Virtualization of virtual memory

G. Lettieri

4 Nov. 2015

1 Target vs Host physical addresses

In hardware-assisted virtualization, the physical memory of the target machine is typically implemented using a subset of the host physical memory. The guest system must only have access to the memory subset the VMM has assigned to it, otherwise it would be able to interfere with other VMs or with the VMM itself. At the same time, the guest must think that it has access to all the physical memory that is installed on the target machine, starting from physical address 0. Figure 1 shows the typical setup. To map the target physical memory (or *guest* physical memory) to a subset of the host physical memory we use the host MMU. Assume that the target machine is in a state where its CPU is accessing physical memory at address F. In the corresponding virtual machine state, the CPU is also accessing address F, since in hardware assisted virtualization we assume that (for most of the time), the virtual CPU and the target CPU are executing exactly the same instruction. In the virtual machine state, however, the address is intercepted by the host MMU which will translate it to F', an address that lies inside the portion of physical memory allocated to this VM. Note that, in Figure 1 and in all the remaining Figures, the gray areas represent the *whole* in-memory translation data structure (for 32 bit Intel machines this includes the page directory and all the present pages tables).

Note that this is very similar to what multiprogrammed kernels do with processes. The similarity works because the target machine we are considering has no MMU.

2 Virtual MMU

Now consider a target machine which itself has an MMU. This means that software running on the target machine should be able to prepare and use its own translations from virtual address to physical addresses, and expect them to work. How can we emulate this, and still make sure that the VM will never access host physical memory outside the allocated region?

Assume the guest software has prepared a G mapping that maps *guest* virtual address V to *guest* physical address F. At the same time, the VMM has a mapping H that maps F to host physical address F'. The combined effect is

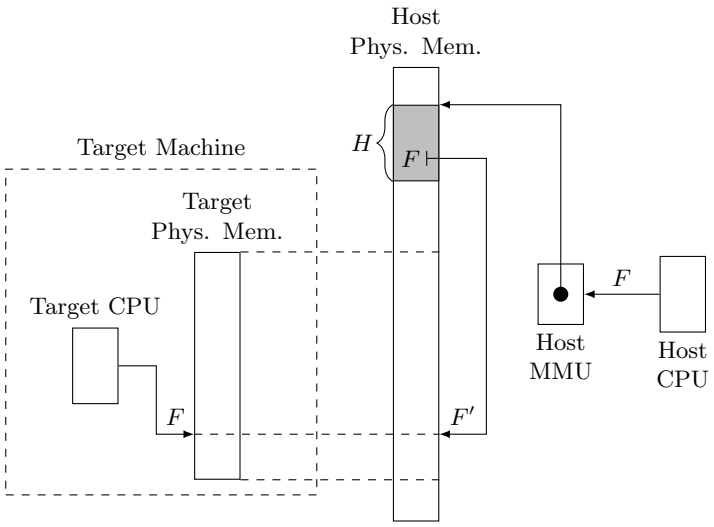

Figure 1: Implementation of the target physical memory using the host MMU for a target machine with no MMU.

that, whenever the guest software tries to access address V, it must actually access address F'. We can obtain this if the VMM is able to build the page tables that implement the $H \circ G$ mapping and let the host MMU always point to them while the guest software is running. Fig. 2 shows the new setup. The target machine has an MMU. The target MMU points to some guest prepared page tables that map V to F. The host MMU does not point to the page tables prepared by the guest software. Instead, it points to some page tables prepared by the VMM and unaccessible to the guest. These host page tables implement the composition of the $V \mapsto F$ and $F \mapsto F'$ mappings.

How can the VMM prepare the host page tables? And what happens if the guest software tries to modify the guest page tables (i.e., the G mapping)? We now study two methods that the VMM may use to keep the host page tables in sync with what the guest is doing.

2.1 The brute force method

With this method we try to update the host page tables as soon as the guest modifies the guest page tables, i.e., the G mapping.

The guest may modify the G mapping in two ways:

1. changing the pointer in the MMU (writing something into `%cr3` on Intel);
2. changing the entries in the page tables in memory (the gray area in the target physical memory in Fig. 2, corresponding to the gray area in the lower part of the host physical memory).

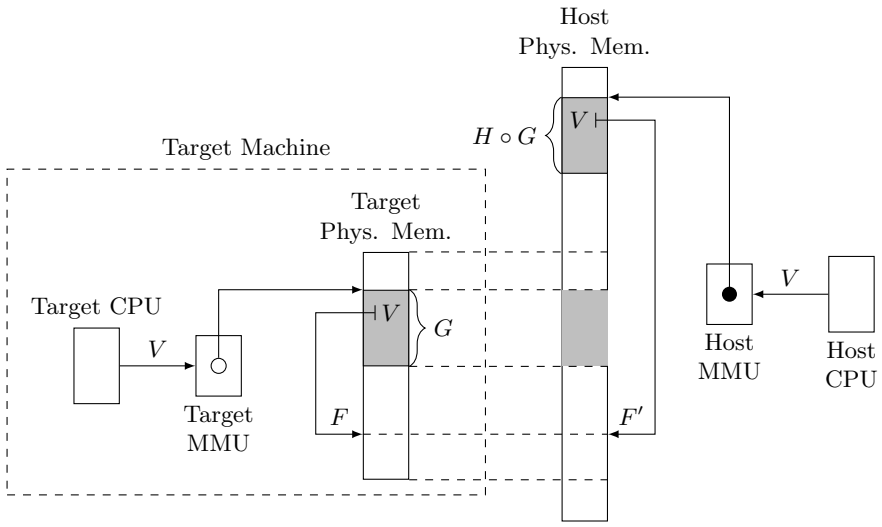

Figure 2: Implementation of the target virtual and physical memory using the host MMU for a target machine with an MMU.

The VMM must setup the VM in order to trap both actions. We assume that we are using Intel VMS and that the guest runs in non-root mode.

For the first kind of action, the VMM may setup the VMCS so that each write to `%cr3` from non-root/system mode causes a VM exit. When the VMM regains control as an effect of the VM exit, it can examine the instruction that the guest was trying to execute. Assume it is `mov %eax, %cr3`. The VMM may read the content of `%eax` and learn the guest physical address of the page directory that the guest was trying to install. Then, it may use the H map to convert this address into an host physical address, and then read the page directory that the guest had prepared. From there, it may learn the guest physical address of all the present page tables, and so on. The VMM uses all this information to prepare the host page directory and page tables, by translating through H all the physical addresses contained in the present page descriptors. Then, it writes into `%cr3` the host physical address of the page directory it has created, it modifies the `%eip` field in the guest state part of the VMCS, in order to skip the guest `movl %eax, %cr3` instruction, and finally re-enters the VM from the new state. The guest will think that it has successfully updated the `%cr3` register.

The guest may performs the second kind of action with any instruction that writes to memory, if the address memory corresponds to some active page directory or page tables. To cause VM exits for these actions (and possibily not for any write to memory, which would be terrible perfomance-wise), the VMM may write-protect, in the host page tables, the pages that contain the active guest page directory and page tables. Then, the VMM sets up the VMCS so that any write to a write-protected page causes a VM exit. In reply to any such

exit, the VMM must examine the address that the guest was trying to write and determine whether the addres falls within any active page directory/table or not. When this is the case, the VMM must decode the instruction that the guest is trying to execute to learn how the guest was trying to update its page directory/tables, and update the host page tables accordingly. Then it must update the guest page directory/table by itself and re-enter the VM skipping the trapped instruction.

2.2 The virtual TLB method

The above method is complex to implement and may cause many unnecessary VM exits. A more interesting method, called virtual TLB, is to update the host page tables lazily. The idea is that the host page tables prepared by the VMM work like the TLB in the MMU: even if the system software updates the page tables in memory, the MMU actually translates the address according to what it finds in the TLB. The TLB is a cache of the page tables: if the translation is not in the TLB, the MMU looks up the translation in the page tables and updates the cache; if the system software changes the page tables, it must also invalidate the TLB (as a whole or just one entry, using the `invlpg` instruction).

Now assume that the VMM sets up the VMCS as follows:

1. as in the brute force method, there must be a VM exit on each write to `%cr3`;
2. unlike the brute force method, the guest page directory/tables are not write protected;
3. also unlike the brute force method, there must be a VM exit on each `invlpg`.
4. finally, there must be a VM exit on each page fault.

Since we are still trapping writes for `%cr3`, the VMM may operate as before in this case. Now assume that the guest writes into some page table. Since we have not write protected the page table, this action will not cause a VM exit. Therefore, the host page tables are now out of sync with the guest page tables. Assume that they were perfectly in sync before the guest change. Lets consider, for the sake of simplicity, only changes that involve the Present bit. Two cases may arise:

1. the guest had made present a guest virtual page that was not present before;
2. the guest had made not-present a guest vitual page that was present before.

In the former case, the host page tables still mark the page as not present. When the guest tries to access any virtual address inside the page, there is a page fault and, therefore, a VM exit. The VMM now examines the guest page

tables and finds out that the page was actually present, and updates the host page tables accordingly. Now it may re-enter the VM from the state contained in the VMCS, so that the instruction that had caused the page fault is retried. Note how this case is similar to a TLB miss, with the host tables playing the role of the TLB.

In the latter case (i.e., a non-present page becomes present), the host page tables still mark the page as present. When the guest tries to access any virtual address inside the page, there is no page fault and no VM exit, even if the page should not be present, according to the guest. This case is similar to what happens in the target machine if the system software does not invalidate the TLB. Therefore, it is a possible behaviour of the target machine, and our virtual machine is free to behave in this way. Of course, a correct gust software will invalidate the TLB after removing some virtual page. Since the VMM is trapping both writes to `%cr3` and the `invlpg` instruction, and these are the only methods available to invalidate the TLB, the VMM regains control on the invalidation and may find out what pages are now no longer present, and update the host page tables accordingly.

Excercise: do the A and D bits in the guest page tables work? If not, what the VMM should do to emulate them?

2.3 Extended Page Tables

Both AMD and Intel have added some extensions to their hardware support for virtualization, in order to simplify the task of virtualizing guest virtual memory. The support comes in the form of *extended page tables*. As is usually the case for this kind of hardware extensions, this is available for the host, but is not necessarily part of the target machine. The idea, illustrated in Fig. 3, is that the host MMU now holds two pointers: one to the guest page tables, and another one to the host page tables. The guest page tables are manipulated by the guest, with no intervention from the VMM; they contain the G mapping. The host page tables are created by the VMM, as before, but they only contain the H mapping. The composition of the G mapping with the H mapping is performed in hardware by the host MMU. At each memory access, the host MMU first performs the G translation, obtaining the guest physical address, then it translates the guest physical address using the H mapping, obtaining the host physical address.

Note that, by using extended page tables, there is no longer any need for VM exits. However, address translation becomes much more expensive. We must be aware, in fact, that each address used during the translation according to G is a *guest* physical address, and it must itself be translated according to H to obtain the corresponding host physical address. Let us count the number of memory accesses needed to translate V to F':

- First, there must be an access to an entry in the guest page directory. To obtain the host physical address of this entry, the host MMU must traverse the host page tables (the H mapping). This adds 2 memory accesses, plus

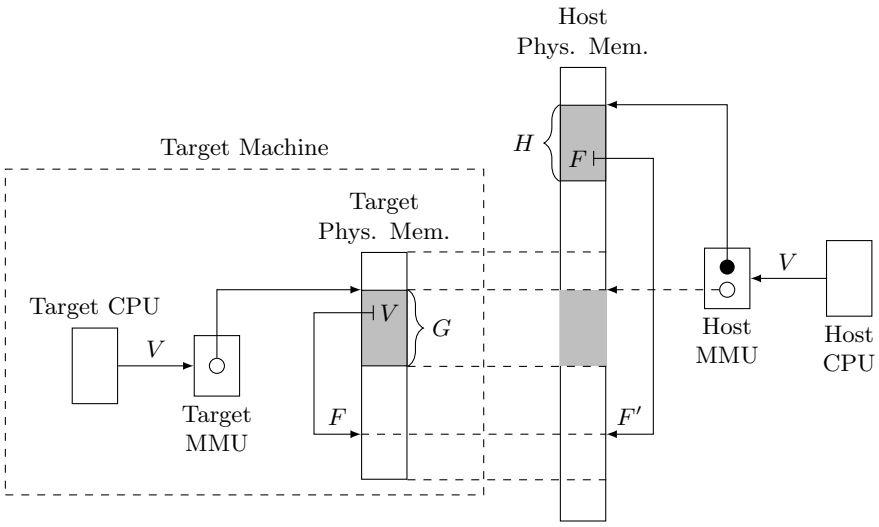

Figure 3: Implementation of the target virtual and physical memory using extended page tables. The hollow dot in the Host MMU is a guest physical address (the same address stored in the Target MMU). The dashed arrow points to the host physical address obtained after the translation through H.

the access to the guest page directory;

- Then, there must be an access to an entry in the guest page table pointed to by the guest page directory entry read at the previous step. To obtain the host physical address of this entry, the host MMU must again traverse the host page tables (the H mapping). This adds another 2 memory accesses, plus the access to the guest page table;

Therefore, for each access to guest virtual memory, the host MMU must perform 6 additional memory accesses for the translation. The situation is much worse for 64 bit CPUs: there, there are four levels of page tables instead of just 2, and the translation needs $(4 + 1) \times 4 = 20$ memory accesses to be completed. Of course, the MMU is equipped with several TLBs to try to avoid most of these accesses.