Hardware Assisted Virtualization: Introduction

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 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 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. Figure 1 shows the target machine that we want to emulate. In the most simple case, the host machine has just one host processor, and this is the example we are going to focus on.

Here, the *T-state* contains the state of the shared memory and the registers of all virtual processors. The *V-state* contains the state of the shared memory, 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.

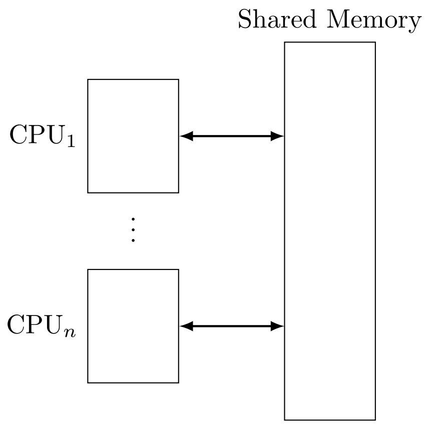

Figure 1: The target machine for the multiprogramming example.

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.

Why do we need hardware assistance to implement this virtual machine? Part of the virtual machine logic is typically implemented in software. This is 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 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.

In Figure 2 we have shown an example host machine which can be used to implement the virtual machine sketched above. The host CPU is similar to the CPUs of the target machine, but it also has additional features. There is a sys/usr flag that records whether the CPU is running in *system* or *user* mode.

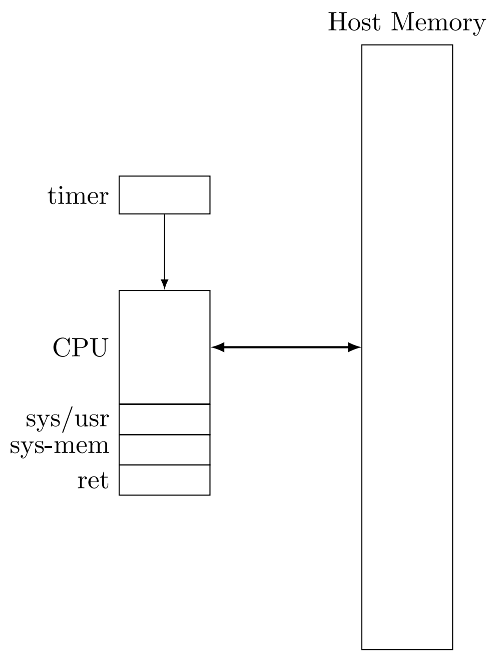

Figure 2: The host machine for the multiprogramming example.

There is also a "sys-mem" register that contains a memory address. When in user mode, the CPU can only access the host memory at addresses below the sys-mem address. In system mode there is no such restriction. Finally, there is a "ret" register that can also store a memory address. The contents of the sys/usr flag and of the sys-mem and ret registers can only be changed while the CPU is running in system mode. The host machine also contains a timer device, which can send a periodic interrupt to the CPU. When the CPU receives an interrupt it disables further interrupts, switches to system mode, saves the current instruction pointer in the ret register and jumps at the address stored in the sys-mem register. A special instruction can be used to jump back to the address stored in the ret register, returning to user mode and re-enabling the interrupts.

Figure 3 shows how the host machine of Figure 3 can be used to emulate the target machine of Figure 1. The sys-mem register points to the address of the "system code" that, whenever the timer triggers:

- saves the state of the CPU into the current virtual-cpu slot (the contents of the instruction pointer are taken from the ret register);
- selects a new virtual cpu to run (also updating the "current" variable);
- loads the state of the selected virtual cpu into the host CPU (the contents of the virtual instruction pointer are stored in the ret register);

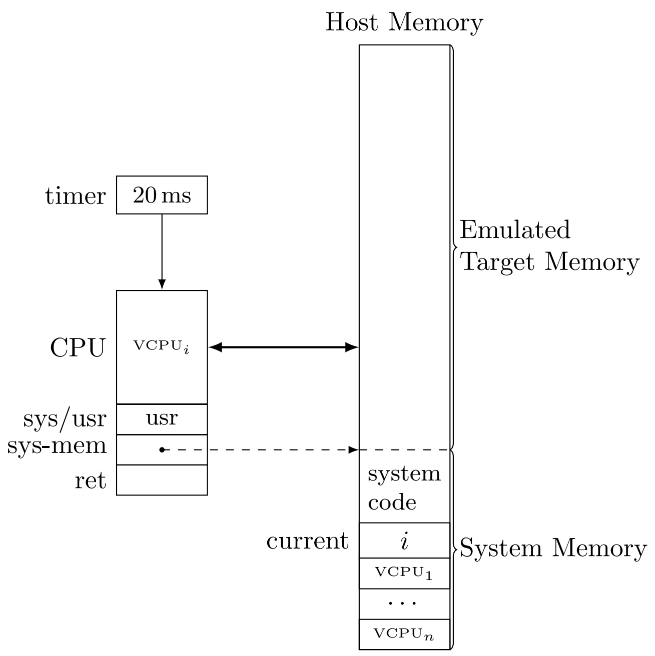

Figure 3: The virtual machine for the multiprogramming example.

- executes the special instruction that jumps to the address stored in ret, returning into user mode.

Note how the host machine needs additional hardware capabilities that are not available in the target machine. The additional host hardware (the sys/usr, sys-mem and ret register, the privilege level checks, the timer, ...) *assists* the software in the implementation of the virtual machine. Note also that, unlike the emulation and binary-translation software, the hardware-assisted virtual machine software (which in this example coincides with multiprogrammed kernel) is *privileged*, since it must have access to the additional hardware.

1.2 The virtual memory example

Also 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

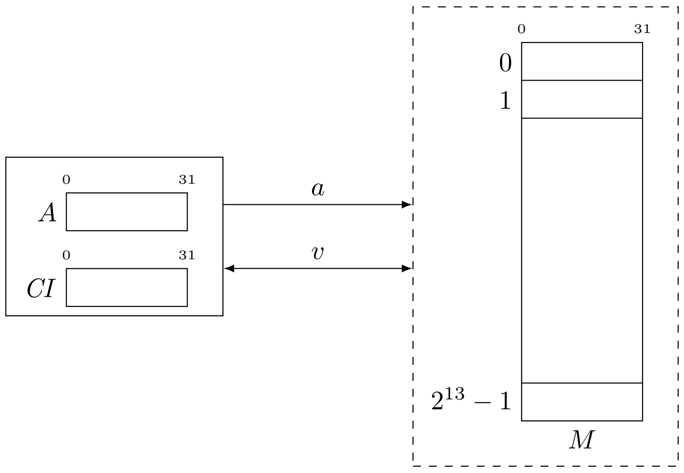

Figure 4: The target machine.

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 $2^{13} = 8192$ 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 8192 words. See Fig. 4, 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. To use this machine 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. Instead, we regard the machine of Figure 4 as our target 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. 5 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.

Fig. 6 shows the architecture of the virtual machine. Whenever a memory operation is initiated at an address a (on 13 bits), the MMU must compare the most significant 8 bits of a (i.e., the number $\lfloor a/32 \rfloor$) with the contents of the $\mathcal{P}$ register: in case of match, the requested word is in the tube at address

Figure 5: The magnetic drum memory of the CEP, 1961. It has a capacity of 16384 36-bit words. (credit: www.cep.cnr.it).

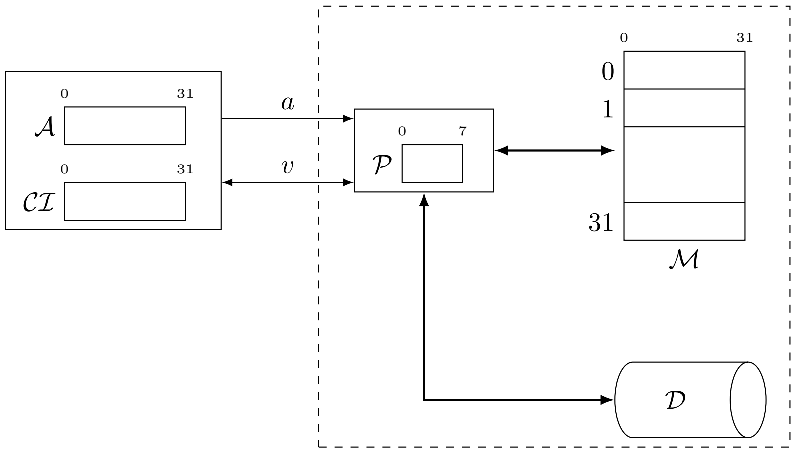

Figure 6: The virtual machine.

$a \bmod 32$; otherwise, a swap must be performed: the 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 $\lfloor a/32 \rfloor$, 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 (Figure 4) has no drum and no MMU, it is just a Manchester Baby machine with a bigger tube memory.

Section 2 contains a complete formalization of the example, using the framework introduced in the first lecture, that shows that the virtual machine implementation is correct, i.e., that it preserves the correspondence between the target and virtual states.

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: where the program wants to read or write any location of memory, the corresponding instruction will just contain the desired address. In the virtual machine, however, we cannot use this address as is: we must inspect it, to see whether it belongs to the page currently in the tube and, if not, a swap must be performed. 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 this case the virtual and host machines coincide. 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. In this more realistic case we can identify an host machine which is distinct from the virtual machine: the host machine can potentially run any software and make different uses of the MMU, or even not use it at all. The virtual machine is the host machine plus the software that implements the virtual memory. 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.

2 Appendix

Let us try to formalize the Manchester Baby virtual memory example, 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.

2.1 The target machine

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 all the 8192 words memory, which we can regard as a vector M, with elements $M_0, \ldots, M_{8191}$:

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

The definition of the $T\text{-}next\colon T\text{-}state \to T\text{-}state$ function is as follows:

$$T\text{-}next(\langle A, CI, M \rangle) = \begin{cases} \langle A, M_a + 1, M \rangle & \text{if } o = 0; \\ \langle A, CI + M_a + 1, M \rangle & \text{if } o = 1; \\ \langle -M_a, CI + 1, M \rangle & \text{if } o = 2; \\ \langle A, CI + 1, M\{A/a\} \rangle & \text{if } o = 3; \\ \langle A - M_a, CI + 1, M \rangle & \text{if } o = 4 \text{ or } o = 5; \\ \langle A, CI + 2, M \rangle & \text{if } o = 6 \text{ and } A = 0; \\ \langle A, CI + 1, M \rangle & \text{if } o = 6 \text{ and } A \neq 0; \\ \langle A, CI, M \rangle & \text{if } o = 7; \end{cases} \tag{1}$$

where:

$$\begin{aligned} o &= (M_{CI} >> 13)\,\&\,\texttt{0x7}, \\ a &= M_{CI}\,\&\,\texttt{0x1FFF}. \end{aligned}$$

Intuitively, $T\text{-}next(s)$, where s is any $T\text{-}state$, must return the state of the target machine after the execution of the instruction pointed to by CI in s.

In the definition we have used the following notation: if X is any vector, then $X\{v/a\}$ is another vector with exactly the same elements as X, except for element number a, which is v:

$$X\{v/a\}_b = \begin{cases} X_b & \text{if } b \neq a, \\ v & \text{if } b = a. \end{cases} \tag{2}$$

The notation has been used in the case for $o = 3$, to represent the state of the target memory after the store operation has been completed.

2.2 The virtual machine

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, $\mathcal{M}_0 \dots \mathcal{M}_{31}$. For the drum, we assume that we can regard it as a vector $\mathcal{D}$ of 8192 words, $\mathcal{D}_0, \dots, \mathcal{D}_{8191}$:

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

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. We say that all the addresses with $\lfloor a/32 \rfloor = \mathcal{P}$ are *accessible* in the VM memory. Note that an accessible address a is stored in location $a \bmod 32$ of the tube memory.

To make an address a accessible we can *swap-in* the page that contains it, swapping out the current page from main memory. We can model this operation with a function $swap_a$ that returns the state of the VM memory such that a given address $0 \leq a < 8192$ is accessible:

$$swap_a(\langle \mathcal{M}, \mathcal{P}, \mathcal{D} \rangle) = \begin{cases} \langle \mathcal{M}, \mathcal{P}, \mathcal{D} \rangle & \text{if } \mathcal{P} = \lfloor a/32 \rfloor, \\ \Big\langle \big(\mathcal{D}_{32\lfloor a/32 \rfloor}, \dots, \mathcal{D}_{32\lfloor a/32 \rfloor + 31}\big), \lfloor a/32 \rfloor, & \text{otherwise.} \\ \quad \mathcal{D}\{\mathcal{M}_0/32\mathcal{P}\} \cdots \{\mathcal{M}_{31}/32\mathcal{P}+31\} \Big\rangle & \end{cases} \tag{3}$$

If address a is not accessible, the current contents of the tube memory are copied in drum locations $32\mathcal{P}, \dots, 32\mathcal{P}+31$ while drum locations $32\lfloor a/32 \rfloor, \dots, 32\lfloor a/32 \rfloor + 31$ are copied into the tube memory. Note that the new $\mathcal{P}$ resulting from an application of $swap_a$ is always equal to $\lfloor a/32 \rfloor$.

The $V\text{-}next\colon V\text{-}state \rightarrow V\text{-}state$ function is defined as follows:

$$V\text{-}next(\langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D} \rangle) =
\begin{cases}
\langle \mathcal{A}, \mathcal{M}''_{a \bmod 32} + 1, \mathcal{M}'', \mathcal{P}'', \mathcal{D}'' \rangle & \text{if } o = 0;\\
\langle \mathcal{A}, \mathcal{CI} + \mathcal{M}''_{a \bmod 32} + 1, \mathcal{M}'', \mathcal{P}'', \mathcal{D}'' \rangle & \text{if } o = 1;\\
\langle -\mathcal{M}''_{a \bmod 32}, \mathcal{CI} + 1, \mathcal{M}'', \mathcal{P}'', \mathcal{D}'' \rangle & \text{if } o = 2;\\
\langle \mathcal{A}, \mathcal{CI} + 1, \mathcal{M}''\{\mathcal{A}/a \bmod 32\}, \mathcal{P}'', \mathcal{D}'' \rangle & \text{if } o = 3;\\
\langle \mathcal{A} - \mathcal{M}''_{a \bmod 32}, \mathcal{CI} + 1, \mathcal{M}'', \mathcal{P}'', \mathcal{D}'' \rangle & \text{if } o = 4 \text{ or } o = 5;\\
\langle \mathcal{A}, \mathcal{CI} + 2, \mathcal{M}', \mathcal{P}', \mathcal{D}' \rangle & \text{if } o = 6 \text{ and } \mathcal{A} = 0;\\
\langle \mathcal{A}, \mathcal{CI} + 1, \mathcal{M}', \mathcal{P}', \mathcal{D}' \rangle & \text{if } o = 6 \text{ and } \mathcal{A} \neq 0;\\
\langle \mathcal{A}, \mathcal{CI}, \mathcal{M}', \mathcal{P}', \mathcal{D}' \rangle & \text{if } o = 7;
\end{cases} \tag{4}$$

where:

$$\begin{aligned}
\langle \mathcal{M}', \mathcal{P}', \mathcal{D}' \rangle &= swap_{\mathcal{CI}}(\langle \mathcal{M}, \mathcal{P}, \mathcal{D} \rangle),\\
a &= \mathcal{M}'_{\mathcal{CI} \bmod 32} \,\&\, \texttt{0x1FFF},\\
o &= (\mathcal{M}'_{\mathcal{CI} \bmod 32} >> 13) \,\&\, \texttt{0x7},\\
\langle \mathcal{M}'', \mathcal{P}'', \mathcal{D}'' \rangle &= swap_a(\langle \mathcal{M}', \mathcal{P}', \mathcal{D}' \rangle).
\end{aligned}$$

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. A swap is performed and the value and the access is completed like in step 1 above.

Note also that a single step of $V\text{-}next$ may cause two swaps: one to bring in the page containing the next instruction, and another one to bring in the page containing the operand.

2.3 Interpretation of the virtual state

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

$$m\text{-}interp(\langle \mathcal{M}, \mathcal{P}, \mathcal{D} \rangle)_a = \begin{cases} \mathcal{M}_{a \bmod 32} & \text{if } \mathcal{P} = \lfloor a/32 \rfloor,\\ \mathcal{D}_a & \text{otherwise,} \end{cases} \tag{5}$$

for all $0 \leq a < 8192$.

The complete *interp* function can be defined as

$$\textit{interp}\big(\langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big) = \Big\langle \mathcal{A}, \mathcal{CI}, \textit{m-interp}\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)\Big\rangle, \tag{6}$$

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

2.4 Preservation of the interpretation

We now give a complete proof of the correctness of the Manchester Baby virtual-memory implementation.

First, we need a few Lemmas. Provided that the address is accessible in the VM, the word at address a of the target memory can be read at address $a \bmod 32$ in a corresponding VM memory.

Lemma 1. *Let M be a target memory, $\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle$ a VM memory and $0 \leq a < 8192$. If $M = \textit{m-interp}\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)$ and $\mathcal{P} = \lfloor a/32 \rfloor$, then*

$$M_a = \mathcal{M}_{a \bmod 32}.$$

Proof. By hypotesis, $M_a = \textit{m-interp}\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)_a$. But $\textit{m-interp}\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)_a = \mathcal{M}_{a \bmod 32}$ by definition 5, since $\mathcal{P} = \lfloor a/32 \rfloor$ by hypothesis. □

Similarly, writing the same value at address $a \bmod 32$ of the VM memory and at address a of the target memory preserves interpretation, provided that a is accessible.

Lemma 2. *Let M be a target memory, $\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle$ a VM memory, $0 \leq a < 8192$ an address and v a value. If $M = \textit{m-interp}\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)$ and $\mathcal{P} = \lfloor a/32 \rfloor$, then*

$$M\{v/a\} = \textit{m-interp}\big(\langle \mathcal{M}\{v/a \bmod 32\}, \mathcal{P}, \mathcal{D}\rangle\big).$$

Proof. To show that the two memories are equal, we need to show that they contain the same value at all addresses. First, note that

$$\textit{m-interp}\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)_b = \textit{m-interp}\big(\langle \mathcal{M}\{v/a \bmod 32\}, \mathcal{P}, \mathcal{D}\rangle\big)_b$$

for all addresses $b \neq a$. Therefore, we only need to check the value at a, for which we have $\mathcal{M}\{v/a\}_a = v$ on one hand, and

$$\textit{m-interp}\big(\langle \mathcal{M}\{v/a \bmod 32\}, \mathcal{P}, \mathcal{D}\rangle\big)_a = \mathcal{M}\{v/a \bmod 32\}_{a \bmod 32} = v$$

on the other hand, since $\mathcal{P} = \lfloor a/32 \rfloor$ by hypothesis. □

Swapping can be performed at any time, since it does not change the interpretation of the VM memory subsystem:

Lemma 3. *Assume that* $M = \textit{m-interp}\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)$. *Then*

$$M = \textit{m-interp}\Big(\textit{swap}_a\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)\Big)$$

for all $0 \leq a < 8129$.

Proof. This is obvious whenever $\lfloor a/32 \rfloor = \mathcal{P}$, so assume $\lfloor a/32 \rfloor \neq \mathcal{P}$.

Let

$$\langle \mathcal{M}', \mathcal{P}', \mathcal{D}'\rangle = \textit{swap}_a\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)$$

be the state of the VM memory after the swap. Note that $\mathcal{P}' = \lfloor a/32 \rfloor \neq \mathcal{P}$. The $\textit{m-interp}\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)$ and $\textit{m-interp}\big(\langle \mathcal{M}', \mathcal{P}', \mathcal{D}'\rangle\big)$ vectors are the interpreted VM memories before and after the swap, respectively. We know that $X = M$ and we need to show that also $\textit{m-interp}\big(\langle \mathcal{M}', \mathcal{P}', \mathcal{D}'\rangle\big) = M$. We do the latter by showing that $\textit{m-interp}\big(\langle \mathcal{M}', \mathcal{P}', \mathcal{D}'\rangle\big)_x = M_x$ for all $0 \leq x < 8192$. There are three cases:

$\lfloor x/32 \rfloor = \mathcal{P}$ (x was in the swapped-out page); we have

$$\begin{aligned}
\textit{m-interp}\big(\langle \mathcal{M}', \mathcal{P}', \mathcal{D}'\rangle\big)_x &= \mathcal{D}'_x && \text{by (5) since } \mathcal{P}' \neq \mathcal{P} = \lfloor x/32 \rfloor \\
&= \mathcal{D}'_{32\lfloor x/32 \rfloor + x \bmod 32} && \text{decomposing } x \\
&= \mathcal{D}'_{32\mathcal{P} + x \bmod 32} && \text{since } \lfloor x/32 \rfloor = \mathcal{P} \\
&= \mathcal{M}_{x \bmod 32} && \text{by (3)} \\
&= \textit{m-interp}\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)_x && \text{by (5) since } \mathcal{P} = \lfloor x/32 \rfloor \\
&= M_x && \text{by hypothesis;}
\end{aligned}$$

$\lfloor x/32 \rfloor = \mathcal{P}'$ (x was in the swapped-in page); we have

$$\begin{aligned}
\textit{m-interp}\big(\langle \mathcal{M}', \mathcal{P}', \mathcal{D}'\rangle\big)_x &= \mathcal{M}'_{x \bmod 32} && \text{by (5) since } \lfloor x/32 \rfloor = \mathcal{P}' \\
&= \mathcal{D}_{32\lfloor a/32 \rfloor + x \bmod 32} && \text{by (3)} \\
&= \mathcal{D}_{32\lfloor x/32 \rfloor + x \bmod 32} && \text{since } \lfloor x/32 \rfloor = \mathcal{P}' = \lfloor a/32 \rfloor \\
&= \mathcal{D}_x && \text{re-composing } x \\
&= \textit{m-interp}\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)_x && \text{by (5) since } \mathcal{P} \neq \mathcal{P}' = \lfloor x/32 \rfloor \\
&= M_x && \text{by hypothesis;}
\end{aligned}$$

$\lfloor x/32 \rfloor \neq \mathcal{P}$ and $\lfloor x/32 \rfloor \neq \mathcal{P}'$ (x was in neither page); we have

$$\begin{aligned}
\textit{m-interp}\big(\langle \mathcal{M}', \mathcal{P}', \mathcal{D}'\rangle\big)_x &= \mathcal{D}'_x && \text{by (5) since } \mathcal{P}' \neq \lfloor x/32 \rfloor \\
&= \mathcal{D}_x && \text{by (3)} \\
&= \textit{m-interp}\big(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)_x && \text{by (5) since } \mathcal{P} \neq \lfloor x/32 \rfloor \\
&= M_x && \text{by hypothesis.}
\end{aligned}$$

There are no other cases, so this concludes the proof. □

Finally, we can prove the theorem that we need.

Theorem 1. *Let* $t = \langle A, CI, M\rangle$ *and* $v = \langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle$ *be such that* $t = \mathit{interp}(v)$*. Then*

$$T\text{-}\mathit{next}(t) = \mathit{interp}\big(V\text{-}\mathit{next}(v)\big).$$

Proof. First we can show that both machines, starting from their respective states, will execute the same instruction. Let $\langle \mathcal{M}', \mathcal{P}', \mathcal{D}'\rangle = \mathit{swap}_{\mathcal{CI}}(\langle \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle)$. It is sufficient to show that $M_{CI} = \mathcal{M}'_{\mathcal{CI} \bmod 32}$. Indeed, by hypothesis we know that $CI = \mathcal{CI}$ and, by definition of *swap*, we have $\mathcal{P}' = \lfloor CI/32\rfloor$. Moreover, by Lemma 3, the *m-interp* function is preserved by the swap. Then, the conclusion follows by Lemma 1.

So, the opcode o and the operand address a decoded by both machines will be the same.

Now the proof is by cases on the opcode of the next instruction in state v.

Assume $o = 0$. Then

$$\begin{aligned}
&\mathit{interp}\Big(V\text{-}\mathit{next}\big(\langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)\Big) \\
&\quad = \mathit{interp}\big(\langle \mathcal{A}, \mathcal{M}''_{a \bmod 32} + 1, \mathcal{M}'', \mathcal{P}'', \mathcal{D}''\rangle\big) && \text{by (4)} \\
&\quad = \Big\langle \mathcal{A}, \mathcal{M}''_{a \bmod 32} + 1, m\text{-}\mathit{interp}\big(\langle \mathcal{M}'', \mathcal{P}'', \mathcal{D}''\rangle\big)\Big\rangle && \text{by (6)} \\
&\quad = \langle \mathcal{A}, \mathcal{M}''_{a \bmod 32} + 1, M\rangle && \text{by hyp. and Lemma 3} \\
&\quad = \langle A, \mathcal{M}''_{a \bmod 32} + 1, M\rangle && \text{by hypothesis} \\
&\quad = \langle A, M_a + 1, M\rangle && \text{by hyp. and Lemma 1} \\
&\quad = T\text{-}\mathit{next}\big(\langle A, CI, M\rangle\big) && \text{by (1)}
\end{aligned}$$

Assume $o = 1$. Then

$$\begin{aligned}
&\mathit{interp}\Big(V\text{-}\mathit{next}\big(\langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)\Big) \\
&\quad = \mathit{interp}\big(\langle \mathcal{A}, \mathcal{CI} + \mathcal{M}''_{a \bmod 32} + 1, \mathcal{M}'', \mathcal{P}'', \mathcal{D}''\rangle\big) \\
&\quad = \langle A, CI + M_a + 1, M\rangle \\
&\quad = T\text{-}\mathit{next}\big(\langle A, CI, M\rangle\big).
\end{aligned}$$

Assume $o = 2$. Then

$$\begin{aligned}
&\mathit{interp}\Big(V\text{-}\mathit{next}\big(\langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)\Big) \\
&\quad = \mathit{interp}\big(\langle -\mathcal{M}''_{a \bmod 32}, \mathcal{CI} + 1, \mathcal{M}'', \mathcal{P}'', \mathcal{D}''\rangle\big) \\
&\quad = \langle -M_a, CI + 1, M\rangle \\
&\quad = T\text{-}\mathit{next}\big(\langle A, CI, M\rangle\big).
\end{aligned}$$

Assume $o = 3$. Then

$$\begin{aligned}
&\mathit{interp}\Big(V\text{-}\mathit{next}\big(\langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)\Big) \\
&\quad = \mathit{interp}\big(\langle \mathcal{A}, \mathcal{CI}+1, \mathcal{M}''\{\mathcal{A}/a \bmod 32\}, \mathcal{P}'', \mathcal{D}''\rangle\big) \\
&\quad = \Big\langle \mathcal{A}, \mathcal{CI}+1, \\
&\qquad\qquad m\text{-}\mathit{interp}\big(\langle \mathcal{M}''\{\mathcal{A}/a \bmod 32\}, \mathcal{P}'', \mathcal{D}''\rangle\big)\Big\rangle \\
&\quad = \big\langle A, CI+1, M\{A/a\}\big\rangle && \text{by hyp. and Lemma 2} \\
&\quad = T\text{-}\mathit{next}\big(\langle A, CI, M\rangle\big).
\end{aligned}$$

Assume $o = 4$ or $o = 5$. Then

$$\begin{aligned}
&\mathit{interp}\Big(V\text{-}\mathit{next}\big(\langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)\Big) \\
&\quad = \mathit{interp}\big(\langle \mathcal{A} - \mathcal{M}''_{a \bmod 32}, \mathcal{CI}+1, \mathcal{M}'', \mathcal{P}'', \mathcal{D}''\rangle\big) \\
&\quad = \langle A - M_a, CI+1, M\rangle \\
&\quad = T\text{-}\mathit{next}\big(\langle A, CI, M\rangle\big).
\end{aligned}$$

Assume $o = 6$ and $\mathcal{A} = 0$. Then

$$\begin{aligned}
&\mathit{interp}\Big(V\text{-}\mathit{next}\big(\langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)\Big) \\
&\quad = \mathit{interp}\big(\mathcal{A}, \mathcal{CI}+2, \mathcal{M}'', \mathcal{P}'', \mathcal{D}''\rangle\big) \\
&\quad = \langle A, CI+2, M\rangle \\
&\quad = T\text{-}\mathit{next}\big(\langle A, CI, M\rangle\big) && \text{since } A = \mathcal{A} = 0.
\end{aligned}$$

Assume $o = 6$ and $\mathcal{A} \neq 0$. Then

$$\begin{aligned}
&\mathit{interp}\Big(V\text{-}\mathit{next}\big(\langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)\Big) \\
&\quad = \mathit{interp}\big(\mathcal{A}, \mathcal{CI}+1, \mathcal{M}'', \mathcal{P}'', \mathcal{D}''\rangle\big) \\
&\quad = \langle A, CI+1, M\rangle \\
&\quad = T\text{-}\mathit{next}\big(\langle A, CI, M\rangle\big) && \text{since } A = \mathcal{A} \neq 0.
\end{aligned}$$

Assume $o = 7$. Then

$$\begin{aligned}
&\mathit{interp}\Big(V\text{-}\mathit{next}\big(\langle \mathcal{A}, \mathcal{CI}, \mathcal{M}, \mathcal{P}, \mathcal{D}\rangle\big)\Big) \\
&\quad = \mathit{interp}\big(\mathcal{A}, \mathcal{CI}, \mathcal{M}'', \mathcal{P}'', \mathcal{D}''\rangle\big) \\
&\quad = \langle A, CI, M\rangle \\
&\quad = T\text{-}\mathit{next}\big(\langle A, CI, M\rangle\big). && \square
\end{aligned}$$