The evolution of virtual machine architecture*

by J. P. BUZEN and U. O. GAGLIARDI

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

INTRODUCTION

In the early 1960's two major evolutionary steps were taken with regard to computing systems architecture. These were the emergence of I/O processors and the use of multiprogramming to improve resource utilization and overall performance. As a consequence of the first step computing systems became multiprocessor configurations where nonidentical processors could have access to the common main memory of the system. The second step resulted in several computational processes sharing a single processor on a time-multiplexed basis while vying for a common pool of resources.

Both these developments introduced very serious potential problems for system integrity. An I/O processor executing an "incorrect" channel program could alter areas of main memory that belonged to other computations or to the nucleus of the software system. A computational process executing an "incorrect" procedure could cause similar problems to arise. Since abundant experience had demonstrated that it was not possible to rely on the "correctness" of all software, the multi-processing/multiprogramming architectures of the third generation had to rely on a completely new approach.

DUAL STATE ARCHITECTURE

The approach chosen was to separate the software into two classes: the first containing a relatively small amount of code which was presumed to be logically correct, the second containing all the rest. At the same time the system architecture was defined so that all functionality which could cause undesirable interference between processes was strictly denied to the second class of software.

Essentially, third generation architectures created two distinct modes of system operation (privileged/non-privileged, master/slave, system/user, etc.) and permitted certain critical operations to be performed only in the more privileged state. The critical operations restricted to privileged state typically include such functions as channel program initiation, modification of address mapping mechanisms, direct monitoring of external interrupts, etc.

Experience has shown that this solution can be quite effective if the privileged software is limited in quantity, is stable in the sense that few changes are made over long periods of time, and is written by skilled professional programmers.

While this architectural principle has proven its value by fostering the development of computing systems with true simultaneity of I/O operations and high overall resource utilization, it has generated a whole host of problems of its own. These problems arise from the fact that the only software which has complete access to and control of all the functional capabilities of the hardware is the privileged software nucleus.

Probably the most serious difficulty arises in the area of program transportability since non-privileged programs are actually written for the extended machine formed by the privileged software nucleus plus the non-privileged functions of the hardware. These extended machines are more difficult to standardize than hardware machines since it is relatively easy to modify or extend a system whose primitives are in part implemented in software. This has frequently resulted in a multiplicity of extended machines running on what would otherwise be compatible hardware machines. A user who wishes to run programs from another installation which were written for a different extended machine is faced with either scheduling his installation to run the "foreign" software nucleus for some period of time or converting the programs to his installation's extended machine. Neither of these alternatives is particularly attractive in the majority of cases.

Another problem is that it is impossible to run two versions of the privileged software nucleus at the same time. This makes continued development and modification of the nucleus difficult since system programmers often have to work odd hours in order to have a dedicated machine at their disposal. In addition to the inconvenience this may cause, such procedures do not result in

* This work was sponsored in part by the Electronic Systems Division, U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract Number F19628-70-C-0217.
very efficient utilization of resources since a single programmer who is modifying or debugging a system from a console does not normally generate a very heavy load.

A final problem is that test and diagnostic software has to have access to and control of all the functional capabilities of the hardware and thus cannot be run simultaneously with the privileged software nucleus. This in turn severely curtails the amount of testing and diagnosis that can be performed without interfering with normal production schedules. The ever increasing emphasis on computer system reliability will tend to make this an even more serious problem in the future.

THE VIRTUAL MACHINE CONCEPT

Figure 1 illustrates the conventional dual state extended machine architecture which is responsible for all the difficulties that were cited in the preceding section. As can be seen in the Figure, the crux of the problem is that conventional systems contain only one basic machine interface* and thus are only capable of running one privileged software nucleus at any given time. Note, however, that conventional systems are capable of running a number of user programs at the same time since the privileged software nucleus can support several extended machine interfaces. If it were possible to construct a privileged software nucleus which supported several copies of the basic machine interface rather than the extended

* A basic machine interface is the set of all software visible objects and instructions that are directly supported by the hardware and firmware of a particular system.
programs but rather allows them to run directly on the bare machine for much of the time. However, the VMM will occasionally trap certain instructions and execute them interpretively in order to insure the integrity of the system as a whole. Control is returned to the executing program after the interpretive phase is completed. Thus program execution on a virtual machine is quite similar to program execution on an extended machine: the majority of the instructions execute directly without software intervention, but occasionally the controlling software will seize control in order to perform a necessary interpretive operation.

VIRTUAL MACHINES AND EMULATORS

Figure 2 is not intended to imply that the basic machine interface supported by the VMM must be identical to the interface of the bare machine that the VMM runs on. However, these interfaces often are identical in practice. When they are not, they are usually members of the same computer family as in the case of the original version of CP-67, a VMM which runs on an IBM 360 Model 67 (with paging) and supports a virtual IBM 360 Model 65 (without paging) beneath it.

When the two interfaces are distinctly different the program which supports the virtual interface is usually called an emulator rather than a virtual machine monitor. Aside from this comparatively minor difference, virtual machines and emulators are quite similar in both structure and function. However, because they are not implemented with the same objectives in mind, the two concepts often give the appearance of being markedly different.

Virtual machine monitors are usually implemented without adding special order code translation firmware to the bare machine. Thus, most VMM's project either the same basic machine interface or a restricted subset of the basic machines interface that they themselves run on. In addition, VMM's are usually capable of supporting several independent virtual machines beneath them since many of the most important VMM applications involve concurrent processing of more than one privileged software nucleus. Finally, VMM's which do project the same interface as the one they run on must deal with the problem of recursion (i.e., running a virtual machine monitor under itself). In fact, proper handling of exception conditions under recursion is one of the more challenging problems of virtual machine design.

Emulators, by contrast, map the basic machine interface of one machine onto the basic machine interface of another and thus never need be concerned with the problem of recursion. Another point of difference is that an emulator normally supports only one copy of a basic machine interface and thus does not have to deal with the scheduling and resource allocation problems which arise when multiple independent copies are supported. Still another implementation difference is that emulators must frequently deal with more complex I/O problems than virtual machine monitors do since the emulated system and the system that the emulator is running on may have very different I/O devices and channel architecture.

Modern integrated emulators exhibit another difference from the virtual machine monitor illustrated in Figure 2 in that an integrated emulator runs on an extended machine rather than running directly on a bare machine. However, it is possible to create virtual machine monitors which also run on extended machines as indicated in Figure 3. Goldberg refers to such systems as Type II virtual machines. Systems of the type depicted in Figure 2 are referred to as Type I virtual machines.

It should be apparent from this discussion that virtual machines and emulators have a great deal in common and that significant interchange of ideas is possible. For a further discussion of this point, see Mallach.

ADDITIONAL APPLICATIONS

It has already been indicated that virtual machine systems can be used to resolve a number of problems in program portability, software development, and "test and diagnostic" scheduling. These are not the only situations in which virtual machines are of interest, and in fact virtual machine systems can be applied to a number of equally significant problems in the areas of security, reliability and measurement.

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 basic machine interface operating under its control. In particular, a programming error in one privileged software nucleus will not affect the operation of

![Figure 3—Type II virtual machine organization](image-url)
another privileged software nucleus 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.9

The high degree of isolation that exists between independent virtual machines also makes these systems important in certain privacy and security applications.7 Since a privileged software nucleus has, in principle, no way of determining whether it is running on a virtual or a real machine, it has no way of spying on or altering any other virtual machine that may be coexisting with it in the same system. Thus the isolation of independent virtual machines is important for privacy and security as well as system reliability.

Another consideration of interest in this context is that virtual machine monitors typically do not require a large amount of code or a high degree of logical complexity. This makes it feasible to carry out comprehensive checkout procedures and thus insure high overall reliability as well as the integrity of any special privacy and security features that may be present.

The applications of virtual machines to the measurement of system behavior are somewhat different in nature. It has already been noted that existing virtual machine monitors intercept certain instructions for interpretive execution rather than allowing them to execute directly on the bare machine. These intercepted instructions typically include I/O requests and most other supervisory calls. Hence, if it is desired to measure the frequency of I/O operations or the amount of supervisory overhead in a system, it is possible to modify the virtual machine monitor to collect these statistics and then run the system under that modified monitor. In this way no changes have to be made to the system itself. A large body of experimental data has been collected by using virtual machine monitors in this fashion.8,10,11

EARLY VIRTUAL MACHINES

Virtual machine monitors for computers with dual state architecture first appeared in the mid 1960's. Early VMM's were most noteworthy for the manner in which they controlled the processor state, main memory and I/O operations of the virtual machines which ran under their control. This section presents a brief description and analysis of the special mapping techniques that were employed in these early systems.

**Processor state mapping**

The mapping of processor state was probably the most unusual feature of early virtual machine monitors. If a VMM did not maintain proper control over the actual state of the processor, a privileged software nucleus executing on a virtual machine could conceivably enter privileged mode and gain unrestricted access to the entire system. It would then be able to interfere at will with the VMM itself or with any other virtual machine present in the system. Since this is obviously an unacceptable situation, some mapping of virtual processor state to actual processor state was required.

The solution that was adopted involved running all virtual machine processes in the non-privileged state and having the virtual machine monitor maintain a virtual state indicator which was set to either privileged or non-privileged mode, depending on the state the process would be in if it were executing directly on the bare machine. Instructions which were insensitive to the actual state of the machine were then allowed to execute directly on the bare machine. Instructions which were insensitive to the actual state of the machine were then allowed to execute directly on the bare machine with no intervention on the part of the VMM. All other instructions were trapped by the VMM and executed interpretively, using the virtual system state indicator to determine the appropriate action in each case.

The particular instructions which have to be trapped for interpretive execution vary from machine to machine, but general guidelines for determining the types of instructions which require trapping can be identified.14 First and most obvious is any instruction which can change the state of the machine. Such instructions must be trapped to allow the virtual state indicator to be properly maintained. A second type is any instruction which directly queries the state of the machine, or any instruction which is executed differently in privileged and non-privileged state. These instructions have to be executed interpretively since the virtual and actual states of the system are not always the same.

**Memory mapping**

Early virtual machine monitors also mapped the main memory addresses generated by processes running on virtual machines. This was necessary because each virtual machine running under a VMM normally has an address space consisting of a single linear sequence that begins at zero. Since physical memory contains only one true zero and one linear addressing sequence, some form of address mapping is required in order to run several virtual machines at the same time.

Another reason for address mapping is that certain locations in main memory are normally used by the hardware to determine where to transfer control when an interrupt is received. Since most processors automatically enter privileged mode following an interrupt generated transfer of control, it is necessary to prevent a process executing on a virtual machine from obtaining access to these locations. By mapping these special locations in virtual address space into ordinary locations in real memory, the VMM can retain complete control over the

From the collection of the Computer History Museum (www.computerhistory.org)
actual locations used by the hardware and thus safeguard the integrity of the entire system.

Early VMM’s relied on conventional paging techniques to solve their memory mapping problems. Faults generated by references to pages that were not in memory were handled entirely by the VMM’s and were totally invisible to processes running on the virtual machines. VMM’s also gained control after faults caused by references to addresses that exceeded the limits of a virtual machine’s memory, but in this case all the VMM had to do was set the virtual state indicator to privileged mode and transfer control to the section of the virtual machine’s privileged software nucleus which normally handles out-of-bounds memory exceptions. These traps were thus completely visible to the software running on the virtual machine, and in a sense they should not have been directed to the VMM at all. More advanced virtual machine architectures permit these traps to be handled directly by the appropriate level of control. 15,16

It should be noted that the virtual machines supported by early VMM’s did not include paging mechanisms within their basic machine interfaces. In other words, only privileged software nuclei which were designed to run on non-paged machines could be run under these early virtual machine monitors. Thus these VMM’s could not be run recursively.

I/O mapping

The final problem which early VMM’s had to resolve was the mapping of I/O operations. As in the case of main memory addresses, there are a number of reasons why I/O operations have to be mapped. The primary reason is that the only addresses which appear in programs running on virtual machines are virtual (mapped) addresses. However, existing I/O channels require absolute (real) addresses for proper operation since timing considerations are the only addresses which appear in programs runn­ing on virtual machines are virtual (mapped) addresses. These traps were thus completely visible to the software running on the virtual machine, and in a sense they should not have been directed to the VMM at all. More advanced virtual machine architec­tures permit these traps to be handled directly by the appropriate level of control. 15,16

It should be noted that the virtual machines supported by early VMM’s did not include paging mechanisms within their basic machine interfaces. In other words, only privileged software nuclei which were designed to run on non-paged machines could be run under these early virtual machine monitors. Thus these VMM’s could not be run recursively.

One of the drawbacks of copying channel programs into private work areas and executing the absolutized copies is that channel programs which dynamically modify themselves during execution sometimes do not operate correctly. Hence it was not possible to execute certain self-modifying channel programs in early VMM’s. However, since the majority of commonly used channel programs are not self-modifying, this lack of functionality could frequently be tolerated without serious inconvenience.

Channel program absolutization is not the only reason for VMM intervention in I/O operations. Intervention is also needed to maintain system integrity since an incorrectly written channel program can interfere with other virtual machines or with the VMM itself. The need for intervention also arises in the case of communication with the operator’s console. This communication must clearly be mapped to some other device since there is normally only one real operator’s console in a system.

A final point is that VMM intervention in I/O operations makes it possible to transform requests for one device into requests for another (e.g., tape requests to disk requests) and to provide a virtual machine with devices which have no real counterpart (e.g., a disk with only five cylinders). These features are not essential to VMM operation, but they have proven to be extremely valuable by-products in certain applications.

Summary

In summary, early VMM’s ran all programs in non-privileged mode, mapped main memory through paging techniques, and performed all I/O operations interpretively. Thus they could only be implemented on paged computer systems which had the ability to trap all instructions that could change or query processor state, initiate I/O operations, or in some manner be “sensitive” to the state of the processor. 15 Note that paging per se is not really necessary for virtual machine implementation, and in fact any memory relocation mechanism which can be made in­visible to non-privileged processes will suffice. However, the trapping of all sensitive instructions in non-privileged mode is an absolute requirement for this type of virtual machine architecture. Since very few systems provide all the necessary traps, only a limited number of these VMM’s have actually been constructed. 12,13,17,19

PAGED VIRTUAL MACHINES

It has already been noted that early VMM’s did not support paged virtual machines and thus could not be run on the virtual machines they created. This lack of a recursive capability implied that VMM testing and development had to be carried out on a dedicated processor. In order to overcome this difficulty and to achieve a more satisfying degree of logical completeness, CP-67 was modified so that it could be run recursively. 18
The major problem which had to be overcome was the efficient handling of the additional paging operation that took place within the VMM itself.\textsuperscript{20} To put the problem in perspective, note that early VMM's used their page tables to map addresses in the virtual machine's memory into addresses in the real machine's memory. For example, virtual memory address $A'$ might be mapped into real memory address $A''$. However, processes running on paged virtual machines do not deal with addresses which refer directly to the virtual machine's memory the way address $A'$ does. Rather, an address $A$ used by such a process must be mapped into an address such as $A'$ by the page table of the virtual machine. Thus, in order to run a process on a paged virtual machine, a process generated address $A$ must first be mapped into a virtual machine memory address $A'$ by the virtual machine's page table, and then $A'$ must be mapped into a real address $A''$ by the VMM's page table.

In order to carry out this double mapping efficiently, the VMM constructs a composed page table (in which virtual process address $A$ is mapped into real address $A''$) and executes with this map controlling the address translation hardware. When the VMM transfers a page out of memory, it must first change its own page table and then recompose the composed map. Similarly, if the privileged software nucleus changes the virtual machine's page table, the VMM must be notified so that the composed map can be recomputed.

This second consideration poses some difficulties. Since the virtual machine's page tables are stored in ordinary (virtual) memory locations, instructions which reference the tables are not necessarily trapped by the VMM. Thus changes could theoretically go undetected by the VMM. However, any change to a page table must in practice be followed by an instruction to clear the associative memory since the processor might otherwise use an out of date associative memory entry in a subsequent reference. Fortunately, the instruction which clears the associative memory will cause a trap when executed in non-privileged mode and thus allow the VMM to recompose the composed page table. Therefore, as long as the privileged software nucleus is correctly written, the operation of a virtual machine will be identical to the operation of the corresponding real machine. If the privileged software nucleus fails to clear the associative memory after changing a page table entry, proper operation cannot be guaranteed in either case.

\section*{TYPE II VIRTUAL MACHINES}

VMM's which run on an extended machine interface are generally easier to construct than VMM's which run directly on a bare machine. This is because Type II VMM's can utilize the extended machine's instruction repertoire when carrying out complex operations such as I/O. In addition, the VMM can take advantage of the extended machine's memory management facilities (which may include paging) and its file system. Thus Type II virtual machines offer a number of implementation advantages.

\subsection*{Processor state mapping}

Type II virtual machines have been constructed for the extended machine interface projected by the UMMPS operating system.\textsuperscript{21} UMMPS runs on an IBM 360 Model 67, and thus the VMM which runs under UMMPS is able to utilize the same processor state mapping that CP-67 does. However, the instruction in the VMM which initiates operation of a virtual machine must inform UMMPS that subsequent privileged instruction traps generated by the virtual machine should not be acted on directly but should instead be referred to the VMM for appropriate interpretation.

\subsection*{Memory mapping}

The instruction which initiates operation of a virtual machine also instructs UMMPS to alter its page tables to reflect the fact that a new address space has been activated. The memory of the virtual machine created by the VMM is required to occupy a contiguous region beginning at a known address in the VMM's address space. Thus UMMPS creates the page table for the virtual machine simply by deleting certain entries from the page table used for the VMM and then subtracting a constant from the remaining virtual addresses so the new address space begins at zero. If the virtual machine being created is paged, it is then necessary to compose the resulting table with the page table that appears in the memory of the virtual machine. This latter operation is completely analogous to the creation of paged virtual machines under CP-67.

\subsection*{I/O mapping}

I/O operations in the original UMMPS Type II virtual machine were handled by having UMMPS transfer control to the VMM after trapping the instruction which initiated channel program execution. The VMM translated the channel program into its address space by applying the virtual machine's page map if necessary and then adding a constant relocation factor to each address. After performing this translation the VMM called upon UMMPS to execute the channel program. UMMPS then absolutized the channel program and initiated its execution.

In addition to the overhead it entailed, this mapping procedure made it impossible for the virtual machine to execute a self-modifying channel program. A recent modification to the UMMPS virtual machine monitor has been able to alleviate this situation.\textsuperscript{22} This modification involves positioning the virtual machine's memory in real
memory so that the virtual and real address of each location is identical. This eliminates the need for channel program absolutization and thus improves efficiency while at the same time making self-modification of channel programs possible.

One of the difficulties that had to be overcome when making this change to the VMM was that the real counterparts of certain virtual machine memory locations were already being used by UMMPS. The solution that was adopted was to simply re-write the virtual machine’s privileged software nucleus so that most of these locations were never used. A more detailed discussion of this point is provided by Srodawa and Bates. Parmelee describes a similar modification that has been made to CP-67.

SINGLE STATE ARCHITECTURE

One of the more unusual approaches to the problem of creating virtual machine architectures is based on the idea of eliminating privileged state entirely. The proponents of this approach argue that the primary—function of privileged state is to protect the processor’s address mapping mechanism. If the address mapping mechanism were removed from the basic machine interface and thereby made totally invisible to software, there would be no need to protect the mechanism and therefore no need for privileged state.

In these single state architectures all software visible addresses are relative addresses and the mechanism for translating these relative addresses to absolute addresses is always concealed. That is, each software level operates in an address space of some given size and structure but has no way of determining whether its addresses correspond literally to real memory addresses or whether they are mapped in some fashion. Since all addressing including I/O is done in this relative context, there is really no need for software to know absolute address and thus no generality is lost.

The central feature of this architecture is the manner in which software level N creates the address space of software level N+1. Basically, level N allocates a portion of its own address space for use by level N+1. The location of the address space of level N+1 is thus specified in terms of its relative address within level N. After defining the new address space, the level N software executes a special transfer of control instruction which changes the address mapping mechanism so that addresses will be translated relative to the new address space. At the same time, control passes to some location within that new space.

Note that this special instruction need not be privileged since by its nature it may only allocate a subset of the resources it already has access to. Thus it cannot cause interference with superior levels. Level N can protect itself from level N+1 by defining the address space of level N+1 so that it does not encompass any information which level N wishes to keep secure. In particular, the address map that level N sets up for level N+1 is excluded from level N+1’s address space.

When an addressing fault occurs, the architecture traps back to the next lower level and adjusts the address map accordingly. Thus the system must retain a complete catalog of all active maps and must be able to compose and decompose them when necessary. This is relatively easy to do when only relocation/bounds maps are permitted but more difficult when segmentation is involved.

Since each level sees the same bare machine interface except for a smaller address space, each level corresponds to a new virtual machine. Mapping of processor state is unnecessary, mapping of memory is defined by the level N VMM relative to its own address space and is completely invisible to level N+1, and mapping of I/O is treated as a special case of mapping of memory. The two published reports on this architecture are essentially preliminary documents. More details have to be worked out before a complete system can be defined.

THE VIRTUAL MACHINE FAULT

The single state architecture discussed in the preceding section provides a highly efficient environment for the creation of recursive virtual machine systems. However, the basic machine interface associated with this architecture lacks a number of features which are useful when writing a privileged software nucleus. These features, which are present to varying degrees in several late third generation computer systems, include descriptor based memory addressing, multi-layered rings of protection and process synchronization primitives.

A recent analysis of virtual machine architectures for these more complex systems is based on an important distinction between two different types of faults. The first type is associated with software visible features of a basic machine interface such as privileged/nonprivileged status, address mapping tables, etc. These faults are handled by the privileged software nucleus which runs that interface. The second type of fault appears only in virtual machine systems and is generated when a process attempts to alter a resource map that the VMM is maintaining or attempts to reference a resource which is available on a virtual machine but not the real system (e.g., a virtual machine memory location that is not in real memory). These faults are handled solely by the VMM and are completely invisible to the virtual machine itself.

Since conventional architectures support only the former type of fault, conventional VMMs are forced to map both fault types onto a single mechanism. As already noted, this is done by running all virtual machine proc-

---

* Faults caused by references to unavailable real resources were not clearly identified in this paper. The distinctions being drawn here are based on a later analysis by Goldberg.
esses in non-privileged mode, directing all faults to the VMM, and having the VMM "reflect" all faults of the first type back to the privileged software nucleus of the virtual machine.

An obvious improvement to this situation can be realized by creating an architecture which recognizes and supports both types of faults. A preliminary VMM design for a machine with this type of architecture has been proposed. The design relies on static composition of all resource maps and thus requires a trap to the VMM each time a privileged process attempts to alter a software visible map. However, the privileged/non-privileged distinction within a virtual machine is supported directly by the bare machine and a privileged process is allowed to read all software visible constructs (e.g., processor state) without generating any type of fault. The major value of this design is that it can be implemented on an existing system by making only a relatively small number of hardware/firmware modifications.

**DYNAMIC MAP COMPOSITION—THE HARDWARE VIRTUALIZER**

The clear distinction between virtual machine faults (handled by the VMM) and process exceptions (handled by the privileged software nucleus of the virtual machine) first appeared in a Ph.D. thesis by Goldberg. One of the essential ideas of the thesis is that the various resource maps which have to be invoked in order to run a process on a virtual machine should be automatically composed by the hardware and firmware of the system. Since map composition takes place dynamically, this proposal eliminates the need to generate a virtual machine fault each time a privileged process running on a virtual machine alters a software visible map. Thus the only cause of a virtual machine fault is a reference to a resource that is not present in a higher level virtual or real machine.

The thesis contains a detailed description of a "hardware virtualizer" which performs the map composition function. It includes a description of the virtualizer itself, the supporting control mechanisms, the instructions used for recursive virtual machine creation, and the various fault handling mechanisms. These details will not be considered here since they are treated in a companion paper.

It is interesting to note that the work on single state architecture can be regarded as a special case of the preceding analysis in which process exceptions caused by privileged state are completely eliminated and only virtual machine faults remain. Similarly, the earlier work of Gagliardi and Goldberg represents another special case in which map composition is carried out statically by the VMM and where additional virtual machine faults are generated each time a component of the composite map is modified. By carefully identifying the appropriate functionality and visibility of all the maps involved in virtual machine operation, Goldberg's later analysis provides a highly valuable model for the design of virtual machine architectures and for the analysis of additional problems in this area.

**CONCLUSION**

A number of issues related to the architecture and implementation of virtual machine systems remain to be resolved. These include the design of efficient I/O control mechanisms, the development of techniques for sharing resources among independent virtual machines, and the formulation of resource allocation policies that provide efficient virtual machine operation. Many of these issues were addressed at the ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems held recently at Harvard University's Center for Research in Computing Technology.

In view of the major commitment of at least one large computer manufacturer to the support of virtual machine systems, the emergence of powerful new theoretical insights, and the rapidly expanding list of applications, one can confidently predict a continuing succession of virtual machine implementations and theoretical advances in the future.

**ACKNOWLEDGMENT**

We would like to express our appreciation to Dr. R. P. Goldberg for generously providing us with much of the source material that was used in the preparation of this paper.

**REFERENCES**


*Proceedings* may be ordered from ACM Headquarters in New York City.


