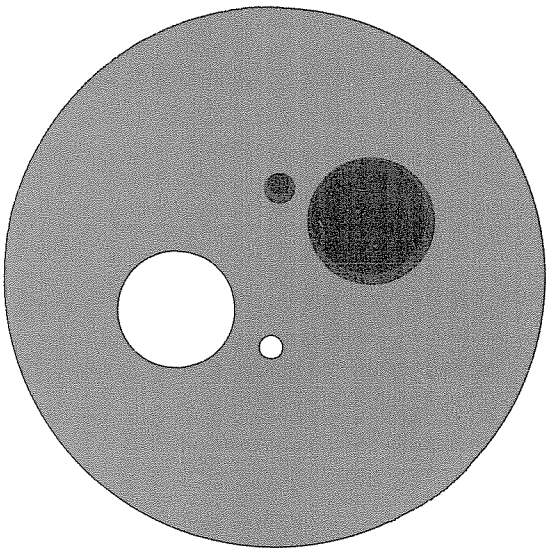
COMPUTER SCIENCES
DEPARTMENT

University of Wisconsin -
Madison



Formal Properties of Recursive Virtual Machine Architectures

by

Gerald Belpaire and Nai-Ting Hsu

Computer Sciences Technical Report #255

May 1975

The University of Wisconsin
Computer Science Department
1210 West Dayton Street
Madison, Wisconsin 53706

Received: May 28, 1975

Formal Properties of Recursive Virtual Machine Architectures

by

Gerald Belpaire and Nai-Ting Hsu

University of Wisconsin - Madison

Technical Report #255

ABSTRACT

A formal model of hardware/software architectures is developed and applied to Virtual Machine Systems. Results are derived on the sufficient conditions that a machine architecture must verify in order to support VM systems. The model deals explicitly with resource mappings (protection) and with I/O devices. Some already published results are retrieved and other ones, more general, are obtained.

INTRODUCTION

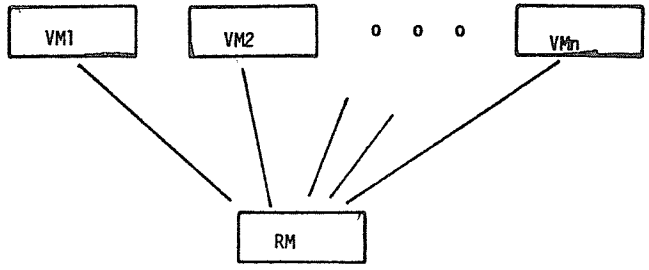
The principles and functions of Virtual Machine Systems have been quite extensively described in the literature. In this study, the interest we have in them resides less in the ability of implementing hierarchical structures of identical machines than in the theoretical and practical fall-outs of such implementations. In particular, the study of VM systems has uncovered interesting relations between the system structure and the host machine architecture. Based upon these findings, several new machine architectures were proposed for VM system design [1, 3, 4, 5] and theoretical results were obtained on the sufficient conditions that the hardware should verify in order to support a VM system.

In this paper, we further investigate the properties of the hardware/software interface of VM systems. We adopt a classical definition of VM systems and then proceed to examining (with the appropriate formalism) what machine architectures are needed to implement them. Some already known results are retrieved [6] and some new ones are obtained. Examples of architectures are discussed.

Our aim is to reach a better understanding of the Virtual Machine "phenomenon", but also to provide practical guidelines for machine architects and system designers.

II. DEFINITION OF A VIRTUAL MACHINE SYSTEM

A (non-recursive) VM system simply consists of a perfect multiplexing of the real machine.



The time-sharing aspect of the multiplexing is secondary; what imports is the concurrent co-existence of several virtual machines. This multiplexing includes the scheduling of the CPU but also the sharing of the memory (virtual memory is a necessary element of virtual machines), and of other resources of the RM. Classically this will be implemented by a software nucleus called Virtual Machine Monitor (VMM). Furthermore we need some facilities that will be the equivalent, for the Virtual Machines, of the hardware starting and halting functions on the Real Machine.

Three essential characteristics must be possessed by such a VM system:

1. The execution environment of each VM is, from a logical standpoint, identical to the one of the Real Machine. Said otherwise, what runs on the RM can run on the VM. It accepts the same instructions and has an identical resource space, i.e., it can access an identical address space and identical peripherals.

2. The performance of a VM with respect to the one of the RM, is degraded only by the need for resource sharing.

3. The different VMs are protected by impassable walls, i.e., they cannot interfere with each other.

If the VMM can run on a Virtual Machine to generate another level of VMs then the system is a Recursive Virtual Machine System. Its structure is a potentially infinite tree of VMs. Each node of the tree must verify the same three characteristics of VM systems.

These definitions have been given, with minor differences only, in a number of papers on the subject. However the second characteristic as stated here is rather stronger than usual. In particular it prohibits any interpretive execution of instructions by the VMM. Interpretation can be rapidly catastrophic when the system has several levels of "recursion", as its detrimental effect is spread throughout all the levels. Early implementations of VM systems have compromised on this point for practical reasons: no suitable hardware was available. Even recent designs allow the interpretive execution of privileged instructions. Our present aim is, in part, to characterize the type of hardware needed to avoid these drawbacks. This is not to mean that no compromises should be made but that we want to evaluate at what price they can be avoided.

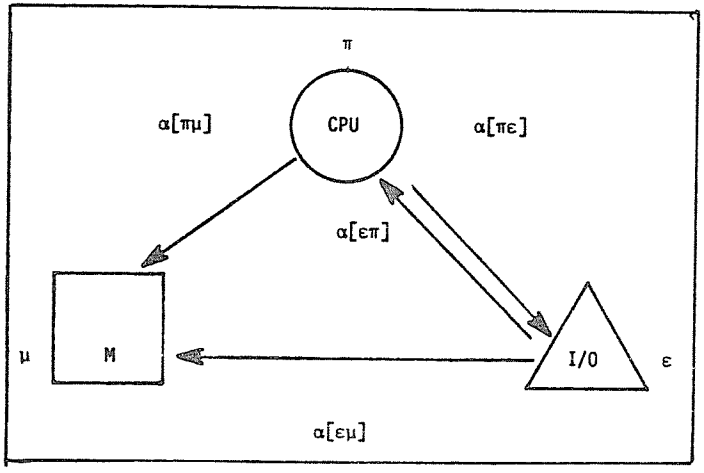
Since performances of the system depend upon the level of resource sharing, it might seem possible to increase these performances by increasing the amount of resources, i.e. by

decreasing the level of sharing. However, this is not immediately true in general since increasing the number of resources extends the real machine and therefore imposes a sharing of this extended machine.

In order to examine more formally these aspects of the problems, we shall define in the next section an ideal real machine that will perfectly verify the three criteria of a VM system. This architecture is hypothetical but, by imposing restrictions on its structure, we shall be able to determine under which conditions its properties are invariant, i.e., under which conditions the system remains a VM system.

III. THE INFINITE RECURSIVE ARCHITECTURE (IRA).

We first consider a simple machine architecture with one processing unit, one central memory, and one I/O peripheral. The primitiveness of this machine is taken for reason of simplicity; it however presents the minimum set of features necessary to illustrate our point.



It consists of:

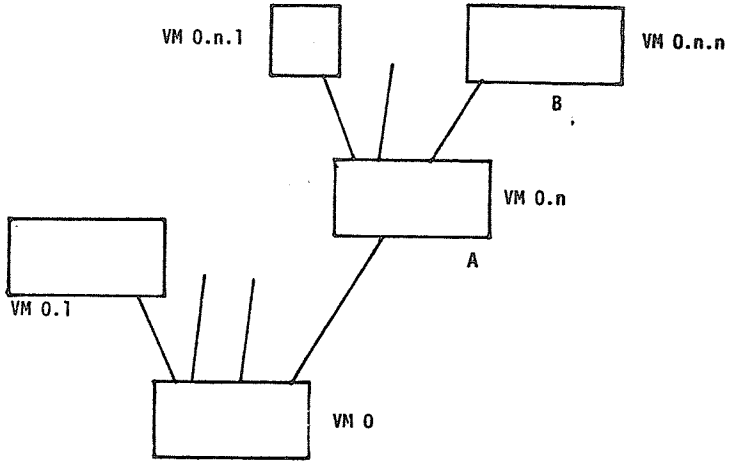
1. one CPU (central processing unit) that we denote by π.
2. one I/O unit denoted by ε.
3. one central memory denoted by μ.

Basic communication links exist between these devices. They are:

1. memory accesses by the CPU and by the I/O device (e.g. a channel):
   α[πμ] and α[εμ] (resp.)
2. the I/O commands:
   α[πε]
3. the interrupt mechanism:
   α[επ]

The purpose of these notations is to represent the existence of such communications rather than the mechanism of their functions. For example, the fact that I/O commands can be done by memory accesses is irrelevant at this point. It will be detailed at a later stage. The different execution modes (supervisor, kernel,...) of the CPU will also be defined later.

Using an infinite number of such modules we can define an arbitrary tree whose nodes are the hereabove defined elementary machines.



The parent machine has the ability to start and to halt its child machines. This provides for the only communications (in this machine) between nodes belonging to different levels.

Let T(A) be the machine consisting of all the elementary machines belonging to the sub-tree whose root is A.

We certainly have:

1. The execution environment of T(A) is identical to the one of T(VM 0).
2. The performance of T(A) is identical to the one of T(VM 0).
3. T(A) is isolated from all sub-trees whose roots are at the same level as A.

This shows that this architecture exhibits the structure and the properties of a VM machine system in an idealized manner. We call it the Infinite Recursive Machine (IRA). Of course, these properties were obtained by using an infinite number of resources and therefore the IRA does not deserve the Virtual Machine label. However there is no way for a program to determine whether it runs on the IRA or on a perfect VM system. In that sense the IRA captures the notions common to all VM systems. Our task now is to examine how these notions and properties remain invariant while the number of resources is reduced.

In the IRA the intra-node architecture is completely independent on the VM system structure. In particular, no restrictions are made upon the memory mappings or on the processor instructions.

It therefore seems reasonable to state that only the reduction of the number of resources, which is effectively a mapping of all resources of the VM system into one node, will be the cause of imposed limitations on the intra-node architecture. These limitations will be necessary to enforce a proper working of the Virtual Machine Monitor, i.e., they give the VMM the power to control the resources of the system. The processes, other than the VMM, running on a VM can in principle use any form of protection, independently upon the VMM form of protection. But this would impose that the VMM has different power than these processes, a fact not likely to be accepted because it would mean that virtualizable machines have a specific feature to support the Virtual Machine Monitor, different from the ones supporting the other processes. In conclusion, although the intra-node architecture is in principle not affected by the virtualization, the need to consider the VMM as any other process will limit the architecture.(*)

*An example of a VM design that gives different powers to the VMM than to other processes is the Hardware Virtualizer [4].

IV. VIRTUALIZATION OF THE IRA.

Typically, a VM system with the structure of the IRA has the property that all the resources are mapped into the resources of the root node VM 0 (which then is the Real Machine). By mapping is meant that each resource is available to a Virtual Machine only when it is mapped into a real resource: the mapping describes the allocation of real resources to virtual resources. When this allocation holds, we say that the virtual resource is realized.

Formally this will be defined by a time dependent mapping between the set of virtual resources and the set of real resources. This mapping will have the following meaning:

1. For the memory, it is the usual virtual memory address translation.
2. For the CPU, it abstracts the program status registers and the assignments of these registers.
3. For the I/O, it specifies the device status words and the device buffers.

Whenever an attempt is made to use a virtual resource without it being realized, a VM fault occurs and is routed to the VMM of the parent machine. The VMM has the functions of properly handling VM faults and resources allocation. In some VM systems it also has the function of interpreting some instructions. For reasons explained earlier we momentarily assume that the latter function is nonexistent.

The proper handling of the resources allocation is a problem of local program correctness of the VMM. We need only to guarantee

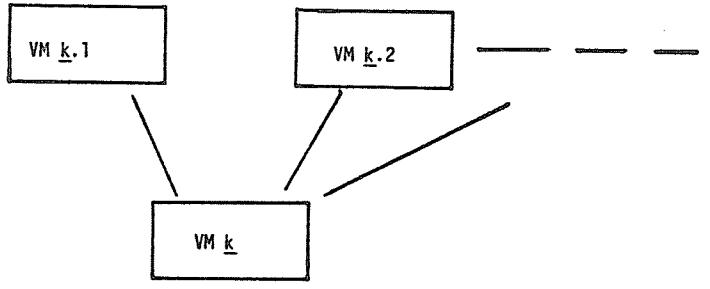
that the architecture provides the right features for the VMM to implement its functions. This is to say that the different resource mappings must be represented in the machine and accessible to the VMM.

The problem of VM protection cannot be solved by assumption on the correct behavior of the system as we must consider possible occurrences of errors and of malicious attitudes. The architecture must therefore include features that will render these "errors", even in the worst case, harmless.

We now proceed to the formal treatment of the virtualization and consider first the case where the CPU has only one execution mode (paradoxically the less frequent in practice but the simplest one).

First Case: One Execution Mode.

We treat first an arbitrary pair of levels of Virtual Machines.



$\underline{k} = 0.k_1.k_2.k_3. \ldots$ is an arbitrary node label.

Let f_t be the notation for the realization map (also called resource map [4]). It represents the mapping as it holds at time t.(*).

To define f_t precisely, we have three cases to consider:

1. The memory.

   $f_t[\mu_{\underline{k}.i}]: \mu_{\underline{k}.i} \rightarrow \mu_{\underline{k}} \cup \{\omega\}$

   To a virtual address x corresponds the location

   $f_t[\mu_{\underline{k}.i}]\ (x)$ of $\mu_{\underline{k}}$

   If $f_t[\mu_{\underline{k}.i}]\ (x) = \omega$ then the address x is not realized and a VM fault will occur when x is accessed.

2. The CPU.

   $f_t[\pi_{\underline{k}.i}]: \{\pi_{\underline{k}.i}\} \rightarrow \{\pi_{\underline{k}}, \omega\}$

   If $f_t[\pi_{\underline{k}.i}]\ (\pi_{\underline{k}.i}) = \omega$ then $\pi_{\underline{k}.i}$ is not realized and VM $_{\underline{k}.i}$ is not currently running.

3. The I/O device.

   The definition is analog to the CPU map:

   $f_t[\varepsilon_{\underline{k}.i}]: \{\varepsilon_{\underline{k}.i}\} \rightarrow \{\varepsilon_{\underline{k}}, \omega\}$

*The time appears here as a parameter of the mapping, i.e. for different instants there are different mappings. This representation is equivalent to a representation in terms of state and it was chosen for simplicity. For more details on the time vs. state duality, cf.[7].

We also define $\bar{\mu}_{\underline{k}}$ as:

$$\bar{\mu}_{\underline{k}} = \mu_{\underline{k}} - \bigcup_i f_t[\mu_{\underline{k}.i}]\ (\mu_{\underline{k}.i})$$

i.e., $\bar{\mu}_{\underline{k}}$ at time t, is the part of $\mu_{\underline{k}}$ that is not allocated to the virtual machines.

To define the "recursive" aspect of the VM system, we need only to take the functional composition of the mappings. For instance, if x is a virtual address for the machine VM $0.k_1.k_2 \ldots k_n$, the real location realizing x is:

$$f_t[\mu_{0.k_1}] \circ f_t[\mu_{0.k_1.k_2}] \circ \ldots \circ f_t[\mu_{0.k_1.k_2 \ldots k_n}]\ (x)$$

if this composition is defined, i.e., if no resulting value is ω.

The mapping must be taken at the same instant t. This does not mean that it must be the same "clock" instant, but the same "system" instant, i.e. that between the first and the last effective mapping calculations the state of the system does not change.

At this stage of our developments two remarks can be made:

1. The virtual macine at one level is "virtual" for the lower level and "real" for a higher level. Mappings are nested as well as VM faults [4]. Similar observations were made for hierarchies of processes [2].

2. The CPU is preemptible and re-usable and for these reasons it can be shared on a time dependent basis. The core memory is also preemptible and re-usable, but this is not true for the memory taken globally (core + secondary memory). Indeed to preempt secondary memory is inevitably to loose information. Therefore the memory is sharable only on a space dependent basis and it is impossible that each VM can be allocated the total (core + secondary) memory. In this sense, the virtualization considered hereabove is not acceptable since a VM cannot share the entire Real Machine. This is a rather theoretical point, as in practice the secondary memory is considered as a memory "sink" rather than as a part of the real machine.

We now make the assumption that all accesses within a particular VM are made through memory locations. This is a quite realistic hypothesis as some actual implementation are working on this principle. It was also taken as hypothesis in other papers [4, 6, 1].

In this case it is easy to define formally the different accesses characterized earlier:

1. Let $m_{\underline{k}.i} \subseteq \mu_{k.i}$ the set of virtual memory locations used for the regular memory accesses by the CPU and the I/O devices. They correspond to the accesses:

   $$\alpha[\pi_{\underline{k}.i}\ \mu_{\underline{k}.i}] \text{ and } \alpha[\varepsilon_{\underline{k}.i}\ \mu_{\underline{k}.i}]$$

2. Let $s_{\underline{k}.i}$ be the set of virtual addresses by which the CPU controls the I/O devices (by the access $\alpha[\pi_{\underline{k}.i}\ \varepsilon_{\underline{k}.i}]$).

3. Let $v_{\underline{k}.i}$ be the set of (virtual) interrupt vectors. (access $\alpha[\varepsilon_{\underline{k}.i}\ \pi_{\underline{k}.i}]$).

The protection requirements (third characteristic of a VM system) imply:

1. For all pair i, j, at any time t; and for each mode VM$\underline{k}$

   $$\text{if } r_{\underline{k}.i} = m_{\underline{k}.i} \cup s_{\underline{k}.i} \cup v_{\underline{k}.i}$$

   $$f_t[\mu_{\underline{k}.i}](r_{\underline{k}.i}) \cap f_t[\mu_{\underline{k}.j}](r_{\underline{k}.j}) = \phi$$

   this means that the address spaces are mapped into disjoint address spaces. This property is enforced by the VMM program, i.e. this is a condition of correctness of the VMM.

2. The mapping f_t must be represented in the machine to be accessible to the VMM and to the VMM only. A sufficient condition is that it resides in the virtual address space of the VMM.

3. Let $\rho(f_t)$ be a notation for the representation of f_t, i.e. $\rho(f_t[\mu_{\underline{k}.i}])$ is a set of memory locations.

We can formulate the following theorem to summarize the preceding conditions:

Theorem 1.

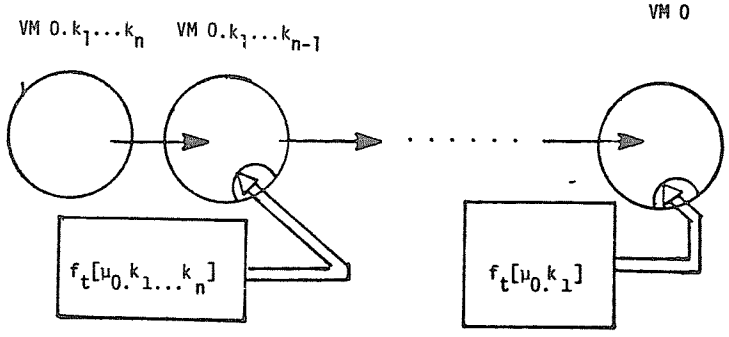
A sufficient condition for a proper working of a VM system of the type defined here above is:

1. The representation of the realization mapping of each node resides in the address space of its VMM:

   $$\rho(f_t[\mu_{\underline{k}.i}]) \subseteq \bar{\mu}_{\underline{k}} \quad \text{at all time t.}$$

2. If the mapping is not defined when used at a time t, it generates a VM fault in the VMM address space.

We can schematize the situation by the following figure:



It is important to notice that the representation of the mapping is itself in virtual memory. This means that any use of the memory mapping will be recursive. This should not be surprising since the structure of the system is itself recursive. As such an address calculation by means of the mapping can cause a VM fault if the mapping representation is not realized at a lower level. It is evident that such a design, although secure in principle, will cause a tremendous overhead as it forces the address calculation to be done interpretively in Virtual Memory. The conditions stated earlier are thus minimal theoretical conditions. "Accomodations" must be made to render it practical.

One solution that was taken as basis for a number of proposed solutions is to force the representation of the mappings to reside in fixed physical location and to make it time independent. This means that the mappings are not in virtual memory anymore and that they remain constantly in the same location. Only one node of the tree of VMs can be active at a time (because there is only one CPU) and therefore only one "path" of mappings must be available at that time. The inactive mappings can therefore reside anywhere in the virtual memory of the VMM.

The simplest solution using these principles is the Hardware Virtualizer (HV) [4]. In this scheme, the mapping representations are in Virtual memory and a hardware register (the Virtual Machine Identifier register, VMID) indicates how to access them. This design however generates a problem: how to protect the VMID. Goldberg

describes in his paper a way to remedy this problem.

In Lauer's proposal [5], the mapping representations are also in virtual memory, and are accessed through a stack of registers (the Display). A register containing the currently realized level of VM ensures that only the appropriate registers of the Display are accessed by the VMM. Only the active mappings are pointed at by the Display.

A more recent proposal [1] generalized the preceding ones by using a stack of registers representing the mappings (multiple copies of RMRs, Resource Management Registers). Here too a level register ensures proper accesses to the RMRs.

Which solution should be adopted is determined at this point by efficiency considerations only.

Before considering the case with two execution modes, several remarks could be made on this result:

1. The virtualization does not require any interpretation that would not be needed on the real machine, if one excepts the resource mapping calculations.

2. No particular paging or segmentation mechanism is suggested for the memory management. The program counter is considered to be an intra-VM feature, working only in virtual addresses, and therefore is not protected by the resource mapping.

3. All I/O accesses must be made by virtual addresses. In particular no instruction such as the RESET on the PDP 11 is tolerated. This is not a restriction, as the RESET function can be implemented differently.

4. When an interrupt occurs, it will be routed to the appropriate VM by the memory mapping provided that the

VM mode label is memorized in a location associated with the device. If any interrupt mask is set (also done through some virtual memory address), the interrupt will be appropriately inhibited for the corresponding machine. However interrupts are accesses of a particular kind as they must be acknowledged (if not masked). This means that a VM currently being realized on the CPU might be interrupted to serve another VM. This situation is abnormal with respect to the VM system characteristics and it led to suggestions that other mechanisms than interrupts should be devised [5].

Second Case: Two Execution Modes

We assume that the CPU can be in two execution modes: privileged (P) and non-privileged (NP) mode. Typically in the P-mode the CPU may execute special instructions not executable in the NP-mode. We represent this formally by defining in the P-mode accesses of the machine that are different from the ones in the NP-mode. Indeed the privileged instructions are usually used for specific accesses to the processor status, the memory management registers, or the I/O devices. The effects of these accesses is not as important as the ability of having such accesses, at least for our present purpose.

We conserve in this section all the previously defined concepts and objects and assume that they denote the non-privileged mode features. To represent the privileged mode features we simply use the corresponding non-privileged mode notations with a star. For example:

$\pi_{\underline{k}}$ is a virtual CPU in the NP mode

$\pi_{\underline{k}}^*$ is the same CPU in the P mode.

We furthermore define the access of the CPU to its own execution mode attributes (e.g., status word) by:

$\alpha[\pi_{\underline{k}}\pi_{\underline{k}}]$ and $\alpha^*[\pi_{\underline{k}}\pi_{\underline{k}}]$ for the VM $\underline{k}$

The access $\alpha[\varepsilon\pi]$, which was to represent the effects of the interrupts, had in the case with one execution mode the effect of branching the processor to a new instruction in the virtual memory. In the case of two execution modes the interrupts might have the effect of changing the execution mode. This can be "automatic", e.g. after an interrupt the processor is always in the P-mode, or it can be done by fetching a new processor status word.

In a first step, we assume that all accesses are made through memory as for the case with one execution mode. This gives the following virtual memory areas for the virtual machine VM $\underline{k}.i$:

1. The virtual addresses: $m_{\underline{k}.i}$ and $m^*_{\underline{k}.i}$ for the accesses $\alpha[\pi_{\underline{k}.i}\ \mu_{\underline{k}.i}]$, $\alpha[\varepsilon_{\underline{k}.i}\ \mu_{\underline{k}.i}]$, $\alpha^*[\pi_{\underline{k}.i}\ \mu_{\underline{k}.i}]$, and $\alpha^*[\varepsilon_{\underline{k}.i}\ \mu_{\underline{k}.i}]$
2. The virtual addresses by which the CPU controls the I/O devices:
   $s_{\underline{k}.i}$ and $s^*_{\underline{k}.i}$
3. The interrupt vectors: $v_{\underline{k}.i}$ and $v^*_{\underline{k}.i}$. The possible change of execution mode is done by fetching a new processor status word from one of the locations $v_{\underline{k}.i}$ or $v^*_{\underline{k}.i}$.
4. The virtual addresses of the processor status words (accesses $\alpha[\pi_{\underline{k}.i}\ \pi_{\underline{k}.i}]$ and $\alpha^*[\pi_{\underline{k}.i}\ \pi_{\underline{k}.i}]$):
   $p_{\underline{k}.i}$ and $p^*_{\underline{k}.i}$.

In most of the machine, simplifications will be made, e.g. $m_{\underline{k}.i} = m^*_{\underline{k}.i}$ or $v_{\underline{k}.i} = v^*_{\underline{k}.i}$. These are not needed for our present purpose. In the P mode the f_t^* mappings will be used and in the NP mode f_t will be used; this fact is familiar to the users of the PDP 11/45 where different sets of memory management registers are used for different execution modes.

We make a further assumption: A P-mode VM$\underline{k}.i$ is mapped into a P-mode VM$\underline{k}$, and a NP-mode VM$\underline{k}.i$ is mapped into a NP-mode VM$\underline{k}$. With this it is easy to generalized the Theorem 1 to the case with two execution modes:

Theorem 2.

A sufficient condition for the proper working of a VM system with two execution modes is:

1. The representation of the realization mappings of each mode resides in the VMM address space:

   $\rho(f_t[\mu_{\underline{k}.i}]) \subseteq \bar{\mu}_{\underline{k}}$ and $\rho(f^*[\mu_{\underline{k}.i}]) \subseteq \bar{\mu}^*_{\underline{k}}$

   this must be true at all time t.

2. If the mapping is not defined when used at a time t, it generates a VM fault in the VMM address space

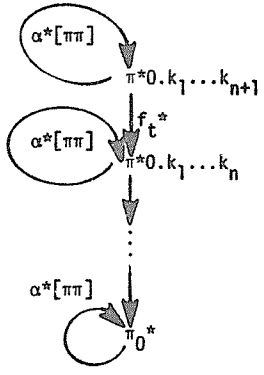
   $\bar{\mu}_{\underline{k}}$ or $\bar{\mu}_{\underline{k}}^*$.

We can conclude from this theorem that the execution mode is a virtual execution mode in the case where all accesses are made through virtual memory. No interpretive execution of the privileged instructions is needed. This is true for the particular implementations described earlier (HV, Display, Multiple Copies of

RMRs) and shows formally their superiority upon the classical third generation architectures.

Let us assume now that the accesses $\alpha[\pi\pi]$ and $\alpha^*[\pi\pi]$ are made through a status register accessed by a special instruction. The address of the register is absolute and a VM using the special instruction will modify its contents.

This is a case where an access is not made through the virtual memory mappings. The scheme of operation, whenever a status register is accessed, is as follows:



Whenever $\pi^*_{0.k_1\ldots k_n}$ executes the special instruction accessing the status register, the access $\alpha^*[\pi\pi]$ is performed at the highest level. But since the register is common to all Virtual Machine levels, this will cause the performing of an access $\alpha^*[\pi\pi]$ at all levels or, said otherwise, this will cause a change of the execution mode of all the Virtual Machines, including the Real Machine VM0.

This does not verify the VM system characteristics and must be prevented.

The classical solution to this problem proceeds as follows:

1. All the VMs are in non-privileged mode, including the Real Machine.

2. Whenever an access to the status register is made, it generates a VM fault at level 0, and the VMM at that level can interprete the fault and take the appropriate action. This is possible only if:

   a) The special instruction is a privileged instruction, i.e. it generates a VM fault when executed in NP mode.

   b) the VM fault vector of the privileged instruction provokes a change of mode to P mode, and branches to a location in the VMM space of level 0. (i.e. in $\tilde{\mu}_0^*$)

The "special" instruction considered here above is a particular case of a class of instructions called sensitive instructions (*).

A sensitive instruction is any instruction that bypasses the accesses through the virtual memory mapping.

It includes the following examples:

1. Access to a absolute processor status word

2. Access to physical absolute address (bypass of the memory map) or access to memory by another mapping (Move from Previous Instruction Space, PDP 11).

3. Instructions whose effect depends on the physical address

(*) Defined by Popek & Goldberg [6]. Their definitions do not recover completely ours but we keep the same name.

(the address is virtual but the effect of the instruction depends on the physical address).

4. Access to I/O devices by absolute addresses of status and command words.

5. Accesses to memory by I/O devices made in absolute addresses.

6. Interrupt vectors in absolute addresses.

7. Interrupts provoking automatically a change of an absolute processor status word.

8. Accesses to absolute relocation bound registers.

The list is not exhaustive. Actually, in this paper, contrary to what is done in [6], a sensitive instruction is not defined "positively": it is defined as anything that does not behave in a "virtual" manner.

If one generalizes the solution given hereabove for the virtualization of machines with sensitive instructions, one can state the following theorem (due to Popek and Goldberg):

Theorem 3.

A sufficient condition for the proper working of a VM system with sensitive instructions is:

1. all sensitive instructions must be privileged instructions. (they must generate VM fault when used in NP mode).

2. The VM fault provokes a change to P mode of the real machine and branches to a location in $\tilde{\mu}_0^*$, the address space of the VMM at level 0.

If a machine encompasses features of various forms, Theorem 2 should be applied to some of them and Theorem 3 to other ones.

It is evident that in spite of its simplicity this theorem defines a virtualization that will involve interpretive execution of the

privileged instructions. This has two negative consequences: 1. it decreases the performances of the system, and 2. it adds complexity to the Virtual Machine Monitor (i.e. an increase in programming costs).

V. CONCLUSIONS

We have set forth a method of formal analysis of VM systems that proceeds in two steps: 1. We specify the structure of the system without concern for the sharing of resources, i.e. the number of resources is assumed to be unlimited, 2. We examine the restrictions needed to preserve the structure of the system while reducing the number of resources and establishing resource sharing. This leads to the formulation of sufficient conditions that the machine architecture should verify in order support the VM system.

When considering a generalized resource mapping mechanism one can observe that a VM system can be supported with a minimum of overhead and with a relative simplicity of the Virtual Machine Monitor functions. The problem of the resource management overhead can be resolved by suitable implementations of multiple copies of Resource Management Registers. If one uses classical machine architectures with a lot of complex and not easily defined sensitive instructions, one obtains both a significant inefficiency of the system and a definite complexity of the design.

From this a lesson can be drawn: simple and sound hardware enhances simple software structures, "wild" architectures generate "wild" programming problems.

REFERENCES

1. Belpaire, Gerald, and Hsu, Nai-Ting. Hardware Architecture for Recursive Virtual Machines. Tech. Rep. #242, Computer Sciences Dept., University of Wisconsin - Madison.

2. Belpaire, G., and Wilmotte, J.P. Correctness of Realizations of Levels of Abstraction in Operating Systems, in Operating Systems, Proceedings of an Int. Symposium, Rocquencourt, April 23-24, 1974. Gelenbe, E. and Kaiser, C. (Eds.), Lecture Notes in Computer Science 16, pp. 1-14, Springer-Verlag, Berlin-Heidelberg, 1974.

3. Goldberg, R.P. Hardware Requirements for Virtual Machine Systems. Fourth Hawaii International Conference on System Sciences, Honolulu, Hawaii, Jan. 1971, pp. 449-451.

4. Goldberg, R.P. Architecture of Virtual Machines. Proc. NCC 1973, AFIPS Press, Montvale, N.J., pp. 309-318.

5. Lauer, H.C., and Wyeth, D. A Recursive Virtual Machine Architecture. Technical Report #54, Computing Laboratory, University of Newcastle upon Tyne, G.B., 1973.

6. Popek, G.J., and Goldberg, R.P. Formal Requirements for Virtualizable Third Generation Architectures. Comm. ACM 17, 7, July 1974, pp. 412-421.

7. Mesarovic M.D., Takahara Y. General Systems Theory: Mathematical Foundations. Academic Press, New York, 1975.