19. Fabry, R.S. A user's view of capabilities. *ICR Quart. Rep. 15* (Nov. 1967), ICR, U. of Chicago, Sec. 1C.
20. Fabry, R.S. Preliminary description of a supervisor for a machine oriented around capabilities. *ICR Quart. Rep. 18* (Aug. 1968), ICR, U. of Chicago, Sec. 1B.
21. Fabry, R.S. List-structured addressing. Ph.D. Th., U. of Chicago, 1971.
22. Feustal, E.A. The Rice research computer—a tagged architecture. Proc. AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J. pp. 369–377.
23. Feustal, E.A. On the advantages of tagged architecture. *IEEE Trans. on Computers C-22*, 7 (July 1973), 644–656.
24. Graham, G.S., and Denning, P.J. Protection—principles and practice. Proc. AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., pp. 417–429.
25. Halton, D. Hardware of the System 250 for communication control. Presented at the Internat. Switching Symp., Cambridge, Mass., June 6–9, 1972, 7 pp.
26. Hamer-Hodges, K.J. Fault resistance and recovery within System 250. Presented at I.C.C. Conf., Washington, D.C., Oct. 1972, 6 pp.
27. Iliffe, J.K. *Basic machine principles.* American Elsevier, New York, 1968.
28. Iliffe, J.K., and Jodeit, J.G. A dynamic storage allocation scheme. *Comput. J. 5* (Oct. 1962), 200–209.
29. Jones, A.K. Protection structures. Ph.D. Th., Carnegie-Mellon U., 1973.
30. Lampson, B.W. On reliable and extendable operating systems. In Techniques in Software Engineering, NATO Science Committee Workshop Material, Vol. II, Sept. 1969.
31. Lampson, B.W. Dynamic protection structures. Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Montvale, N.J., pp. 27–38.
32. Lampson, B.W. Protection. Proc. 5th Ann. Princeton Conf., Princeton U., Mar. 1971, pp. 437–443.
33. LeClerc, J.Y. Memory structures for interactive computers. Project GENIE document No. 40.10.110, U. of California, Berkeley, 1966.
34. Needham, R.M. Protection systems and protection implementations. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 571–578.
35. Organick, E.I. *Computer System Organization—the B5700 B6700 Series.* Academic Press, New York, 1973.
36. Organick, E.I. *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, Mass., 1972.
37. Saltzer, J.H. Traffic control in a multiplexed computer system. MAC-TR-30, Proj. MAC, MIT, Cambridge, Mass., 1966.
38. Schroeder, M.D. Performance of the GE-645 associative memory while Multics is in operation. Proc. Workshop on System Performance Evaluation, Cambridge, Mass., 1971, pp. 227–245.
39. Schroeder, M.D. Cooperation of mutually suspicious subsystems in a computer utility. Ph.D. Th., MIT, 1972.
40. Sevick, K.C., et al. Project SUE as a learning experience. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N. J., pp. 331–339.
41. Shepherd, J. Principal design features of the multi-computer. (The Chicago Magic Number Computer). *ICR Quart. Rep. 19* (Nov. 1968), ICR, U. of Chicago, Sec. 1-C.
42. Sturgis, H.E. A postmortem of a time sharing system. Ph.D. Th., U. of California, Berkeley, 1973.
43. Wilkes, M.V. *Time Sharing Computer Systems.* 2nd ed., American Elsevier, New York, 1972.
44. Wilner, W.T. Design of the Burroughs B1700. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 489–497.
45. Wilner, W.T. Burroughs B1700 memory utilization. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 579–586.
46. Wulf, W.A., et al. HYDRA: The kernel of a multiprocessor operating system. Carnegie Mellon U., Comput. Sci. Dep. rep., June 1973.
47. Yngve, V.H. The Chicago Magic Number Computer. *ICR Quart. Rep. 18* (Nov. 1968), ICR, U. of Chicago, Sec. 1-B.

Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

Key Words and Phrases: operating system, third generation architecture, sensitive instruction, formal requirements, abstract model, proof, virtual machine, virtual memory, hypervisor, virtual machine monitor
CR Categories: 4.32, 4.35, 5.21, 5.22

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973.
This research was supported in part by the U.S. Atomic Energy Commission, Contract No. AT(11-1) Gen 10, Project 14 and in part by the Electronic Systems Division, U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract Number F19628-70-0217.

Authors' addresses: Gerald J. Popek, Computer Science Department, University of California, Los Angeles CA 90024; Robert P. Goldberg, Honeywell Information Systems, Waltham, MA 02154.

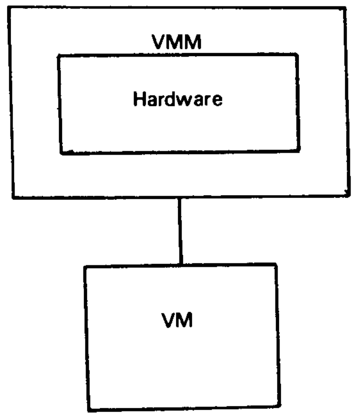
1. Virtual Machine Concepts

There are currently a number of viewpoints suggesting what a virtual machine is, how it ought to be constructed, and what hardware and operating system implications result [1, 6, 7, 9, 12]. This paper examines computer architectures of third-generation-like machines and demonstrates a simple condition which may be tested to determine whether an architecture can support a virtual machine. This condition may also be employed in machine design. In the following, we specify intuitively what is meant by the above, then develop a more exact model of third-generation-like machines, and finally state and prove a sufficient condition for such a system to be virtualizable.

A virtual machine is taken to be an *efficient, isolated duplicate* of the real machine. We explain these notions through the idea of a *virtual machine monitor* (VMM). See Figure 1. As a piece of software a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.

By an "essentially identical" environment, the first characteristic, is meant the following. Any program run under the VMM should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly, with the possible exception of differences caused by the availability of system resources and differences caused by timing dependencies. The latter qualification is required because of the intervening level of software and because of the effect of any other virtual machines concurrently existing on the same hardware. The former qualification arises, for example, from the desire to include in our definition the ability to have varying amounts of memory made available by the virtual machine monitor. The identical environment requirement excludes the behavior of the usual time-sharing operating system from being classed as a virtual machine monitor.

Fig. 1. The virtual machine monitor.



The second characteristic of a virtual machine monitor is efficiency. It demands that a statistically dominant subset of the virtual processor's instructions be executed directly by the real processor, with no software intervention by the VMM. This statement rules out traditional emulators and complete software interpreters (simulators) from the virtual machine umbrella.

The third characteristic, resource control, labels as resources the usual items such as memory, peripherals, and the like, although not necessarily processor activity. The VMM is said to have complete control of these resources if (1) it is not possible for a program running under it in the created environment to access any resource not explicitly allocated to it, and (2) it is possible under certain circumstances for the VMM to regain control of resources already allocated.

A *virtual machine* is the environment created by the virtual machine monitor. This definition is intended not only to reflect generally accepted notions of virtual machines, but also to provide a reasonable environment for a proof.

Before going on to specify a machine model, it is worth pointing out several implications of the definition. First, a VMM as defined is not necessarily a time-sharing system, although it may be. However, the identical-effect requirement applies regardless of any other activity on the real computer, so that isolation, in the sense of protection of the virtual machine environment, *is* meant to be implied. This requirement also distinguishes the virtual machine concept from virtual memory. Virtual memory is just one possible ingredient in a virtual machine; and techniques such as segmentation and paging are often used to provide virtual memory. The virtual machine effectively has a virtual processor, too, and possibly other virtual devices.

We now describe a more formal specification of a third-generation-like computer and a virtual machine monitor before stating and demonstrating the sufficient conditions that the computer must fulfill in order to host a VMM.

2. A Model of Third Generation Machines

The picture described below is intended to reflect a simplified version of a conventional third generation machine, such as the IBM 360, Honeywell 6000, or DEC PDP-10, with a processor and linear, uniformly addressable memory. For the purposes of the formal part of this paper, we assume that I/O instructions and interrupts don't exist, although they may be added as extensions.

The computer is presented by stating several necessary assumptions about its behavior, describing its state-space, and specifying how changes of state may occur.

The processor is a conventional one with two modes of operation, supervisor and user. In supervisor mode, the complete instruction repertoire is available to the precessor. In user mode, it is not. Memory addressing is done relative to the contents of a relocation register. The instruction set consists of the usual complement of instructions for doing arithmetic, testing, branching, moving data in memory, and the like. In particular, with these instructions, it is possible to perform a table lookup on a table of arbitrary size, key, and value, and having obtained the value, move it anywhere in memory (the *table look-up and copy* property).[1]

The machine can exist in any one of a finite number of states where each state has four components: executable storage E, processor mode M, program counter P, and relocation-bounds register R.

$S = \langle E, M, P, R \rangle$

Executable storage is a conventional word or byte addressed memory of size q. The notation $E[i]$ will refer to the contents of the ith unit of storage in E, i.e. $E = E'$ if and only if $E[i] = E'[i]$ for any $0 \leq i < q$. The relocation-bounds register, $R = (l, b)$ is always active, regardless of the machine's current mode. The relocation part l of the register gives an absolute address, which will correspond to the apparent address 0. The bounds part b will give the absolute size (not the largest valid address) of the virtual memory. If it is desired to access all of memory, the relocation must be set to 0 and the bounds to $q - 1$.

If an instruction produces the address a, the address development is as follows:

```
**if** $a + l \geq q$ **then** *memorytrap* **else**
**if** $a \geq b$ **then** *memorytrap*
**else** *use* $E[a + l]$.
```

The meaning of "memorytrap" used here will be discussed in detail in the next section.

The mode M of the processor is either s or u, supervisor or user. The program counter P is an address relative to the contents of R, which acts as an index into E, indicating the next instruction to be executed. Note that the state S is intended to specify the current state of the real computer system, not some portion of it, or some virtual machine.

The contents of the triplet $\langle M, P, R \rangle$ are often referred to as the *program status word*, or PSW. To make our proof easier, we will assume that a PSW can be recorded in one storage location. This restriction can be easily removed. We shall have occasion to use $E[0]$ and $E[1]$ to store an *old*-PSW and fetch a *new*-PSW respectively.

Each component of S can take on only a finite number of values. Call the finite set of states C.

Then an *instruction* i is a function from C to C. $i : C \rightarrow C$. So, for example, $i(S_1) = S_2$, or $i(E_1, M_1, P_1, R_1) = (E_2, M_2, P_2, R_2)$.

So far, this specification of a conventional third generation computer should not be too surprising. After superficial complexities in such systems are removed, what remains is generally a primitive protection system built around a supervisor/user mode concept, and a simple memory allocation system built around a relocation-bounds system. In this model, for simplicity, we have departed slightly from most common relocation systems by assuming it to be active in the supervisor as well as user mode. This difference will not be important to the proof of our result. Note also that *all* references made by the processor to memory are assumed to be relocated.

One key restriction in the model is the exclusion of I/O devices and instructions. While it is commonplace now to provide users with an extended *software* machine without explicit I/O devices or instructions, there is one late third generation *hardware* machine that exhibits this appearance. In the DEC PDP-11, I/O devices are treated as memory cells and I/O operations are performed by doing the proper memory transfer to the appropriate cell.

Traps

We continue with the model of the third generation machine by defining the action of a *trap*. An instruction i is said to *trap* if $i(E_1,M_1,P_1,R_1) = (E_2,M_2,P_2,R_2)$ where

$$\begin{aligned} E_2[j] &= E_1[j], \quad \text{for } 0 < j < q, \\ E_2[0] &= (M_1,P_1,R_1) \\ (M_2, P_2, R_2) &= E_1[1]. \end{aligned}$$

Hence, when an instruction traps, storage is left unchanged, except for location zero in which is put the PSW that was in effect just before the instruction trapped. The PSW to be in effect after the instruction trapped is taken from location one. In the software of most third generation machines, one expects that $M_2 = s$ and $R_2 = (0,q-1)$.

Intuitively, a trap automatically saves the current state of the machine and passes control of a prespecified routine by changing the processor mode, the relocation bounds register, and the program counter to the values specified in $E_1[1]$. Our definition could be relaxed to include cases in which the trap does not *block* the instruction but rather gains control immediately afterward or even some number of instructions later, providing that the state of the machine is stored in such a way as to be reversible to the point at which the instruction causing the trap was about to be executed.

It will be convenient to have defined several particular varieties of traps. One such is a memory trap. A *memory trap* is a trap caused as a result of an attempt by an instruction to develop an address which is greater than the bounds in R or physical memory. From above, the micro-sequence would be

```
**if** $a + l \geq q$ **then** *trap*;
**if** $a \geq b$ **then** *trap*
```

3. Instruction Behavior

In the following, we classify instructions on the basis of their behavior as a function of the state S of the machine. Which groups an instruction falls into will determine whether the real machine is virtualizable.

Instruction i is *privileged* if and only if for any pair of states $S_1 = \langle e, s, p, r\rangle$ and $S_2 = \langle e, u, p, r\rangle$ in which $i(S_1)$ and $i(S_2)$ do not memory trap: $i(S_2)$ traps and $i(S_1)$ does not.

The states S_1 and S_2 differ only in that the mode of S_1 is supervisor and the mode of S_2 is user. The trap that occurs under these conditions is often called a privileged instruction trap.

This notion of a privileged instruction is close to the conventional one. Privileged instructions are independent of the virtualization process. They are merely characteristics of the machine which may be determined from reading the principles of operation. Note, however, that the way we have defined privileged instructions *requires* them to *trap*. Merely NOPing the instruction without trapping is insufficient. The latter case should not be called a privileged instruction; maybe "user mode NOP" would be more accurate.

Examples of privileged instructions in common third generation machines:

```
(1) if $M = s$ then load_PSW      IBM System/360 LPSW
          else trap;
(2) if $M = s$ then load_R        {Honeywell 6000 LBAR,
          else trap;               {DEC PDP-10 DATAO APR
```

Another important group of instructions will be called *sensitive instructions* [4]. The members of this group will have a major bearing on the virtualizability of a particular machine. We define two types of sensitive instructions.

An instruction i is *control sensitive* if there exists a state $S_1 = \langle e_1, m_1, p_1, r_1\rangle$, and $i(S_1) = S_2 = \langle e_2, m_2, p_2, r_2\rangle$ such that $i(S_1)$ does not memory trap, and either:

it was mentioned that complete control over system resources was required. Control sensitive instructions are those which affect, or potentially affect, that control. In this simplified view of third generation machines, the only resource is memory.[3]

Second, ours is a simplified machine. There are no isolated condition codes or other complications by which instructions can interact, other than through the contents of the PSW. For actual machines on which instructions such as ADD or DIVIDE trap on exception conditions, the definition of control sensitivity should exclude those traps as well as memory traps.

In order to describe a second variety of sensitivity, we first introduce a bit of notation. Earlier, the relocation-bounds register with values $r = (l,b)$ was defined. For x an integer, we define an operator $\oplus$, such that $r' = r \oplus x = (l+x,b)$. The relocation register has had its base value shifted by the value of x.

At this point, we note that the only part of memory that can be accessed from a particular state is that specified by the relocation-bounds register R. So for the purposes of examining the effect of an instruction, we can just as well include in the state description only that portion of memory to which we are restricted by R. The notation $E \mid R$ will mean the contents of that part of memory. For $r = (l,b)$, $E \mid r$ stands for the contents of that section of memory from location l to location $l + b$. So, for example, we might essentially specify a state by the notation $S = \langle e \mid r, m, p, r\rangle$.[4] What then does $E \mid r \oplus x$ mean? Combining the two pieces of notation, it represents the contents of that section of memory from $[l + x]$ to $[l + b + x]$.

Then to say that $E \mid r = E' \mid r \oplus x$ means that for $0 \leq i < b$, $E[l + i] = E'[l + x + i]$.

Intuitively, we are getting ready to describe conditions akin to those which occur when programs are moved about in executable storage.

After this unfortunately notation-laden tangent, we

[1] This property will be used in the proof.

[2] Certain machines may have instructions that can store old and new PSWs directly; that is, reference e[0] or e[1], regardless of the values in the relocation register R. In that case, one might wish to add to the two control sensitivity conditions a third one: that $e_1[i] \neq e_2[i]$ for i = 0,1.

[3] We do not treat the processor as a resource. In its simplest form, the virtual machine concept does not require multiprogramming or time-sharing, so that it is not necessary to control allocation of the processor. In most practical systems, however, this assumption is not accurate, so that when I/O is introduced, it will

are now ready for a definition of the second kind of sensitive instruction. An instruction i is *behavior sensitive*[5] if there exist an integer x and states:

(a) $S_1 = \langle e \mid r, m_1, p, r \rangle$, and
(b) $S_2 = \langle e \mid r \oplus x, m_2, p, r \oplus x \rangle$,

where

(c) $i(S_1) = \langle e_1 \mid r, m_1, p_1, r \rangle$,
(d) $i(S_2) = \langle e_2 \mid r \oplus x, m_2, p_2, r \oplus x \rangle$, and
(e) neither $i(S_1)$ or $i(S_2)$ memorytrap,

such that either

(a) $e_1 \mid r \neq e_2 \mid r \oplus x$, or
(b) $p_1 \neq p_2$, or both.

Intuitively, an instruction is behavior sensitive if the effect of its execution depends on the value of the relocation-bounds register, i.e. upon its location in real memory, or on the mode. The other two cases, where the location-bounds register or the modes do not match after the instruction is executed, fall into the class of control sensitive instructions.

In our model, there are really two kinds of behavior sensitivity. In one case, which might be called location sensitivity, an instruction's execution behavior depends on its location in real memory. In the other, an instruction's behavior is affected by the machine's mode.

Example of behavior sensitive instructions:

Location sensitive—load physical address (IBM 360/67 LRA).

Mode sensitive—move from previous instruction space (DEC PDP-11/45 MFPI). (This instruction forms its effective address from information that depends on the current mode.)

By definition, we shall say that an instruction i is *sensitive* if it is either control sensitive or behavior sensitive. If i is not sensitive, then it is *innocuous*.

Now that we have classified instructions, we need to specify the virtual machine monitor more exactly.

4. The Virtual Machine Monitor

The virtual machine monitor will be a particular piece of software, which we shall call a *control program*, that exhibits certain properties. That program consists of several modules. The necessary properties of those modules are presented. It will then be demonstrated that a control program which meets the stated properties can be constructed for third-generation-like machines whose instruction set fulfills one particular constraint.

The control program modules fall into three groups which we present fairly informally. First is a *dispatcher* D. Its initial instruction is placed at the location to which the hardware traps: the value of P in location 1. Note that although not included in our simple trap definition, certain machines trap to one of several locations depending on the type of trap. Such behavior causes no real difficulty since there may be several "first" instructions (entry points) to the dispatcher.

The dispatcher can be considered as the top level control module of the control program. It decides which module to call. It may invoke one from either the second or third set of modules.

The second set in this skeletal specification has one member, an *allocator* A. It is the allocator's task to decide what system resource(s) are to be provided. In the case of a single VM, the allocator needs only to keep the VM and the VMM separate. In the case of a virtual machine monitor which hosts several VMs, it is also the allocator's task to avoid giving the same resource (such as part of memory) to more than one VM concurrently. It is assumed that any usual third-generation-like machine has the capabilities to build an allocator with the appropriate resource tables, etc.

The allocator will be invoked by the dispatcher whenever an attempted execution of a privileged instruction in a virtual machine environment occurs which would have the effect of changing the machine resources associated with that environment. Attempting to reset the R (relocation-bounds) register is the primary example in our skeletal model. If the processor were to be treated as a resource, a halt would be another.

The third set of modules in the control program can be thought of as *interpreters* for all of the other instructions which trap, one interpreter routine per privileged instruction. The purpose of each such routine is to simulate the effect of the instruction which trapped. To specify further, recall that in our current notation, $i(S_1) = S_2$ means that state S_1 is mapped into state S_2 by the instruction i. We will agree that $ij(S_1) = S_2$ means that there exists a state S_3 such that $i(S_1) = S_3$ and $j(S_3) = S_2$. The meaning of a sequence of instructions $ij \cdots k(S_1)$ should then be clear.

Let v_i represent such a series of instructions. Then we may represent the set of interpretive routines as a set of v_i, indicated notationally as $\{v_i\}$, $i = 1$ to m, where m is the number of privileged instructions. Of course the dispatcher and allocator are also sequences of instructions.

A control program is thus specified by its three parts: $CP = \langle D, A, \{v_i\} \rangle$.

The only control programs of interest to us will be those which satisfy the properties we are about to discuss. Since it is a fairly common practice in actual systems, we assume, for simplicity, that the control program will run in supervisor mode. That is, the PSW in location 1, which is loaded by hardware when a trap occurs, has mode set to *supervisor* and program

[5] The results of this paper are still true if the definition of behavior sensitivity is restricted to the cases where $m_2 \neq s$. Changes in instruction behavior due to relocation in supervisor mode does not affect virtual machine code, since that code is run in user mode.
[6] See for example [6, pp. 108–113] for a discussion of other alternatives to these assumptions.

counter set to the first location of the dispatcher. Furthermore, we will agree that *all* other programs will run in user mode.[6] That is, the PSW, which the control program loads as its last operation, turning control back to the running program, will have its mode set to *user*. Hence it will be necessary that one location in the control program be used to record the simulated mode of the virtual machine.

5. The Virtual Machine Properties

There are three properties of interest when any arbitrary program is run while the control program is resident: efficiency, resource control, and equivalence.

The efficiency property. All innocuous instructions are executed by the hardware directly, with no intervention at all on the part of the control program.

The resource control property. It must be impossible for that arbitrary program to affect the system resources, i.e. memory, available to it; the allocator of the control program is to be invoked upon any attempt.

The equivalence property. Any program K executing with a control program resident, with two possible exceptions, performs in a manner indistinguishable from the case when the control program did not exist and K had whatever freedom of access to privileged instructions that the programmer had intended.

As mentioned earlier, the two exceptions result from timing and resource availability problems. Because of the occasional intervention of the control program, certain instruction sequences in K may take longer to execute, so assumptions about the length of time required for execution might lead to incorrect results. In our simple system we will assume for the time being that there are no such difficulties.

The resource availability problem is the following. It might be the case, for example, that the allocator does not satisfy a particular request for space (an attempt to change the relocation-bounds register). The program may then be unable to function in the same manner as it would if the space were made available. The problem could easily occur, since the control program itself takes space.

One way around this difficulty is to realize that the virtual machine environment being produced is a "smaller" version of the actual hardware: logically the same, but with a lesser quantity of certain resources. Then the equivalence to be guaranteed is that between running on an actual smaller hardware machine and the environment we have created. On a paged machine, the resource consumed is more likely drum space to hold the pages of the VMM. In any case, we will specify this equivalence property more precisely. But first, a definition and the statement of our major theorem are in order.

We say that a *virtual machine monitor* (VMM) is any control program that satisfies the three properties of efficiency, resource control, and equivalence. Then functionally, the environment which any program sees when running with a virtual machine monitor present is called a virtual machine. It is composed of the original real machine and the virtual machine monitor. This informal definition should agree with the intuitive description early in this paper.

That done, we may now state our basic theorem.

THEOREM 1. *For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

6. Discussion of Theorem

Before discussing the import of this theorem, it would be appropriate to clarify what is meant by "conventional third generation computer." This phrase is intended to imply all the assumptions made so far in this paper regarding the operation of: relocation mechanisms, supervisor/user mode, and trap mechanisms. The assumptions were chosen to provide both clarity and a reasonable reflection of the relevant practices in common third generation machines. Also, the phrase is meant to imply that the instruction set is of general purpose enough to allow the construction of a dispatcher, allocator, and a generalized table lookup procedure. The need of the last will appear later in this discussion.

The theorem provides a fairly simple condition sufficient to guarantee virtualizability, assuming, of course, that the requisite features of "conventional third generation machines" are present. However, those features which have been assumed are fairly standard ones, so the relationship between the sets of sensitive and privileged instructions is the only new constraint. It is a very modest one, easy to check. Further, it is also a simple matter for hardware designers to use as a design requirement. Of course, we have not characterized the requirements which result from interrupt handling or I/O. They are of a very similar nature.

It will be useful in the proof to characterize the equivalence property in terms of a homomorphism on possible states in C, the collection of machine states. Partition C into two parts. The first set C_v contains all those states for which the VMM is present in memory and the value of P in the PSW stored in location 1 is equal to the first location of the VMM. The second set C_r contains the remaining states. The two sets reflect the possible states of the real machine with and without a VMM, respectively.

Each instruction in the processor set can be thought of as a unary operator on the set of states: $i(S_j) = S_k$. Likewise, each instruction sequence $e_n(S_1) = ij \cdots k(S_1) = S_2$ can also be thought of as a unary operator on C. Consider all the instruction sequences of finite length. Call that set of instruction sequences I. This set contains the operators with which the homorphism will be concerned.

A *virtual machine map* (VM map) $f : C_r \rightarrow C_v$ is a one-one homomorphism with respect to all the operators e_i in the instruction sequence set I.

That is, for any state $S_i \in C_r$ and any instruction sequence e_i, there exists an instruction sequence e_i' such that $f(e_i(S_i) = e_i'(f(S_i))$. This correspondence is shown in Figure 2.

There are two related properties included in the definition of a VM map. First is the mathematical existence of a particular mapping from the states of the real machine to the virtual machine system. Nothing, however, is said about the ability to construct such a map, by hardware or any other way. Second is the *actual* existence of instruction sequences e_i' on the C_v domain that correspond to the sequences e_i on the C_r domain. We demand as part of the definition of a VM map that for each e_i, the appropriate e_i' can be found and executed. Hence it is not immediately clear that a VM map exists for a particular machine.

As part of the definition, it is also necessary that f be one-one. This requirement is equivalent to saying that f has a (left) inverse. Call that inverse g. It will be needed in the proof.

To make this mapping concept more precise, we will demonstrate a particular VM map.

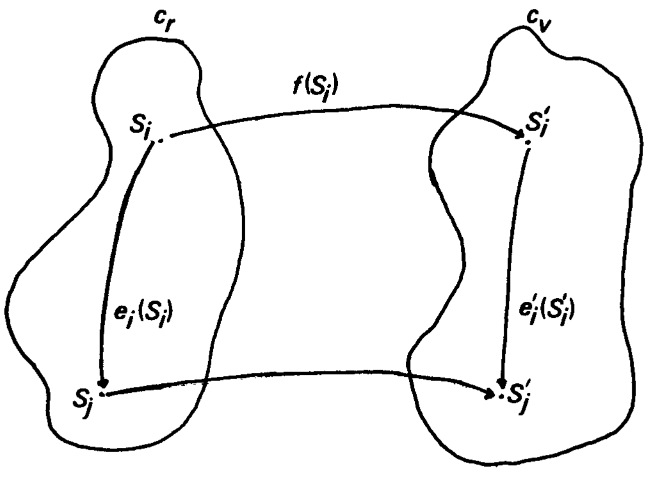
Let the control program occupy the first k locations of physical memory. That is, $E[0]$ and $E[1]$ are reserved for PSWs, so the control program takes locations 2 through $k - 1$. The next w locations will be used for a virtual machine. We assume that $k + w \leq q$. So $f(E, M, P, R) = (E', M', P', R')$ where $S = \langle E, M, P, R \rangle$ is the machine without a VMM present. It is assumed that in this real machine (the machine to which we will compare the activity in the virtual machine), the value of b in $r = (l,b)$ is always less than w. Then

$E'[i + k] = E[i]$, for $i = 0, w - 1$,
$E'[1]$ = the control program, for $i = 2$ to $k - 1$,
$E'[i] = \langle m',p',r' \rangle$,
  m' = supervisor,
  p' = first location of the control program,
  $r' = (0,q - 1)$,
$E'[0] = \langle m,p,r \rangle$ as last set by trap handler,
$M' = u$ (user),
$P' = p$,
$R' = (l + k, b)$, where $R = (l, b)$.

Notice that the VM map specified above only maps states after the completion of one instruction in the real machine and before the beginning of the next.

This virtual machine map is a fairly simple one; it is certainly possible to create much more complex functions which display the properties of a VM map required so far. However, the above will be taken as the standard VM map, and for the remainder of this paper any

Fig. 2. The virtual machine map.



Now we can state what is meant by "equivalence," or "essentially identical effect" more precisely. Suppose the two machines are started, one in state S_1, the other in state $S_1' = f(S_1)$. Then the environment provided by the virtual machine monitor is equivalent to the real machine if and only if, for any state S_1, if the real machine halts in state S_2; then the virtual machine halts in state $S_2' = f(S_2)$. By the virtual machine halting, we mean that an attempt is made in the virtual machine system to execute a halt from a location j where $j \geq k$, that is, by the user program. Again see Figure 2.

This definition is chosen for several reasons. First, halts are used as comparison points rather than counts of the number of instructions executed, for example, because certain instructions will be interpreted by the virtual machine system, using potentially long instruction sequences. Also, since the VM map f is so simple and the difference from the user's point of view so inconsequential, we argue that it is not necessary to actually apply g to determine whether $g(S_2') = S_2$ in order to check for equivalence. Showing that $f(S_2) = S_2'$ is enough.

Proof Sketch

The proof of the theorem consists of demonstrating that a control program can be constructed which has the three properties of equivalence, resource control, and efficiency as now defined.

We construct a control program that obeys the three requisite properties. It is the CP outlined earlier. The only constructive part not demonstrated was the ability to provide the appropriate interpretive routines for all privileged instructions. We demonstrate below more practical techniques. The effect of any privileged instruction (in general, any instruction) depends only on $M, P, R, E[1]$ and $E \mid R$; that is, not on all of memory, but only on location 1 and that part specified by the relocation bounds register R. The maximum size of $E \mid R$ is w. Then the effect of any privileged instruction can be specified in a table of two-tuples where the length of the table is the number of possible states that $\langle E \mid R, M, P, R\rangle$ can describe. The first entry in each tuple is a state, the second the state corresponding to the effect of the particular privileged instruction executed in the first state.

Such a state transition table can be extremely large, and there is a table for each privileged instruction. The VMM, that is to say k, can be quite large. Without going through the arithmetic, we argue however that the tables can be made small by limiting the size of the the real machine. That is, w can be chosen small.

We have assumed that third generation machines have an instruction set capable of managing these tables. Hence, interpretive routines are guaranteed constructable. Note of course that such state tables are a last resort, for those privileged instructions of an extremely arcane nature which are in fact arbitrary algorithms. By limiting the size of "real" memory though, the number of nonequivalent such programs is also limited, hence the appropriate tables are also of limited size. In all real cases today, *much* simpler and more efficient routines exist, and should be used.

This completes the description of the control program, so it remains to discuss the three properties.

Guarantees of the resource control and efficiency properties are trivially dispensed with. By the definition of sensitive instruction and the subset requirement of the theorem, any instruction that would affect the allocation of resources traps and passes control to the VMM. Efficiency has been taken to mean the direct execution of innocuous instructions; we have constructed the VMM to provide that behavior.

Only equivalence remains. It is necessary to demonstrate that, for any instruction sequence $t = ij \cdots k$ where k is a halt and any state S_1 of a real machine, the following is true.

Let $S_1' = f(S_1)$ and $S_2 = t(S_1)$. Then $f(S_2) = t(S_1')$. Again, see Figure 2.

First, we demonstrate that the equivalence property is true for single instructions; that is, for $t =$ any instruction i. We consider two cases, innocuous instructions and sensitive instructions. Both cases are easy, and demonstrated in detail in the Appendix as lemmas 1 and 2. The innocuous case follows from the definition of an innocuous instruction and direct application of the definition of VM map. The sensitive case follows from the fact that all sensitive instructions are privileged, from the existence of correct interpretation sequences and the VM map definition.

Since single instructions "execute correctly," it now remains only to show that finite sequences also do. That is for any instruction sequence $e_m = ij \cdots k$, $e_m(f(S)) = f(e_m'(S))$. This fact follows from lemmas 1 and 2, and the definition of the VM map f as a one-one homomorphism. It is a fairly standard proof and is demonstrated in the Appendix as lemma 3.

The proof is now complete, since for third-generation-like machines in which sensitive instructions are a subset of privileged instructions, we have demonstrated that a control program can be constructed which obeys the required three properties. That is, we have exhibited a VMM. Q.E.D.

Note that there are several reasons why the *necessity* direction of this theorem is not true in general. That is, under certain conditions it may still be possible to virtualize a machine even if the conditions of the theorem are not fulfilled. As a case in point, architectures that include location sensitive instructions may still support a virtual machine system if it is possible to construct a VMM that resides in high core, letting other programs execute unrelocated. Location sensitivity then would not matter.

In addition, there may be instructions that are not true privileged instructions as defined earlier, but which still trap when an undesirable action would result. An example of such a case is an instruction that is able to change the relocation bounds register, but can only decrease the bounds value when executed from user mode.

7. Recursive Virtualization

A number of related results can quickly follow from this approach. One simple example is the idea of recursive virtualization. Is it possible for a virtual machine system to run under itself a copy of the VMM, and will that copy also exhibit all the properties of a VMM? If this procedure can be repeated until the resources of the system are consumed (since each control program takes up space), then the original machine is *recursively virtualizable* [2, 6].

THEOREM 2. *A conventional third generation computer is recursively virtualizable if it is: (a) virtualizable, and (b) a* VMM *without any timing dependencies can be constructed for it.*

PROOF. This property is nearly trivial to demonstrate. A VMM is guaranteed, by definition, to produce an environment in which a large class of programs run with effect identical to that on the real machine. Then it is merely necessary to demonstrate that a VMM which belongs to that class of programs can be constructed. If it can, then the performance of the VMM running on the real machine and under other VMMs will be indistinguishable.

The only programs excluded from the class of identically performing programs are those which are resource bound, or have timing dependencies. The second limitation is mentioned in the statement of the theorem. The resource bound for our skeletal model is only memory, and it just limits the depth (number of nested VMMS) of the recursion, as pointed out in the definition of recursive virtualization. Hence the VMM as constructed earlier qualifies as a member of that "large class of programs." Q.E.D.

8. Hybrid Virtual Machines

As remarked earlier, there exist very few third generation architectures which are virtualizable [5, 6]. For that reason, we relax the definition to yield a related, more general, but less efficient form which we label a *hybrid virtual machine* system (HVM) [6]. Its structure is almost identical to a virtual machine system, but more instructions are interpreted rather than being directly executed. Hence the HVM is less efficient than a VM, but as a result, more actual third generation architectures qualify. For example, the PDP-10 can host a HVM monitor, although it cannot host a VM monitor [3].

To specify the relaxed conditions, it is necessary to divide the class of sensitive instructions into two not necessarily disjoint subsets.

An instruction i is said to be *user sensitive* if there exists a state $S = \langle E, u, P, R \rangle$ for which i is control sensitive or behavior sensitive.

That is, an instruction i is *user control sensitive* if the definition given earlier for control sensitivity holds, with m_1 in that definition set to user. The instruction i is *user behavior sensitive* if the definition for location sensitivity holds with the mode of states S_1 and S_2 equal to user. Then i is *user sensitive* if it is either user control sensitive or user location sensitive. Intuitively, these are instructions which cause difficulty when executed from user mode.

In a parallel fashion, an instruction i is *supervisor sensitive* if there exists a state $S = \langle E, s, P, R \rangle$ for which i is control sensitive or behavior sensitive.

THEOREM 3. *A hybrid virtual machine monitor may be constructed for any conventional third generation machine in which the set of user sensitive instructions are a subset of the set of privileged instructions.*

In order to argue the validity of the theorem, it is first necessary to characterize the HVM monitor. The difference between a HVM monitor and a VMM is that, in the HVM monitor, *all* instructions in virtual supervisor mode will be interpreted. Otherwise the HVM monitor is the same as the VM monitor. Equivalence and control can then be guaranteed as a result of two

To demonstrate the utility of the concept of a HVM monitor, we present the following.

Example. The PDP-10 instruction JRST 1, (return to user mode) is a supervisor control sensitive instruction which is not a privileged instruction. Hence the PDP-10 cannot host a VMM. However, since all user sensitive instructions are privileged, it can host a hybrid virtual machine monitor [3].

9. Conclusion

In this paper, we have developed a formal model of a third generation computer system. Using the model we have derived necessary and sufficient conditions to determine whether a particular third generation machine can support a virtual machine monitor. While previous authors [4, 5] have speculated about architectural characteristics required of third generation virtual machines, we have been able, using the formal approaches of this paper, to establish much more precisely the mechanisms to be used and the requirements to be met. These results have been used at UCLA, for example, to evaluate the DEC PDP-11/45, and make modifications to it so that a virtual machine system could be constructed [13].

While the model does capture much of the essence of third generation virtual machines, there have been a number of simplifications introduced for purposes of presentation. It has been indicated empirically that some of these omissions, such as I/O resources and instructions, asynchronous events, or more complex memory mapping schemes can be added as straightforward extensions to the basic model and our major result extended [6, 12].

Very recent work in computer systems architecture has included proposals for virtualizable architectures [2, 6, 8, 10, 11] which directly support virtual machines while avoiding the need for traditional VMM interpretive software overhead. The formal techniques, as sketched in this paper, may be applied to these new architectures to verify that they are virtualizable as claimed.

Acknowledgments. The authors would like to thank their colleagues at both UCLA and Harvard for many helpful discussions on various aspects of virtual computer systems. Special thanks are due to Professor G. Estrin of UCLA and Dr. U.O. Gagliardi of Harvard University and Honeywell Information Systems for their advice and encouragement during the prepara-

Appendix

Several results were used in the statement of the proof without being explicitly demonstrated. They are the lemmas which follow.

LEMMA 1. *Innocuous instructions, as executed by the virtual machine system, obey the equivalence property.*

PROOF SKETCH. Let i be any innocuous instruction. Let S be any state in the real machine, and $S' = f(S)$. $S = (e \mid r, m, p, r)$ and $S' = (e' \mid r', m', p', r')$. However, from the definition of f, $e' \mid r' = e \mid r$ and $p' = p$, and the bounds in both r' and r are the same. By definition, $i(S)$ cannot depend on m or l (the relocation part of r), and all other parameters are the same for both S and S'. Hence it must be the case that $i(S) = i(S')$. Q.E.D.

LEMMA 2. *Sensitive instructions, as interpreted by the virtual machine system, obey the equivalence property.*

PROOF SKETCH. By assumption, any sensitive instruction i traps. By construction, the interpretation is done correctly, given all necessary parameter specifications. The values of locations $E \mid R$ are not changed by the trap. The values of P and R are saved in $E[0]$. The "simulated mode" value M is stored by the VMM. Hence all necessary information is present, so proper interpretation can be performed. Q.E.D.

LEMMA 3. *Given that all single instructions obey the equivalence property, any finite sequence of instructions also obeys the equivalence property.*

PROOF. The proof is by induction on the length of the instruction sequence. Each sequence can be thought of as a unary operator on the set C of states. The basis of the lemma is true by the hypothesis in the statement of the lemma.

In the following, parentheses will be used only sparingly. Hence $f(g(h(S)))$ may be written $fgh(S)$.

Induction Step. Let i be any instruction, and t any sequence of length less than or equal to k, and t' the instruction sequence corresponding to t.

Then by the induction and lemma hypothesis, we have that, for any state S, there exists an instruction sequence t' such that

$$f(t(S)) = t'(f(S)) \quad \text{and} \quad f(i(S)) = i'(f(S))$$

where the primed operators may or may not be the same instructions or sequences as the unprimed operators. The instruction sequences may differ since some of the instructions expressed by the unprimed operators may be sensitive. The primed operator includes the interpretation sequences for those instructions.

We are given

$$ft(S) = t'f(S). \tag{1}$$

Clearly then,

$$i'ft(S) = i't'f(S). \tag{2}$$

But, for any S, we are given

$$i'f(S) = fi(S). \tag{3}$$

So, letting $t(S)$ in (2) be S in (3), we have, combining (3) with the left side of (2):

$$fit(S) = i't'f(S).$$

Since the sequence may be any sequence of length $k + 1$, and the above is the desired induction step result, the lemma is proven. Q.E.D.

References

1. Buzen, J.P., and Gagliardi, U.O. The evolution of virtual machine architecture. Proc. NCC 1973, AFIPS Press, Montvale, N.J., pp. 291–300.
2. Gagliardi, U.O., and Goldberg, R.P. Virtualizable architectures, Proc. ACM AICA Internat. Computing Symposium, Venice, Italy, 1972.
3. Galley, S.W. PDP-10 Virtual machines. Proc. ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, Cambridge, Mass., 1969.
4. Goldberg, R.P. Virtual machine systems. MIT Lincoln Laboratory Rept. No. MS-2686 (also 28L-0036), Lexington, Mass., 1969.
5. Goldberg, R.P. Hardware requirements for virtual machine systems. *Proc. Hawaii Internat. Conference on Systems Sciences*, Honolulu, Hawaii, 1971.
6. Goldberg, R.P. Architectural principles for virtual computer systems. Ph.D. Th., Div. of Eng. and Applied Physics, Harvard U., Cambridge, Mass., 1972.
7. Goldberg, R.P. (Ed). Proc. ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, Cambridge, Mass., 1973.
8. Goldberg, R.P. Architecture of virtual machines. Proc. NCC 1973, AFIPS Press, Montvale, N.J., pp. 309–318.
9. IBM Corporation. IBM Virtual Machine Facility/370: Planning Guide, Pub. No. GC20-1801-0, 1972.
10. Lauer, H.C., and Snow, C.R. Is supervisor-state necessary? Proc. ACM AICA Internat. Computing Symposium, Venice, Italy, 1972.
11. Lauer, H.C., and Wyeth, D. A recursive virtual machine architecture. Proc. ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, Cambridge, Mass., 1973.
12. Meyer, R.A., and Seawright, L.H. A virtual machine time-sharing system. *IBM Systems J. 9*, 3 (1970).
13. Popek, G.J., and Kline, C. Verifiable secure operating system software. Proc. NCC 1974, AFIPS Press, Montvale, N.J., pp. 145–151.