an instruction that triggers VM exits.

3 Literature Review

Virtualization has become an essential part of modern computing systems, as it allows for efficient resource management and the simultaneous execution of multiple operating systems on a single host machine. The hypervisor, which lies at the heart of virtualization, manages the connections between virtual machines and the underlying hardware resources. Researchers in this field seek to enhance the performance and efficiency of hypervisors, to reduce the overhead typically associated with virtualization.

The study in question highlights the development of a custom, lightweight hypervisor, tailored specifically for Linux platforms. Implemented using the C language and shell scripting. The hypervisor was built using Intel VT-x technology, which provides hardware support for virtualization and enables the management and creation of virtual machines.

Previous studies have explored the performance and efficiency aspects of Type 1 (bare-metal) and Type 2 (hosted) hypervisors. While Type 1 hypervisors offer better performance and security, Type 2 hypervisors are largely used for desktop virtualization and testing environments. This research project focuses on creating a custom, lightweight Type 2 hypervisor for Linux environments, aimed at enhancing efficiency and performance.

Intel Virtualization Technology (VT-x) has been the focus of many studies, as it provides support for hardware virtualization and helps create complex virtualized environments. The VMX (Virtual Machine eXtensions) instruction set enables the management and control of virtual machines on Intel processors. VMX functions allow the hypervisor to configure virtual machine resources, control access to physical resources, and intercept unauthorized access attempts.

Analyzing studies on improving virtualization techniques, such as those conducted by Smith and Nair [1], and research on VMX technology by Kivity et al. [2], reveals the potential for optimizing VMX technology to support nested virtualization. This system, which allows virtual machines to run other virtual machines, has practical applications in cloud computing platforms that require multiple layers of virtualization to manage and control resources [3]. Past research in this area has aimed at identifying performance bottlenecks and proposing optimization techniques to improve the performance and efficiency of hypervisors in these contexts [4],[5].

Leveraging advancements in the realm of virtualization research, the study presents a custom, lightweight hypervisor that aims to improve performance on Linux platforms. By evaluating and optimizing the latencies of various VMX operations, such as VMLAUNCH and VMRESUME, the authors are successful in reducing virtualization overhead by employing their custom implementation [6]. This innovative approach opens the door for future research aimed at further optimization and its potential impacts on other platforms.

4 Methodology

4.1 Checking VT-x support and enabling virtualization

In order to fully utilize the full extent of virtualization, we must ensure the processor supports virtualization and is available. This is essential since systems are primarily designed to be operated only by the host OS. In order to turn on the virtualization, it must be enabled at the BIOS level.

However, we can check whether VMX is enabled by checking whether the VMX bits are set. This can be done using Assembly code, which can also be executed using C language. The 'cpuid' instruction is a special instruction in x86 and x86-64 processors that provides information about the processor itself. The cpuid instruction is commonly used by software to determine the capabilities of the processor and to optimize code for specific processor features. It is often used during system initialization and can also be used to determine the number of cores, cache sizes, and other hardware information. Bit 5 in the ECX register that indicates whether the processor supports Intel Virtualization Technology (VT-x), which is also known as Intel VT. This helped in checking whether the hardware virtualization is supported.

Now that we have ensured the virtualization is supported, VMX must be enabled in the processor so that VMX operations can be used. VMXON is controlled by the IA32_FEATURE_CONTROL MSR (MSR address 0x3A). For VMXON operation, Bit 0 (Lock bit) and Bit 2 (VMXON outside SMX operation) must be set to allow logical processors to allow VMX operations. The RDMSR instruction is used to read the Model-Specific Register (MSR) for IA32 Feature Control and check the respective bits.

4.2 Virtual Machine Control Structure (VMCS)

A Virtual Machine Control Structure interacts with the logical processor while performing VMX operations. A VMCS manages transitions into and out of VMX non-root operations as well as processor behavior during these operations.

A VMCS data structure has six logical groups:

- Guest-state area - saves processor state during VM Exit and loaded back during VM Entry.
- Host-state area - processor state is loaded from the host-state area on VM Exit.
- VM-execution control fields - These fields control processor behavior in VMX non-root operation. They determine in part the causes of VM exits.
- VM-exit control fields - These fields control VM exits.
- VM-entry control fields - These fields control VM entries.
- VM-exit information fields - These fields receive information on VM exits and describe the cause and the nature of VM exits.

4.2.1 Loading the VMCS:. The revision identifier for the VMCS (Virtual Machine Control Structure) is a 32-bit value that identifies the format of the VMCS data structure used by the processor. The revision identifier is stored in the IA32_VMX_BASIC MSR (0x480), which is a control register used to configure the processor's virtualization features.

A bitwise AND operation is performed with the MSR value and 0x8fffffff. This is done because the uppermost bit of the IA32_VMX_BASIC MSR indicates whether VMX is allowed outside of SMX operation.

Before initializing the values in the registers, we must first load the VMCS structure to memory using their physical address. VMPTRLD is a privileged instruction in Intel VT-x virtualization technology used to load the VMCS pointer into the VMCS address field of the processor. It takes a single memory operand that points to the physical address of the VMCS region to be loaded.

4.2.2 Initializing the VMCS:. We use segment registers that are used to specify a base address and a limit for memory segments, which are contiguous blocks of memory that can be accessed by the processor. The segment registers are:

- ES (Extra Segment) - Used for addressing extra data segments.
- SS (Stack Segment) - Used for addressing the stack segment, which holds the program's runtime stack.
- DS (Data Segment) - Used for addressing the data segment, which holds the program's initialized global and static data.
- GS (General Segment) - Used for addressing the general segment, which is typically used for thread-local storage in modern operating systems.
- FS (F Segment) - Used for addressing the F segment, which is typically used for operating system specific purposes, such as accessing thread-local storage in older operating systems.
- CS (Code Segment) - Used for addressing the code segment, which holds the program's executable code.

For each of these segments, we also specify which memory address to be used by the host when the VM exits (VMCS_HOST), the guest when the VM is running (VMCS_GUEST), the base address of the segment for the guest (VMCS_GUEST_BASE), the maximum offset of the segment for the guest (VMCS_GUEST_LIMIT), the access rights for the ES segment for the guest, set to unusable (0x10000) to indicate that the segment is not usable (VMCS_GUEST _ACCESS_RIGHTS).

Similarly, we must initialize the addresses and their values for the Task registers, I/O Bitmap, VM Entry/Exit and Guest stack size.

4.3 Memory Management in Virtual Machines

Memory management is a crucial aspect of virtualization, as virtual machines (VMs) execute in isolated environments, sharing the same physical resources as the host machine. The hypervisor, which manages these interactions between VMs and their resources, plays a vital role in orchestrating memory management during VM Entry (VMLAUNCH) and VM Exit (VMEXIT) with respect to the operating system.

4.3.1 Memory Management during VM Entry (VMLAUNCH):. During VM Entry, or VMLAUNCH, the hypervisor sets up and configures the virtual environment for the VM. It creates the necessary data structures, such as shadow page tables, and establishes the initial conditions for controlling the execution of the VM. This configuration process involves:

1. Mapping guest physical memory addresses to host physical memory addresses, while ensuring isolation.
2. Enabling access to virtual devices and resources needed by the VM to execute properly.
3. Establishing the VM control structures, which help determine the control flow and state of operation during VM Entry and Exit.

alloc_page is a function that is used to allocate a new page of memory. When $alloc_page$ is called with *GFP_KERNEL* | *__GFP_ZERO*, it will allocate a new page of memory that is suitable for kernel use and zero out the contents of the page before returning it. This is often used in situations where the kernel needs to allocate memory for data structures or buffers that require a known initial state

The *GFP_KERNEL* flag indicates that the memory should be allocated from the kernel's normal memory pool, which is optimized for performance and typically used for long-lived kernel data structures. The *__GFP_ZERO* flag indicates that the page should be zeroed out before it is returned, which helps to ensure that sensitive data is not left in memory that is reused from a previous allocation.

kmap is a function that is used to map a page of memory into the kernel's virtual address space. When *kmap* is called with p->page, it maps the physical page associated with the struct page pointed to by p into the kernel's virtual address space, and returns a kernel virtual address that can be used to access the page's contents. This is often used in situations where the kernel needs to access the contents of a page of memory that has been allocated for use by a device driver or other kernel component.

page_to_phys is a function that is used to retrieve the physical address of a page of memory. The function takes a pointer to a struct page as its argument, and returns the physical address of the page.

When *page_to_phys* is called with p->page, it retrieves the physical address of the page associated with the struct page pointed to by p. This is often used in situations where the kernel needs to perform low-level operations on the physical memory associated with a page, such as when setting up direct memory access (DMA) buffers or accessing hardware registers that are mapped to physical memory.

4.3.2 Memory Management during VM Exit (VMEXIT):.

When a VM exits execution, the hypervisor performs necessary cleanup and de-allocations. This process includes:

1. Releasing any memory allocated to the guest VM.
2. Saving and restoring context on the hardware to maintain state separation between the host and guest.
3. Checking whether any collected performance metrics need to be aggregated, logged, or stored for later analysis.

kunmap is a function that is used to unmap a page of memory from the kernel's virtual address space. The function takes a pointer to a page of memory as its argument, and releases any resources associated with the mapping.

When *kunmap* is called with p->page, it unmaps the physical page associated with the struct page pointed to by p from the kernel's virtual address space, and releases any resources associated with the mapping. This is typically done after the kernel has finished accessing the page's contents using *kmap*.

__free_page is a function that is used to release a page of memory that was previously allocated using *alloc_page*. The function takes a pointer to a struct page as its argument, and releases the page back to the kernel's memory allocator.

When *__free_page* is called with p->page, it releases the physical page associated with the struct page pointed to by p back to the kernel's memory allocator. This makes the page available for use by other kernel components that may need to allocate memory.

It is important to note that *__free_page* should only be called on pages that were allocated using *alloc_page*, and that the page should not be accessed after it has been released.

By employing efficient memory management strategies during VM Entry and Exit, hypervisors not only improve performance but also maintain security and resource isolation, contributing to a seamless virtualization experience.

4.4 VMXON and VMXOFF Operations

VMX provides the VMXON and VMXOFF instructions which are used to enable and disable Intel's Virtual Machine Extensions (VMX).

Before performing any VMX-specific operation, the host processor must enter VMX operation using the VMXON instruction. If VMX is supported, the CR4.VMXE bit is set. This can be checked using MOVQ instruction to read the value of the CR4 register on an x86 processor. The CR4 register is a control register in the x86 architecture that controls various system features, such as page translation and virtualization support.

The VMXON instruction is then used to enable the Virtual Machine Monitor. It takes a single operand, which is the physical address of the VMCS region.

Upon completion of VM-related tasks, the VMXOFF instruction allows the processor to disable VMX operation and clear the contents of the VMCS. This ensures that no further virtual machines can be executed on the processor, and that the processor returns to its original state. This instruction must be executed while the processor is in privileged mode (VMX root operation), similar to the VMXON operation. Furthermore, the CR4.VMXE bit should also be cleared after deallocating the VMCS memory and successfully disabling the virtualization.

4.5 Saving and Restoring Host State

When a virtual machine is running, the processor switches between executing the guest operating system and the hypervisor or virtual machine monitor (VMM) that is managing the virtual machine. To ensure that the guest operating system cannot modify the state of the hypervisor or VMM, the hypervisor or VMM must save the state of the host system before entering the guest environment.

There are two parts of the host state that need to be saved before entering the guest (non-root) environment:

1. Global Descriptor Table Register (GDTR) : The GDTR is a special-purpose register in the x86 architecture that contains the base address and size of the Global Descriptor Table (GDT). In virtualization, it is used to manage the memory protection and access privileges of individual segments of memory.

2. Interrupt Descriptor Table Register (IDTR) : The IDTR is a special-purpose register in the x86 architecture that contains the base address and size of the Interrupt Descriptor Table (IDT). IDT is a crucial part of the interrupt handling system in x86 processors, and it is used to manage the handling of hardware and software interrupts.

To save the contents of the GDTR and IDTR during virtualization, the hypervisor or VMM can use the *sgdt* and *sidt* instructions to load the contents of the GDTR and IDTR respectively, into memory. The memory location can then be saved as part of the host state.

When the hypervisor or VMM needs to return to the host environment, it can use the *lgdt* and *lidt* instructions to load the contents of the GDTR and IDTR back into the processor, restoring the original memory protection and access settings.

4.6 VM Entry and Exit

Once the host state is saved, we can launch the VM and enter the guest state. To do so, VMX provides two instructions *vmentry_prepare* and *vmlaunch*. *vmentry_prepare* is a preliminary step before entering a virtual machine, and it is used to prepare the processor for executing in the guest environment. This instruction sets up the processor's registers and flags for virtualization, and it checks that the processor is in the correct state to enter the guest environment. After *vmentry_prepare* is executed, the *vmlaunch* instruction is used to launch the virtual machine. *vmlaunch* transfers control to the guest operating system and starts execution in the guest environment. If successful, *vmlaunch* executes the guest operating system in the virtual machine, and the processor switches between executing the guest and the hypervisor or VMM.

Once the guest execution is completed, the VMCLEAR instruction is executed. The *vmclear* instruction takes a single operand, which is the physical address of the VMCS that is to be cleared. The instruction is then used to clear the VMCS that is located at the physical address. Following that, the VMXOFF operation is executed and the memory allocated to the VM is freed.

During an unexpected termination of the guest operation or the Virtual Machine, an error code is set in the VMCS. Specifically, it is saved in the $VM\ Exit\ Instruction - Information\ Field$ (0x4400) to provide information about the instruction that caused the exit. This code can be obtained using the VMREAD operation to determine the cause of the VM execution failure.

5 Latency Measurement Methodology

We conduct a comprehensive check to verify machine support for VMX and ensure that VMX is enabled, and the VMX capabilities are cached. Next, memory is allocated for VMX regions, and VMXON and VMCS regions are set up with the correct revision identifier. VMXON is enabled, and the VMCS region is loaded using the VMPTRLD instruction.

The VMCS is initialized with appropriate values, including the host and guest state, and the host's GDT and IDT are saved before the VMLAUNCH operation. Then, the main function '_vmlatency()' measures latencies for VMLAUNCH and VMRESUME operations in a loop with exponentially increasing iterations (2^n) for n iterations.

Effective error handling is implemented to address VM exit issues by using 'handle_early_exit()' and 'handle_vmexit()' functions. After latency measurements are completed, the VMCS is cleared with the VMCLEAR instruction, the VMX off state is restored using VMXOFF, and the host's GDT and IDT are restored. Finally, allocated memory for VMX regions is freed, and the execution concludes.

Latency measurements from VMLAUNCH and VMRESUME operations are presented in a tabular format, illustrating latencies for each number of iterations in the loop. This methodology provides an accurate and comprehensive approach to executing a sequence of VMX operations on x86-64 systems while ensuring the correct system state and contingency measures are in place.

6 Results

As part of our test, we ran multiple guest operations to check the latency in performance. We tested two levels of virtualization. One through direct Host OS and the other through a virtual machine on VMWare. We used a dual-booted system (Ubuntu and Windows) with 16GB RAM and Intel 8th Gen i7-8705G Kaby-Lake G Processor.

Table 1 shows the performance of the hypervisor on the host machine directly. The host OS is Ubuntu 22.10. The table shows two operations performed- Get CPU ID and Matrix Multiplication. From the table, we can see that the number of cycles increase for Matrix multiplication as the process is more complex than fetching CPU information.

As the number of executions increases, the latency for the *Host_Mat_Mul* process tends to fluctuate slightly, such as in the cases where it decreases after 4096 executions. However, from the 65536 and 262144 executions, we can see that the latency starts stabilizing gradually.

In contrast, the latency of the *Host_Get_CPU_ID* process experiences a consistent decrease with an increasing number of executions. This decline in latency suggests that the process is becoming more efficient as the number of executions increases. General trends can be observed, and more in-depth analyses can help further optimize the processes.

No. of executions	Host_Mat_Mul	Host_Get_CPU_ID
1	836	584
4	855	567
16	866	559
64	868	559
256	867	559
1024	867	558
4096	869	561
16384	779	538
65536	791	539
262144	761	529

Table 1. Latency results (in no. of CPU cycles) for 2 operations on the Host machine.

No. of Executions	Host's Get_CPU_ID	VMWare Get_CPU_ID
1	584	867
4	567	848
16	559	839
64	559	839
256	559	839
1024	558	838
4096	561	842
16384	538	814
65536	539	816
262144	529	804

Table 2. Latency results (in no. of CPU cycles) for direct Host machine access and through VMWare.

We also tested the hypervisor latency when run through another VM. We used VMWare Workstation 17 Pro running on Windows with the same hardware configurations. The Virtual Machine, however, had only 4 CPU cores and 4 GB of RAM, running Ubuntu 22.10. Table 2 shows our results for the second run.

It is worth noting that there is a 45% increase in the number of CPU cycles in this case since although the Virtualization is enabled in VMWare, there is still a significant overhead while relaying control from the guest to host. Furthermore, the host OS limitations restricted us from exploring a much higher VM configuration for testing.

7 Conclusion

The results of our research demonstrate the potential for performance improvement in hypervisors by utilizing the VMLatency benchmarking tool to measure and optimize VM execution latencies. Our custom lightweight hypervisor achieved a 45% higher latency on VM compared to Host. Future work could further explore hypervisor optimization methods and their potential impacts on other platforms.

References

[1] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, Intel Corporation, August 2023.

[2] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, no. 8. Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[3] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *ACM Sigplan Notices*, vol. 41, no. 11, pp. 2–13, 2006.

[4] 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," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[5] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, 2005.

[6] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[7] C. A. Waldspurger, "Memory resource management in vmware esx server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.

[8] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 1–14.