Hardware Assisted Virtualization

G. Lettieri

21 Oct. 2015

1 Introduction

In the *hardware-assisted* virtualization technique we try to execute the instructions of the target machine directly on the host processor, as much as possible. If we are able to execute a large fraction of the target machine instructions in this way, we clearly obtain a large speedup w.r.t. the other techniques we have already examined.

Hardware-assisted virtualization is only possible if the host machine understands a superset of the target machine instructions. In some cases, the target and and host machine have exactly the same architecture. As we have seen, this may make perfect sense when the motivations for the use of virtualization are related to cost, flexibility and security, rather than on the unavailability of the target machine hardware. In most cases, however, the two architectures are not exactly the same. To understand why, let us now consider some well known examples from computer architecture.

1.1 The virtual memory example

In *virtual memory* the target machine is almost, but not exactly the same as the host machine we already have. Of course, the two machines may differ in the amount of installed memory, but this is not the only difference. Another difference between the host and target machines lies in the hardware (and maybe software) that is unique to the host and that is needed to build the virtual machine that emulates the target. For example, the host machine typically has an MMU (Memory Management Unit), but the target machine has no such thing.

To focus these ideas, let us consider a simple example, again based on the Manchester Baby machine. The machine has only 32 words of memory, but the *addr* field in the instruction format is 13 bits wide, allowing for a maximum of $2^{13} = 8196$ addressable words. Let us imagine that we are in the '50s and we want a machine that is exactly the same as the Manchester Baby, only with a memory of 8196 words. We may build other 255 tube memory devices, plus all the logic to access them, but this is going to be very expensive, so we take another route. We regard the 8192-words Manchester Baby as our target

Figure 1: The magnetic drum memory of the CEP, 1961. It has a capacity of 16384 36-bit words. (credit: www.cep.cnr.it).

machine and we create a virtual machine that emulates it on the standard, 32-words Manchester Baby, with the addition of a *magnetic drum memory*. This is a relatively less expensive, but also much slower, memory than the Williams-Kilburn tubes (see Fig. 1 for an example). The idea is to store all the memory of the target machine in the inexpensive drum. We organize this memory into 256 (i.e., 8192/32) *pages*, each one consisting of 32 consecutive words. At any time, only one page is available in the 32-words tube memory of the Manchester Baby. Our virtual machine must swap pages between the tube and the drum as needed, in order to emulate the bigger memory of the target machine. To implement swapping, we choose to add another piece of hardware to the host Baby: an MMU, intercepting all accesses to main memory. Our MMU must contain an 8 bit register, $\mathcal{P}$, that records which one of the possible 256 pages is currently stored in the tube. Whenever a memory operation is initiated at an address a (on 13 bits), the MMU must compare the most significant 8 bits of a with the contents of the $\mathcal{P}$ register: in case of match, the requested word is in the tube at address a mod 32; otherwise, a swap must be performed: the

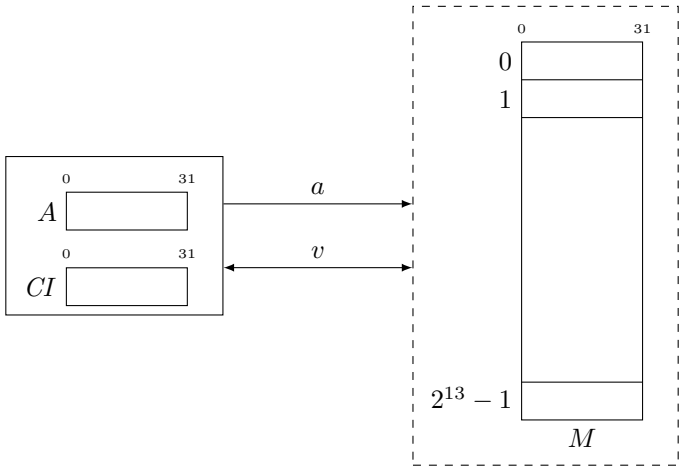

Figure 2: The target machine.

current content of the tube must be copied on the drum, the requested page must be copied from the drum to the tube, $\mathcal{P}$ must be updated accordingly. Now $\mathcal{P}$ matches the 8 most significant bits of a, and the memory access can be completed.

The drum and the MMU are pieces of hardware that we have added in order to implement the virtual machine, but are not part of the target machine that our virtual machine emulates. The target machine has no drum and no MMU, it is just a Manchester Baby machine with a bigger tube memory.

Let us try to formalize what we have done, using the framework we introduced in the first lecture. We need to define the states together with the state transition function for both the target and the virtual machine, and then we need to define the function that maps each virtual state to the corresponding target state. Then, we should check that interpretation is preserved at every step.

We take a snapshots just before the fetch of a new instruction, like we did in the first lecture. The target state contains the state of the accumulator, A, the instruction pointer, CI, and the state of the whole 8192 words memory, which we can regard as a vector M, with elements $M_0, \ldots, M_{8191}$:

$$T\text{-}state = \langle A, CI, M \rangle.$$

Also see Fig. 2, where we have also shown the address, a, coming from the control unit and going to the memory, and the data, v, returned by the memory during a read operation, or provided by the control or arithmetic unit during a write operation. The virtual states need to also show the current contents of the drum and the $\mathcal{P}$ register of the MMU. Memory is a vector $\mathcal{M}$ of just 32 words, $\mathcal{M}_0 \ldots \mathcal{M}_{31}$. For the drum, we assume that we can regard it as a vector

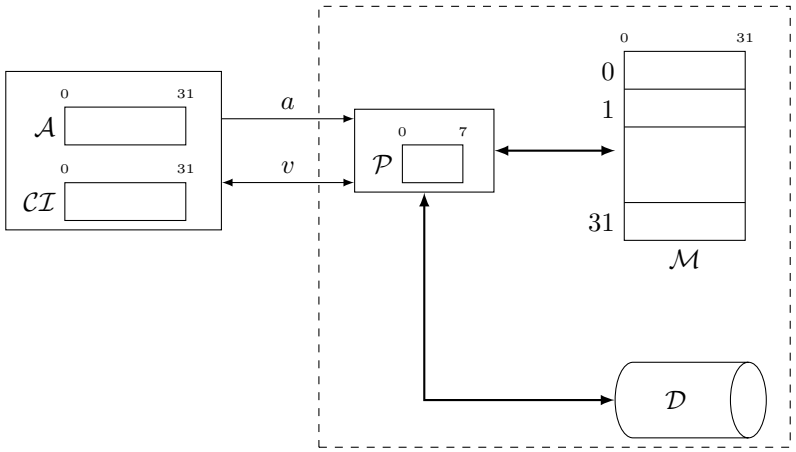

Figure 3: The virtual machine.

$\mathcal{D}$ of 8192 words, $\mathcal{D}_0, \ldots, \mathcal{D}_{8191}$:

$$V\text{-}state = \langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D} \rangle.$$

Fig. 3 shows the architecture of the virtual machine.

Now we need an *interp* functions that maps a *V-state* to a *T-state*. First, let us define a $interp_M$ functions that just maps the memory subsystem part. If we are given $\mathcal{M}$, $\mathcal{P}$ and $\mathcal{D}$, then $X = interp_M(\mathcal{M}, \mathcal{P}, \mathcal{D})$ is the 8192-elements vector that represents the contents of the corresponding target memory. The elements of X are defined as follows:

$$X_a = \begin{cases} \mathcal{M}_{a \bmod 32} & \text{if } \mathcal{P} = \lfloor a/32 \rfloor, \\ \mathcal{D}_a & \text{otherwise,} \end{cases} \tag{1}$$

for all $0 \leq a < 8192$. Each word of the target machine is stored in the main memory, if the word is inside the page that is currently loaded; otherwise, the word is stored in the drum. The complete *interp* function can be defined as

$$interp(\langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D} \rangle) = \langle \mathcal{A}, \mathcal{CI}, interp_M(\mathcal{M}, \mathcal{P}, \mathcal{D}) \rangle,$$

i.e., the accumulator and instruction pointer are interpreted as themselves, while the memory is interpreted as above.

We omit the definitions of the *T-next* and *V-next* functions. Intuitively, *T-next*(s), where s is any *T-state*, must return the state of the target machine after the execution of the instruction pointed to by CI in s. The same is true for *V-next*(s). Note that memory accesses in the virtual machine are as follows, where a is the requested memory address and $\bar{a} = \lfloor a/32 \rfloor$:

1. if $\bar{a} = \mathcal{P}$, then the page currently loaded in the tube memory is the one containing the requested word. In this case there is an access to $\mathcal{M}_{a \bmod 32}$ (either for reading or writing) and the operation completes.

2. if $\bar{a} \neq \mathcal{P}$, the requested page is on the drum. The current contents of the tube memory are copied in drum locations $32\mathcal{P}, \ldots, 32\mathcal{P}+31$ while drum locations $32\bar{a}, \ldots, 32\bar{a}+31$ are copied into the tube memory. The $\mathcal{P}$ register is updated with the $\bar{a}$ value and the access is completed like in step 1 above.

We can prove the following (we give only informal proofs, since we have not given the formal definition of *T-next* and *V-next*).

Lemma 1. *Let M be the state of the target machine memory and $\mathcal{M}$, $\mathcal{P}$, $\mathcal{D}$ the states of the virtual machine memory, MMU and drum, respectively. Assume that*

$$M = interp_M(\mathcal{M}, \mathcal{P}, \mathcal{D}) \qquad (*)$$

and we start a read operation at address a in both memories. Then we obtain the same value v from both. Moreover, if M', $\mathcal{M}'$, $\mathcal{P}'$ and $\mathcal{D}'$ are the new states after the completion of the operation, it still holds that $M' = interp_M(\mathcal{M}', \mathcal{P}', \mathcal{D}')$.

Proof. Note that the read operation on the target machine memory returns M_a and the memory is not changed, i.e., $M' = M$. For the virtual machine memory we must consider two cases, where $\bar{a} = \lfloor a/32 \rfloor$:

1. if $\bar{a} = \mathcal{P}$, the virtual machine memory returns $\mathcal{M}_{a \bmod 32}$. Since we are assuming (*), this is equal to M_a by definition (1). Moreover, no state is changed, and therefore the interpreting relation is preserved.

2. If $\bar{a} \neq \mathcal{P}$ the virtual machine memory will perform a swap, as explained above. At the end of the swap, the new state is

$$\begin{aligned}
\mathcal{M}'_0 &= \mathcal{D}_{32\bar{a}}, \\
&\vdots \\
\mathcal{M}'_{31} &= \mathcal{D}_{32\bar{a}+31}, \\
\mathcal{P}' &= \bar{a}, \\
\mathcal{D}'_{32\mathcal{P}} &= \mathcal{M}_0, \\
&\vdots \\
\mathcal{D}'_{32\mathcal{P}+31} &= \mathcal{M}_{31}, \\
\mathcal{D}'_i &= \mathcal{D}_i \qquad \text{for } 0 \leq i < 32\mathcal{P} \text{ or } i \geq 32(\mathcal{P}+1).
\end{aligned}$$

   The read operation will return the word

$$\mathcal{M}'_{a \bmod 32} = \mathcal{D}_{32\bar{a}+a \bmod 32} = \mathcal{D}_{32\lfloor a/32 \rfloor + a \bmod 32} = \mathcal{D}_a,$$

   but this is equal to M_a by (*) and (1). Note that this time the state of virtual machine memory has changed, so we have to prove that $M' = interp_M(\mathcal{M}', \mathcal{P}', \mathcal{D}')$ holds. This can be easily checked by using the equations above, hypothesis (*) and definition (1).

□

Lemma 2. *Let M be the state of the target machine memory and $\mathcal{M}$, $\mathcal{P}$, $\mathcal{D}$ the states of the virtual machine memory, MMU and drum, respectively. Assume that $M = interp_M(\mathcal{M}, \mathcal{P}, \mathcal{D})$ and we start a write operation of word v at address a in both memories. If M', $\mathcal{M}'$, $\mathcal{P}'$ and $\mathcal{D}'$ are the new states after the completion of the operation, it still holds that $M' = interp_M(\mathcal{M}', \mathcal{P}', \mathcal{D}')$.*

Proof. Exercise. □

With these two lemmas we can prove the following.

Theorem 1. *Let s be any T-state and t a V-state such that $s = interp(t)$. Let $s' = T\text{-}next(s)$ and $t' = V\text{-}next(t)$. Then $s' = interp(t')$.*

Proof. Starting from s, the target machine will try to fetch the instruction at address CI. The virtual machine, starting from t, will try to fetch the instruction at $\mathcal{CI}$. Since $s = interp(t)$, we know that $CI = \mathcal{CI}$ and, by Lemma 1, we know that both machines will read the same instruction. Moreover, by the same Lemma, the two memories will be left in an equivalent state. After fetching the instruction, the two machines will start to decode and execute them. Since their control and arithmetic unit is exactly the same (the two machines only differ in their memory subsystem), they will perform exactly the same actions until they will need to either read or write from memory. They will present their memories with exactly the same address and, in case of a write operation, also the same value to be written. Now we can apply either Lemma 2 or 1 again and see that the two machines will go into equivalent states even after this new memory operation. The two (identical) control and arithmetic units will again perform exactly the same actions until the point where they will need to fetch another instruction. It is clear that the new states at this point, t' and s', will satisfy $s' = interp(t')$. □

Why is this technique called *hardware assisted* virtualization? In this example we can see that we need help from the hardware to maintain the correspondence between the virtual machine state and the target state. Our problem, here, is that the host hardware is not exactly the same as the target one: we have a drum memory instead of the bigger tube memory. The target-machine programs will try to access the bigger tube memory, and have no knowledge of the drum. In emulation and binary translation we have the opportunity to change the target-machine program instructions before executing them (either when we interpret them or when we translate them). But now we are running the *unmodified* target-machine programs directly on the host hardware, so we need assistance from the host hardware itself to hide all the differences. In this example, it is the MMU (a part of the host hardware) that translates all the memory accesses generated by the target machine programs, so that they have the same effect as if they were completed on the target machine. In our formalization, the MMU takes care of updating that part of the *V-state* that differs from the *T-state*, so that the *V-state* continues to be equivalent to the *T-state*.

Note that, in this simple example, we have assumed that all the virtual machine logic can be implemented in the MMU. This is feasible since the Baby has just one physical page, so there is no need for complex data structures and algorithms. In a more realistic example, part of the virtual machine logic would be implemented in software: typically, the MMU only translates addresses for the pages that are loaded in memory, and raises an exception for all the other ones. The exception causes the execution of virtual machine software that, in this cases, implements the swapping. Note that, even when part of the virtual machine is software based, we still need help from the hardware: it is the hardware that must raise the exception that triggers the execution of the software module. Keep in mind that we always need to maintain the correspondence between the *T-state* and *V-state*. The hardware may be able to perform some of the necessary translation by itself; if it cannot, it must stop the execution and invoke the software.

1.2 The virtual processor example (multiprogramming)

In *multiprogramming* we emulate a target machine which has a greater number of processors than the host machine. Each *process* runs on its own *virtual processor*, and virtual processors are multiplexed on the host machine *physical* processors (*host* processors from now on). Let us assume that the target machine uses a shared memory model, otherwise we would have to virtualize the private memory of each target process as well.

In the most simple case, the host machine has just one host processor. Here, the *T-state* contains the registers of all virtual processors. The *V-state* state contains the registers of the host processor and a set of data structures in the host memory, including:

- one data structure for each virtual processor, containing a copy of the virtual processors registers;
- one variable containing the identifier of the virtual processor that is currently running on the host processor.

Our virtual machine emulates one target processor at a time. This is obtained by loading the registers of the host processor with the values stored in the virtual processor data structure and then letting the host processor continue the execution. At some later time (maybe determined by a timer), execution is paused, the virtual processor data structure is updated with the current contents of the host registers, a new virtual processor is selected for execution and so on.

A complete definition of *T-state*, *V-state* and *interp* for a simple processor is left as an exercise.

Why do we need hardware assistance to implement this virtual machine? Much like the virtual memory example, part of the virtual machine logic may be implemented in software. This is typically the case for the "context switching", i.e., loading and unloading the host processor registers. However, we still need help from the hardware to force the invocation of this software, e.g., with a

timer based interrupt. But there is also another kind of help that we need: the data structures that store the contents of the virtual processors must only be accessible to the software that implements the virtual machine, and not to the (target) software running on the virtual processors. Otherwise, one virtual processor would be able to read and write into the registers of another virtual processor, something that is not possible in the target machine. The standard solution is to introduce *privilege levels* in the host processor. The processor may run in one of at least two different privilege levels: in the "system" level it has access to all the memory, in the "user" level it has access to only a memory subset that does not include the virtual processors data structures. The timer interrupt switches the host processor to system level and causes a jump to the virtual machine software that performs the context switch. One special instruction, only available at system level, then causes the return to user mode. If there is an attempt to access the privileged memory while the processor is in user mode, the accesses is denied by the hardware.

2 Virtual machines

When we talk of virtual *machines*, we want to virtualize a complete computing system, composed of full processors, memory and I/O peripherals. In the virtual memory and virtual processors examples we have virtualized only a part of the complete system. In particular, the virtual processor (multiprogramming) example only virtualizes *part* of a full processor—namely, only the userspace visible part. In a virtual machine we want to emulate the full processor, since we want to run all the software that runs in the target machine: this includes the operating system software that, in the target machine, has access to the privileged registers and instructions of the processor. For the Intel x86 processor these would be the `%cr3` register (pointer to the page directory), the `%idtr` register (pointer to the interrupt descriptor table), ..., plus the instructions that manipulate them. It should be clear that we cannot directly execute any instruction of the target-machine programs that modifies any of these registers, since modifying them affects the state of the entire host machine. Each virtual machine will have its own virtual copy of these registers, and it is these copies that should be updated, not the host registers. At the same time, we want to execute target-machine instructions directly on the host hardware, as much as possible. Again, we need help from the hardware to handle these problematic instructions when they show up in the stream of target instructions.

2.1 Trap-and-emulate

A special case of hardware assisted virtualization is when the target and host architecture is exactly the same, i.e., we emulate the full processor by reusing the hardware mechanisms that are already available, and nothing else. The idea is to use the two levels of privilege, user and system, and reserve the system level for the virtual machine monitor (the software that implements the virtual machine)

and the user level for all the software running inside the virtual machine (i.e., the software originally written for the target machine). The effect of this is that the target-machine system software will run at user privilege. This scheme may work if our processor raises an exception whenever a privileged instruction is used while running at user level: the virtual machine monitor may intercept the fault and emulate the effect of the privileged instruction on the virtual state. Non privileged operations, which typically are a large fraction of the stream of instructions, may be executed directly. This is called a "trap-and-emulate" virtual machine monitor.

The trap-and-emulate virtual machine monitor can be implemented on some architectures but, unfortunately, not on the (original) Intel x86 processors. The problem is that some privileged operations do not raise an exception when executed at user level. One example is the `popf` instruction. This instruction pops one double-word from the stack and stores it in the `EFLAGS` register. This is a privileged operation since the `EFLAGS` register contains the `IF` flag, that, when zeroed, disables the external interrupts on the processor. Assume that the target system software, running inside our virtual machine, tries to execute this instruction. We would like to intercept the instruction so that we can disable the "virtual interrupts" of the virtual machine, and (of course) not the real interrupts of the host machine. Unfortunately, the x86 processor does not raise an exception in this case, but it simply does not updated the `IF` flag when the `popf` instruction is run at user level. Therefore, we obtain only a part of what we need: the host interrupts are not disabled, but also the virtual interrupts are not disabled, since the virtual machine monitor has no way to know that the system software was trying to disable the target machine interrupts. The former part is good, but the latter part is bad, since our *V-state* no longer matches the *T-state*: the target machine has disabled its interrupts, but our virtual machine has not. Other problems arise when the target system software tries to access the privileged registers, such as `%cr3`: writes from userspace cause an exception, but reads are allowed. Therefore, the software running inside the virtual machine may detect that it is not running on the target machine, by comparing what it writes with what it reads from `%cr3`: on the target machine the two things would match, but on the virtual machine they would most certainly not (since `%cr3` contains the pointer to the page directory used by the host, not the one used by the virtual machine).

VMware solved these (and many other) problems using hardware-assisted virtualization for all the target userspace software, and switching to binary translation for the target system software. Today, both AMD and Intel have added virtualization extensions to their processors, in order to allow for efficient hardware assisted virtualization implementations.