Intel VT-x Virtualization experiments with concurrent hypervisors

Antonio Cardace, Michele Cucchi

Università di Bologna, Dipartimento di Informatica - Scienza e Ingegneria
`antonio.cardace2@studio.unibo.it, michele.cucchi4@studio.unibo.it`

Abstract. VT-x is the hardware virtualization support for the x86 architecture, it allows a virtual machine to have near-native performance in terms of execution speed. In this document we introduce a limitation of the current technology as implemented in the Linux Kernel 4.x and we try to get a deeper understanding through some experiments we have conducted.

1 VT-x Explained

VT-x Intel is the acronym of Virtualization Tecnology for x86, it is implemented with a subset of new x86 instructions able to assist the virtualization process. The new instructions are the following [1]:

- VMXON and VMXOFF
- VMPTRLD and VMPTRST
- VMCLEAR
- VMREAD and VMWRITE
- VMCALL
- VMLAUNCH and VMRESUME

1.1 VT-x Architecture

The architecture Intel has defined for its virtualization technology is fairly simple, it basically consists of two software entities :

- **Virtual machine monitor**: the VMM acts as a host and has full control of the processor(s) and other platform hardware. A VMM presents guest software with an abstraction of a virtual processor and allows it to execute directly on a logical processor, it is able to retain selective control of processor resources, physical memory, interrupt management, and I/O. This software entity must be unique in the whole system at any given moment since the VT-x support is not meant to be multiplexed.
- **Guest software**: Each virtual machine is a guest software environment that supports a stack consisting of operating system (OS) and application software. Each operates independently of other virtual machines and uses on the same interface to processor(s), memory, storage, graphics, and I/O provided by a physical platform. The software stack acts as if it were running on a platform with no VMM.

1.2 Activating VMX

To enable a processor to run VT-x assisted virtualization the OS must execute the **VMXON** instruction which activates a new processor execution mode called *VMX operation mode* which has two submodes called *VMX root mode* and *VMX non-root mode*.
To exit the VMX mode the **VMXOFF** instruction can be called from kernel space.

1.3 VMXON and vmxon region

The **VMXON** instruction needs as an argument a pointer to a VMCS memory area (which will be covered in the next section) in which it saves some hypervisor specific data, this area must be 4kB aligned and every processor or core in the system must have its own area. No other software should modify this area while vmx mode is active otherwise unpredictable behavior may occur.

1.4 VMX Execution modes

The VMX root mode is dedicated to running the virtual machine monitor software also known as the *hypervisor*, VMX non-root mode instead is the *guest mode* dedicated to running the virtual machines, whose machine code is not emulated but executed on the bare hardware.
Both of these modes have the normal x86 privilege modes separation in *rings*. *Ring 0* is the most privileged, dedicated to running the kernel and *ring 3* is the least privileged normally dedicated to running th user mode software.
The software running in guest mode is agnostic of VMX execution mode, from its point of view the processor mode and state is equal to the normal not-virtualized environment granting unmodified software execution. In case of exceptions the guest mode execution is stopped and the processor state is saved before switching to hypervisor mode, where the virtual machine monitor can handle the cause of the exception.

1.5 VM entry and exit

The hypervisor can start a virtual machine switching the processor to *guest mode* through the instructions VMLAUNCH or VMRESUME. VMLAUNCH must be used upon virtual machine creation, VMRESUME instead every time switching to guest mode.
The *VM exit* operations to VMX root mode will occur every time the processor execute a special instruction or at the occurrence of a particular event such as an exception. The hypervisor can specify by setting the appropriate bits in the vmxon region which instructions should cause a VM exit.
VM exit events are handled in a similar way to the normal processors exceptions like interrupts and traps. The actual process execution is suspended and the CPU state is saved in a dedicated data structure in memory called *virtual machine control structure VMCS*.

1.6 VMCS

The VMCS structure maintains all the data to correctly handle transitions between VMX states of every virtual processors, in particular it stores six areas of data:

- **Host state area**: where the host processor state is saved during the hypervisor to guest mode transition
- **Guest state area**: where the guest processor state is saved during guest to hypervisor mode transition
- **VM execution control fields**: determines the processor behavior in guest mode
- **VM entry control fields**: control VM entry
- **VM exit control fields**: control VM exit
- **VM exit information fields**: a read only area containing information about the reasons of a VM exit

The structure must be aligned on a 4KByte memory page boundary. Every virtual processor must have its own VMCS structure, forcing the hypervisor with multiple virtual processors to use different structures for every single virtual CPU.

The hypervisor can handle the VMCS through its address that can be set as the pointer to the *current VMCS* for the running processor with the **VMPTRLD** instruction which copies an address to the processor specific pointer. The VMCS current pointer can be replaced with a new VMCS pointer or can be cleared with VMCLEAR instruction, in that case there is no current VMCS active.

The other instructions:

- VMLAUNCH
- VMPTRST
- VMREAD
- VMRESUME
- VMWRITE

can operate only on the current active VMCS. The execution of one of the previous instructions without setting current VMCS will cause an exception.

2 KVM (Linux implementation)

2.1 Architecture

The KVM architecture is quite simple, it consists of just two loadable kernel modules, the first one is a generic kvm module which exposes many hooks that abstract the inner mechanism (very dependent by the underlying hardware) of the hypervisor, the second one is a module which implements the functions defined by the hooks of the generic one and actually executes the CPU instructions needed to make the hardware virtualization work.

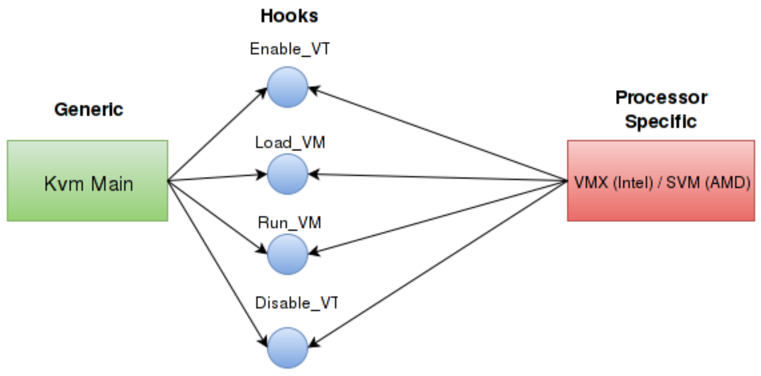

Fig. 1. Kvm architecture

2.2 Kernel Integration

The KVM approach at building a hypervisor is really smart, what it does is not trying to reinvent the wheel. As a matter of fact this Linux implementation of a virtual machine monitor integrates tightly with the kernel and presents itself as a simple process running in the system and waiting for the user to make use of it to spawn virtual machines, without the need to re-implement a scheduler, an interrupt handler, a memory manager, ecc.

What we've just described is possible because what KVM does when Qemu uses its API to create a virtual machine is not a fork() system call but just a load of all the necessary CPU register with the correct values which allows to execute the VM, therefore KVM actually becomes the virtual machine and let the Kernel handle all the complex tasks as usual, this is a radical approach compared to what other hypervisor do (XEN).

Listing 1.1. vmx.c (Linux 4.3)

```
/* Load guest registers.  Don't clobber flags.*/
mov %c[rax](%0), %% _ASM_AX
mov %c[rbx](%0), %% _ASM_BX
mov %c[rdx](%0), %% _ASM_DX
mov %c[rsi](%0), %% _ASM_SI
mov %c[rdi](%0), %% _ASM_DI
mov %c[rbp](%0), %% _ASM_BP
...
...
 /* Enter guest mode */
jne 1f \n\t
__ex(ASM_VMX_VMLAUNCH)
jmp 2f \n\t
1:   __ex(ASM_VMX_VMRESUME)
2:
/* Save guest registers, load host registers, keep flags*/
mov %0, %c[wordsize](%% _ASM_SP )
...
```

2.3 Execution Cycle

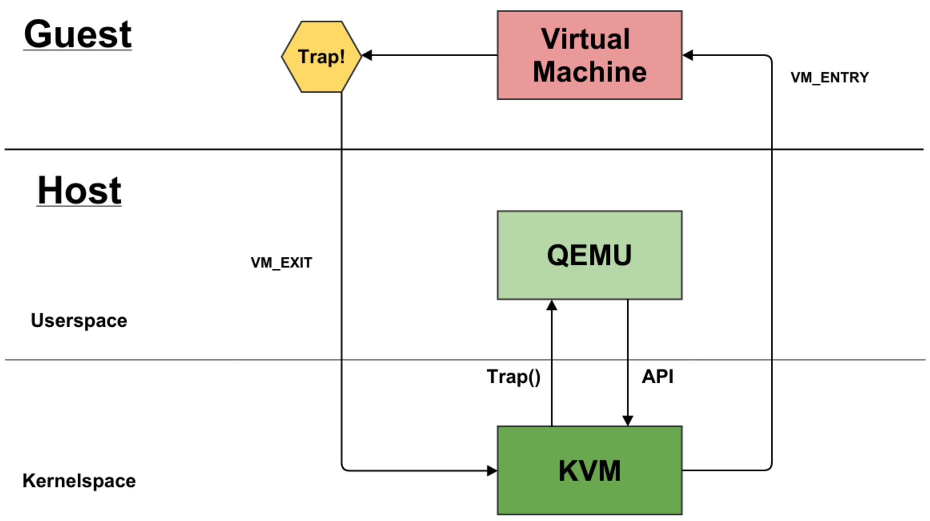

Fig. 2. Kvm execution cycle

In this section we are going to describe how KVM manages the execution of a virtual machine.
The execution cycle is simply composed of these steps:

1. Run VM
2. Trap
3. Emulate/Process operation

The process is actually initiated by the QEMU process which uses the KVM APIs to create a virtual CPU as to spawn a virtual machine image on it.

Then KVM takes place and executes the real vmx instructions which tell the CPU to instantiate a new virtual machine and to start it, this is called VM_ENTRY. After a VM_ENTRY the guest OS start its execution which can be stopped by either the end of its time-slice or a trap, which is an illegal opcode executed by the virtual machine, when any of the two conditions occur there is a VM_EXIT, after which the control returns to KVM.

At this point KVM can dismiss the operation done by the virtual machine or emulate it (it usually does the latter), if the choice is to emulate it what happens most of the times is that there is an upcall towards QEMU which actually emulate the instruction using one of its device emulators or by doing some other generic operation (like creating a file).

This scheme continues for the whole lifecycle of the virtual machine.

3 Hypervisors Concurrency

As stated the purpose of this work was to make the execution of 2 or more different hypervisors (using the VT-x technology) possible, therefore we studied the implementation of the hardware virtualization support on Linux trying to understand if there was any lack of functionality or just some things which could have been done better. In order to report all we have found out about VT-x and KVM we will list everything we have discovered about this technology related to hypervisors concurrency.

3.1 About The Hardware

The one real problem that keeps the concurrent execution of hypervisors from happening resides in how the hardware implementing VT-x was designed and more specifically is about the way a virtual machine monitor gains the power of spawning VMs.
To understand this a little recap is needed, a VMM in order to enable the VT-x functionalities has to prepare a page in memory, called the vmxon region, which contains all the specific setting of the hypervisor, and this is precisely what KVM does, once one hypervisor has configured its vmxon region and activated

the VT-x no other process can because of this, moreover if a virtual machine is scheduled on a different core from which it was first created KVM repeats the same process on the new core, not only acquiring the resource of the new core but adding some overhead to the virtual machine scheduling since a copy of the VMCS structure and the vmxon region is needed. It is clear that Intel never thought about the coexistence of two or more processes making use of the hardware virtualization support, otherwise the activation process would have been designed differently.

3.2 About KVM

Even if Intel did not think about concurrency KVM did, as a matter of fact there is a kernel module parameter called *vmm_exclusive* which tells the Linux Kernel hypervisor if there is another process supposed to be using the VT-x technology at the same time.
The inner workings are simple, when *vmm_exclusive* is true KVM just goes along with its standard and well-tested execution flow, on the other hand when the parameter is set to false KVM tries to let another hypervisor in the game by switching on and off everytime any of its virtual machines are scheduled-in or scheduled-out.
Obviously this mechanism is working as long as any other hypervisor currently executing is doing the same, thus this approach is highly susceptible to what other hypervisor are doing and some of them might not actually play this nice. Moreover following this approach incurs in a non-avoidable performance penalty, in fact switching some piece of hardware on and off continuously does not come for free, we tried to measure the time spent doing this operation at each time-slice of the virtual machine and the overhead resulted approximately around 400 nanoseconds.

3.3 About Partial Virtualization

One of the many objectives of this project was to discover if the virtualization hardware support offered by Intel could be exploited to implement partial virtualization with near-native performances, the answer is simply no and the reasons are quite straightforward.

First the current VT-x technology has no support or functionality to trap the SYSCALL instruction, even though Intel included features to trap many other single instructions this one has simply not been taken under consideration, thus we do not have any reliable tool to understand and intercept when a user-space process is making a request to the Kernel.

Second even if the just described feature was present it would be just useless because the Intel hardware virtualization support has been clearly designed to implement full virtualization, as a matter of fact when the hypervisor launches a virtual machine the environment in which it starts is in real-mode (16-bit), thus a bootloader and full OS is needed to actually boot the machine and this is just too much bloatware and probable issues to implement partial virtualization using VT-x.

4 Experiments

4.1 Introduction

The goal of the following experiments is to explore the possibility of running concurrent different hypervisors on the same system without significant modifications to the Linux kernel subsystems.

4.2 Host Kernel Patch Experiment

To store the VMCS pointer between context switches we modified the host kernel scheduler and the kvm kernel module adding data structures to maintain the VMCS active pointer between processes interleaving.

Oracle Virtualbox Open Source Edition The second Hypervisor which we took into consideration was Oracle Virtualbox (opensource edition). We made a little modification to the Virtualbox kernel module **vboxdrv** to remove the *VMXON* call, as to prevent the hypervisor to start if the vmx mode had been already started.

Test We started KVM and Virtualbox at same time with and without the host kernel patch and with the KVM exclusive parameter activated or not but all the tests resulted with the host kernel crashing with oops or panic output. Sometimes the host system freezed or the behavior was unpredictable resulting in a running system which kept refusing all the inputs.
This did not a seem a good idea anyway because the behavior of the hardware is just unpredictable doing what just described above since there is no information about what we tried on the official Intel documentation [1].

4.3 Exploiting the Scheduler

We tried running different hypervisors using the system call offered by the Linux Kernel to set a thread's CPU affinity mask, namely *sched_setaffinity()*, this call is used on multi-processors systems to obtain performance benefits by dedicating one CPU to a particular thread.

This workaround tried exploiting the hardware design of VT-x and the Linux scheduler to firmly bind the hypervisor process to one or more cores of the CPU, this kind would be really flexible and efficient for many reasons:

1. The solution is hypervisor-independent;

2. No modification to the kernel or the hypervisor, everything is taken care of in user space;
3. The same hypervisor can be assigned as many cores as it needs (virtual machines with more than one core);
4. The same hypervisor can run different VMs on the same core/cores, they will play nice between each other;
5. binding the hypervisor process to a core boosts the performances, flushing the TLB or migrating VMs is no longer needed;

Usage The following is a simple usage example of the application developed.
Run QEMU-KVM on a single core:

```
$ start-hypervisor --core 0 qemu -enable-kvm vm.qcow2
```

Allocate core 0 and 2 for KVM:

```
$ start-hypervisor --core 0,2 qemu -enable-kvm vm.qcow2
```

Allocate core 0,1,2,3 for KVM:

```
$ start-hypervisor --core 0-3 qemu -enable-kvm vm.qcow2
```

Limitations This solution would allow to run different hypervisors concurrently but still has to play fair with the limitation imposed by the hardware, therefore it is obvious that the maximum number of different hypervisors a system can run concurrently is equal to the number of available cores, given that only one core has been allocated for each VMM.

Test We tried to run different hypervisors on the same system by using the just described system call, it simply won't work because after analyzing KVM's source code we found out that the Kernel virtual machine monitor when told to create a virtual cpu under the hood instead executes a VMXON on every available processor, this obviously cancels the possibility of exploiting the Linux scheduler for running multiple hypervisors.
Given KVM's behavior we patched its code to only execute a VMXON instruction on the CPU where it is currently executing, nevertheless this approach did not work either resulting in an unstable system with other hypervisors allocated on different cores refusing to start because some other process activated the hardware support before them.

5 Conclusions

Concurrently running different hypervisors using the VT-x architecture without switching off the vmx mode before vmm interleaving is probably impossible with the current hardware design.

The main issue is the strict binding between the VMXON operation and the hypervisor that called the instruction. VMM switching operations without turning off VT-x are unable to maintain coherent VMCS data, resulting in unpredictable results or the host system crash.
The current design of the VT-x hardware results messy under every aspects, the system behaves strangely when experimenting with these features, one example is the VMXON instruction which is described as having effect on a single core (where it is actually executed), instead it has side-effects on the other cores making the system unstable and unpredictable.
In conclusion the VT-x technology as of today is surely powerful and efficient but it lacks of a broader view about the virtualization matter making their usage very restricted and most of the times confusing.

References

1. Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manuals*, 2015. Volume 3 http://www.intel.com.
2. *The Architecture of Virtual Machines*, James E.Smith University of Wisconsin-Madison ; Ravi Nair IBM T.J. Watson Research Center ; May 2005 IEEE Computer Society
3. *Formal Requirements for Virtualizable Third Generation Architectures*, Gerald J. Popek University of California, Los Angeles and Robert P. Goldberg Honeywell Information Systems and Harvard University