Hardware passthrough

G. Lettieri

22 Nov. 2018

1 Introduction

I/O emulation poses the biggest performance challenge in virtual machines. In all the virtualization techniques that we have seen so far, the Virtual Machine Monitor (VMM) has to participate in all the interactions between the guest software and the I/O peripherals. In hardware assisted virtualization, this causes a lot of expensive Virtual Machine exits.

One way to overcame these problems is to give a VM direct access to an host peripheral. We say that the peripheral is *passed through* to the VM. In terms of our model, the target machine contains a device which is the same as a device in the host machine, and the virtual machine maps the target device directly onto the host device. Note that the device is now dedicated to that particular VM, and it cannot be used by either other VMs or the host.

This is an expensive solution, since we are essentially giving up all the benefits of virtualization for the passed-through peripheral: we cannot share it, we have removed the software layer that gave us more flexibility, and we also have to face a security problem, since we must now guarantee that the VM is only able to access that peripheral and nothing else. Nonetheless, passthrough may be a cost effective solution, especially when coupled with other hardware techniques like PCI SR-IOV[1]. This technique is often used for network cards, to establish a fast data path between VMs and the network, and it is very important for Cloud service providers, which offer their customers VM access over the Internet.

To implement passthrough in hardware assisted virtualization we need, again, help from hardware. This is because the VMM software is not running, in general, when the interactions between the guest and the peripheral take place, and therefore the VMM needs help from the hardware to guarantee that these interactions are properly handled.

The interactions involve three mechanisms: reads and writes to I/O registers, DMA and interrupts. Let's examine each of these in turn.

[1] This is a feature of some PCI devices that may offer several virtual instances of themselves. The instances may be used independently, even if they internally share some hardware.

2 I/O registers

Assume we want to passthrough a device D to a VM M. For reads/writes to/from the registers of D, we want the hardware to complete the operation without VMM intervention, accessing the real registers of D. For all other I/O reads and writes, we want the hardware to intercept the accesses as usual, yielding control back to the VMM.

This is easy to implement using Intel VMX for registers that live in the I/O address space. The Virtual Machine Control Structure (VMCS) contains a pointer to an I/O bitmap, with one bit for each possible I/O address (there are 65536 I/O addresses, therefore the size of the bitmap is just 8 KiB). While in non-root mode, the CPU will check the bitmap for all the addresses used by all `in` and `out` instructions. The CPU will either complete the instruction or cause a VM exit, according to the value of the corresponding bit. The VMM may prepare the bitmap for the M VM by setting all the bits that correspond to registers of the D device, and resetting all the other ones.

For memory mapped I/O registers, the VMM must use the host MMU to map some guest physical addresses to the host physical addresses where the registers reside. Since the mapping is per-page, it is imperative that each page only contains registers from at most one device. It is not necessary that the guest and host physical addresses are the same, but there is usually no reason for them to differ.

3 DMA

Assume that the passed-through device is DMA-capable, i.e., it is able to perform read and write operations on the system memory by itself. Of course, the DMA operations must be programmed by software and, in particular, the software must tell the device the memory addresses to read and/or write, e.g., by writing the addresses into some device registers.

Consider now the target machine in Fig 1. The software, running in the target CPU, writes address F into the register of the target device as part of the programming of a DMA operation. At a later time, the device will access memory at address F. This is the expected behaviour of the DMA operation that we need to reproduce if we want to run the same software inside a virtual machine. But, if we use only the mechanisms that we have seen so far, we run into a problem. Recall that, before the execution of each instruction, the target and virtual state are equivalent and, in particular, each guest visible CPU register and each location of the virtual machine physical memory contain the same value as the corresponding element in the target machine. Moreover, according to the previous section, when we pass through a device D, we let the host CPU directly execute all write operations into the registers of D. Therefore, if we consider the state when both the target machine and the virtual machine are about to execute the write of F into the device register, we can see that, at the end of the operation, also the *host* device register will contain F (since

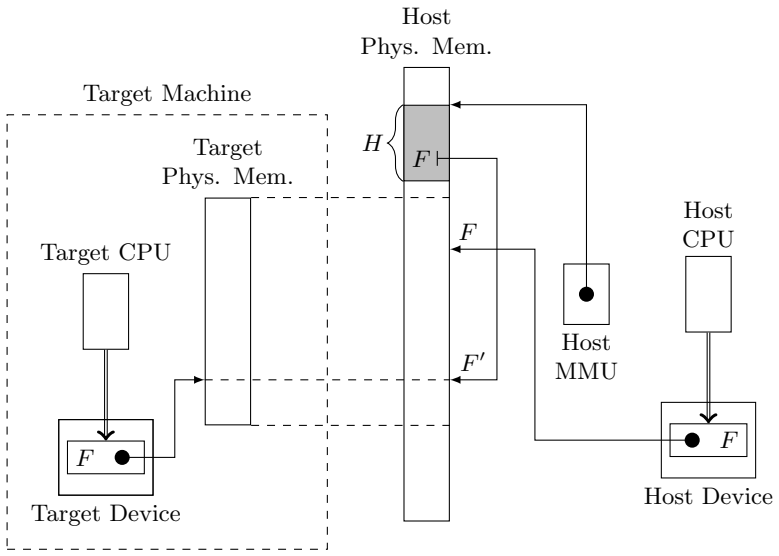

Figure 1: Mismatch between guest and physical addresses in device passthrough DMA.

the initial target and virtual states are the same, the instruction is the same and the processor is the same). The problem, of course, is that F is a *guest* physical address, but the host device will use it as a *host* physical address. In fact, guest physical addresses are translated into host physical addresses by the host MMU (see mapping H in Fig. 1) but I/O devices are not connected to the MMU. Therefore, very different things are going to happen when the target and host devices will both try to access address F in memory. To match the target machine behaviour, the virtual machine should access host physical address F', not F.

Note that the above is not only an error in the simulation, but also a security problem. In fact, a malicious guest may write whatever it wants into the host device register, and therefore gain indirect access to parts of the host memory that the VMM had not assigned to it. These may include the emulated physical memory of other virtual machines, or the reserved memory of the VMM itself.

Note also that the problem is caused by the fact that we have granted the VM direct access to the device registers. If, instead, the write operation to the above device register causes a VM exit, the VMM may regain control, see that the guest was about to write address F, translate it through H to obtain F' and write the latter value into the device register. This would solve the problem, but it again would require a VM exit for every read and write operation to the device registers, at least those that contain memory addresses.

One may think to solve the problem in hardware by letting all addresses go through the host MMU, not only the addresses coming from the CPU, but also those coming from I/O peripherals. However, this does not work and it is

important to understand why.

A first problem is that, if the target machine has an MMU and the Extended Page Tables are not available, the host MMU will point to the $H \circ G$ translation, while the DMA addresses must be translated using only the H translation (the guest will have already performed the G translation before passing the address to address to the device). We must assume, therefore, that either the guest has no MMU (or it does not use it), or that the VMM is using EPTs. In the rest of the discussion, we assume that EPTs are available. For the host MMU, we only show the pointer to the H translation.

There is a second problem. Consider Fig. 2, where we have done two things: we have let the address generated by the host device go through the host MMU, but we have also added a second virtual machine to the system. Each virtual machine uses its own subset of the host physical memory to implement the corresponding target physical memory. Two guest-host address translations are available: H_1 for the first virtual machine and H_2 for the second. Since there is only one host CPU, only one virtual machine at a time will be running. When the host CPU is used by virtual machine #1, the host MMU must point to translation H_1 and, similarly, when the host CPU is used by virtual machine #2, the host MMU must use translation H_2. The VMM will take care of switching between the two translations whenever it orders a switch from one VM to the other.

Now assume, again, that we pass through an host device to VM #1 and that the guest software running inside it writes the guest #1 physical address F into the device register. For correct operation, this address must be translated to F' when the device later tries to access the host memory. This would be the case if, at that time, the MMU were using translation H_1. The problem is, however, that the we have no guarantee that this is indeed the case, since the device operates asynchronously with the rest of the system. It may well be the case that, when the device is ready to access host memory, the VMM has scheduled VM #2 and, therefore, the current active translation is now H_2, as shown in the Figure. Address F would thus translate to F'', an address inside the emulated memory of VM #2.

What we need is another kind of MMU, that is used only by I/O devices, is capable of operating several possible translations, and chooses the correct translation depending on the I/O device which has issued the memory access request. This is called IOMMU and is now available on most machines based on Intel and AMD processors (for PCs you may need to enable it in the BIOS).

Fig. 3 illustrates the idea. In the Figure we have two virtual machines, one emulating Target Machine #1 and another emulating Target Machine #2. Host Device A as been assigned to VM #1 (the one that emulates Target Machine #1) and Host Device B has been assigned to VM #2 (emulating Target Machine #2). The IOMMU translates all addresses coming from all devices. In order to perform the translation it uses a data structure (in the Host Physical Memory) that maps each device to a set of page tables. The IOMMU is an additional device and does not replace the MMU. The MMU continues to be attached to the Host CPU. In the Figure, the Host CPU is running VM #1, and therefore

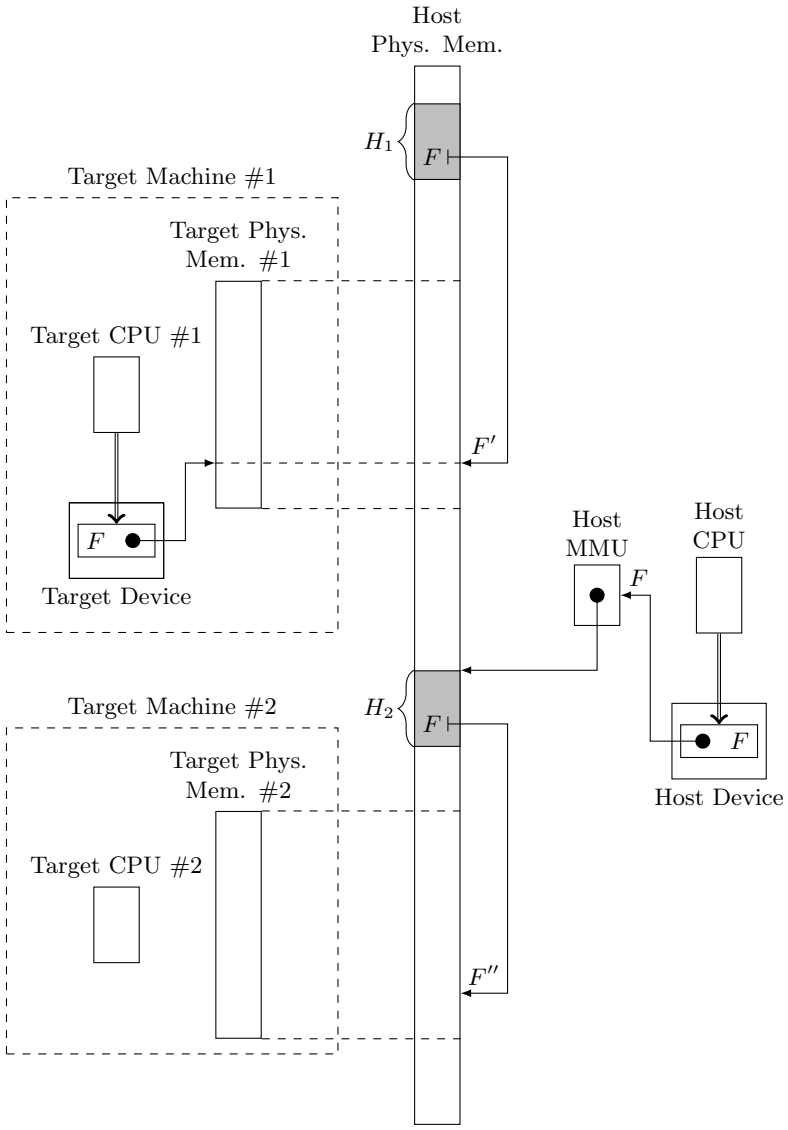

Figure 2: Using the host MMU is not sufficient for DMA operation in passthrough.

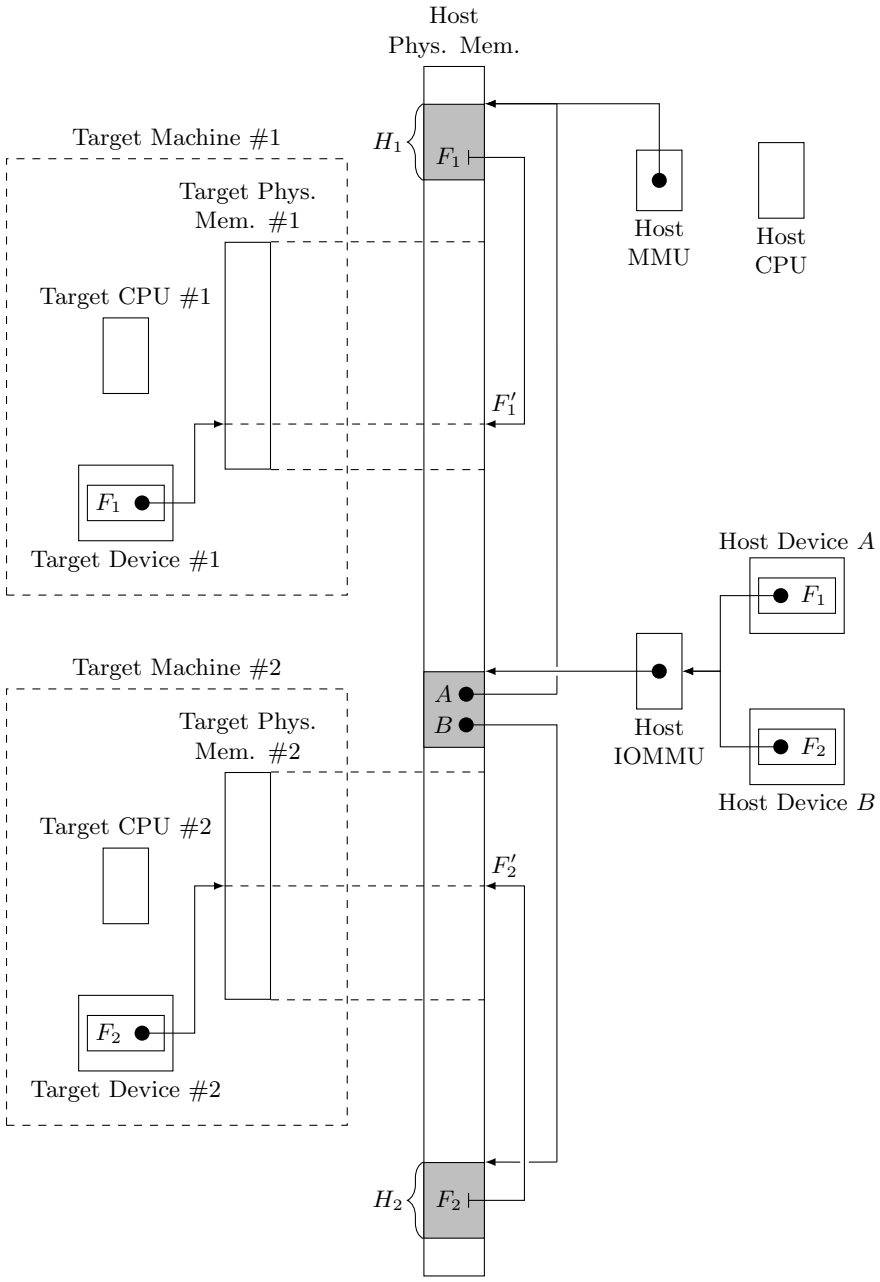

Figure 3: Two virtual machines, each one with a passed-through device, served by an IOMMU.

the MMU points to translation H_1. The IOMMU, however, is always able to perform both translations. To allow the IOMMU to choose the right translation, the host devices must identify themselves in each memory operation they issue on the bus. This is automatically done in PCI express systems, since each transaction contains the bus/device/function triple that uniquely identifies the transaction originator in the system. Since Host Device A is assigned to VM #1 (to implement Target Device #1), the IOMMU data structure maps it to translation H_1. Similarly, Host Device B is mapped to translation H_2. All the VMM has to do when it needs to passthrough a device to a VM is to add an entry for the device in the IOMMU data structure. (Devices that have no entry in the data structure are assumed to belong to the host, and therefore the IOMMU lets their address to reach memory untranslated).

Intel and AMD IOMMUs are also capable of issuing page faults by interrupting the Host CPU if they find some unset present bit during the translation. Therefore, there is no need for the guest physical memory to be always resident in host physical memory. Translations can be cached inside the IOMMU, and there is also the support for TLBs that reside inside the devices themselves, to improve the caching scalability.

The IOMMU is a general device which may be used not only by VMMs, but also by ordinary multiprogrammed systems that use virtual memory for their processes. In much the same way as we have just seen, they can use the IOMMU to program DMA accesses between devices and swappable userspace buffers.

4 Interrupts

Until now, we have seen only two options for the management of interrupts coming while the processor is in non-root mode: either each external interrupt causes a VM exit (i.e., they are handled by the VMM), or none of them does (i.e., they are handled by the VM).

For passthrough, we would like the VM to handle the interrupts coming from the passed-through device and the VMM to handle all the other ones. Moreover, we need to take care of the fact that the passed-through interrupt may come while the VM is not running on the CPU. Indeed, a VM CPU may be in one of several running states, much like a process. Fig. 4 shows the possible states of a VM CPU. In the "Running" state the VM CPU is actually using the host CPU (in non-root mode). In the "Ready" state the VM CPU is stopped because the host CPU is currently being used by the VMM, some other VM (or even an host process, in the systems that run VMs alongside normal processes). A VM CPU may switch between the Running and Ready states for VM-exits and scheduling decisions operated by the VMM. When the VM is Ready, the VMM will schedule it at some future time. While in the Running state, the guest software running inside the VM may execute the `hlt` instruction, thus halting the VM CPU. Typically, the VMM will intercept the instruction, put the VM CPU into the "Halted" state and schedule something else. A VM CPU may exit from the Halted state only if it receives an interrupt. Note that here we refer

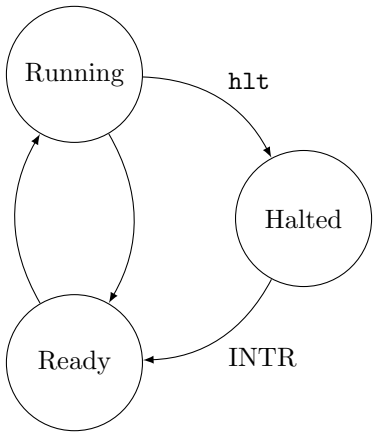

Figure 4: Running states of a Virtual Machine CPU.

to an interrupt directed to the *VM CPU*. This is, typically, a virtual interrupt generated by the front end of a software emulated device.

Now let's assume that a device D that has been passed-through to VM M generates an (host) interrupt. This is what we would like to happen:

1. if the M CPU is Running, we want the interrupt to be handled by the processor, jumping to the guest interrupt handler, without any VM exit (and, therefore, without any intervention of the VMM);

2. if the M CPU is Ready, the host CPU is currently running something else, e.g., another VM CPU; in this case we want to store the interrupt request somewhere, so that the M CPU may handle it at the later time when the M VM is scheduled and becomes Running;

3. if the M CPU is Halted, we need to store the interrupt request, as above, and also make the M CPU become Ready.

Note that we still want the VMM to handle all the interrupts that do not come from the passed-through device.

The *posted interrupt* mechanism, available in very recent CPUs, allows us to implement this idea. The mechanism is able to avoid any VMM intervention also in case 2, besides case 1. For case 3, instead, we still need the VMM to intercept the interrupt.

4.1 Posted interrupts

The *posted interrupt* feature must be supported by both the processor and the interrupt controller, that now uses an Interrupt Remapping Table (IRT). The idea is to keep a Posted Interrupt Descriptor (PID) in system memory for each VM. The running status of the VM is stored in the PID. When the interrupt

controller needs to deliver the interrupt to the VM, it reads the running status of the VM in the PID and either interrupts the processor, or it just stores (*posts*) the interrupt request in the PID itself.

The PID is a 64 byte data structure that contains several fields. The most important ones are the following:

- Posted Interrupt Request (PIR): a 256 bits field (just like the IRR and ISR registers of the APIC), one bit for each possible interrupt vector; this is where the interrupts are posted;
- Suppress Notification (SN): one bit that determines whether the controller, after posting the interrupt in the PIR, must also notify (see below) the CPU (SN=0) or not (SN=1); this bit is used to encode the running state of the VM;
- Notification Vector (NV): the CPU must be notified with an interrupt and this field contains the interrupt vector that must be used.

The IRT has an entry for each possible interrupt request, mapping requests to interrupt vectors and, optionally to a PID. Whenever the controller has to deliver an interrupt that is mapped to a PID, it behaves as follows:

1. it sets the proper vector bit in the PIR;
2. if SN is 1, it does nothing else;
3. otherwise, it interrupts the processor using interrupt vector NV.

Note that the interrupt vector of the incoming request is always stored in the PIR, and the actual vector that the processor sees is always the one stored in NV.

The PID is only used by the processor in non-root mode, and a pointer to the PID must be stored in the currently active VMCS. The VMCS execution control section must specify that external interrupts should cause VM exits (note that interrupts delivered through a PID may still not cause a VM exit, see below). The VMCS exit control section must specify that, on receiving an external interrupt, the processor must obtain the interrupt vector before exiting.

For simplicity, assume that interrupts are always disabled when the processor is in root mode (i.e., the VMM runs with interrupts disabled). While in non-root mode, the processor looks into the PID only when it receives an interrupt with a special vector, called the Active Notification Vector (ANV); otherwise, the processor continues to handle interrupts as always (i.e., with a VM exit in this case). If the ANV is received, instead, the processor automatically (i.e., via a microprogram) looks at the PIR and handles the interrupt vectors that it finds set, without leaving non-root mode.

Now let us see how the VMM may setup the PID to implement the desired behaviour for interrupt passthrough. Assume that interrupt I has to be passed through to VM M with vector V. The VMM creates a PID and writes its address in the VMCS of M. Moreover, it fills the entry for I in the IRT with

the vector V and the pointer to the PID. The VMM prepares an handler for a vector WNV (Wakeup Notification Vector), which must be different from ANV. Then, the VMM updates the PID as follows:

- when M goes into the Running state, the VMM sets SN=0 and NV=ANV;
- when M goes into the Ready state, the VMM sets SN=1;
- when M goes into the Halted state, the VMM sets SN=0 and NV=WNV.

Therefore, when VM M is Running, it will receive the passed-through interrupt without any intervention of the VMM. If M is Ready, the interrupts will be posted in the PIR, again without any intervention of the VMM. The VMM, however, must be notified when the interrupt comes while M is halted. The reason is that the VMM needs to know that M is now eligible for execution and has to update its own data structures accordingly. At the very least, it has to move the VM back into the list of the Ready VMs, so that it may be scheduled for the Running state. For this reason, the VMM changes the value of NV when it moves a VM to the Halted state: since NV is now different from ANV, interrupts delivered through the PID will not trigger the posted interrupt processing, but normal behaviour: a VM exit to the VMM, which can then process the event.

There is also another action that the VMM must perform. When it chooses a VM in the Ready state to make it Running, it must look at the PIR. If there is any bit set, it must inject the ANV when entering the VM. In this way the processor will look at the PIR and process the interrupts that were posted while the VM was not Running.

Note that, in order to setup the IRT, the VMM must know the vector V, but this vector is determined by the guest. The VMM may discover V by intercepting guest writes to the interrupt controller registers.