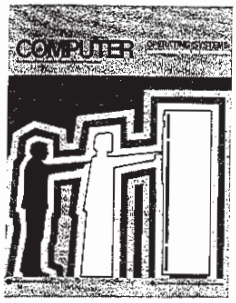
COMPUTER

June 1974 | Contents | Volume 7 Number 6



Cover design, "Operating Systems — Communicating with the Bare Machine," by Frank Yanai.

20 **OPERATING SYSTEMS**
Guest Editor *Richard R. Muntz*

22 **PROTECTION STRUCTURES**
Gerald J. Popek

34 **A SURVEY OF VIRTUAL MACHINE RESEARCH**
Robert P. Goldberg

46 **INTERPROCESS COMMUNICATION FACILITIES FOR NETWORK OPERATING SYSTEMS**
Eralp Akkoyunlu, Arthur Bernstein, and *Richard Schantz*

2 PRESIDENT'S MESSAGE
3 VICE PRESIDENT'S SPECIAL REPORT
4 CALENDAR
5 CALL FOR PAPERS
6 UPDATE
18 READERSHIP SURVEY BOX SCORE
56 NEW PRODUCTS
62 NEW APPLICATIONS
64 REPOSITORY
67 CLASSIFIED
68 PRODUCT PROFILE
73 SHORT COURSES
74 ADVERTISERS INDEX
75 COMPUTER SOCIETY PUBLICATIONS

Reader Service and Order Form Cards Inside Back Cover

EDITOR
True Seaborn
Assistant Editor
Marilyn Potes

Technical Editor **Dr. Jack E. Shemer**
Cookbook Editor **James H. Haynes**
New Products Editor **Milton G. Bienhoff**
Art Director **Jay Simpson**
Education Editor **Dr. David Robinson**
Advertising **Dawn Peck**
Circulation **Dana Gordon**

EDITORIAL BOARD

Dr. Jack E. Shemer
Xerox Corporation

Chairman
Dr. Warren L. Semon
Syracuse University

True Seaborn
IEEE Computer Society

Donald O. Baechler
Bell Telephone Labs

Dr. Albert L. Hopkins
MIT Instrumentation Laboratory

Editor-in-Chief
Dr. Robert A. Short
Oregon State University

Dr. John R. Lehmann
National Science Foundation

CIRCULATION:
Computer magazine is published monthly by the IEEE Computer Society, 5855 Naples Plaza, Suite 301, Long Beach, California 90803. Annual subscription included in Society member dues. Non-member prices: available on request. Single copy prices: members $3.00; non-members $6.00. This journal is also available in microfiche form. UNDELIVERED COPIES: Address all correspondence regarding undelivered copies of *Computer* to: IEEE Headquarters, Technical Services, 345 East 47th Street, New York, New York 10017. POSTMASTER: Send form 3579 to *Computer* magazine, IEEE Headquarters, Technical Services, 345 East 47th Street, New York New York, 10017. Second class postage is paid at New York, New York and at additional mailing offices. Copyright © 1974 by the Institute of Electrical and Electronics Engineers, Inc.

ADVERTISING:
For information about display, conference or product profile advertising in *Computer*, contact Advertising Manager, *Computer*, 5855 Naples Plaza, Suite 301, Long Beach, CA 90803.

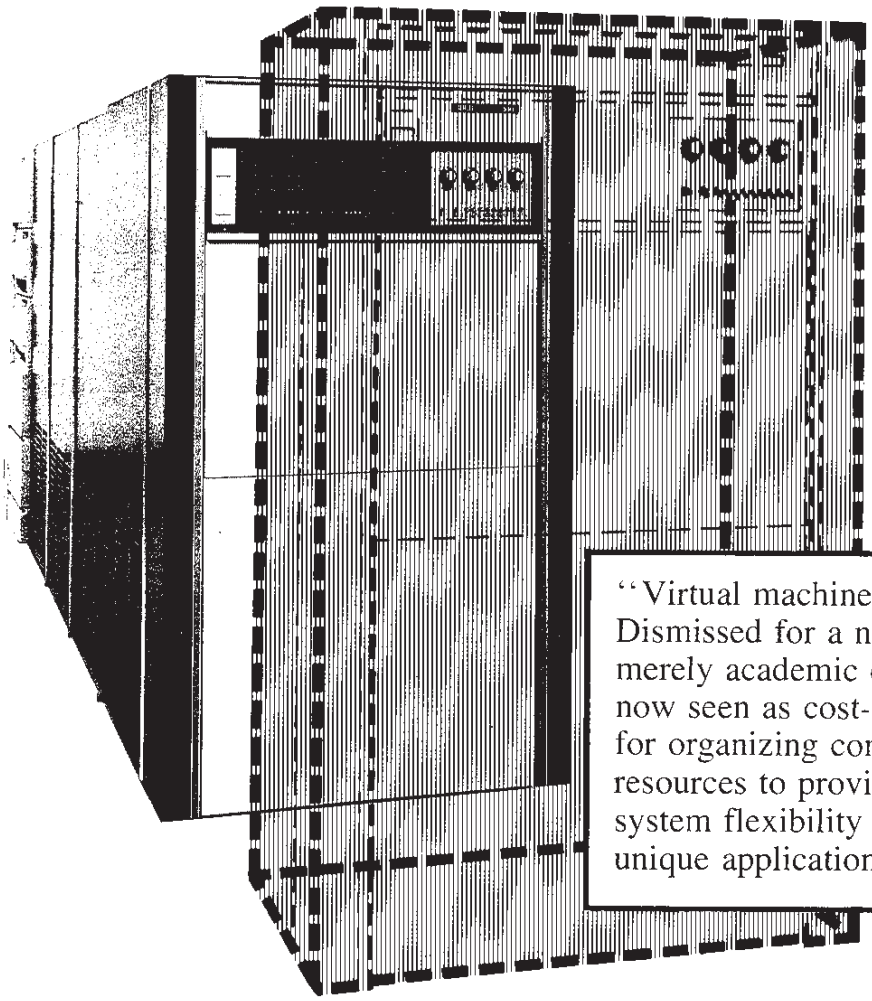
New York Representative: Edwin F. Hoffman, Jr., American Marketing Publications, 342 Madison Avenue, New York, NY 10017, (212) 687-2074.

Chicago Representative: Allan S. Young, P.O. Box 128, Winnetka, IL 60093, (312) 446-0686.

In Europe contact Sweatman and Fordham, Byron House, 7 St. James Street, London, England, SW1 01-930 3637.

In Asia contact Dempa Publications, Inc., 11-15 Higashi, Gotanda 1-chome, Shinagawa-Ka, Tokyo 141, Japan.

EDITORIAL:
Authors interested in publishing in *Computer* should submit their manuscripts to: Dr. Jack E. Shemer, Xerox Research, Advanced Development Lab, 701 South Aviation Blvd., Mail Station C3-50, El Segundo, CA 90245.
Four copies of each manuscript should be submitted, each one complete with all illustrations. Include your address and phone number, your biography, and photo. Please indicate whether the manuscript has been presented, published, or submitted for publication elsewhere. Manuscripts should be typewritten, double or 1-1/2 spaced, on one side of sheet only. See Call for Papers section.
Unless otherwise stated, bylined articles in *Computer* magazine reflect the author's opinion; inclusion in this publication does not necessarily constitute endorsement by the IEEE or the Computer Society.



"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,

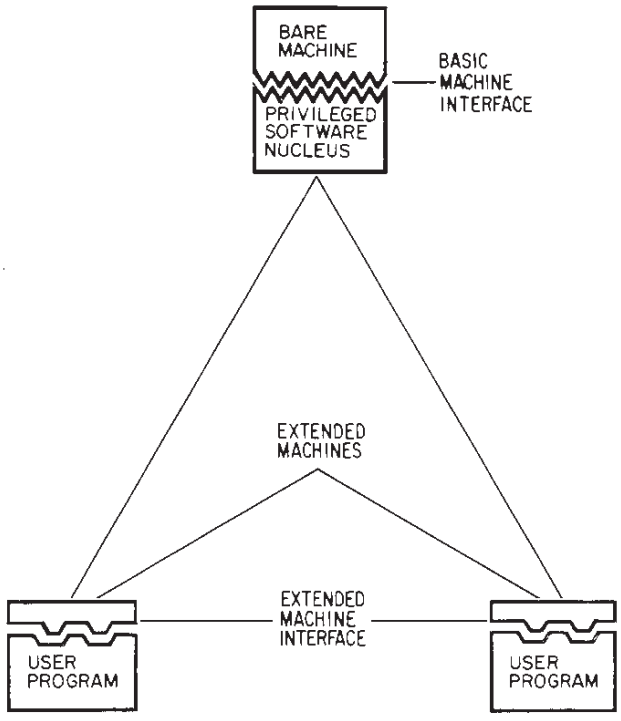


Figure 1. Conventional Extended Machine Organization

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]

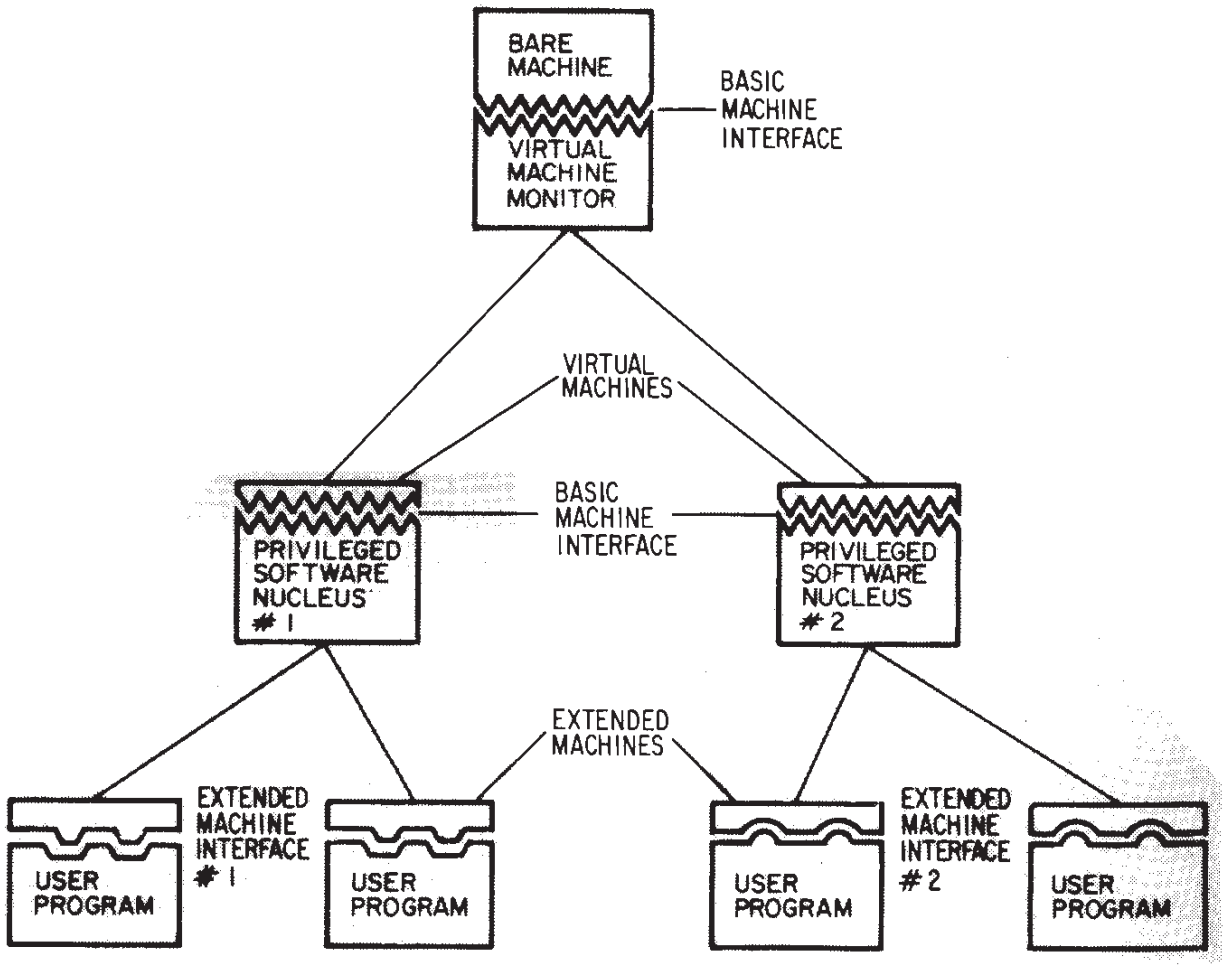


Figure 2. Virtual Machine Organization

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.

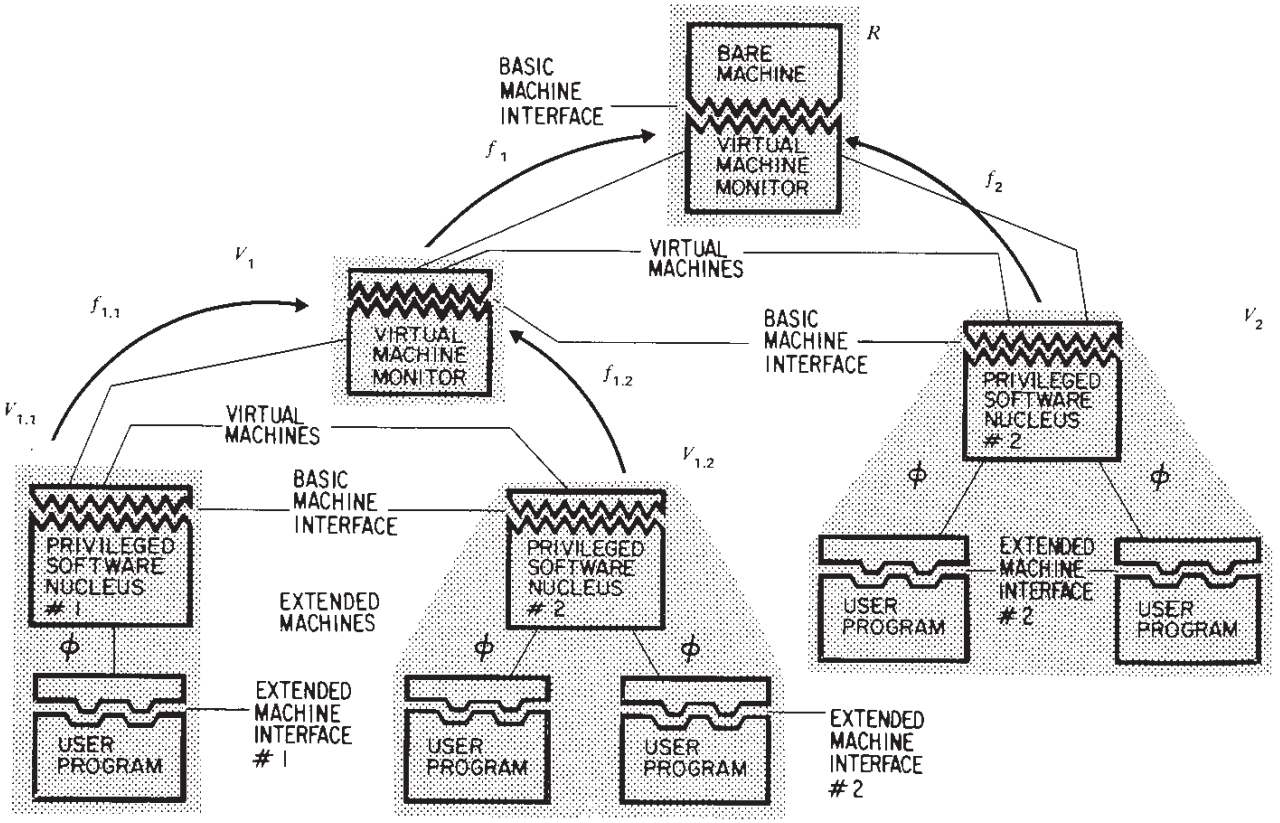


Figure 3. Virtual Machine Model with Recursion

In Goldberg[33] the map composer is sketched using a small number of scratchpad and associative registers which "remember" the most recently composed (mapped) resource names. It is claimed that because of program locality considerations,[68] the hit ratios for these associative registers will be high, and hence the instruction execution rate with the Hardware Virtualizer should be comparable to the real machine's rate for a wide range of f-maps and target architectures.

Performance

There have been numerous studies of the performance characteristics of the virtual machine systems CP-67,[6,19] VM/370,[7] and also the virtual machine facility under UMMPS.[70] However, since these three systems are also multiprogramming systems and virtual storage systems, much of the work has related to these other characteristics, rather than the purely virtual machine aspects.

In this section we will investigate some of the sources of overhead in virtual machine systems. Then we will examine some of the techniques that have been used to improve VM performance. Finally, we will look at some interesting performance problems which are unique to virtual machines.

Sources of Overhead There are demonstrable penalties in running a jobstream on a virtual machine instead of a real machine. These penalties include the extra resources — e.g., main memory and processor cycles needed by the VMM — and the potential drop in system throughput which results. The extra CPU cycles are sometimes called processor overhead, or just *overhead*, and the additional amount of time to process a jobstream is called *stretchout*.[59,60]

Studies often run jobstreams under a single virtual machine in order to separate out unusual multiprogramming effects. For example, Young has reported that a measurement run of a System/360 DOS jobstream run under control of VM/370 produced better throughput, due to multiprogramming, than running the same jobs serially on the same computing system.[72]

Some of the principal sources of overhead in virtual machine systems include:

- *Maintaining the status of the virtual processor.* The complete integrity of *all* visible registers, status bits, and reserved memory (interrupt control) locations must be preserved.
- *Support of privileged instructions.* Third-generation virtual machine systems have expended processor overhead to trap and simulate privileged instructions.
- *Support of Paging Within Virtual Machines.* Software techniques are currently used to transform a paged address in a VM into an address in the VM and finally into a real memory address.
- *Console Functions.* The operator's panel and lights are simulated in software. This overhead is not invoked as frequently as the others cited above.

Additional sources of overhead include the reflection of exceptions and I/O interrupts to the virtual machines, support of virtual timers and clocks, and the translation of I/O channel programs before the VMM initiates I/O. For virtual machines supported with paged memory mapping, channel program translation can be a significant source of overhead.

Improving Performance A number of techniques for improving the performance of virtual machine systems have been developed at various installations. While some are ad hoc approaches aimed at reducing overhead arising from third-generation architectural weaknesses, others have general applicability to virtualizable architectures as well. The techniques can roughly be divided into the following classes:

- policies,
- compromises to virtual machine architecture,
- improved or new mechanisms.

Policy Policy approaches to performance improvement have addressed both overhead and installation management issues. For example, real system resources can be dedicated in order to guarantee a certain performance level for a particular *preferred virtual machine.*[61] Resources include percentage of CPU time, a real I/O channel and its devices, and page frames of main memory.[43]

An interesting resource allocation policy concerns the so-called "virtual = real" option which assigns virtual addresses to identical real addresses even though page tables are still used to establish and limit addressability.[43,61,70] This option arises because existing virtual machine hardware architectures feature CPUs which support paging but I/O channels which do not. By allocating the same virtual and real addresses, it may be possible to eliminate the overhead normally incurred in channel program translation.*

Other policy approaches to virtual machine performance improvement affect the virtual machine definition and system generation of an operating system for it. Paging operations by the VMM are typically more efficient than file I/O operations by the operating system on the virtual machine.[6,19] Thus, it is sometimes a good strategy to have a very large virtual machine memory definition even though it would increase the amount of paging.[46] This will decrease the number of file I/O operations and may yield a net performance improvement.

Another approach is to "streamline" the VM definition.[43,72] If it is known that software running on a virtual machine will never use certain features, the VMM can be informed and may be able to provide more efficient support. In existing virtual machine systems, which support real-time interval timers, certain self-modifying channel programs, or paging in the VM's, is very costly. Performance is improved by ensuring that an operating system running on a particular virtual machine will not use these features.

Compromising the Interface Another performance improvement approach compromises the VMM/OS interface.[72] The VM architecture is routinely altered for certain specific operating systems by moving functions from that operating system to the VMM.[43] Thus, in addition to providing pure VM's for those applications which need it, the VMM also provides a *slightly extended virtual machine* which has a few new instructions which are effectively supervisor calls to the VMM. In CP-67 these calls were implemented via the customer engineering *diagnose* instruction, which is normally a model-dependent illegal instruction. A number of installations have experimented with specialized console[55] and file system[46,55] support provided via diagnose.

*There still may be a need to check addresses for bounds violations.

The unfortunate consequence of this approach is that it introduces the danger of producing a VMM with an incompatible interface. Thus, an operating system written for this interface may not run on any other "virtual machine" or even on the real machine.*

One proposed solution to this dilemma is to let the software determine whether it is running on a real or virtual machine.[72] If an operating system believes it is on a real machine, it behaves normally. If it discovers that it is on a virtual machine, it can take shortcuts which might reduce overhead. For example, it might ignore error codes which the virtual machine knows have been set by the VMM or use diagnose-type support for certain functions.

The debate over "pure" vs. "impure" virtual machines has been going on for several years.[10,27,39] The "purists" argue that an impure virtual machine will suffer the same disadvantages as a conventional operating system's non-standard extended machine interface. The "impurists" counter by pointing to the performance improvements over conventional virtual machine systems. Furthermore, they observe that the "impure" approach leads to clear, hierarchically organized operating systems, even if they are not virtual machines.[9,38]

Improved or New Mechanisms Improved hardware architectures for virtual machines arise from performance considerations, as well as system organization. Much of the overhead, identified above, is introduced because of an inadequate hardware base in present machines. Some of the techniques that have been developed are merely ad hoc approaches to these difficulties. A particular source of overhead in third-generation virtual machines is the trap and simulation of privileged instructions normally performed by the VMM. The System/370 virtual storage operating systems make significantly heavier use of certain privileged instructions than did their OS/360 real memory predecessors.[72] As a result, IBM has measured greater processor overhead in running the virtual storage operating systems under VM/370 Release 1. The solution has been to develop VM-assist, a firmware modification to the System/370 which eliminates the need for much of the privileged instruction software simulation.[44] Preliminary performance information for VM/370 Release 2 indicates success in reducing this processor overhead.[13,42]

Similar success is anticipated for virtualizable architectures such as the Hardware Virtualizer[34] or the Newcastle Recursive Virtual Machine.[48] For certain choices of resource maps, these machines should eliminate other forms of processor overhead. In particular, page table composition, channel program absolutization, virtual timers, and other virtual machine functions can be performed directly by hardware/firmware.

Unique Performance Problems for Virtual Machines Even if the direct overhead problems of VMs are solved (as above) a number of unique virtual machine performance problems may remain. Operating systems include algorithms for effectively managing the real resources of the computer system. In virtual machine systems these virtual resources may not correspond precisely to real resources, and consequently unusual performance problems may arise.[72] Two principal penalties are incurred in this case: (1) CPU time and other resources are wasted utilizing an algorithm which cannot work effectively, and (2) the actual results of the algorithm may be counterproductive – i.e., a random allocation algorithm might be better.

Some example situations where this phenomenon has been observed are:

- disk optimization for a disk which itself is mapped;
- spooling of unit record I/O in a virtual machine and then additional spooling by the VMM;
- paging by an operating system on the VM and paging by the VMM.

The last situation arises, for example, when the IBM OS/VS2 operating system is run on a virtual machine under VM/370. In that case there are two independent page replacement algorithms – one on the virtual machine and one by the VMM. In a recent analysis[37] it has been demonstrated that there are operating regions (i.e., sizes of memory) and page replacement algorithms for which each page fault handled in the VM will cause a second page fault which must be handled by the VMM. Furthermore, the well-known "least recently used" (LRU) page replacement algorithm is susceptible to this phenomenon.

Practice

The versatility and flexibility of virtual machine systems is only beginning to be understood. Virtual machines have been successfully utilized in a number of application areas on small, medium, and large-scale computer systems. We will examine a few of these uses.

Installation Management Virtual machines allow significant scheduling flexibility by permitting privileged software development, test and diagnostic functions, and multiple operating system execution concurrently with production uses of the system. The individual operating systems need not be identical (e.g., IBM's OS/360 and DOS/360 under VM/370). They may also be different versions of the same system (e.g., Release n and Release n+1). In the first case, VM's can aid in unifying installation procedures. In the second case they can greatly simplify the new release installation and conversion period.

An example of installation unification is the integration of a batch operating system and an interactive terminal system on common hardware.[56] Srodowa and Bates[70] have reported on their success in using VM techniques to unify IBM's OS/360 with the Michigan Terminal System (MTS) under UMMPS[2] at Wayne State University. VM/370 has been used to unify DOS/VS with CMS.[43]

Virtual machines can alleviate the new release "trauma" by permitting system generation and testing of the new release simultaneously with production uses of the old release, and by permitting the old and new releases to be run concurrently for an extended time period to permit users to convert their programs. In addition, when most users are finally converted to the new release, it is still possible to run the old release for the user who runs a program so infrequently that conversion is not justified.[59]

Figure 4 illustrates how user population shift from an old release to a new release can be accomplished using virtual machine techniques. As time advances, the relative percentage of the user population running under the new release increases (as represented by the larger "new" boxes to the right of the figure).

*In contrast, an operating system for a streamlined interface can be run on a real machine.

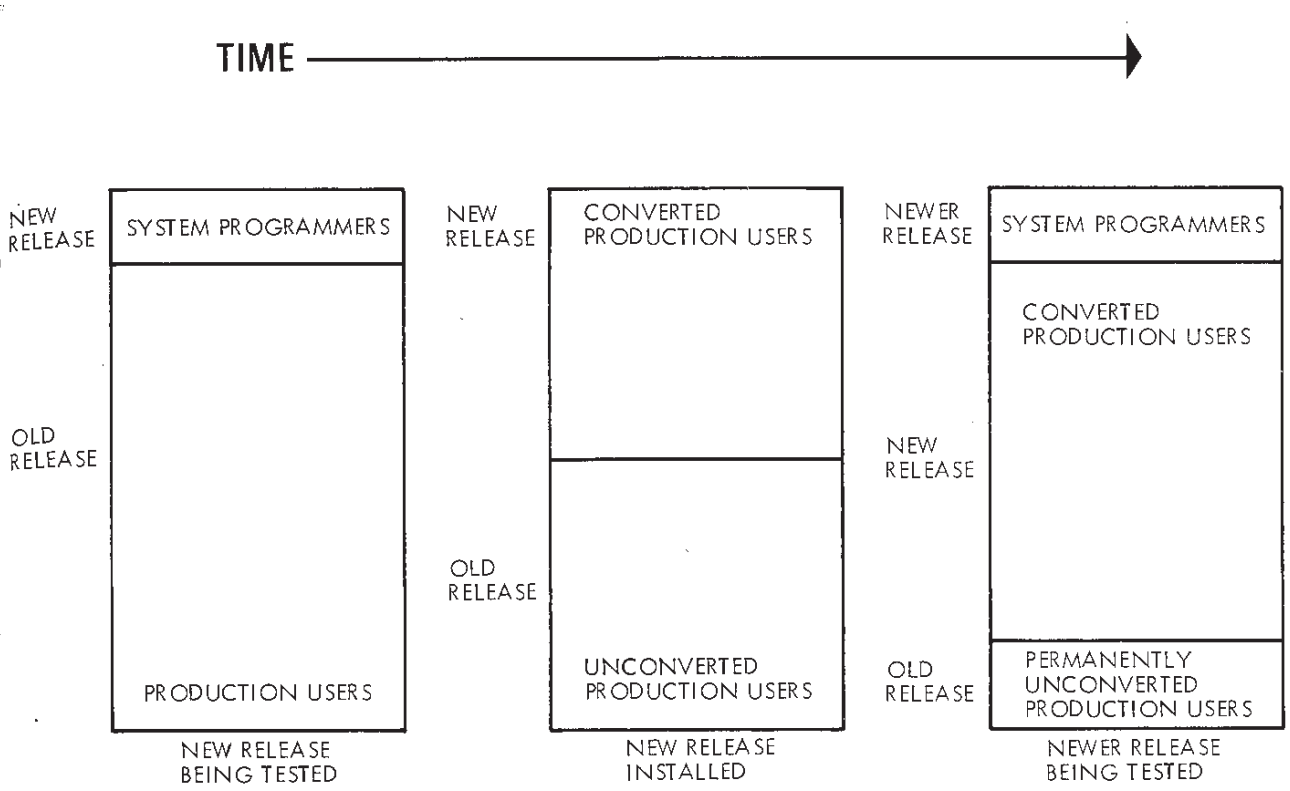


Figure 4. Virtual Machine Support for Multiple Releases of an Operating System

Virtual machines can also be used to *retrofit* old operating systems with new features. For example a new peripheral can be introduced into a computer system without changing the operating system software by making the VMM transform the new device into a virtual device which is already known to the operating system. As a result, the user will be able to take advantage of the new device's characteristics – e.g., technological improvement or lower cost – but will not have to add a new device handler to the operating system. While this approach may require a modification to the VMM, it should be a much less complex task since the VMM is a significantly simpler and smaller software construct. (See Figure 5.) This subject is studied in detail in Buzen and Goldberg.[18]

Another system retrofit that has been performed is adding virtual storage through VMM support rather than by modifying the operating system itself. This is effectively what was done in running CMS under CP-67.[57]

Privileged Software Development and Testing The improved testing of privileged software is a virtual machine application appropriate to any size system.[69] Since minicomputers are often used in OEM applications as part of larger systems, a great deal of "user" software produced for minicomputers includes privileged I/O routines. With the increase in size and power of minicomputers there have been a number of proposals for VM's for minicomputer software development.[23,48,63]

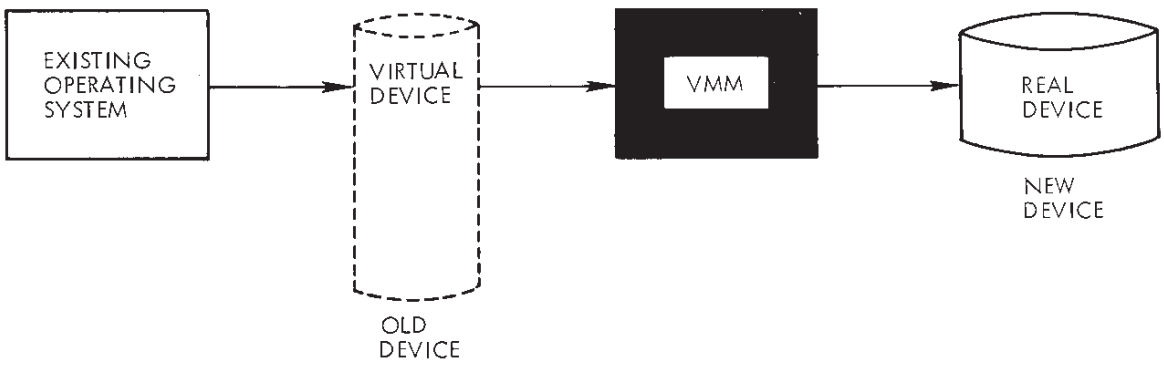


Figure 5. Virtual Device Support

Software development applications on larger systems are well-known.[27,71] IBM uses VM/370 internally to develop and maintain VM/370 and the IBM VS operating systems.[13] This approach eliminates much stand-alone debugging, with its inherently poor resource utilization and inelegant debugging tools. Before System/370 hardware was available, IBM used a similar technique with the CP-67 system on the 360/67. Since System/370 is so similar to System/360, it was possible to modify the standard CP-67 software to produce virtual 370's instead of 360's.[13]

Another software development possibility concerns the testing of computer network software on a single physical machine. Virtual machine systems make it possible to configure a network of virtual machines in which messages are sent between virtual machines. These messages may be routed over simulated transmission control units (see Figure 6a) or they may even be sent out over physical communication lines before getting to the receiving virtual machine (see Figure 6b). The geographical locality of such a system provides unique aid in checking out network software. Virtual machines have also been used in a number of other network[63,71] and multiple processor[5,33,43] applications.

User software and particularly system software development activities have been aided by the availability of unusually powerful debugging and performance monitoring tools. These tools have ranged from facilities as simple as a simulated VM operator console (called "console functions")[21,43,57] illustrated in Figure 7a, to complex debugging languages and systems, such as the SPY system[12,49] shown in Figure 7b. With virtual machine recursion, it is even possible to debug an operating system under a specially enhanced "debugger" version of the VMM which itself is running on a virtual machine under the standard VMM (see Figure 7c).[33,34,48] Galley has incorporated some of these virtual machine debugging tools into an interactive graphics package to produce an unusually powerful operating system debugging environment for the DEC PDP-10.[29] The power and flexibility of virtual machine systems is, in part, mirrored by the variety of debugging styles and tools which are available.

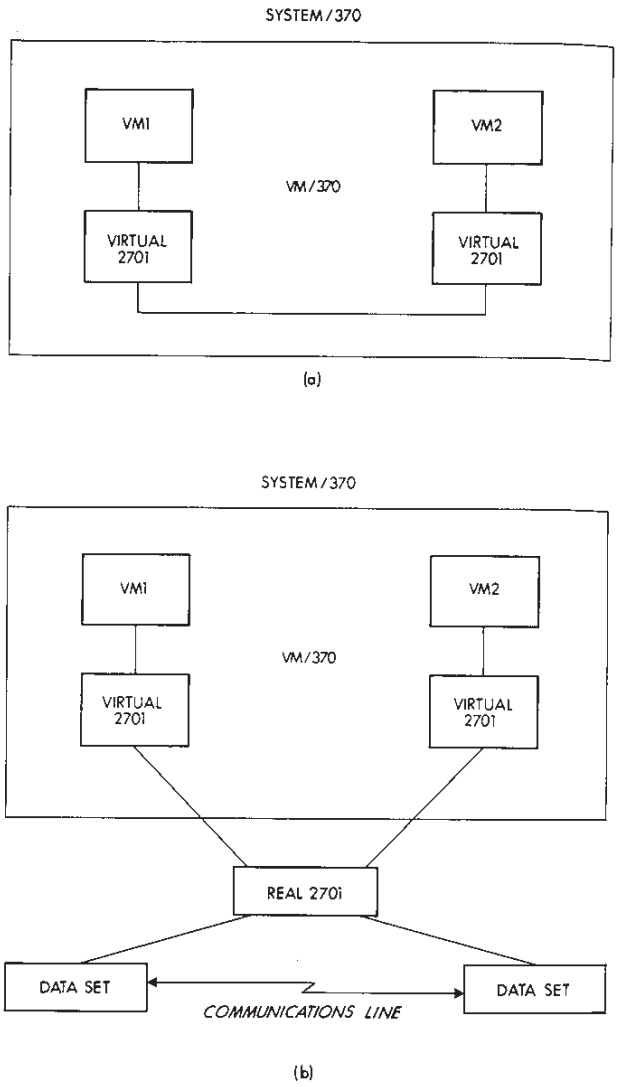


Figure 6. Virtual Machine Network with VM/370

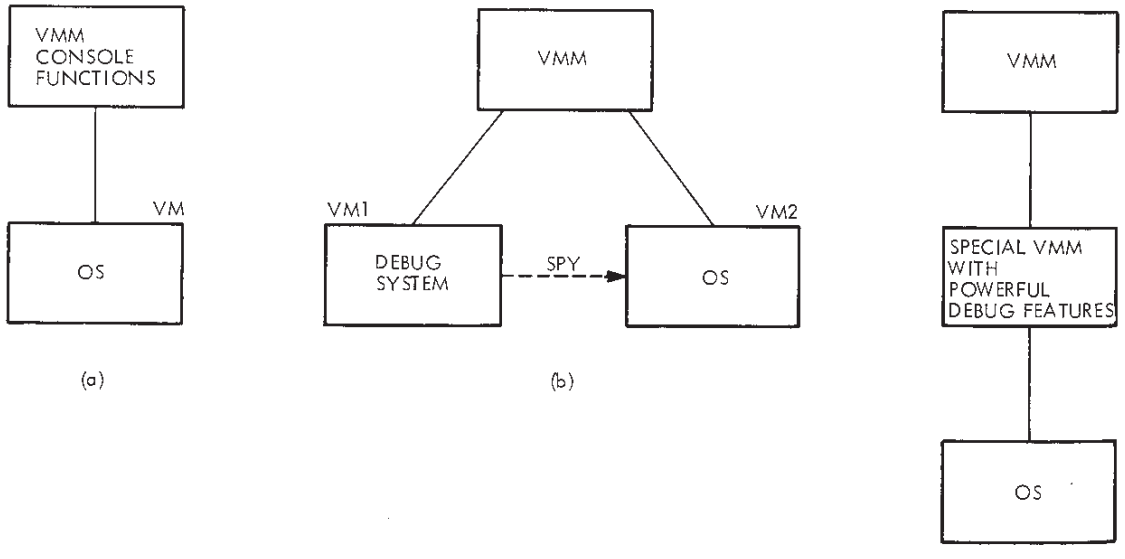


Figure 7. Virtual Machine Debugging Styles

Education Virtual machines provide a unique opportunity to revolutionize computer systems architecture and systems programming education. It now becomes possible to construct sample student operating systems which must run and interface with the bare hardware. Students will be able to gain practical experience in changing an I/O device handler, modifying a page replacement algorithm or even developing a computer network (see above).

Software Reliability From the standpoint of reliability one of the most important aspects of virtual machine systems is the high degree of isolation that a virtual machine monitor provides for each virtual machine operating under its control. In particular, a programming error in one operating system will not affect the operation of another operating system running on an independent virtual machine controlled by the same monitor. Thus virtual machine monitors can localize and control the impact of operating system errors in much the same way that conventional systems localize and control the impact of user program errors. In multiprogramming applications where both high availability and graceful degradation in the midst of failures are required, virtual machine systems can, for a large class of utility functions, be shown to have a quantifiable advantage over conventionally organized systems.[14,15,16]

A key principle in the analysis of software reliability is that the VMM is likely to be correct – i.e., probability of failure is near zero. This assumption is reasonable because the VMM is likely to be a very small program with limited functionality and can be largely checked out by running the system diagnostics. Furthermore, advanced architectural developments such as the Hardware Virtualizer discussed above will further reduce the amount of VMM software.

Data Security Virtual machine techniques for data security is a topic of considerable current interest.[63,64] Some of the reasons for the pursuit of work in this field are related to the isolation and "system hygiene" discussion presented above for reliability.[4,62] Madnick and Donovan[51] have suggested that independent redundant security mechanisms contribute to the empirically observed "good" security of VM/370. For example, in VM/370 the Dynamic Address Translation (DAT) set by the VM/370 control program (VMM) and the storage locks and keys set by OS/360 provide redundant main memory security mechanisms.

Conclusion

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.[27] Virtual machines have been endorsed by IBM (in the continuing support provided for VM/370)[43,44] and are under active study by the various data-security-conscious organizations.[64] In March 1973, 100 specialists attended an ACM sponsored Workshop on Virtual Computer Systems[35] and in June 1973, 500 individuals attended the Virtual Machines session at the National Computer Conference. As a result of the substantial interest in the field shown by manufacturers, computer scientists, and users, we feel certain that there will be a further spread of virtual machine systems, a development of more efficient virtualizable architectures, and a succession of significant new applications.

Acknowledgments

The author would like to thank the many people (some of whose names appear in the bibliography) with whom he has discussed virtual machine research over the years. Special thanks are due to U. O. Gagliardi, J. P. Buzen, S. E. Madnick, G. J. Popek, and H. S. Schwenk. Finally, the author would like to acknowedge the cooperation and support of the guest editor, R. R. Muntz, during the preparation of this paper.

Robert P. Goldberg is a member of the Honeywell Information Systems Technical Office in Waltham, Massachusetts, and also a Lecturer on Computer Science at Harvard University. His current research interests include computer architecture, operating systems design and evaluation, and data management systems.

From 1966 to 1971 he was a member of the research staff at MIT, first at Lincoln Laboratory and then at Project MAC. From 1971 to 1972, Dr. Goldberg served as consultant to the Director of Engineering at Honeywell's Boston Computer Operations. His teaching experience also includes lectureships at Brandeis University and Northeastern University.

Dr. Goldberg is a member of the ACM. He was the organizer of the Virtual Machines session at the 1973 National Computer Conference, was the Program Chairman and Proceedings Editor for the ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, 1973, and has written and lectured extensively on many different aspects of virtual machine systems.

Dr. Goldberg received the BS degree in Mathematics from MIT in 1965 and the MA and Ph.D. degrees in Applied Mathematics from Harvard University, in 1969 and 1973, respectively.

References and Bibliography

This bibliography lists many of the papers which have been written about virtual machines, but it is by no means an exhaustive list.

1. Adair, R., R. U. Bayles, L. W. Comeau, and R. J. Creasy, "A Virtual Machine System for the 360/40." *Cambridge Scientific Center Report No. G320-2007*, May 1966.

2. Alexander, M. T., *Time Sharing Supervisor Programs*. University of Michigan Comp. Center, May 1969, revised May 1970.

3. Ancilotti, R., R. Cavina, and N. Lijtmaer, "Virtual Input-Output in a Virtual Environment." *ACM AICA International Computer Symposium Proceedings*, Venice, Italy, April 12-14, 1972, pp. 302-312.

4. Attansio, C. R., "Virtual Machines and Data Security." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, Cambridge, MA, 1973.

5. Auroux, A. and C. Hans, "Le Concept de Machines Virtuelles." *Revue Francaise d'Informatique et de Recherche Operationelle*, 2e annee, 15, 1968, pp. 45-51.

6. Bard, Y., "Performance Criteria and Measurement for a Time-Sharing System." *IBM Systems Journal,* Vol. 10, No. 3, 1971, pp. 193-216.

7. Bard, Y., "An Analytic Model of CP-67 – VM/370." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

8. Bairstow, J. N., "Many From One: The Virtual Machine Arrives." *Computer Decisions,* January 1970, pp. 29-31.

9. Bellino, J., and C. Hans, "Virtual Machine or Virtual Operating System." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

10. Bellino, J., *Mecanismes de Base Dans Les Systemes Superviseurs: Conception et Realisation d'un Systeme a Acces Multiples.* These, L'Universite Scientifique et Medicale de Grenoble, September 28, 1973.

11. Bellino, J., and Ph. Potin, "Mechanismes d'un Hyperviseur." *Report of IBM Scientific Center,* Grenoble, France.

12. Berthaud, M., M. Jacolin, Ph. Potin, and H. Savary, "Coupling Virtual Machines and System Construction." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

13. Bury, C., Conversations with R. P. Goldberg, 1973-1974.

14. Buzen, J. P., P. P. Chen, and R. P. Goldberg, "Virtual Machine Techniques for Improving Software Reliability." *Proceedings IEEE Symposium on Computer Software Reliability,* New York, 1973.

15. Buzen, J. P., P. P. Chen, and R. P. Goldberg, "A Note on Virtual Machines and Software Reliability." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

16. Buzen, J. P., and U. O. Gagliardi, "The Evolution of Virtual Machine Architecture." *AFIPS Conference Proceedings, 1973* NCC, AFIPS Press, Montvale, N. J.

17. Buzen, J. P., and U. O. Gagliardi, "Introduction to Virtual Machines." *Honeywell Computer Journal,* Vol. 7, No. 4, 1973.

18. Buzen, J. P., and R. P. Goldberg, "Virtual Machine Techniques for Introducing Peripherals into Computer Systems." *Computer Peripherals – Benefactor or Bottleneck? Digest of Papers COMPCON 74,* San Francisco, February 1974, pp. 157-160.

19. Calloway, P. H. "Performance Considerations for the Use of the Virtual Machine Capability." *Report RC-3360,* IBM Corporation, T. J. Watson Research Laboratory, Yorktown Heights, NY, May 12, 1971.

20. Calloway, P. H., J. P. Considine, and C. H. Thompson, "Uses of Virtual Storage Systems in a Scientific Environment." *IBM Systems Journal,* Vol. 11, No. 3, 1972, pp. 200-218.

21. Casarosa, V., and C. Paoli, "VHM: A Virtual Hardware Monitor." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

22. Denning, P. J., "Third Generation Computer Systems." *Computing Surveys,* Vol. 3, No. 4, December 1971.

23. Dickman, L. I., "Small Virtual Machines: A Survey." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

24. Field, M. S., "Multi Access Systems – The Virtual Machine Approach." *IBM Cambridge Scientific Center Report 320-2033,* September 1968.

25. Frasson, C., "Simulation of Input-Output Units." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

26. Fuchi, K., H. Tanaka, Y. Namago, and T. Yuba, "A Program Simulator by Partial Interpretation." *2nd Symposium on Operating Systems Principles,* Princeton, NJ, October 1969, pp. 97-104.

27. Gagliardi, U. O., "Chairman's Report of the Architecture Panel Recommendation." *Tri-Service Workshop on the High Cost of Software,* Monterey, CA, 1973.

28. Gagliardi, U. O., and R. P. Goldberg, "Virtualizable Architectures." *Proceedings of 1972 ACM AICA International Comp. Symposium,* Venice, Italy, April 1972, pp. 527-538.

29. Galley, S. W., "PDP-10 Virtual Machines." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

30. Goldberg, R. P., "Virtual Machine Systems." *MIT Lincoln Laboratory Report No. MS-2687* (also 28L-0036), Lexington, MA, September 1969.

31. Goldberg, R. P., "Virtual Machines: Semantics and Examples." *Proceedings IEEE Computer Society Conference,* Boston, MA, September 1971, pp. 141-142.

32. Goldberg, R. P., "Hardware Requirements for Virtual Machine Systems." *HICSS-4, Hawaii International Conference on System Sciences,* Honolulu, January 1971.

33. Goldberg, R. P., *Architectural Principles for Virtual Computer Systems.* Ph.D. Thesis, Division of Engineering and Applied Physics Harvard University, Cambridge, MA, 1972.

34. Goldberg, R. P., "Architecture of Virtual Machines." *AFIPS Conference Proceedings,* 1973 NCC, AFIPS Press, Montvale, NJ.

35. Goldberg, R. P. (ed.), *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems.* Cambridge, MA, 1973.

36. Goldberg, R. P., "Virtual Machines Architecture." *Honeywell Computer Journal,* Vol. 7, No. 4, 1973.

37. Goldberg, R. P., and R. Hassinger, "The Double Paging Anomaly." *AFIPS Conference Proceedings,* 1974 NCC, AFIPS Press, Montvale, NJ.

38. Hans, C., et al, "GMS Guide de l'Utilisateur." *Report of IBM Scientific Center of Grenoble,* Grenoble, France, July 1972.

39. Hans, C., *Contribution a l'Architecture de Mecanismes Elementaires Pour Certains Systemes Generateurs de Machines Virtuelles.* These, L'Universite Scientifique et Medicale de Grenoble, November 24, 1973.

40. Hoernes, G. E. and H. Hellerman, "An Experimental 360/40 for Time-Sharing." *Datamation,* Vol. 14, No. 4, April 1958, pp. 39-42.

41. Hogg, J., and P. Madderom, "The Virtual Machine Facility – How to Fake a 360." University of British Columbia, University of Michigan Computer Center, Internal Note.

42. Horton, F. R., "Virtual Machine Assist: Performance." *Guide 37,* Boston, MA, 1973.

43. *IBM Virtual Machine Facility/370 Planning Guide.* IBM Corporation, Publication No. GC20-1801-0, 1972.

44. *IBM Virtual Machine Facility/370: Release 2 Planning Guide.* IBM Corporation Publication No. GC20-1814-0, 1973.

45. Keefe, D. D., "Hierarchical Control Programs for Systems Evaluation." *IBM Systems Journal,* Vol. 7, No. 2, 1968, pp. 123-133.

46. Kogut, R. M., "The Segment Based File Support System." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

47. Lauer, H. C. and C. R. Snow, "Is Supervisor-State Necessary?" *Proceedings ACM AICA International Computing Symposium,* Venice, Italy, 1972.

48. Lauer, H. C., and D. Wyeth, "A Recursive Virtual Machine Architecture." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

49. Lefebvre, P., *Spy: Un Systeme de Controle.* These, L'University Scientifique de Medicale de Grenoble, July 5, 1972.

50. Madnick, S. E., "Time-Sharing Systems: Virtual Machine Concept vs. Conventional Approach." *Modern Data,* Vol. 2, No. 3, March 1969, pp. 34-36.

51. Madnick, S. E., and J. J. Donovan, "Application and Analysis of the Virtual Machine Approach to Information System Security and Isolation." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

52. Madnick, S. E., and J. J. Donovan, *Operating Systems.* McGraw-Hill, New York, 1974.

53. Mallach, E. G., "Emulation – A Survey". *Honeywell Computer Journal,* Vol. 6, No. 4, 1972.

54. Mallach, E. G., "On the Relationship between Emulators and Virtual Machines." *Proceedings ACM SIGOPS-SIGARCH Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

55. March, J. H., "The Design and Implementation of a Virtual Machine Operating System Using a Virtual Access Method." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

56. McGrath, M., "Virtual Machine Computing in an Engineering Environment." *IBM Systems Journal,* Vol. 11, No. 2, 1972, pp. 131-149.

57. Meyer, P. A., and L. H. Seawright, "A Virtual Machine Time-Sharing System." *IBM Systems Journal,* Vol. 9, No. 3, 1970, pp. 199-218.

58. O'Neill, R. W., "Experience Using a Time-Shared Multi-Programming System with Dynamic Address Translation." *AFIPS Conference Proceedings,* Vol. 30, 1967, AFIPS Press, Montvale, NJ.

59. Parmelee, R. P., "Virtual Machines – Some Unexpected Applications." *Proceedings IEEE Computer Society Conference,* Boston, MA, September 1971.

60. Parmelee, R. P., T. I. Peterson, C. C. Tillman, and D. J. Hatfield, "Virtual Storage and Virtual Machine Concepts." *IBM Systems Journal,* Vol. 11, No. 2, 1972, pp. 99-129.

61. Parmelee, R. P., "Preferred Virtual Machines for CP-67." *IBM Cambridge Scientific Center Report No. G320-2068.*

62. Parnas, D. L., and W. R. Price, "The Design of the Virtual Memory Aspects of a Virtual Machine." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

63. Popek, G. J., and C. Kline, "Verifiable Secure Operating Systems Software." *AFIPS Conference Proceedings,* 1974 NCC, AFIPS Press, Montvale, NJ.

64. Popek, G. J., "A Survey of Protection Structures." *Computer,* Vol. 7, No. 6, June 1974.

65. Popek, G. J., and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures." *Communications of the ACM,* Vol. 17, No. 7, 1974.

66. Sayre, D., "On Virtual Systems." *IBM Corporation T. J. Watson Research Laboratory,* Yorktown Heights, NY, April 15, 1966.

67. Sayre, D., "Adding Computers Virtually." *IBM Corporation Computer Report,* Vol. 3, No. 2, March 1967, pp. 12-15.

68. Schroeder, M. D., "Performance of the GE 645 Associative Memory While Multics is in Operation." *Proceedings ACM-SIGOPS Workshop on System Performance Evaluation,* Cambridge, MA, April 1971, pp. 227-245.

69. Schwenk, H., "Virtual Micromachines." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.

70. Srodawa, R. J., and L. A. Bates, "An Efficient Virtual Machine Implementation." *Proceedings AFIPS National Computer Conference 1973.*

71. Winett, J. M., "Virtual Machines for Developing Systems Software." *Proceedings IEEE Computer Society Conference,* Boston, MA, September 1971.

72. Young, C. J., "Extended Architecture and Hypervisor Performance." *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems,* Cambridge, MA, 1973.