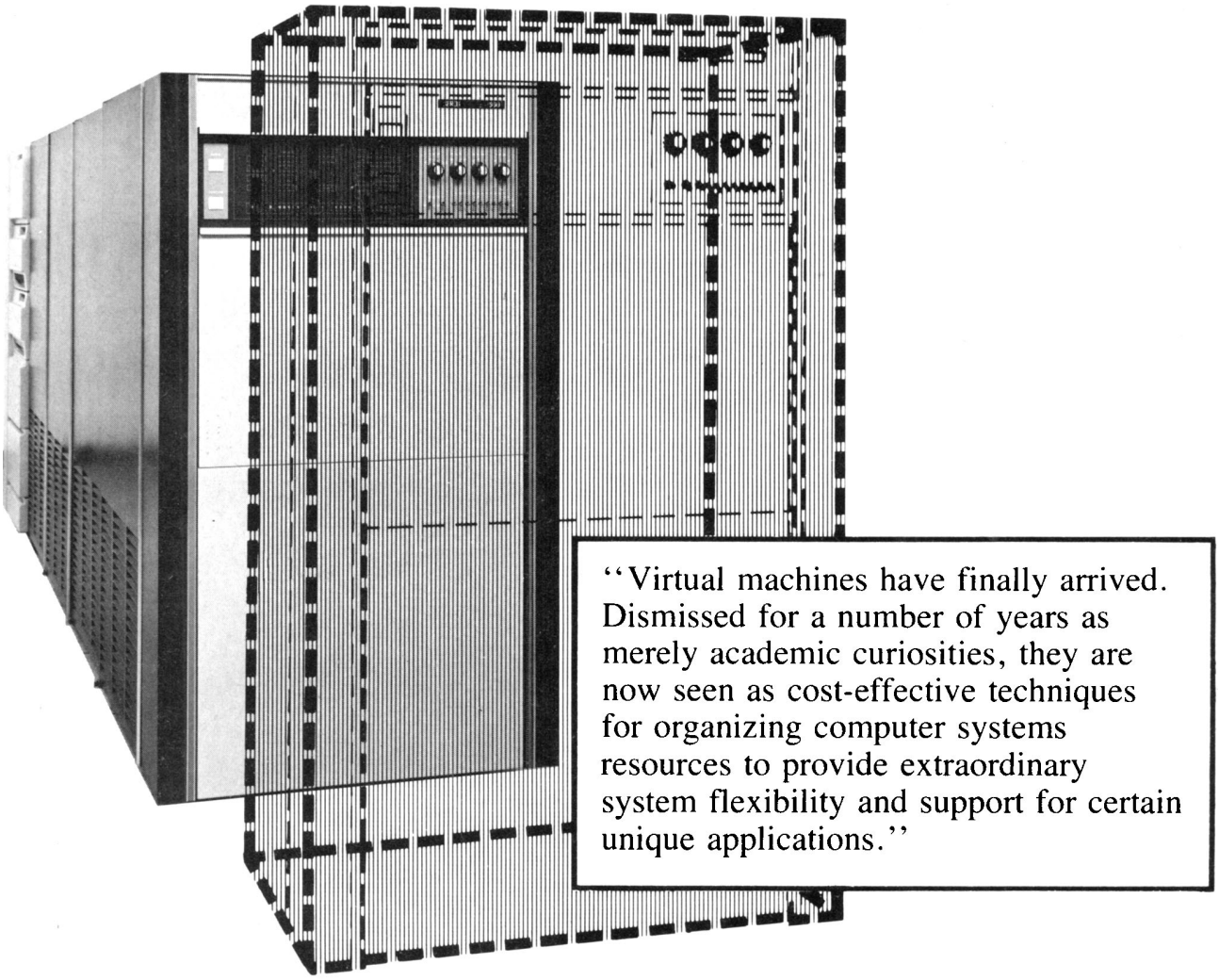

"Virtual machines have finally arrived. Dismissed for a number of years as merely academic curiosities, they are now seen as cost-effective techniques for organizing computer systems resources to provide extraordinary system flexibility and support for certain unique applications."

Survey of Virtual Machine Research

Robert P. Goldberg

Honeywell Information Systems
and Harvard University

Introduction

The complete instruction-by-instruction simulation of one computer system on a different system is a well-known computing technique. It is often used for software development when a hardware base is being altered. For example, if a programmer is developing software for some new special purpose (e.g., aerospace) computer X which is under construction and as yet unavailable, he will likely begin by writing a simulator for that computer on some available general-purpose machine G. The simulator will provide a detailed simulation of the special-purpose environment X, including its processor, memory, and I/O devices. Except for possible timing dependencies, programs which run on the "simulated machine X" can later run on the "real machine X" (when it is finally built and checked out) with identical effect. The programs running on X can be arbitrary — including code to exercise simulated I/O devices, move data and instructions anywhere in simulated memory, or execute any instruction of the simulated machine. The simulator provides a layer of software filtering which protects the resources of the machine G from being misused by programs on X.

If several different programmers are developing software for X concurrently, it may be possible to run a number of copies of the simulator under an operating system on G. Alternatively, a special, more powerful version of the simulator may be developed which itself is a time-sharing system and supports multiple users. In either case, the result would be the illusion of multiple copies of the hardware-software interface of machine X on machine G.

Since machines X and G may be arbitrarily chosen, they may be significantly different in structure. This may imply a very large simulation program and significant overhead for

the simulation of each of X's instructions. As a result, it is possible to find the machine slowed down by as much as 1000 to 1. Consequently, simulation is generally used only for software development and almost never in a production mode.

While X and G may be arbitrarily different, it is also possible to choose them to be identical – i.e., X=G. In this case we would be supporting many copies of the hardware-software interface of G on one machine G. Each user would have his own private copy of a machine G and could select the operating system of his choice to run on his "private" computer. He could also choose to develop or debug his own operating system. As before, since each instruction for the simulated G is actually being interpreted by software on the real G, there can be no way for one simulated machine to interfere with another.

If the real and simulated machines are identical then it may be possible to construct a simulator in which programs run with a slow-down of only about 20 to 1.* While this may be a considerable improvement over the more general simulator, it seems odd that programs being run on native hardware, i.e., the machines they were written for, should have to be slowed down at all. Considerations of this kind have led to the development of much more efficient simulators for multiple copies of a machine on itself.** In these systems, much of the software for the simulated machine executes directly on the hardware without software interpretation. Systems of this kind are called *virtual machine systems,* the simulated machines are called *virtual machines* (VMs), and the simulator software is called the *virtual machine monitor* (VMM).

Whether or not it is possible to construct a VMM depends upon the subject machine's architecture. Even for systems in which virtual machine monitors have been constructed, there still remain many interesting questions concerning performance and use.

IBM's improved virtual machine support for System/370 (i.e., VM/370 Release 2),[43,44] the application of virtual machine systems to significant problems in data security/reliability,[4,14,51,63] and the use of virtual machine techniques to reduce software development costs[13,27] are just some of the reasons for the widespread current interest in virtual machines.

In this paper we will take up these issues in connection with some of the recent work on virtual machine *principles, performance,* and *practice.* In particular, we shall examine the rationale for virtual machines, discuss the implications of virtual machines on new architectural designs, consider virtual machine performance costs, and finally explore some of the unique applications which virtual machines make possible. The tutorial papers by Buzen and Gagliardi,[16,17] Parmelee et al,[60] and Meyer and Seawright,[57] as well as Chapters 1-3 of the author's Ph.D. dissertation[33] may be read for additional background material. Finally, the recent textbook by Madnick and Donovan[52] includes an excellent introduction to virtual machines as part of a course on operating systems.

*The simulator is typically oriented around the use of an execute-type instruction for simulating each central processor instruction.

**Somewhat different considerations have led to the development of *emulators* which are efficient hardware or firmware assisted simulators for dissimilar machines. See Mallach.[53,54]

Principles

Virtual machine systems were originally developed to correct some of the shortcomings of the typical third-generation architectures and multi-programming operating systems – e.g., OS/360.[22] The principal architectural characteristics of these systems was the dual-state hardware organization with a privileged and a non-privileged mode. In privileged mode all instructions are available to software, whereas in non-privileged mode they are not. The operating system provided a small resident program called the *privileged software nucleus.* User programs could execute the non-privileged hardware instructions or make supervisory calls – e.g., SVCs – to the privileged software nucleus in order to have *privileged* functions – e.g., I/O – performed on their behalf. The set of non-privileged instructions together with the supervisory calls effectively defines an *extended machine* which is similar to but *not identical* to the bare machine. (See Figure 1.) The extended machine is, in theory, better human-engineered and easier to program than the original bare machine.

The extended machine approach has been quite successful in many computer systems installations, but there still are a number of problems associated with it. While Figure 1 illustrates multiple extended machine interfaces, only one bare machine interface is provided. Thus, only one privileged software nucleus can be run at a given time. Consequently, it is not possible to run other operating systems, certain diagnostic programs, or any software which requires a bare machine interface instead of an extended machine interface. This rigidity may have significant impact on the transportability of user software (written for other operating systems), modification and testing of the operating system (privileged software), and the running of test and diagnostic (T & D) programs. In the face of these obstacles the installation's management usually solves this problem with shift scheduling: operating system debugging,

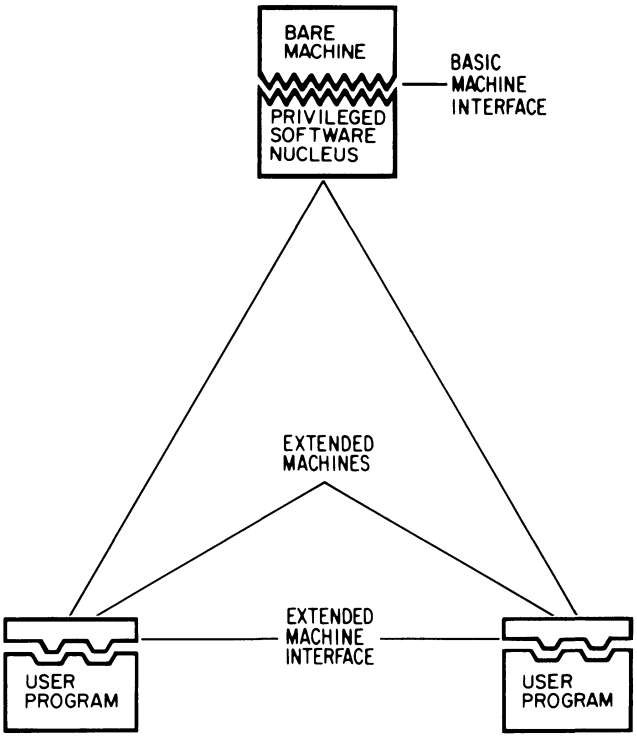

Figure 1. Conventional Extended Machine Organization

T & D, unusual or old release operating systems, and normal system use scheduled for separate blocks of time during the day (and night).

The major innovation of *virtual machines* (VMs) was to solve the above problem. The heart of a VM system is the virtual machine monitor (VMM) software which transforms the single machine interface into the illusion of many. Each of these interfaces (virtual machines) is an efficient replica of the original computer system, complete with all of the processor instructions (i.e., both privileged and non-privileged instructions) and system resources (i.e., memory and I/O devices). By running each operating system on its own virtual machine it becomes possible to run several different operating systems (privileged software nuclei) concurrently. (See Figure 2.)

Perhaps the best known virtual machine system is IBM's VM/370.[43,44] On each virtual 370 a user may run any of the System/360 or System/370 operating systems, such as DOS/360, OS/VS1, OS/VS2, or any version of OS/360. The user may also run the Conversational Monitor System (CMS), a simple monoprogramming operating system which was developed specifically for use on virtual machines.

Other virtual machine and virtual machine-like systems which have been developed include:

- M44/44X – A virtual machine-like system developed for a specially modified IBM 7044.[58,66,67]
- CP-40 – A virtual machine system developed for a specially modified IBM 360/40, forerunner of CP-67.[1,40]
- CP-67 – A virtual machine system developed for the IBM 360/67, forerunner of VM/370.[8,24,57]
- 360/30 – A single virtual machine supported on a specially modified IBM 360/30, used for system measurement.[45]
- HITAC 8400 – A single virtual machine supported on a HITAC 8400 (RCA Spectra 70/45), used for special software development.[26]
- UMMPS – One or several virtual machines (360) supported concurrently with UMMPS on 360/67, normally used to provide OS/360 support.[2,41,70]
- PDP-10 – A virtual machine-like system running under the ITS operating system on a special PDP-10 at MIT.[29]

Other virtual machine systems currently under development include:

- UCLA-VM – A virtual machine system being developed for specially modified PDP-11/45. Will be used for data security studies.[63,64]
- Newcastle Recursive VM – Burroughs B1700 is being microprogrammed to define a machine architecture for which a VMM is being written.[47,48]

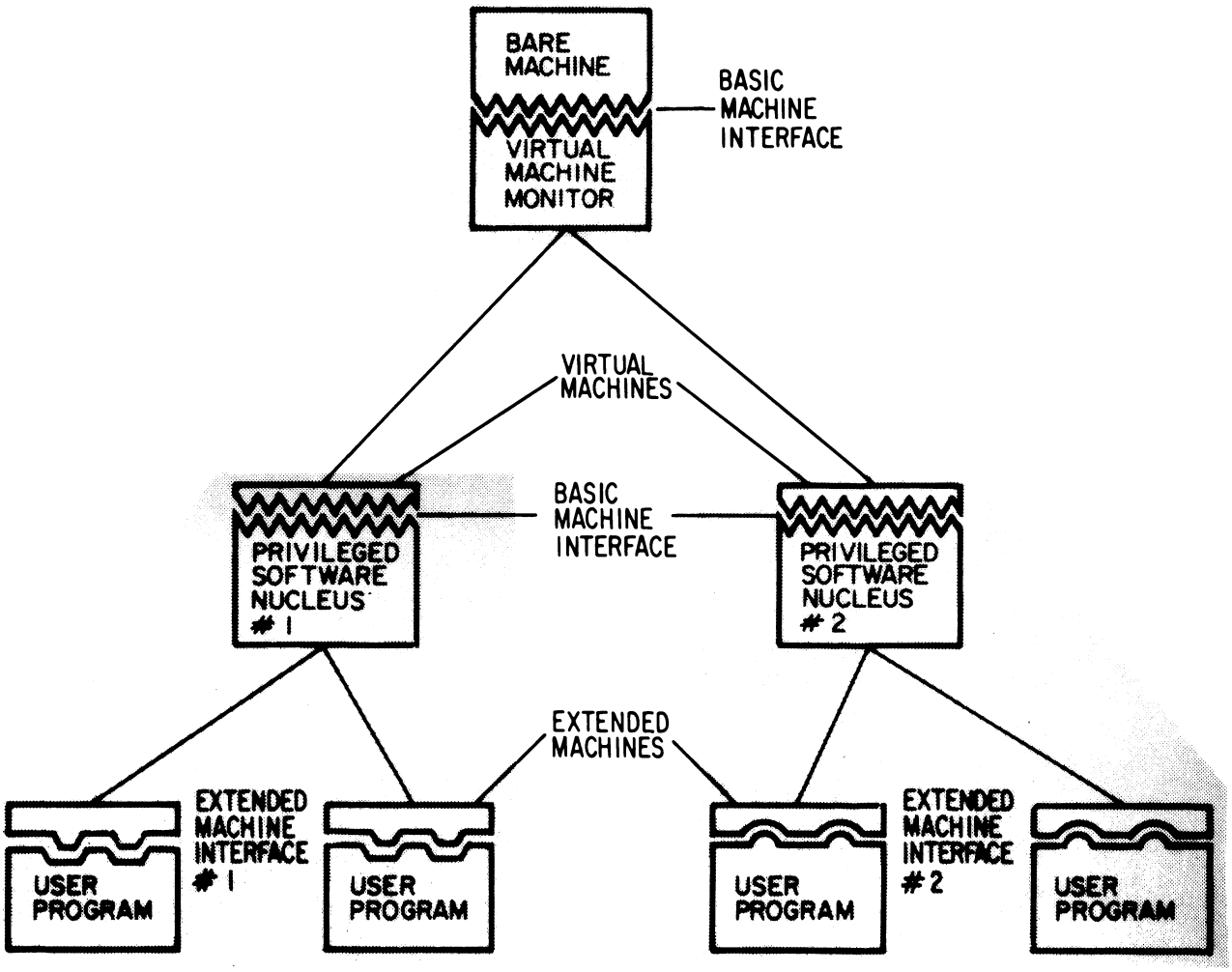

Figure 2. Virtual Machine Organization

While virtual machines, multiprogramming, and virtual storage are independent concepts,[31] they form a very powerful construct when combined together.[60] A virtual machine provides an efficient, isolated replica of a computer system's environment. With multiprogramming it becomes possible to multiplex among several virtual machines concurrently on a single hardware system.[50] Finally, with virtual storage, it is possible to support virtual machines whose memory requirements exceed the actual resources available.[20]

Despite the power of the virtual machine concept, only a very limited number of virtual machine systems have actually been implemented. This situation is in part due to the architectural characteristics of third-generation machines which were not designed to support virtual machines.[30,32,65] Consequently, these systems do not provide the appropriate architectural support and force the existing VMMs to rely on somewhat contrived software techniques.

As with the purely simulated machine discussed in the introduction, support of a virtual machine requires faithful reproduction of the processor, memory, I/O system, and even the operator's console. Furthermore, to satisfy the efficiency requirements which are an essential part of the virtual machine concept, it is necessary to execute a significant portion of the virtual CPU's instructions directly on the host hardware. Since the instructions to be executed on the virtual machine might include the privileged instructions which can alter the mode of the machine, perform I/O, etc., complete direct execution of software by the virtual machine might permit it to interfere with the VMM or other virtual machines. In order to prevent this situation from occurring it is necessary for the VMM to maintain proper control over the state of the real processor.

Third-Generation Implementation Issues The solution that was adopted in third-generation architectures involved running all software for virtual machines in the non-privileged mode and having the virtual machine monitor maintain a virtual mode bit in a software table.[16,17,30] The virtual mode bit indicated the state which the machine would be in if the software were executing directly on the bare machine. Instructions which were insensitive to the actual mode of the machine were allowed to execute directly on the bare machine without VMM intervention. All other instructions were trapped by the VMM and simulated in software using the virtual mode bit to determine the appropriate action in each case.

In general, the non-privileged instructions are executed directly and certain privileged instructions must be trapped and simulated. However, this cannot always be done since there may be some instructions which are sensitive to the processor mode mapping yet are not privileged – i.e., not automatically trapped when executed in non-privileged mode. As a result, it is often impossible to support virtual machine systems using this partial software construction.[30,32,33]

On third-generation virtual machine systems, the virtual machine's memory is usually supported through use of the system's memory mapping mechanism. The memory of the virtual machine must retain the properties of real memory, such as linear addresses from zero and special meanings to certain interrupt control locations. Memory mapping used in current systems has been both simple relocation and paging. If the host machine is paged, the virtual machines *may* include the paging mechanism as well.[30,60] In this case, the VMM must manipulate the page tables in order to map paged addresses within the virtual machines into their corresponding real addresses. Current techniques utilize some awkward and unnecessary software overhead but recent advances have been made in this area.[33,34]

Since I/O instructions are usually privileged, attempted execution by software on a virtual machine causes a trap to the VMM. At this point the VMM is able to translate device and memory addresses before issuing an I/O instruction on behalf of the virtual machine. When an I/O completion interrupt returns to the VMM, it is reflected back to the appropriate virtual machine. Since I/O operations may occur with a "relatively low frequency," the performance degradation introduced by this VMM software intervention should be tolerable. Current computer architectures require VMM software intervention to maintain system integrity since an improperly written channel program can interfere with other virtual machines or the VMM itself.[3] A side benefit of software intervention is the ability to map I/O requests for one device into requests for another[14,26] or to provide a virtual machine with special devices which have no real counterpart.[14,25]

The considerations of how the virtual machine maps are constructed for various systems and which machines admit of such a mapping has been discussed in the literature.[16,17,30,31,33] A recent study has even used formal mathematical techniques to establish sufficient architectural conditions for third-generation virtual machine support.[65] These results have led a number of researchers to make hardware modifications to current machines in order to support virtual machines.[63,64]

Virtualizable Architectures Recently, a number of researchers have proposed new architectures – i.e., virtualizable architectures – which provide features to directly support virtual machines.[28,33,34,47,48] The arguments for these architectures include:

- *System hygiene.* There is no intrinsic reason why virtual machine support must be based on the trap and simulation approach since it is clumsy and awkward.
- *Software simplicity.* Virtualizable architectures would make the VMM an even smaller and simpler program and further contribute to the reliability/security appeal of VM's.
- *System performance.* Machines designed to support virtual machines should operate even more efficiently than third-generation VM systems.

IBM has recently announced VM/370 Release 2 which includes a firmware modification, called VM-assist, to the standard System/370.[13,42,44] While very little information is currently available about VM-assist, it seems to have some of the characteristics of the virtualizable architectures.

The Hardware Virtualizer In order to illustrate the principles of virtualizable architectures, we will sketch the design of the author's *Hardware Virtualizer* which has been described in detail in the literature.[33,34,36] The theory is based on the following arguments:

- The key issue involved in VM's is the instantaneous relationship between the resources of the virtual and real machines

- We must identify the sets of resources of the virtual machine and the real machine and define a map between them, called an f-map.
- The f-map transforms a virtual resource name into its corresponding real resource name.
- The f-map must be invisible to *all software* executing on the virtual machine.
- The VMM software running on the real machine manipulates and invokes the f-map, and is given control on an f-map violation, called a VM-fault.
- The design extends directly for recursion, in which case the f-map maps adjacent levels of virtual resources. In order to run a VM it is necessary to compose – i.e., combine – all the maps together. Faults must be passed to the VMM at the appropriate level.
- Any other structure – e.g., privileged/non-privileged modes – is independent of virtualization and behaves as it would on the original machine.

The resource sets relevant to the virtual machine model are represented by the shaded areas of Figure 2, shown earlier. These sets are the real resource set and the two virtual resource sets. The corresponding f-maps are not illustrated in the figure.

Figure 3 illustrates the extension of the virtual machine model to include recursion. The model indicates how a VMM may be run on the basic machine interface of a virtual machine – e.g., V_1. This VMM in turn creates two virtual machines, $V_{1.1}$ and $V_{1.2}$, on which are running conventional operating systems – i.e., privileged software nuclei.

The virtual machine model identifies the five shaded areas of the figure as distinct resource sets and indicates the mapping relationship among them. Thus a resource name of V_2 must be mapped by f_2 to be transformed into a real resource of R. On the other hand, a resource of $V_{1.1}$ must be mapped consecutively by both $f_{1.1}$ and f_1 in order to be transformed into its corresponding resource of R. If there is a violation in applying the mapping of $f_{1.1}$, a VM-fault passes control to the VMM in V_1. Similarly, a violation of f_1 faults to the VMM in R. As in the nonrecursive model, local mapping structure pertaining to user programs is hidden within the resource sets and is ignored.

Direct application of this theory yields the design of the Hardware Virtualizer. Goldberg discusses in detail the development of a generic Hardware Virtualizer with arbitrary choices of target architecture and virtual machine map. There are a number of subtle issues which arise in the design but the key concept is the direct mirroring in hardware of the virtual machine model presented above. This requires hardware/firmware support to:

- represent the f-maps,
- activate a virtual machine,
- compose the f-maps (and possibly local maps) together during resource referencing, and
- pass control to the correct VMM on a VM-fault.

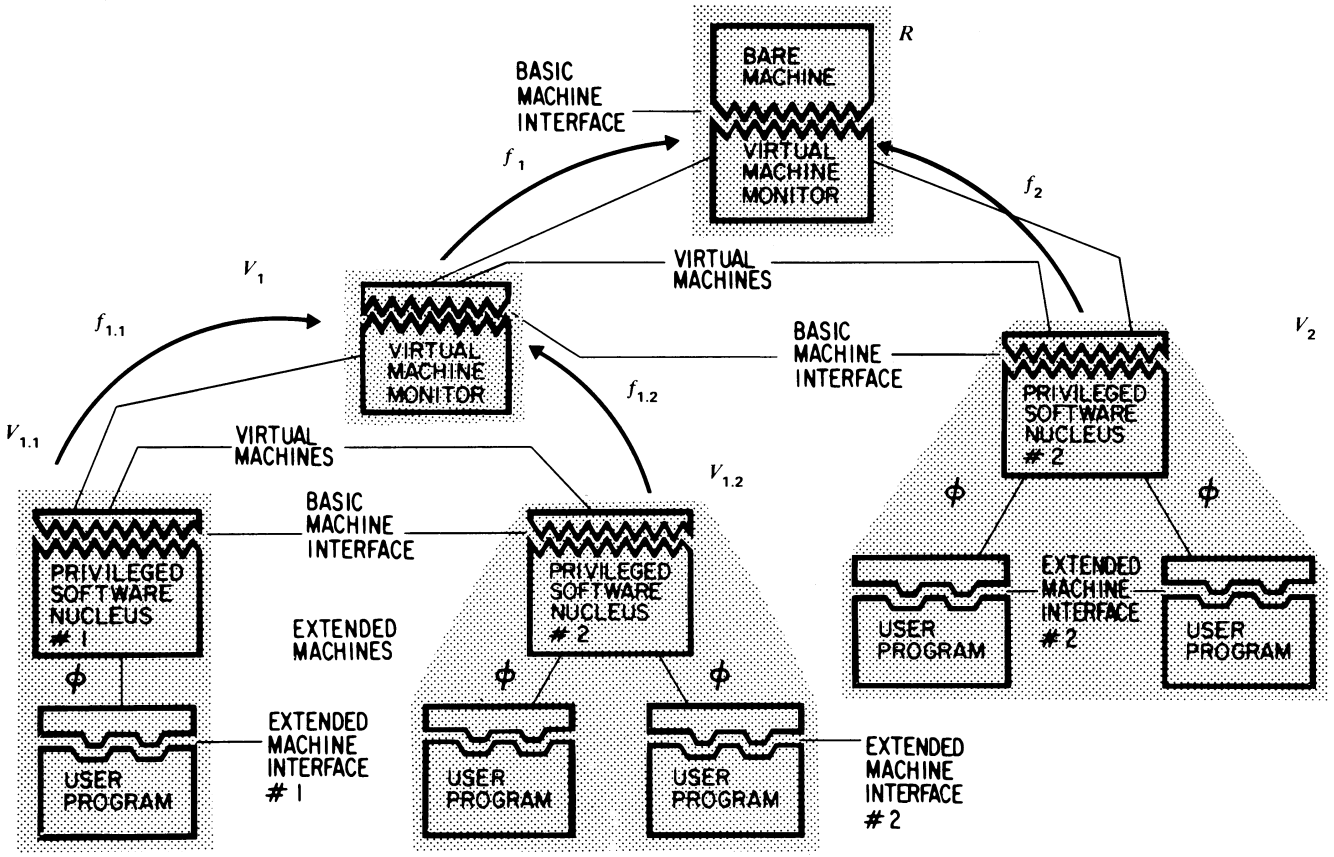

Figure 3. Virtual Machine Model with Recursion

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.