ARCHITECTURE OF VIRTUAL MACHINES*

by R. P. Goldberg

Honeywell Information Systems, Inc.
Billerica, Massachusetts
and
Harvard University
Cambridge, Massachusetts

ABSTRACT

In this paper we develop a model which represents the addressing of resources by processes executing on a virtual machine. The model distinguishes two maps: the ϕ-map which represents the map visible to the operating system software running on the virtual machine, and the f-map which is invisible to that software but which is manipulated by the virtual machine monitor running on the real machine. The ϕ-map maps process names into resource names and the f-map maps virtual resource names into real resource names. Thus, a process running on a virtual machine addresses its resources under the composed map $f \circ \phi$. In recursive operation, f maps from one virtual machine level to another and we have $f \circ f \circ \ldots \circ f \circ \phi$.

The model is used to describe and characterize previous virtual machine designs. We also introduce and illustrate a general approach for implementing virtual machines which follows directly from the model. This design, the Hardware Virtualizer, handles all process exceptions directly within the executing virtual machine without software intervention. All resource faults (VM-faults) generated by a virtual machine are directed to the appropriate virtual machine monitor without the knowledge of processes on the virtual machine (regardless of the level of recursion).

* This work was sponsored in part by the Electronic Systems Division, U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract Number F19628-70-C-0217.

This is the preliminary version of a paper to be presented at the AFIPS National Computer Conference, New York, New York, June 4-8, 1973.

Second edition of Proceedings, July, 1973

INTRODUCTION

Virtual machine (VM) systems are a major development in computer systems design[1]. By providing an efficient facsimile of one or more complete computer systems, virtual machines have extended the multi-access, multi-programming, multi-processing systems of the past decade to be multi-environment systems as well. Thus, many of the advantages in ease of system use previously enjoyed only by application programmers have been made available to systems programmers.

Some of these advantages include support of the following activities concurrently with production uses of the system:

- improving and testing the operating system software[2]
- running hardware diagnostic check-out software[1]
- running different operating systems or versions of an operating system[3,4]
- running with a virtual configuration which is different from the real system, e.g., more memory or processors, different I/O devices[5]
- measuring operating systems[6,7]
- adding hardware enhancements to a configuration without requiring a recoding of the existing operating system(s)[3]

providing a high degree of reliability and security/privacy for those applications which demand it[8,9,10].

While several virtual machine systems have been constructed on contemporary machines[3,7,11,12,13,14], the majority of today's computer systems do not and cannot support virtual machines[15]. The few virtual machine systems currently operational, e.g. CP-67, utilize awkward and inadequate techniques because of unsuitable architectures.

Recent proposals of computer architectures specifically designed for virtual machines, i.e., virtualizable architectures, have suffered from two weaknesses. Either they have been unable to support modern complex operating systems directly on the virtual machines[16,17] or they have been unable to avoid all of the traditional awkwardness associated with virtual machine support[18].

A new proposal[19] called the Hardware Virtualizer, avoids the weaknesses of the previous designs while at the same time incorporating their strong points. Thus, the Hardware Virtualizer applies to the complete range of conventional computer systems and eliminates the awkwardness and overhead of significant software intervention. The Hardware Virtualizer may either be added to an existing computer system design or incorporated directly into a future system design.

In this paper, we develop a model which represents the mapping and addressing of resources by a process executing on a virtual machine. By deriving properties of the model, we can clarify and contrast existing virtual machine systems. However, the most important result of the model is that its proper interpretation implies the Hardware Virtualizer as the direct natural implementation of the virtual machine model. We develop some of the characteristics of the Hardware Virtualizer and then illustrate the operation through the use of a concrete example.

MODEL OF A PROCESS RUNNING ON A VIRTUAL MACHINE

In order to derive the underlying architectural principles for virtual machines, we develop a model that represents the execution of a process on a virtual machine. Since we want these principles to be applicable to the complete range of conventional computer systems-- fromminicomputers, through current general purpose third generation systems, and including certain future (possibly fourth generation) machines -- it is necessary to produce a model which reflects the common points of all of these systems. The model should not depend on the particular map structures visible to the software of the machine under discussion. Features such as memory relocation or supervisor state are characteristics of the existing system and occur whether or not we are discussing virtual machines.

To introduce virtual machines we must define a different, independent mapping structure which captures the notions common to all virtual computer systems. The unifying theme is the concept of a virtual machine configuration and a set of virtual resources. These resources, e.g., the amount of main memory in the virtual machine, are a feature of all virtual machines regardless of the particular virtual processor's form of memory relocation, etc. Thus, the key point is the relationship between the resources in the configuration of the virtual machine and those in the configuration of the real (host) machine. Only after this relationship has been fully understood need we treat the complexities introduced by the existence of any additional mapping structure.

The resource map f

We develop a model of virtual machine resource mapping by defining the set of resources $V = (v_0, v_1, \ldots, v_m)$ present in the virtual machine configuration and the set of resources $R = (r_0, r_1, \ldots, r_n)$ present in the real (host) configuration. [Resource spaces, both real and virtual, are always represented as squares in the figures.] The sets V and R contain all main memory names, addressable processor registers, I/O devices, etc. However, in the discussion which follows, for simplicity, we treat all resource names as if they are memory names. As Lauer and Snow[16] have observed, memory locations can be used to reference other resource names such as processor registers, e.g., DEC PDP-10, or I/O devices, e.g., DEC PDP-11. Therefore, no generality is

lost by treating all resource names as memory names.

Since we assume no a priori correspondence between virtual and real names, we must incorporate a way of associating virtual names with real names during execution of the virtual machine. To this end, we define, for each moment of time, a function

$$f: V \longrightarrow R \cup \{t\}$$

such that if $y \in V$ and $z \in R$ then

$$f(y) = \begin{cases} z & \text{if } z \text{ is the real name for virtual name } y \\ t & \text{if } y \text{ does not have a corresponding real name} \end{cases}$$

The value $f(y) = t$ causes a trap or fault to some fault handling procedure in the machine whose resource set is R, i.e., the machine R. For clarity we always term this event a VM-fault, never an exception.

We call the function f a resource map, virtual machine map, or f-map. The software on the real machine R which sets up the f-map and (normally) receives control on a VM-fault is called the virtual machine monitor (VMM).

The model imposes no requirement that the f-map be a page map, relocation-bounds (R-B) map, or be of any other form. However, when speaking of virtual machines we normally restrict our attention to those cases where both the virtual machine is a faithful replica

of the real machine and the performance of the virtual system can be made comparable to the real one.

Recursion

The resource map model developed above extends directly to recursion by interpreting V and R as two adjacent levels of virtual resources. Then the real physical machine is level 0 and the f-map maps level n+1 tc level n.

Recursion for virtual systems is not only a matter of conceptual elegance or a consideration of logical closure[16,17], it is also a capability of considerable practical interest[18,20]. In its simplest form, the motivation for virtual machine recursion is that although it makes sense to run conventional operating systems on the virtual machine, in order to test the VMM software on a VM,it is also necessary to be able to run at least a second level virtual machine.

In the discussion which follows, we use a PL/I - style qualified name tree-naming convention in which a virtual machine at level n has n syllables in its name[18,19].
This tree-name is used as a subscript for both the virtual resource space, e.g., $V_{1.1}$, and corresponding f-map, e.g., $f_{1.1}$.

Thus, if

$$f_1: V_1 \longrightarrow R$$

$$f_{1.1}: V_{1.1} \longrightarrow V_1$$

Then a level 2 virtual resource name y is mapped into $f_1(f_{1.1}(y))$ or $f_1 \circ f_{1.1}(y)$.* See Figure 1a.

In this function, $f_1 \circ f_{1.1}$, we identify two possible faults:

(1) The level 2 resource (virtual machine) fault to the VMM of level 1, i.e., $f_{1.1}(y) = t$. See Figure 1b.

(2) The level 1 resource (virtual machine) fault to the VMM of level 0 (the real machine), i.e., $f_1 \circ f_{1.1}(y) = t$. See Figure 1c.

In general, a composed f-map may cause either fault. However, there exists a class of maps, called inclusive maps, which can only cause the first fault (level 2 fault). The relocation-bounds map (R-B map) is inclusive but the page map is not. The inclusive property implies the possibility of simple recursive implementation[16,19].

For the general case of level n recursion, we have n-level virtual name y being mapped into

$$f_1 \circ f_{1.1} \circ \cdots \circ f_{1. \ldots .1}(y).$$

See Figure 1d.

The present model may be used to describe the proposals of Lauer and Snow[16] and of Lauer and Wyeth[17] for single state recursive virtual machines. In the former case, the map is f = R-B; in the latter case, it is f = segmentation. See discussion of Table I below.

---

*"o" is the conventional function composition operator of mathematics.

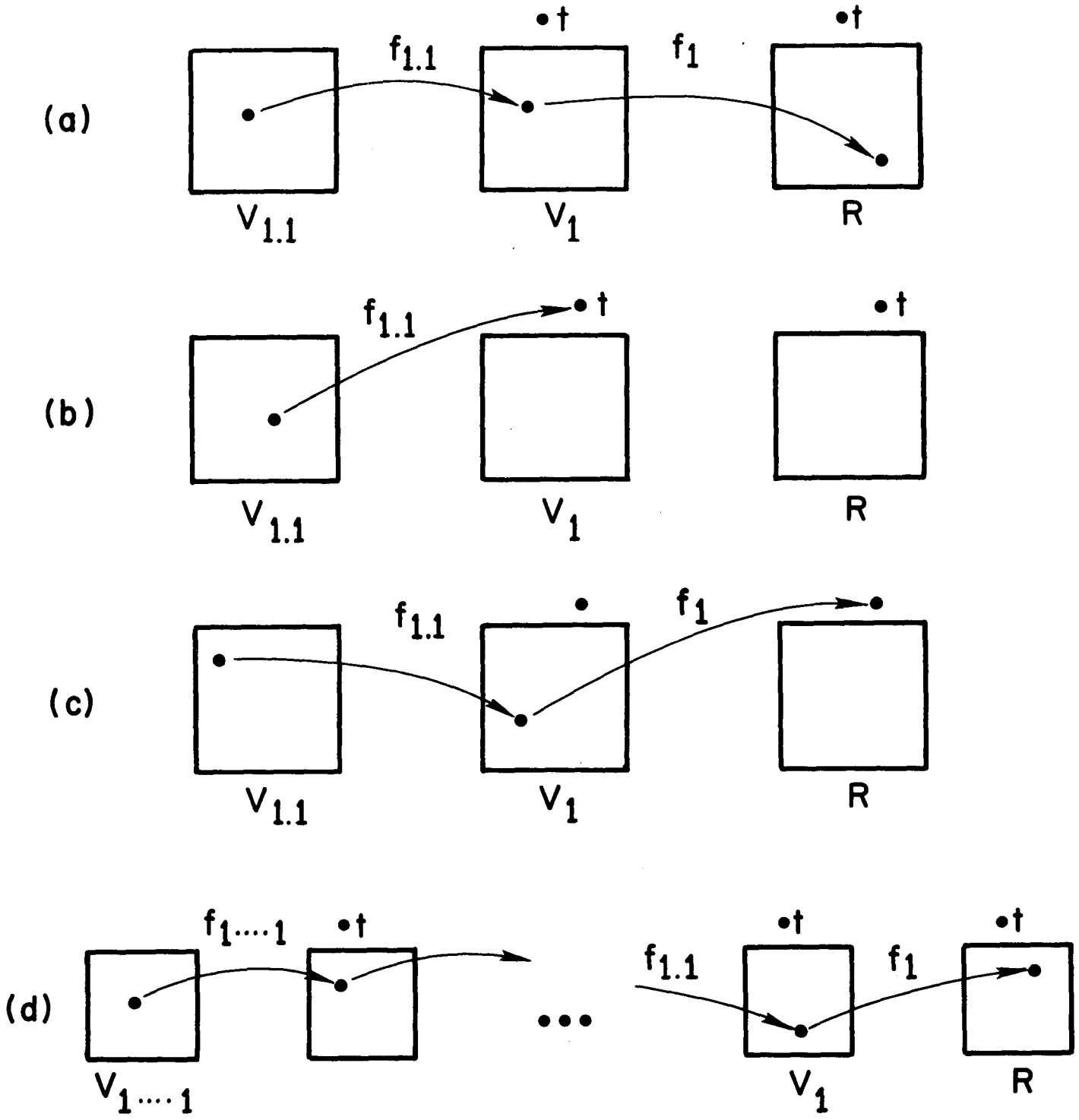


Figure 1 Recursive f-map

The process map ϕ

The model as currently developed represents only the mapping of resources in a computer system. This machinery is sufficient to discuss virtualization of certain mini-computers, e.g., DEC PDP-8, which do not exhibit any local mapping structure. However, most current (third generation) general purpose systems have additional software-visible hardware maps. This additional structure may be as simple as supervisor/problem states (IBM System/360) and relocation-bounds registers (DEC PDP-10 and Honeywell 6000), or as complex as segmentation-paging-rings[21] (Multics - Honeywell 6180). In future fourth generation systems, the maps will likely be even more complex and might feature a formal implementation of the process model[22,23] in hardware-firmware.

The crucial point about each of these hardware (supported) maps is that they are software visible. In certain systems, the visibility extends to non-privileged software[15]. However, in all cases the maps are visible to privileged software[18].

Typically, an operating system on one of these machines will alter the map information before dispatching a user process. The map modification might be as simple as setting the processor mode to problem state or might be as complex as changing the process's address space by switching its segment table. In either case, however, the subsequent execution of the process and access to resources by it will be affected by the current local map.

Therefore, in order to faithfully model the running of processes on a virtual machine, we must introduce the local mapping structure into the model.

We develop a model of the software-visible hardware map by defining the set of process names $P = \{p_0, p_1, \ldots, p_j\}$ to be the set of names addressable by a process executing on the computer system. [Process spaces are always represented as circles in the figures.] Let $R = \{r_0, r_1, \ldots, r_n\}$ be the set of (real) resource names, as before.

Then, for the active process, we provide a way of associating process names with resource names during process execution. To this end, via all of the software visible hardware mapping structure, e.g., supervisor/problem state, segment table, etc., we define, for each moment of time, a function

$$\phi: P \longrightarrow R \cup \{e\}$$

such that if $x \in P$, $y \in R$, then

$$\phi(x) = \begin{cases} y \text{ if } y \text{ is the resource name for process name } x \\ e \text{ if } x \text{ does not have a corresponding resource.} \end{cases}$$

The value $\phi(x) = e$ causes an exception to occur to some exception handling procedure, presumably to a privileged procedure of the operating system on this machine. To avoid confusion with VM-faults (see above), process traps will always be called exceptions.

We call the function ϕ a process map or ϕ-map. The term process map is applied regardless of what form the ϕ-map takes. In future (fourth generation) systems, ϕ might actually represent the firmware implementation of processes, although this is not necessary. The important point about ϕ is that unlike f, which is an inter-level map, ϕ is a local or intra-level map and does not cross a level of resource mapping.

Running a virtual machine: f o ϕ

Running a process on a virtual machine means running a process on a configuration with virtual resources. Thus, if a process $P = \{p_0, p_1, \ldots, p_j\}$ runs on the virtual machine $V = \{v_0, v_1, \ldots, v_m\}$ then

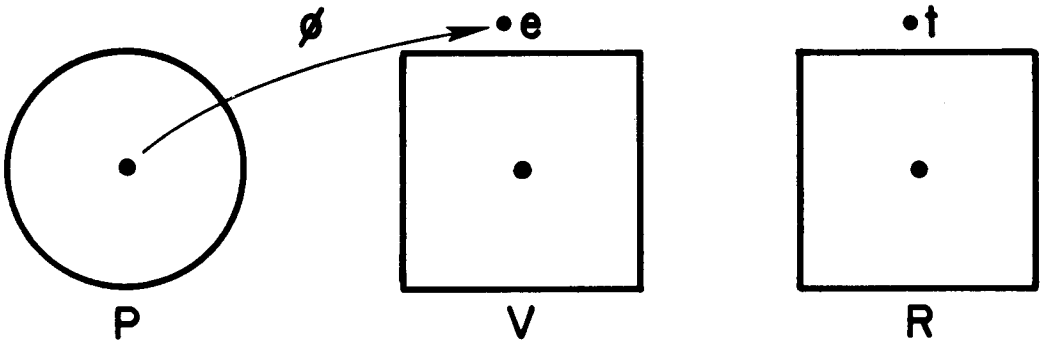
$$\phi: P \longrightarrow V \cup \{e\}$$

as before, with virtual resource names, V, substituted for real ones in the resource range of the map.

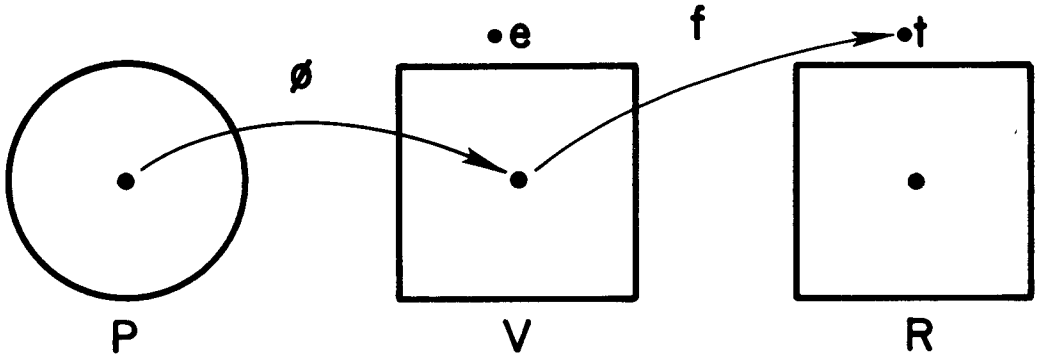
The virtual resource names, in turn, are mapped into their real equivalents by the map, $f: V \longrightarrow R$. Thus, a process name x corresponds to a real resource $f(\phi(x))$. In general, process names are mapped into real resource names under the (composed) map

$$f \circ \phi: P \longrightarrow R \cup \{t\} \cup \{e\}.$$

This (composed) map can fail to take a process name into a real resource name in one of two ways. In the event of a process name exception (Figure 2a), control is given, without VMM knowledge or intervention, to the privileged software of the operating system



(a)



(b)

Figure 2 Process Exception and VM-fault

within the same level. A virtual name fault, however, causes control to pass to a process in a lower level virtual machine, without the operating system's knowledge or intervention (Figure 2b). While this fault handling software in the VMM is not subject to an f-map since it is running on the real machine, it is subject to its ϕ-map just as any other process on the machine.

The ϕ-map may be combined with the recursive f-map result to produce the "general" composed map

$$f_1 \circ f_{1.1} \circ \cdots \circ f_{1.1 \ldots 1.1} \circ \phi.$$

Thus, for virtual machines, regardless of the level of recursion, there is only one application of the ϕ-map followed by n applications of an f-map. This is an important result that comes out of the formalism of distinguishing the f and ϕ maps. Thus, in a system with a complex ϕ-map but with a simple f-map, n-level recursion may be easy and inexpensive to implement.

In the model presented, f-maps map resources of level n+1 into resources of level n. It is equally possible to define an f-map in which resources of level n+1 are mapped into process names of level n (which are then mapped into resource names of level n). This new f-map is called a Type II f-map to distinguish it from the Type I f-map which is discussed in this paper[19,24].

Interpretation of the model

The model is very important for illustrating the existence of two very different basic maps in virtual machines. Previous works have not clearly distinguished the difference or isolated the maps adequately. The key point is that f and ϕ are two totally different maps and serve different functions. There is no a priori requirement that f or ϕ be of a particular form or that there be a fixed relationship between them. The ϕ-map is the interface seen by an executing program whereas the f-map is the interface seen by the resources. In order to add virtual machines to an existing computer system, ϕ is already defined and only f must be added. The choice of whether the f-map is R-B, paging, etc., depends upon how the resources of the virtual machines are to be used. In any case, the f-map must be made recursive whereas ϕ need not be.

If a new machine is being designed, then neither ϕ nor f is yet defined. ϕ may be chosen to idealize the structures seen by the programmer whereas f may be chosen to optimize the utilization of resources in the system. Such a "decoupled" view of system design might lead to systems with ϕ = segmentation and f = paging.

Another intrinsic distinction between the maps is that the f-map supports levels of resource allocation between virtual machines, while the ϕ-map establishes layers (rings, master/slave mode) of privilege within a single virtual machine.

The virtual machine model may be used to analyze and characterize different virtual machines and architectures[19]. As can be seen from Table I, none of the existing or previously proposed systems provides direct support of completely general virtual machines. CP-67 has a non-trivial ϕ-map but no direct hardware support of the f-map; the approach of Lauer and Snow provides direct hardware support of the f-map but has a trivial ϕ-map, i.e., ϕ = identity. Therefore, CP-67 must utilize software plus the layer relationship of the ϕ-map to simulate levels, whereas Lauer and Snow must utilize software plus the level relationship of the f-map to simulate layers.*

The Gagliardi-Goldberg "Venice Proposal" (VP)[18] supports both the layer and level relationships explicitly. However, since the VP does not directly provide hardware support for f (it supports ϕ and f o ϕ), certain software intervention is still required.

In the next section, we shall discuss a design, called the Hardware Virtualizer (HV), which eliminates the weaknesses of the previous designs. As can be seen from Table I, the HV is based directly upon the virtual machine model which we have developed.

*This is not to suggest that the Lauer and Snow approach is inferior. It is only less general in that it will not support modern operating systems running directly on the individual virtual machines.

TABLE I - COMPARISON OF SYSTEMS USING VIRTUAL MACHINE MODEL

SYSTEM	ϕ	f	$\phi_t = f \circ \phi$	$f \circ \phi$ COMPOSER	DIRECT ACCESS TO ϕ PERMITTED?		RECURSION AND RECURSIVE COMPOSER	RECURSIVE VIRTUAL MACHINE STRUCTURE HARDWARE SUPPORTED
					READ	WRITE		
IBM CP-67[3]	Hardware (III generation)	Software	Software support to translated ϕ	Software	No	No	Software composition	---
Gagliardi-Goldberg[18] VP	Hardware (complex IV generation)	Software	Hardware support to store translated ϕ	Hardware	Yes	No	Hardware assisted composition	Tree
Lauer-Snow[16]	---	Hardware (relocation-bounds)	---	---	---	---	Direct hardware static composition	Stack
Lauer-Wyeth[17]	---	Hardware (segmentation, paging)	---	---	---	---	Direct hardware dynamic composition	Stack
Goldberg HV[19]	Hardware (completely arbitrary)	Hardware (completely arbitrary)	Evaluated dynamically	Direct hardware dynamic composi-tion	Yes	Yes	Direct hardware dynamic composition	Tree

HARDWARE VIRTUALIZER (HV)

Despite the value of the virtual machine model in providing insight into existing and proposed systems, perhaps its most important result is that it implies a natural means of implementing virtual machines in all conventional computer systems. Since the f-map and ϕ-map are distinct and (possibly) different in a virtual computer system, they should be represented by independent constructs. When a process running on a virtual machine references a resource via a process name, the required real resource name should be obtained by a dynamic composition of the f-map and ϕ-map at execution time. Furthermore, the result should hold regardless of recursion or the particular form of f and ϕ. We call a hardware-firmware device which implements the above functionality a Hardware Virtualizer (HV). The HV may be conceptually thought of as either an extension to an existing system or an integral part of the design of a new one.

HV design and requirements

The design of a Hardware Virtualizer must consider the following points:

(1) The database to store f

(2) A mechanism to invoke f

(3) The mechanics of map composition

(4) The action on a VM-fault.

In the discussion which follows, we shall develop the basis for a Hardware Virtualizer design somewhat independently of the particular

form of the f-map or ϕ-map under consideration. We assume that the ϕ-map is given (it could be the identity map) and we discuss the additional structure associated with the f-map. Although we shall refer to certain particular f-map structures, such as the R-B or paging form of memory map, the actual detailed examples are postponed until later.

Database to represent f

The VMM at level n must create and maintain a database which represents the f-map relationship between two adjacent levels of virtual machine resources, namely level n + 1 to level n. This database must be stored so that it is invisible to the virtual machine, i.e., level n + 1, including the most privileged software. Let us assume that for economic reason[18] the database must be stored in main memory. Then f may not be in the (virtual) memory of level n + 1, but it must be in the (virtual) memory of level n.

The only requirement on where the f-map is stored in level n memory is that it be possible for the HV to locate it by applying a deterministic algorithm from the beginning (ROOT) of level n memory. The f-maps corresponding to different virtual machines at the same level may be identified either implicitly[16] or explicitly[18]. For explicit identification, we assume a Virtual Machine Table (VMTAB), the ith entry of which points to the Virtual Machine Control Block (VMCB) of virtual machine i (supported at level n). See Figure 3.

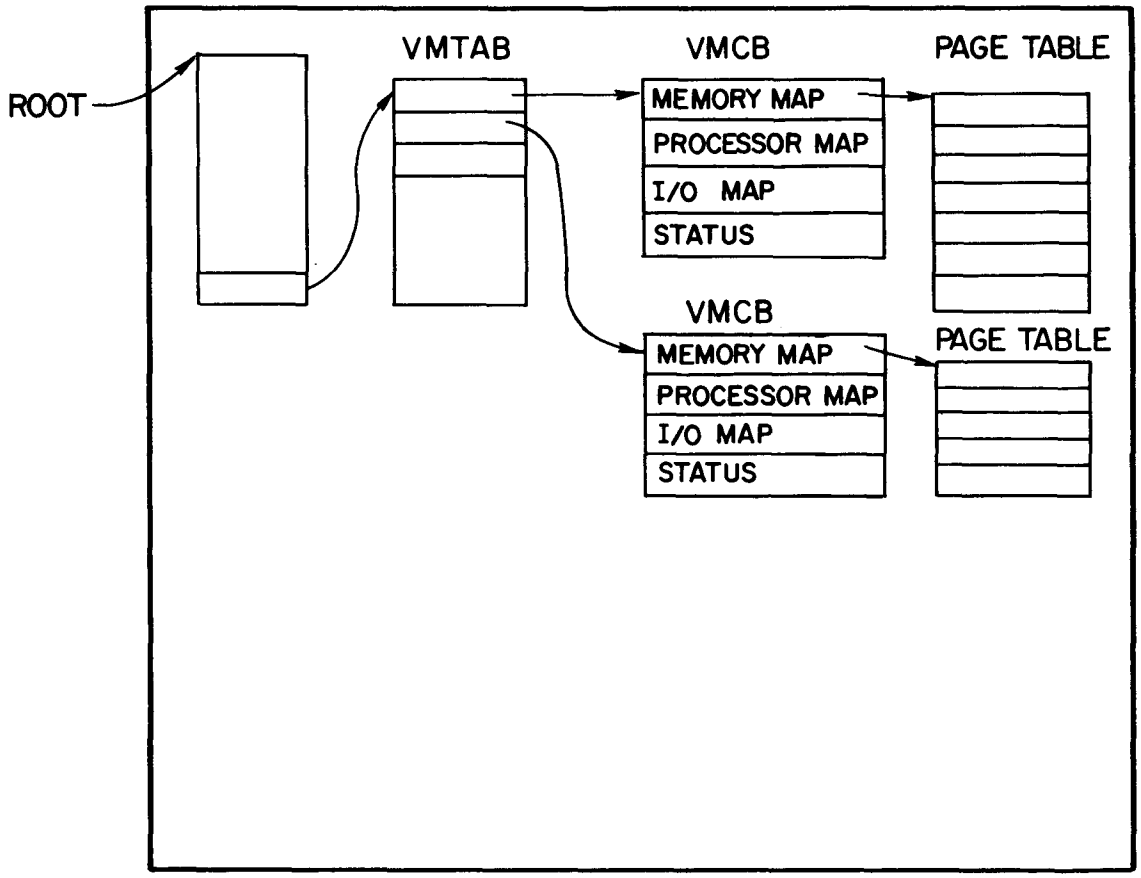


Figure 3 The VMTAB and VMCB's

The VMCB provides the representation of the f-map for the virtual machine. It contains the memory map, processor map, and I/O map. In addition, there may be other status and/or accounting data for the virtual machine.* The specific form of the VMCB is dependent upon the f-map actually used, e.g., R-B, paging, etc.

Additional information possibly kept in the VMCB includes capability information for the virtual processor indicating particular features and instructions, present or absent. These capability bits include, for example, scientific instruction set or virtual machine instruction set (recursion). If recursion is supported, then the VMCB must include sufficient information to automatically restart a higher level virtual machine on a lower level VM-fault (Figure 1c).

Mechanism to invoke f

In order to invoke the f-map, the HV requires an additional register and one instruction for manipulating it. The register is the virtual machine identifier register (VMID) which contains the "tree name" of the virtual machine currently executing. The VMID is a multi-syllabic register, whose syllables identify all of the f-maps which must

*As noted earlier, mapping of I/O and other resources may be treated as a special case of the mapping of memory. Under these circumstances, the VMCB reduces to the memory map component.

be composed together in order to yield a real resource name. The new instruction is LVMID (load VMID) which appends a new syllable to the VMID register. This instruction should more accurately be called append VMID but LVMID is retained for historical reasons.

For the hardware virtualizer design to be successful, the VMID register (and the LVMID instruction) must have four crucial properties[18,19].

(1) The VMID register absolute contents may neither be read nor written by software.

(2) The VMID of the real machine is the null identifier.

(3) Only the LVMID instruction may append syllables to the VMID.

(4) Only a VM-fault (or an instruction which terminates the operation of a virtual machine) may remove syllables from the VMID.

Figure 4 sketches the operation of the LVMID instruction while avoiding implementation details related to a specific choice of map. In the flowchart, we use the VMID as a subscript to indicate the current control block, VMCB [VMID]. Thus SYLLABLE, the operand of the LVMID instruction, is stored in the NEXT_SYLLABLE field of the current VMCB. SYLLABLE is appended to the VMID and this new virtual machine is activated. If the NEXT_SYLLABLE field of the new VMCB is NULL, indicating that this level of machine was not previously active, then the LVMID instruction completes and execution continues within this virtual machine. Otherwise, if it is not null, the lower level was previously active and was suspended due to a VM-fault at a still lower

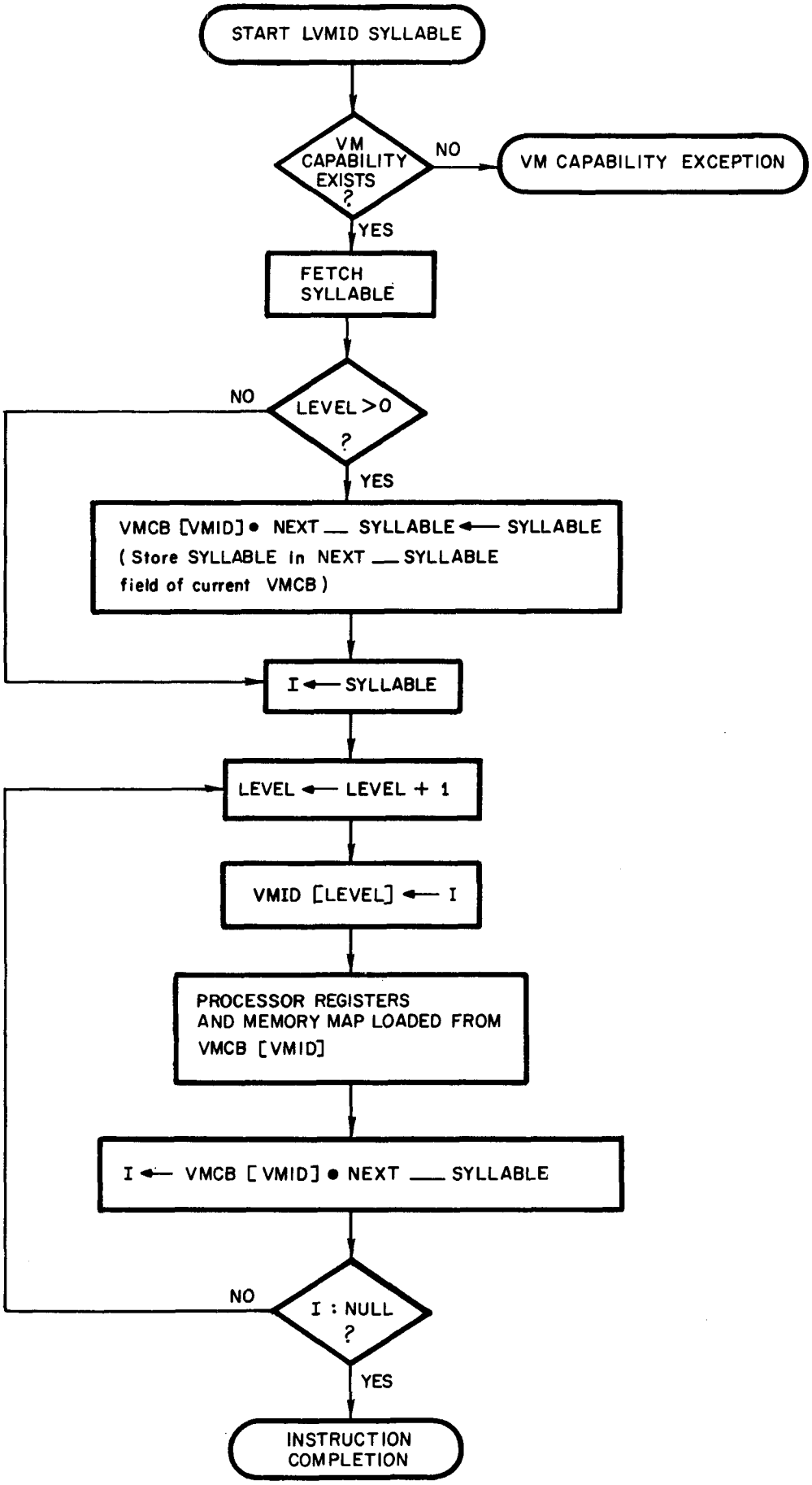


Figure 4 LVMID Instruction

level. In this case, execution of the LVMID instruction continues by appending the NEXT_SYLLABLE field of the new VMCB to the VMID.

Map composer

A map composer is needed to provide the dynamic composition of the ϕ-map (possibly indentity) and the active f-maps on each access to a resource. The ϕ-map is known and the active f-maps, i.e., the VMCB's, are determined from the VMID register. Figure 5 sketches the map composition mechanism while avoiding implementation details related to specific choice of maps. As can be seen, the composer accepts a process name P and develops a real resource name R or causes a VM-fault.

VM-fault

A VM-fault occurs when there does not exist a valid mapping between two adjacent levels of resources. As shown in Figure 5, a VM-fault causes control to be passed to the VMM superior to the level which caused the fault. This is done by removing the appropriate number of syllables from the VMID.

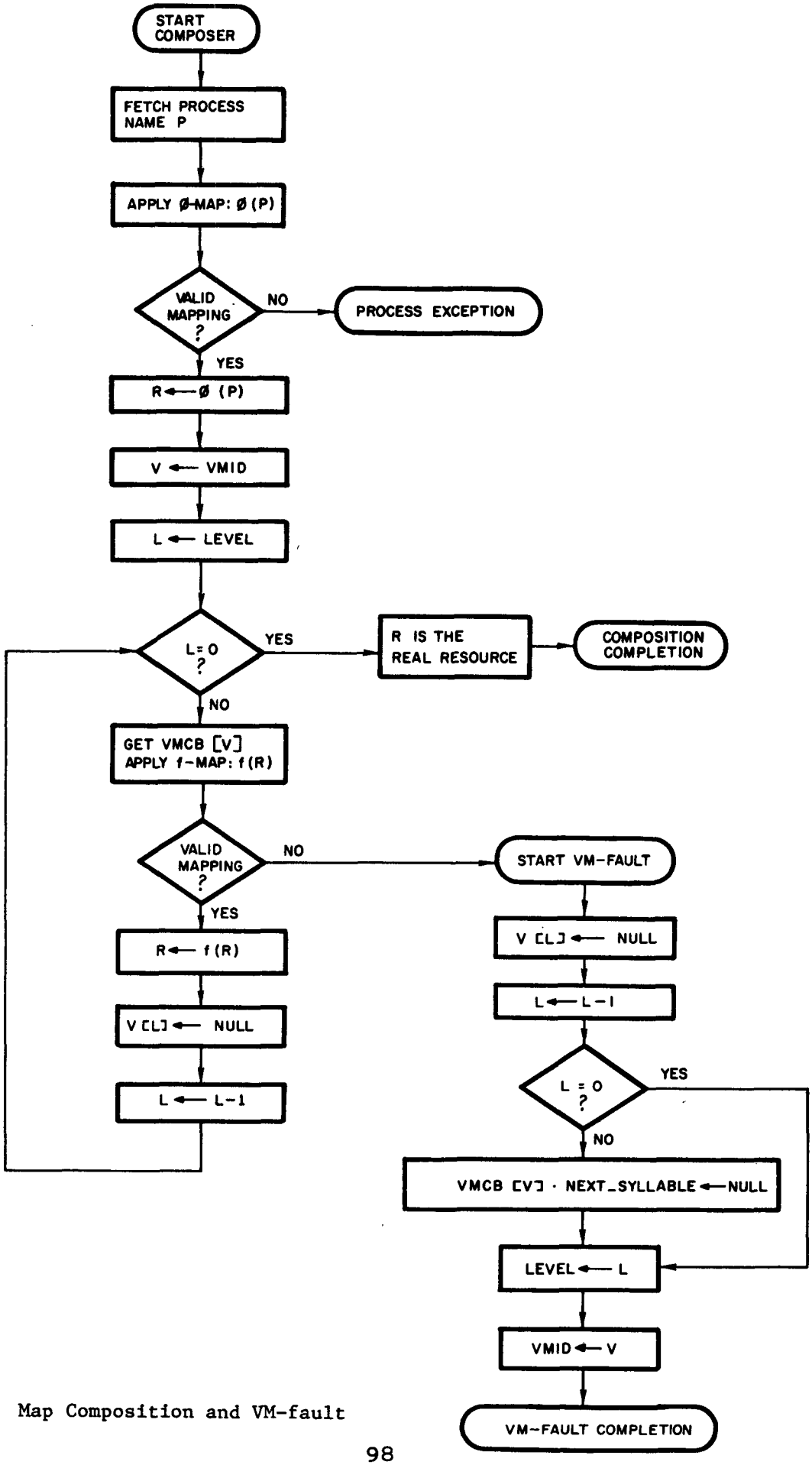


Figure 5 Map Composition and VM-fault

Performance assumptions

The performance of the Hardware Virtualizer depends strongly upon the specific f-map, ϕ-map, and HV implementation technique used. However, there are basic reasons why processes can execute on a virtual machine with efficiency approaching that of the real machine. Most current systems which employ memory mapping (in the ϕ-map) make design assumptions concerning program behavior. We will observe that these assumptions are applicable to virtual machines as well.

From the initial notion of "program locality", Madnick[25] has generalized and identified two specific aspects of locality.

(1) Temporal locality

If the logical addresses $<a_1, a_2, \ldots>$ are referenced during the time interval t-T to t, there is a high probability that these same logical addresses will be referenced during the time interval t to t+T[25].

(2) Spatial locality

If the logical address a is referenced at time t, there is a high probability that a logical address in the range a-A to a+A will be referenced at time t + 1[25].

In modern operating systems, because of the cost to "start up" a process or to change the ϕ-map, it is likely that the scheduler and dispatcher will enforce an additional locality:

(3) Process locality

If the ϕ-map value of the process executing at time t is ϕ_*, then there is a high probability that it will

be ϕ_* at time t + 1.

Virtual machines and the Hardware Virtualizer add a new notion.

(4) Virtual machine locality

If the VMID of the currently executing virtual machine at time t is $x_1.x_2. \ldots .x_{n-1}.x_n$, then there is a high probability that the VMID will be $x_1.x_2. \ldots .x_{n-1}.x_n$ at time t + 1. Furthermore the VMID may change only on a VM-fault or an LVMID instruction.

Combining all of these locality notions, we determine that with proper implementation, multi-level recursive virtual machines need not have significantly different performance from real machines. Another way of phrasing this observation is:

Temporal and spatial locality are name invariant.

Regardless of what a block of memory is called, or how many times it gets renamed (via composed f-maps) there is still an intrinsic probability of reference to it by an executing program. Thus, a virtual machine supported by a map composer and associative store should enjoy comparable performance to the real machine.[19]

If the f-map and ϕ-map are sufficiently simple then the associator may not be needed. For example, if f = R-B, ϕ = identity, then it may be sufficient for the HV to provide "invisible scratchpad registers" to maintain statically composed R-B values which are altered only on a level change[16,19].

If ϕ involves paging or segmentation, then the real machine itself probably required an associator for performance reasons.[26] The HV associator will replace it. If the f-map is simple, e.g., f = R-B, then the HV associator will be very similar; if f includes paging it will be somewhat different. The choice of whether to include the VMID or level as part of the search key of the associator can be made for price-performance reasons.

Interpretation of the HV

As indicated earlier, the Hardware Virtualizer can serve as the central mechanism in the design of a new computer system or as an expansion to an existing computer system. In the latter case, we assume a computer system M with a given ϕ-map. The HV construction, i.e., additional data structures, new instruction (LVMID), VM-fault etc., defines a new machine M' with added functionality. The Hardware Virtualizer guarantees that M' is a recursive virtual machine capable of supporting a hierarchy of M' machines with M machines as terminal nodes where desired.

EXAMPLE OF A HARDWARE VIRTUALIZER

In order to clarify the operation of the Hardware Virtualizer, we demonstrate one example of its use. In the example, we present some features of a typical third generation architecture, indicate the extensions introduced by the Hardware Virtualizer, and then illustrate the execution of some instructions. Many other examples

have been developed in greater detail, including those for very complex (fourth generation) architectures[19] but the principles involved are the same.

Existing Architecture

This example is developed around a canonical third generation computer system, similar to the Honeywell 6000, DEC PDP-10, or IBM System/360. The salient features of the architecture are (1) the privileged/non-privileged mode distinction (master/slave, supervisor/problem, etc.) as part of the instruction counter (IC), (2) a single relocation-bounds register (R-B) whose absolute contents may be loaded in privileged mode, and (3) some fixed locations in main memory where the old and new R-B and IC registers are swapped on a process exception.

To simplify the example we will assume the R-B register is active, even in privileged mode. Furthermore, all instructions will be assumed to be executing in privileged mode. Since mode violations are local process exceptions and are treated identically to R-B violations, there is no need to illustrate them both. The example illustrates execution of central processor instructions only. The extension of the example to include a homogeneous treatment of I/O is possible[17,19] but introduces additional issues of both mechanisms and policies that are best treated in a subsequent paper. Thus, in this example, the R-B map is the ϕ-map.

Extensions to Architecture

The Hardware Virtualizer requires extensions to the third generation architecture. We will illustrate the modifications introduced by the addition of a page f-map (in the memory domain). We will assume 1000-word pages. (See Figure 6.) The modifications include:

(1) database to store f - Some fixed known location, say 0, in the memory of level n points to the virtual machine table (VMTAB) which describes the virtual machines of level n + 1. In this example, each virtual machine control block (VMCB) illustrates a memory map (page table) and a processor map. The processor map includes storage for level n + 1's IC and R-B. Also included but not illustrated is the level n+2 NEXT_SYLLABLE which is stored whenever level n + 1 issues an LVMID instruction.

(2) a mechanism to invoke f - A multi-syllable VMID register and a LVMID instruction are added. When a virtual machine is activated, its IC and R-B are loaded from its control block (VMCB).

(3) a composer - A hardware-firmware composer supported with scratchpad memory and associator (for performance reasons) is added. We do not discuss the details of the implementation.

(4) the action on a VM-fault - The IC and R-B are stored in their VMCB, the appropriate syllable(s) are removed from the VMID and control passes to a fixed known location, say 1, in the VMM.

Note that this example illustrates a Type I f-map in which resources of level n + 1 are mapped into resources of level n. Thus, the relocation-bounds register value of level n does not enter into the mapping. In this example when LVMID is executed,relocation is coincidentally zero, but need not be.

The example

Figure 6 shows the state of main memory in our hypothetical hardware virtualized machine. We show VMCB's together with a number of instructions and data. For purposes of illustration, we assume the existence of a simple instruction, LOAD, that accesses memory. Figure 6 also shows the three registers, VMID, R-B, and IC, but their values are not indicated. Instead, Table II shows six sets of values for VMID, R-B, and IC. For each set, we identify the instruction which is executed and the evaluation sequence used in developing an absolute physical memory address. The table entry

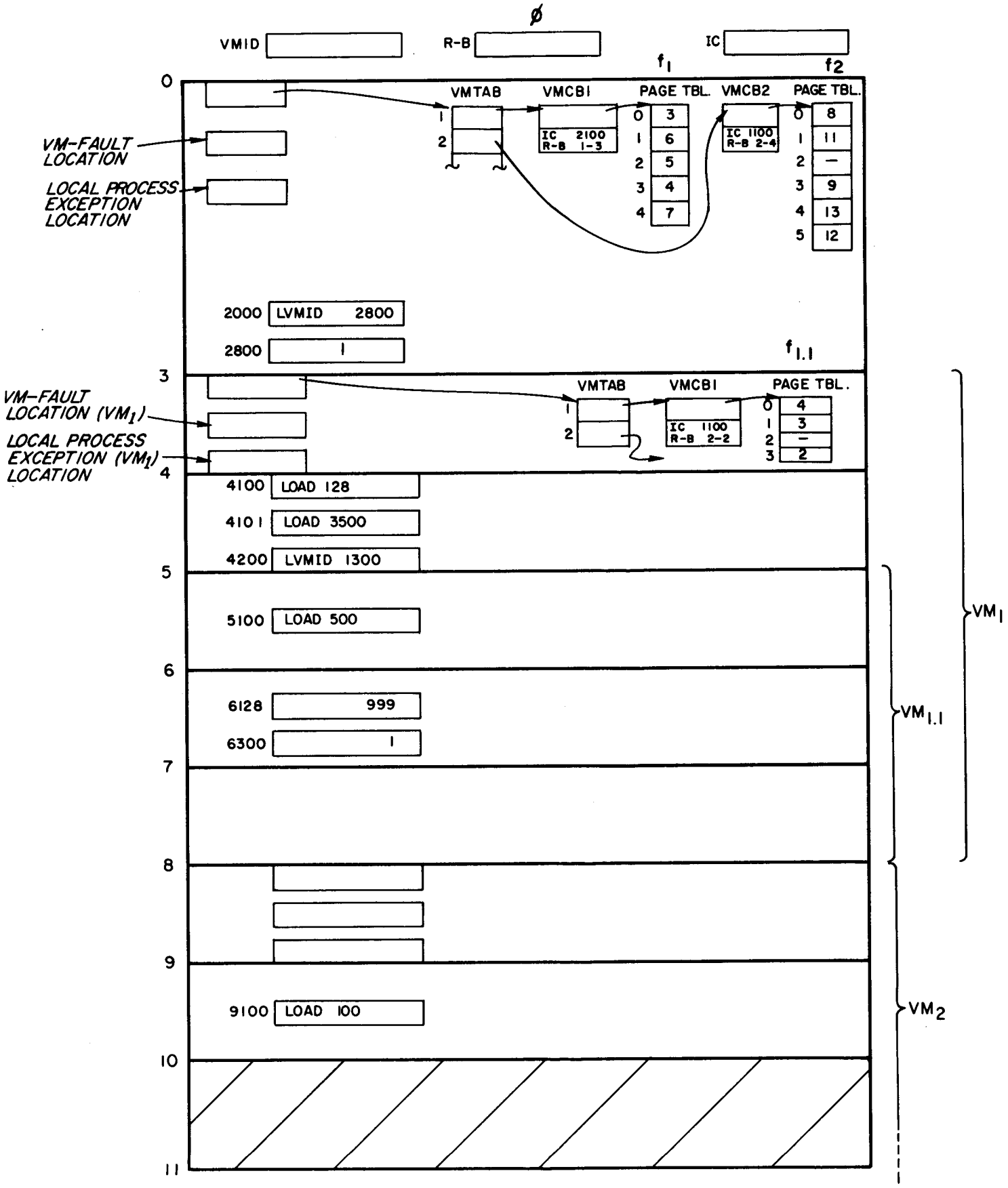


Figure 6 Hardware Virtualizer Example: Main Memory

TABLE II - HARDWARE VIRTUALIZER EXAMPLE: EVALUATION SEQUENCES

LINE	VMID	R-B	IC	EVALUATION SEQUENCE	VMID AFTER EVAL. SEQ.	POINT ILLUSTRATED
1	NULL	0-14	2000	IC is 2000 $\phi(2000) = 2000$ Fetch inst: LVMID 2800 $\phi(2800) = 2800$ Append 1 to VMID	1	LVMID instruction
2	1	1-3	2100	IC is 2100 $\phi(2100) = 3100$ $f_1(3100) = 4100$ Fetch inst: LOAD 128 $\phi(128) = 1128$ $f_1 1128 = 6128$ Load 999	1	Virtual Machine instruction execution
3	1	1-3	2101	IC is 2101 $\phi(2101) = 3101$ $f_1(3101) = 4101$ Fetch inst: LOAD 3500 $\phi(3500) = e$	1	Process exception in virtual machine
4	2	2-4	1100	IC is 1100 $\phi(1100) = 3100$ $f_2(3100) = 9100$ Fetch inst: LOAD 100 $\phi(100) = 2100$ $f_2(2100) = t$	NULL	Virtual Machine fault and different VMID
5	1	0-5	3200	IC is 3200 $\phi(3200) = 3200$ $f_1(3200) = 4200$ Fetch inst: LVMID 1300 $\phi(1300) = 1300$ $f_1(1300) = 6300$ Append 1 to VMID	1.1	LVMID with recursion
6	1.1	2-2	1100	IC is 1100 $\phi(1100) = 3100$ $f_{1.1}(3100) = 2100$ $f_1(2100) = 5100$ Fetch inst: LOAD 500 $\phi(500) = 2500$ $f_{1.1}(2500) = t$	1	Recursive instruction execution and VM-fault

includes indication of a process exception, VM-fault, and any change to the VMID. The R-B register values are represented as r-b where r is the relocation (in thousands of words) and b is the amount of contiguous allocation (in thousands of words).

The six lines of Table II divide into three sets, Lines 1-3, 4, and 5-6. Within these sets, Lines 1-3 execute consecutively and Lines 5-6 also execute consecutively.

Referring to Figure 6 and Table II, let us step through the first several evaluation sequences. In Line 1, we are in the VMM running on the real machine. All control blocks have been set up and it is time to activate virtual machine 1. The instruction counter value is 2000. Since the R-B map is 0-14, we add zero to 2000 and obtain $\phi(2000) = 2000$. The VMID is NULL. Therefore, the resource name 2000 is a real resource and we fetch the instruction at physical location 2000, LVMID 2800. We apply the R-B map to 2800 and eventually fetch 1 which is loaded into the VMID register.

Virtual machine 1 is now activated and its IC and R-B registers are loaded from VMCB1. Thus, IC is now 2100 and R-B is 1-3. Even though the memory of virtual machine 1 is 5000 words (as can be seen from its page table) the R-B register limits this active process to addressing only 3000 words. This limit was presumably set by the operating system of virtual machine 1 because the active process is a standard

(non-monitor) user.

Now we are in Line 2 and the IC is 2100. To apply the ϕ-map, we add 1000, checking that 2100 is less than 3000, and obtain $\phi(2100) = 3100$. Since the VMID is 1, we must apply f_1 to map the virtual resource 3100 to its real equivalent. The page table, pointed at by VMCB1, indicates that virtual page 3 is at location 4000. Therefore, $f_1(3100) = 4100$ and the LOAD 128 instruction is fetched.

The other sequences may be evaluated in the same manner. Line 3 illustrates a process exception to the local exception handler of VM1, Line 5 illustrates activation of recursion, and Lines 4 and 6 illustrate VM-faults to the fault handler of their respective VMMs.

It should be noted that we have added a paged f-map which is invisible to software at level n. The pre-existing R-B ϕ-map remains visible at level n. Thus, operating systems which are aware of the R-B map but unaware of the page map may be run on the virtual machine without any alterations.

Note that the addition of an R-B f-map instead of the paged f-map is possible. This new R-B f-map would be distinct from and an addition to the existing R-B ϕ-map; it would also have to satisfy the recursion properties of f-maps[19]. Similarly, a paged f-map added to a machine such as the IBM 360/67 would be distinct from the existing paged ϕ-map.

CONCLUSION

In this paper we have developed a model which represents the addressing of resources by processes executing on a virtual machine. The model distinguishes two maps: (1) the ϕ-map which maps process names into resource names, and (2) the f-map which maps virtual resource names into real resource names. The ϕ-map is an intra-level map, visible to (at least) the privileged software of a given virtual machine and expressing a relationship within a single level. The f-map is an inter-level map, invisible to all software of the virtual machine and establishing a relationship between the resources of two adjacent levels of virtual machines. Thus, running a process on a virtual machine consists of running it under the composed map $f \circ \phi$.

Application of the model provides a description and interpretation of previous virtual machine designs. However, the most important result is the Hardware Virtualizer which emerges as the natural implementation of the virtual machine model. The Hardware Virtualizer design handles all process exceptions directly within the executing virtual machine without software intervention. All resource faults (VM-faults) generated by a virtual machine are directed to the appropriate virtual machine monitor without the knowledge of processes on the virtual machine (regardless of the level of recursion).

A number of virtual machine problems, both theoretical and practical must still be solved. However, the virtual machine model and the Hardware Virtualizer should provide a firm foundation for subsequent work in the field.

ACKNOWLEDGMENTS

The author would like to thank his colleagues at both MIT and Harvard for the numerous discussions about virtual machines over the years. Special thanks are due to Dr. U. O. Gagliardi who supervised the author's Ph.D. research. In particular, it was Dr. Gagliardi who first suggested the notion of a nested virtual machine fault structure and associated virtual machine identifier (VMID) register functionality.

REFERENCES

1 J P BUZEN U O GAGLIARDI
The evolution of virtual machine architecture
Proceedings AFIPS National Computer Conference 1973

2 M BERTHAUD M JACOLIN P POTIN H SAVARY
Coupling virtual machines and system construction
Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems Cambridge Massachusetts 1973

3 R A MEYER L H SEAWRIGHT
A virtual machine time-sharing system
IBM Systems Journal Vol 9 No 3 1970

4 R P PARMELEE
Virtual machines: some unexpected applications
Proceedings IEEE International Computer Society Conference Boston Massachusetts 1971

5 J M WINETT
Virtual machines for developing systems software
Proceedings IEEE International Computer Society Conference
Boston Massachusetts 1971

6 V CASAROSA C PAOLI
VHM: a virtual hardware monitor
Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer
Systems Cambridge Massachusetts 1973

7 D D KEEFE
Hierarchical control programs for systems evaluation
IBM Systems Journal Vol 7 No 2 1968

8 J P BUZEN P P CHEN R P GOLDBERG
Virtual machine techniques for improving software reliability
Proceedings IEEE Symposium on Computer Software Reliability New York 1973

9 C R ATTANASIO
Virtual machines and data security
Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer
Systems Cambridge Massachusetts 1973

10 S E MADNICK J J DONOVAN
Virtual machine approach to information system security and isolation
Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer
Systems Cambridge Massachusetts 1973

11 R ADAIR R U BAYLES L W COMEAU R J CREASY
A virtual machine system for the 360/40
IBM Cambridge Scientific Center Report No G320-2007 1966

12 R J SRODAWA L A BATES
An efficient virtual machine implementation
Proceedings AFIPS National Computer Conference 1973

13 K FUCHI H TANAKA Y NAMAGO T YUBA
A program simulator by partial interpretation
Proceeding ACM SIGOPS Second Symposium on Operating
Systems Principles Princeton New Jersey 1969

14 IBM CORPORATION
IBM virtual machine facility/370: planning guide
Publication Number GC20-1801-0 1972

15 R P GOLDBERG
Hardware requirements for virtual machine systems
Proceedings Hawaii International Conference on System Sciences
Honolulu Hawaii 1971

16 H C LAUER C R SNOW
Is supervisor-state necessary?
Proceedings ACM AICA International Computing Symposium Venice Italy 1972

17 H C LAUER D WYETH
A recursive virtual machine architecture
Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems Cambridge Massachusetts 1973

18 U O GAGLIARDI R P GOLDBERG
Virtualizable architectures
Proceedings ACM AICA International Computing Symposium Venice Italy 1972

19 R P GOLDBERG
Architectural principles for virtual computer systems
Ph.D. Thesis Division of Engineering and Applied Physics Harvard University Cambridge Massachusetts 1972

20 R P GOLDBERG
Virtual machine systems
MIT Lincoln Laboratory Report No MS-2687 (also 28L-0036) Lexington Massachusetts 1969

21 M D SCHROEDER J H SALTZER
A hardware architecture for implementing protection rings
Communications of the ACM Vol 15 No 3 1972

22 INFOTECH
The fourth generation
Maidenhead, England 1972

23 B H LISKOV
The design of the VENUS operating system
Communications of the ACM Vol 15 No 3 1972

24 R P GOLDBERG
Virtual Machines: semantics and examples
Proceedings IEEE International Computer Society Conference Boston Massachusetts 1971

25 S E MADNICK
Storage hierarchy systems
Ph.D. Thesis Department of Electrical Engineering MIT Cambridge Massachusetts 1972

26 M D SCHROEDER
Performance of the GE-645 associative memory while Multics is in operation
Proceedings ACM SIGOPS Workshop on System Performance Evaluation Cambridge Massachusetts 1971