This paper reviews virtual storage and virtual machine concepts, consolidating and updating earlier discussions. The manner in which actual virtual storage and machine systems have been implemented, and certain problems of current implementations, are described. To better illustrate the material, the virtual machine system CP-67 for the IBM System/360 Model 67 is considered at some length. An annotated bibliography is included.

Virtual storage and virtual machine concepts

by R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield

In recent years, the concepts of virtual storage and virtual machines have been of increasing interest in the computing community. This paper defines these concepts and describes how virtual storage systems and virtual machine systems are implemented and used. To give the reader a more concrete conception of these systems, the IBM virtual machine system CP-67 (Control Program – 67) is considered at some length.

A virtual storage system may be defined generally as any information storage system in which there is, or may be, a distinction between the *logical address* generated by a program and the *physical address* for some real storage device from which information is actually fetched. Similarly, a virtual machine system may be defined as a computing system in which the *instructions* issued by a program may be different from those actually executed by the hardware to perform a given task. Since instructions generally include storage addresses as well as operation codes, a virtual machine system may include virtual storage as well as other virtual hardware features. In this sense, the virtual machine concept is a generalization of the virtual storage concept, and indeed, existing virtual machine systems such as CP-67 do include virtual storage. Thus it is natural and convenient to treat virtual storage and virtual machine concepts in a single paper. Other papers that consider these topics are References 2, 4,6, and 7 on virtual storage and References 47, 68, 70, 91, and 96 on virtual machines.

In this paper, we first give an overview of virtual storage, discussing its advantages and two approaches to its organization and management. Then we describe details of implementation. In the latter part of the paper, we discuss virtual machines. With CP-67 used as the example, we describe the implementation and operation of virtual machines.

Virtual storage

overview

Viewed in a high-level programming context, the definition previously stated for a virtual storage system includes a variety of common storage schemes, e.g., conventional file systems in which data sets are addressed by name rather than directly in terms of device and position information. Normally, however, the term virtual storage refers to the addressing of individual memory words by central processor instructions, and in particular, to systems in which memory addresses are *translated* or *relocated* dynamically by hardware. A simple system of this type is seen in computers where a single relocation constant is added to effective (i.e., logical) addresses, as for example in the Disk Operating System/360 (DOS) emulator of the IBM System/370 Models 135, 145, and 155.[56] A more general form of virtual storage permits virtual address space to be split into pieces, each with its own dynamically changeable relocation constant, so that individual pieces can be swapped back and forth between main and auxiliary storage as deemed appropriate by the system control program. It is this latter form of virtual storage that concerns us in the present paper.

programmer advantages

Before going into the details of implementation, it is of interest to consider the advantages that virtual storage offers and to review some of the systems that employ it. From the programmer's viewpoint, a major advantage of virtual storage is the reduced need for concern about storage management. In particular, since only those portions of virtual storage that are actually in use need occupy main storage at any given time, it is possible to give the programmer much more logical address space than would otherwise be possible. Thus he can avoid working with overlay structures that are often necessary in conventional storage systems. In truth, of course, the overlaying of information takes place in a virtual storage system too, but it is handled automatically by the system and is logically transparent to the programmer. For the programmer who is developing software that must run in a broad range of system configurations, this advantage has particular significance. With virtual storage, a single version of a program can be developed that will run in any amount of main storage. Moreover, this single program can continue to run when a main storage module is taken off-line for maintenance, and it can be expected to perform more efficiently as new modules are acquired. As we shall see later, all of this does not necessarily mean that the programmer may be oblivious to the structure of his program if he wishes it to perform well in a virtual storage environment. However, it is easier to isolate and defer questions of program structure in this environment than in environments where overlays must be considered from the outset. More importantly, improvements in program structure for virtual storage environments tend to be valid independent of the amount of main storage available, and hence have broader payoffs.

system advantages

Virtual storage can also offer significant system advantages, namely, better storage utilization and increased potential for multiprogramming. In conventional storage systems, main storage may be under-utilized because of the fragmentation associated with the allocation of large contiguous regions. In virtual storage systems, the pieces into which virtual storage is divided can be allocated discontiguously through main storage wherever there is room. The resulting reduction in fragmentation, combined with the fact that only the active portions of virtual address space need be in main storage at any given time, can substantially increase storage utilization and, hence, the degree to which a system can be multiprogrammed.

systems with virtual storage

In reviewing systems that employ virtual storage, it is of interest to distinguish between two approaches to virtual storage organization and management. These are *paging*, where virtual storage is allocated to physical storage in fixed-length blocks called *pages*, and *segmentation*, where virtual storage is divided into variable-length *segments* that may or may not be subdivided into pages for allocation to physical storage.

Paging is generally logically transparent to the programmer, and may be considered solely a storage management mechanism. Segmentation may or may not be visible to the programmer as a means of structuring programs and data, depending on system software.

The first virtual storage system was implemented in the early 1960's on the Ferranti ATLAS, where paging was used primarily as a mechanism for "extending" a relatively small main store.[58,64] An early machine with segmented virtual storage was the Burrough's B5000, in which logically distinct program and data elements were allotted to different segments.[67] Other more recent systems implementing virtual storage include:

- IBM System/360 Model 40 modified for paging and used for the experimental virtual machine system CP-40[47,61,66]
- GE 645, a large-scale machine with segmentation and paging for which the MULTICS time-sharing system was implemented[51,52,54]

- IBM System/360 Model 67, a large-scale machine with segmentation and paging for which three time-sharing systems have been developed: TSS/360,[57,63,65,72] MTS,[48] and CP-67/CMS[53,55,68]
- RCA Spectra 70/46 and 70/61, medium-scale paging machines for which the TSOS time-sharing system was developed[69,73]
- XDS Sigma 7 (with memory-mapping option), a medium-scale paging machine.

Implementation of virtual storage

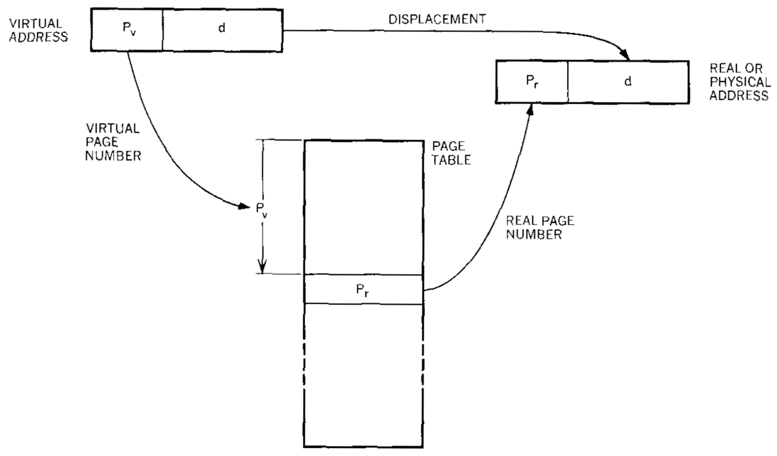
In this section, we consider the manner in which virtual storage systems are implemented and the implications of virtual storage on program performance. The implementation of a virtual storage system generally involves two distinct mechanisms: (1) an *address translation* mechanism for converting logical, or *virtual addresses*, into real addresses, and (2) a *storage management* mechanism for handling the transfer of information between main and auxiliary storage. For convenience of exposition, these mechanisms are treated separately in the following discussion, though in practice they may be highly interrelated.

address translation

The translation of virtual addresses into real addresses is performed in most virtual storage systems by special hardware in conjunction with tables maintained by the system control program. Consider, for example, a byte-addressable machine with an address field of n bits and with a paged but nonsegmented virtual storage that has pages 2^m bytes in length ($m < n$). In such a machine, the left-most (n-m) bits of a virtual address are typically treated as a *virtual page number* and the right-most m bits as a *displacement* within the indicated virtual page. Then, as illustrated in Figure 1, address translation is achieved by using the virtual page number as an index into a *page table* supplying, for each of the $2^{(n-m)}$ possible virtual page numbers, either a corresponding *real page number* (sometimes called a *page frame number*) or an indication that the page does not presently reside in main storage. If we assume that the page does reside in main storage, the address translation hardware merely replaces the virtual page number of the original virtual address with the real page number from the page table.

Similarly, in a machine with segmented but nonpaged virtual storage, certain high-order bits of a virtual address may be used to index a *segment table* having fields not only for the beginning real address of each segment and an indication as to whether the segment is actually in main storage, but also a length field to be compared against the displacement field of the virtual address. Thus a successful translation involves adding the displacement

Figure 1 Address translation in a paged, nonsegmented virtual storage system



field from the original virtual address to the beginning-of-segment address from the segment table.

Finally, the concepts of paging and segmentation may be combined in some machines. In these machines, address translation can be effected by a two-level look-up scheme.

Each of the translation mechanisms described above would obviously cause substantial performance degradation were the required translation tables kept solely in main storage. Indeed, the two-level look-up procedure for systems combining segmentation and paging would, for each storage access, require two additional accesses for address translation. For this reason, address translation hardware generally includes high-speed registers in which the most recently used portions, if not all portions, of the translation tables are maintained. For example, the *dynamic address translation* (DAT) hardware of the IBM System/360 Model 67 uses eight associative storage registers for the eight most recent translations and a ninth register for the real page number of the translated instruction counter.

storage management

As implied above, the tables used for virtual address translation generally contain fields not only for address information *per se*, but also for control information indicating possibly, among other things, whether or not a given portion of virtual address space is presently available in main storage. In machines combining segmentation and paging, such *availability* or *validity indicators* may be present in the segment table as well as the page tables; so the page tables themselves may at times not reside in main storage.

It is, of course, a function of the translation hardware to interrupt processing whenever a translation table entry is encountered that indicates that a storage block is unavailable. On detecting such an interruption, called a *translation exception*, the system control program must initiate the operations needed to bring into main storage the information that is missing. In general, these operations involve:

1. Determining an area in main storage into which the missing information can be placed (a function of the *replacement algorithm*, discussed further below)
2. Writing, if necessary, into auxiliary storage the current contents of the selected area
3. Reading from auxiliary storage the missing information
4. Updating the translation tables to reflect the changes that have been made

Only when these operations have been completed can the system control program resume execution of the instruction causing the translation exception. Note that these operations generally entail reference to (and possibly modification of) additional tables specifying the location of virtual storage blocks on auxiliary storage. In CP-67 these tables are called *swap tables*.

In some virtual storage systems, the transfer of portions of virtual storage between main and auxiliary storage is performed solely when translation exceptions occur. Paged virtual storage systems implemented in this manner are called *demand paging systems*. It should be noted, however, that virtual storage transfers need not be handled entirely on a demand basis. For example, in time-sharing systems with virtual storage, it may be advantageous to immediately initiate the transfer of a particular user's storage blocks from main to auxiliary storage whenever that user is deactivated. This is done, for example, in the IBM Time-Sharing System/360 (TSS/360) and RCA TSOS systems. These two systems also employ *prepaging*, an anticipatory strategy which, in general, involves transferring virtual storage blocks into main storage before they are actually demanded.

In addition to the essential operations of transferring virtual storage blocks between auxiliary and main storage, virtual storage management may entail other operations aimed at increased function or improved performance. For example, in some systems, the storage management mechanism is generalized so that programs and data can be introduced into a user's virtual address space without conventional file I/O operations. In essence, this is accomplished by using the same format for files as is used for virtual storage blocks. Then, by making swap table entries point to the appropriate file blocks when access is desired, the blocks can be brought into main storage on a demand basis. Highly refined mechanisms of this type have been developed in TSS/360, where they are called *virtual access methods* (VAM), and in the MULTICS system, where files are almost always treated as virtual storage segments. As discussed later, a very limited form of VAM is used in CP-67 to establish an operating system in the address space of a virtual machine.

Another storage management function in multiuser virtual storage systems may be the provision of a mechanism for information sharing. This is accomplished quite simply, in principle, by making the translation tables for different users point to the same physical storage areas. In MULTICS and TSS/360, sharing is effected at the segment level; hence, when users share a given segment, pointers to the page table for that segment are placed at the appropriate entry of each user's segment table. In CP-67, the sharing of read-only portions of an operating system is effected at the page level. Thus each user has his own set of page tables, but individual page tables of different users may have common entries.

program performance

Program performance in virtual storage environments has been the subject of numerous and often conflicting articles (see References 8–46). Here we make no attempt to treat this subject exhaustively, but wish to point out some of the factors influencing program performance other than the obvious factor of program size relative to the amount of available main storage. We also indicate certain techniques that may improve performance.

One factor which has received considerable attention is the *replacement algorithm*, mentioned previously as the mechanism determining the area of main storage in which to place a newly demanded virtual storage block. In systems with predominantly demand-based storage management, the replacement algorithm can greatly affect the rate at which translation exceptions occur and, hence, system performance in general. Intuitively it would appear that, to minimize the rate of translation exceptions, the replacement algorithm should choose the storage area containing information that has the longest expected time before being referred to again. This notion, essentially a statement of the *principle of optimality*, is in fact embodied in many of the replacement algorithms that have been implemented and/or theoretically investigated, for example:

- First-in-first-out (FIFO), wherein the storage block that is replaced is the one first brought into main storage
- Least-recently-used (LRU), wherein the storage block that is replaced is the one referenced longest ago
- Working-set (WS) algorithm, wherein the storage block that is replaced is any block that has not been referenced within some specified period[20]
- Optimal (MIN) algorithm, wherein the storage block that is replaced is the one that will, in fact, be referenced farthest in the future

The MIN algorithm, though not realizable in practice, was used in experimental work by Belady as a basis for evaluating other algorithms.[11] Actual virtual storage systems generally employ various approximate forms of the WS and LRU algorithms that typically generate 10 to 50 percent more translation exceptions than the MIN algorithm.[11,17]

While the control of main storage by the replacement algorithm is important to performance, it is generally logically transparent to the application programmer. Thus, more germain from the programmer's view, are the factors of programming style and program structure. A central goal of programming for a virtual storage environment is that of maintaining locality. *Locality* is difficult to specify precisely, but generally it implies staying within a small set of virtual storage blocks for long periods of time. Techniques for achieving locality may be roughly divided into those having to do with procedures and those having to do with data, and will be so divided here, though the distinction is not always clearcut.

procedures

Since procedures tend to stay in one place in virtual storage while data is passed around from one location to another, locality implies compacting procedures internally and clustering those procedures which are frequently used together. Compacting is done by removing areas of seldom-used code that are in line with areas of often-used code, and making each seldom-used area a separate routine that can be assigned to a virtual storage space near other seldom-used code. Clustering the often-used areas can be done on the basis of frequency of use or on the basis of the number of transfers from one area to another. The information needed to perform the clustering may be gathered at little or great expense, depending on the accuracy desired. Automatic techniques have been developed to perform this clustering, and improvements are usually possible through manual or automatic methods. See Comeau[18] and Hatfield and Gerald.[28]

data

Given that the amount of data examined by a program is determined by the problem to be solved, the programmer has choices left in the manner of structuring and accessing the data he will use. The access pattern and storage pattern should be mated, and when one is fixed, the other should conform to it as much as possible. There is no "best" storage structure (e.g., an array) independent of the distribution of data values and access patterns. The array is a reasonable way to store a matrix if no large fraction of its elements has the same value, but if a matrix is 80 percent zeros, it probably should be packed, and if symmetric, only half need be represented. Where possible, data should be stored in the order in which it is to be used and vice versa.

Hash-coding has the advantage over list processing of localizing drastically the storage traversed for the accessing process (i.e., no intermediate pointers) and usually for the updating process as well. Therefore it is preferred when a small fraction of a total data area needs to be examined in a significant period of real time (the time it would take to fetch all the data area). But if most of the data area, or more specifically most of the virtual address space that makes up the data area, must be handled during a relatively short period of time, the cost of explicitly storing redundant structural information in the hash-code method must be considered, and the storage representation that generates the smallest total storage area chosen.

An aid to increasing data locality is to consider the amount of parallelism available in a process. For instance, the order of processing indices is unimportant when initializing or multiplying a matrix, and that order may be chosen which results in the fewest passes over a large virtual storage area. In general, the more flexible is the order of operations between initialization and result, the more possible it is to increase localization by using compact intermediate data areas for storing partial results, so as to reduce the number and scope of accesses over a data base. Parallelism is specifically important in many large data base applications, e.g., sort-merge, query languages, and matrix manipulation. See, e.g., Brawn and Gustavson[15,16] on sort-merge and Guertin[27] on source language array processing.

What can be done to procedures can be done to data, i.e., data areas used together should be placed near one another in virtual storage. Clustering of data areas can be facilitated if there is a level of indirection (a compact pointer area) between the data name and the lower-level array or tree or hash structures. This permits target data areas to be rearranged periodically on the basis of use without global changes to the procedure and data areas used for accessing. An example of a compact, indirect interface is FORTRAN COMMON, which permits rearranging the storage order of an array list by reordering the names in the COMMON statement.

What can be done to data can be done to procedures whenever there is freedom to reorder dynamically the sequence of use of a set of procedures. Whenever possible, the procedures used last in the previous phase of a program should be used first during the current phase. This is true because nearly all replacement algorithms tend to expect that storage blocks used longest ago will not be needed until furthest in the future. Therefore, looping is the worst possible way to repeatedly traverse a large virtual storage area. For example, consider the problem of multiplying two large matrices to produce a third, all stored columnwise. No matter how the program is written, at least one of the multiplied matrices must be gone over in the wrong direction (across its rows). But if we alternate the direction of the paths across successive rows, which can be done because the sum $\sum_k a_{ik} b_{kj}$ is independent of the order of values given to k, we will produce, instead of a loop over a large storage area, a sawtooth that gives less page exceptions with all but random replacement algorithms, especially as the available real memory size approaches that of the virtual storage area needed for the array.

In general, it is possible to contain procedure and data in local virtual storage areas if (1) code is segregated by frequency of use and communication, (2) the order of processing data corresponds to the order of storage and intermediate data areas containing partial results are used when possible, (3) initiation of data is done immediately before the data is used, and then only a few storage blocks at a time if the data is to be used serially, (4) garbage collection is frequent, and (5) returns from long sequences of large jumps through virtual storage are made by reversing the order of the jumps if the start of the sequence has a greater probability of being used in the immediate future than does the end or the middle. Little is known yet of the value of saving in a temporary array more data than the present pass through a data base requires on the assumption that it will be relevant to the next pass. Also, little information is available concerning under what conditions it is better to do a large overlay rather than shift to a new area of virtual storage, or what statistics are needed for dynamically restructuring a data base. But usually programs do not require such esoteric remedies. What seems most helpful, aside from reordering tools that use detailed examination of program activity, is to consider the program as a problem-solving process as free as possible from (1) preconceived representations of the data involved and (2) preconceived orderings of the detailed sequences of data reduction.

Virtual machines

overview

As stated in the beginning of this paper, in a virtual machine system, the instructions issued by a program to perform a given task may differ from those actually executed by the hardware. Typically in such a system, one computer, the host machine, provides functional simulation of one or more other computers, the virtual machines. Goldberg[93] had distinguished two classes of virtual machine systems: *self-virtualizing*, where the virtual machines are identical to the host, and *family-virtualizing*, where the virtual machines are all members of the same computer family (e.g., IBM System/360 Models 30 through 65) as the host. In either case, the virtual machine system must provide functional simulation of at least four components—system control panel, central processing unit(s), I/O system, and storage—the four basic components of a real system. To the extent that components of the virtual machine (VM) have direct or identical counterparts on the host machine, and to the extent that the architectures of both the host and virtual machines permit it, functional simulation can be effected by utilizing real components or features of the host computer; otherwise, a detailed step-by-step simulation must be performed. The use of components of the host computer to effect the functional simulation of the virtual machine depends primarily on the provisions in both the host's and the virtual computers' architectures to segregate and control those components. For example, if the instruction set of the virtual machine and the host computer are identical, then many (perhaps most) of the instructions to be executed by the virtual machine can be handled directly by the host hardware. This can be the case only if there is a means of preventing the virtual machine from directly changing or interrogating its status, where a status change includes, for example, the initiation of an I/O operation. If a mechanism is available for excluding status-related instructions from the instruction set of a virtual machine, then the virtual machine control program can effect the functional simulation of the virtual machine's central processor without recourse to the detailed and highly expensive simulation of each instruction.

utility

Though the concept of a virtual machine does not necessarily imply that the virtual machine is other than a duplicate of the host or that more than one virtual machine is available, the advantages of virtual machines are enhanced in a multiprogramming environment, permitting different members of a family of similar machines to be used. Listed below are some examples of facilities which are only available in such a system, or available at greater convenience than in a more conventional system:

- Concurrent running of dissimiliar operating systems by different users. While one virtual machine is used to develop and test code for the current release level of an operating system, another virtual machine can be using a back-level release of the same system.
- Both system and application programs may be developed and debugged for machine configurations that are different from that of the host machine. Thus a host machine with a modest amount of main storage can provide the environment for development and test of a system to run on a machine with a large amount of main storage.
- One virtual machine is totally insulated from the effects of software failures occurring in other virtual machines.
- The host machine can aid in the measurement of hardware and software usage by the various virtual machines. Specific virtual machines built for monitoring can communicate directly with the host without impacting the machines being monitored.

In providing functional simulation of a nonexistant computer system, a virtual machine system provides lead time for software development and early checkout of a hardware architecture and its software implications prior to actual hardware construction.

implemented virtual machine systems

One of the earliest virtual machine systems was CP-40[47,61,66] mentioned previously as an example of a system with virtual storage. CP-40 was developed in 1965–66 for an IBM System/360 Model 40 augmented by special dynamic address translation hardware. A prototype for the CP-67 system discussed in detail in the following section, CP-40 allowed concurrent running of as many as 15 virtual System/360's. It should be noted that, unlike CP-67, CP-40 did not support virtual machines that used dynamic address translation themselves, i.e., CP-40 was family-virtualizing but not self-virtualizing. Other implemented virtual machine systems include:

- IBM M44/44X, an experimental system that was neither family- nor self-virtualizing, but provided virtual machines similar to the IBM 7044 from which the M44 was derived [70,83]
- System/360 Model 30 hierarchical control program, a systems evaluation tool supporting a single virtual System/360[94]
- MTS (Michigan Terminal System), an operating system for the IBM System/360 Model 67 that supports multiple virtual System/360's[48,93]
- HITAC-8400 program simulator, a system development and debugging aid supporting a single virtual HITAC-8400[88,93]

CP-67[53,55,68] developed in 1967 (currently an IBM program with Class A maintenance), is perhaps the most widely used virtual machine system to date and is discussed below.

CP-67

CP-67 is a multiuser virtual machine system for the IBM Systen/360 Model 67 that provides functional simulation of the System/360 family of computers, including the Model 67 itself. Further, depending on the programs (and operating systems) running within the virtual System/360's, CP-67 can provide an interactive time-sharing environment. Its responsiveness – a term used loosely here – is determined by many factors, including the system operating in the virtual machine, the dispatching algorithms in CP-67, and the demands of other users. Although responsiveness is of importance to the acceptability of CP-67 in various environments (e.g. high throughput, interactive), it is not central to its definition as a virtual machine system. The emphasis here will be on that definition and its interrelationships to virtual storage. The reader interested in CP-67 performance should see References 74, 76, 77, 79, and 84.

To convey a more concrete understanding of how CP-67 operates, a brief discussion of CP-67's simulation of each of the components of a System/360 is given in the succeeding paragraphs. Following these are discussions of the manner in which CP-67 supports virtual machines with address translation hardware and some of the problems arising from the lack of address translation hardware on the I/O channels. The section concludes with a discussion of some of the exploitations of virtual storage that are within the framework of a virtual machine system.

simulation of System/360

For each virtual machine requested by a user, CP-67 maintains a set of tables containing the description and status of these components. Where appropriate, these tables correlate hardware components of the host Model 67 with components of the virtual System/360. Thus, for example, a keyboard device such as an IBM 2741 communications terminal is correlated with the system control panel and operator's console of each virtual machine.

system control panel

CP-67 associates with each virtual machine a keyboard device (either remote or locally attached) and maps onto this device the major portion of the functions available on the system control panel. Thus the RESET button on the System/360 panel becomes the typed character sequence "RESET", which causes CP-67 to initiate a detailed step-by-step simulation that resets the appropriate status data in the tables describing the virtual System/360. In the same fashion, CP-67 simulates other features of the system control panel.

In addition to the various control-panel functions, CP-67 also maps onto the keyboard device of each virtual machine a virtual printer-keyboard IBM 1052-7. As the 1052-7 is an I/O device to a System/360, its support by CP-67 is covered under the discussion on the I/O system.

CPU

The distinction in System/360 between problem and supervisor state enables CP-67 to execute most of a virtual machine's instructions directly. When the central processing unit (CPU) is in problem state, any attempt to execute an instruction that changes or interrogates the state of the system, i.e., a privileged instruction, causes a program interruption. Thus by executing virtual machine instructions only while in the problem state, CP-67 is ensured of regaining control whenever a privileged instruction is encountered. When such an event occurs, CP-67 simulates the appropriate functional effect of the privileged instruction as follows. From a table describing the virtual CPU, its status is determined—specifically, whether it is in problem or supervisor state. If the virtual machine is in problem state, CP-67 must simulate a program interruption to the virtual machine. This entails storing the virtual machine's CPU status in the virtual machine program old PSW (Program Status Word) location, fetching the virtual machine's program new PSW, and updating appropriately the data in CP-67's table for the virtual CPU. If, on the other hand, the virtual machine is in supervisor state, CP-67 must decode the instruction and perform a simulation of that instruction. For example, on a virtual machine's SSK (Set Storage Key) instruction, CP-67 must determine the key value and block address, and then, if the corresponding page is in main storage, set its key to the value specified. If the page is not in main storage, CP-67 must store the key value in the appropriate swap table entry. In either case, CP-67 must update appropriately the tables (and hardware) to reflect the change in the virtual machine's status before it can resume running the virtual machine.

I/O system

As we have mentioned, CP-67 maintains in tables a description of the I/O structure of each virtual machine. These tables indicate not only the existence of each I/O element but also the status of the element (e.g., busy or free) and the real hardware component to which it corresponds. Thus when a virtual machine issues a SIO (Start I/O) instruction, CP-67 must first determine that the I/O address is valid in the virtual machine's I/O structure and that the elements composing the virtual I/O path (channel, control unit, device) are free. CP-67 must then mark the virtual path busy and build an equivalent I/O task for the real hardware. At its simplest level, a virtual machine's SIO to an IBM 2314 direct access storage device at I/O address 190 could result in CP-67 issuing an SIO to a real 2314 at address 332. The real path may, of course, be busy (as when an I/O task for another virtual machine is utilizing the required channel), and if so, the task must be deferred until the real path becomes free. Then CP-67 can issue an SIO instruction and proceed with the instructions following the virtual machine's SIO. When the I/O task is completed (interruption), CP-67 must reflect this fact in the tables describing the virtual machine's I/O structure; in particular, it must indicate that the previously busy virtual path has become free and that an interruption is pending. Then, when the virtual CPU becomes enabled for the interruption, CP-67 must simulate the effects of the interruption, including the updating of the virtual machine's channel status word.

The procedure just described is followed in cases where direct counterparts exist for the elements of the virtual machine's I/O structure. Where no direct counterparts are available, CP-67 must effect detailed simulation of the data flow through the virtual machine I/O structure. Two examples are:

- *Unit record devices.* Though available on the host machine (and attachable to the virtual machine), unit record devices such as printers and card readers are most efficiently utilized if CP-67 simulates I/O to these devices on a detailed basis, using a disk to buffer the flow of data between many virtual machines and the individual real devices.

- *Operator's console.* CP-67 maps the 1052-7 printer-keyboard onto the same keyboard device used to simulate the system control panel. This entails simulating the data flow between a virtual machine and a 1052-7 using a transmission control unit and communications terminal (e.g., an IBM 2703 and 2741). This simulation is complicated by the fact that not only must a range of different terminals be supported, but a terminal must serve the two dissimilar functions of virtual machine I/O and system control panel simulation.

It is often convenient to add a virtual I/O device for which there is no exact equivalent. An example of such a device is a "minidisk"—a logical subset of a direct access storage device such as a 2314. That is, a minidisk may be in every way a 2314, except that it has fewer than 203 cylinders. By partitioning a 203-cylinder 2314 into several smaller equivalents, operational economy is obtained.

storage

CP-67 employs the dynamic address translation hardware on the Model 67 to establish and maintain a virtual address space for each virtual machine. In the tables that CP-67 uses to describe a virtual CPU, there is a set of segment, page, and swap tables describing an address space of up to 16 million bytes. To start running a virtual machine, CP-67 loads a control register with the address of the segment table associated with the virtual machine. Next the PSW is set to problem state, address-translation or "relocate" mode, and enabled for all interruptions; further, the PSW contains the virtual machine's PSW key and instruction counter. The Model 67 is now "running" the virtual machine's CPU and will do so until an interruption is received. As an example, on a translation exception interruption, CP-67 must determine that the virtual address is in the address span of the virtual machine's storage. If it is outside the span, CP-67 must present an addressing exception interruption to the virtual machine. Otherwise, it must make a page frame available (a function of the page replacement algorithm), find in the swap table the location on auxiliary store of the image of the needed page, bring this page image into main storage, and set its storage keys to the values specified by the swap table. Only when these actions are complete can CP-67 resume running the virtual machine.

Though not central to its definition as a virtual machine system, the page replacement algorithm of CP-67 has a profound effect on system utilization and responsiveness.[76] Using storage reference bits set automatically by the hardware, this algorithm tends to keep in main storage virtual machine pages that have recently been used. If enough pages for a virtual machine are in main storage, CP-67 can execute the virtual machine's instruction sequences for substantial periods of time without incurring page exceptions. Further, if enough pages for each of several virtual machines can reside in main storage, then those machines can be multiprogrammed efficiently. It is an objective of the CP-67 dispatching algorithm to run only virtual machines with a reasonable chance of having the required pages in, or brought into, the available main storage. Thus the dispatching algorithm must be complementary to the page replacement algorithm, which has the function of preserving in storage the required pages of each dispatchable virtual machine.

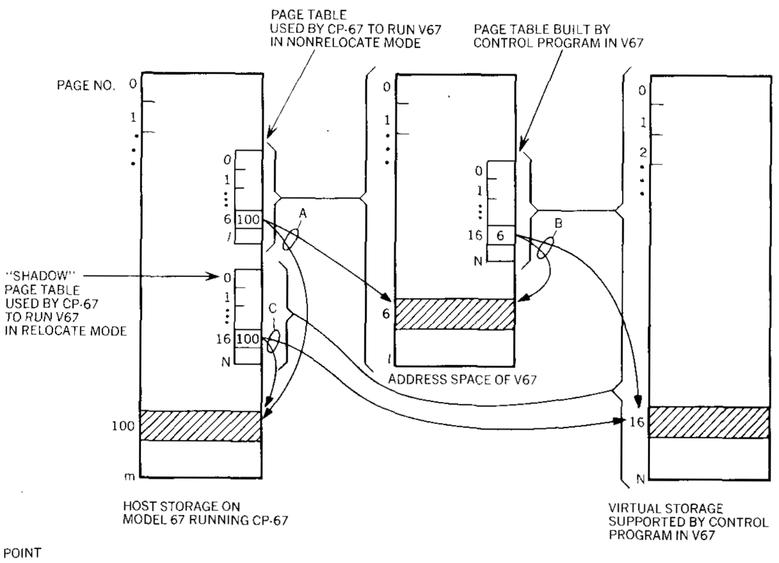
In addition to system utilization being a goal, responsiveness is a goal and is often a crucial aspect of performance. The central philosophy is to ensure that short jobs are not inordinately delayed by long jobs. This gives rise in CP-67 to a dispatching algorithm with both *time-slicing* (i.e., each virtual machine is run a certain length of time, then set aside until others have had a turn) and *multi-queue dispatching* (i.e., on the occurrence of some event, the virtual machine is placed in a high-priority queue and allowed to be dispatched ahead of other virtual machines).

address translation hardware

To support the development of code for the Model 67 itself, there is a need for virtual Model 67's (V67). Except for the simulation of several additional instructions and a new PSW format, the logic for handling the V67's CPU is the same as that for handling any System/360 CPU, and is of little interest here. More interesting is the functional simulation of the DAT hardware. Before discussing V67's on CP-67, let us review briefly the operation of a Model 67. To run in relocate mode, the control program in the Model 67 must among other things load Control Register 0 (the segment table register) with the address of a segment table. It can then load a PSW indicating that address translation is active. Each address translation involves a search of associative storage registers maintained by the DAT hardware, and relies on segment and page tables if this search fails. To support a V67, CP-67 utilizes the host Model 67's DAT hardware to simulate the translation hardware of the V67. The tables of the control program in the V67 indicating how its virtual addresses are to be translated into "real addresses" must be combined with CP-67's tables indicating how the V67's "real addresses" are to be mapped onto the main storage of the host Model 67. In CP-67, the combined tables are called shadow segment and page tables. Shadow tables can be illustrated by the following example.

CP-67's map might indicate that page 6 of a virtual machine's storage is at page 100 of the host Model 67's storage (see Figure 2, point A). If the virtual machine is a V67 in relocate mode, it has a map relating virtual addresses to real addresses which might indicate that "virtual 16 equals real 6." That is, within this

Figure 2 Use of shadow tables to support a virtual Model 67


PAGE TABLE USED BY CP-67 TO RUN V67 IN NONRELOCATE MODE
PAGE TABLE BUILT BY CONTROL PROGRAM IN V67
PAGE NO.
"SHADOW" PAGE TABLE USED BY CP-67 TO RUN V67 IN RELOCATE MODE
ADDRESS SPACE OF V67
HOST STORAGE ON MODEL 67 RUNNING CP-67
VIRTUAL STORAGE SUPPORTED BY CONTROL PROGRAM IN V67
A
B
C
100
16
6

POINT
A: TABLE ENTRY USED BY CP-67 TO MAP PAGE 6 OF V67'S ADDRESS SPACE INTO PAGE 100 OF THE HOST STORAGE
B: TABLE ENTRY WITHIN V67 THAT MAPS VIRTUAL STORAGE PAGE 16 ONTO ITS PAGE 6
C: THE SHADOW TABLE ENTRY THAT MAPS PAGE 16 OF VIRTUAL STORAGE ONTO PAGE 100 OF HOST STORAGE

V67, any virtual address in page 16 is to be converted to one in page 6 of its storage (Figure 2, point B). To effect the functional simulation of the V67's map of virtual addresses to its "real" storage, CP-67 must build a shadow table that maps virtual page 16 to real page 100 (Figure 2, point C). Then, when running a V67 in nonrelocate mode, CP-67 uses the normal page tables, and when the V67 enters relocate mode, CP-67 uses the shadow tables. It is in this fashion that CP-67 maintains a functional simulation of sufficient fidelity such that both CP-67 itself and TSS/360 can be run in a V67.

channel program translation

As mentioned in the discussion on the I/O system, a virtual machine's start I/O instruction to a nonsimulated device results in CP-67 constructing an I/O task equivalent to that demanded by the virtual machine. A major technical factor is that the virtual machine's channel program is defined in the virtual storage of that virtual machine. That is, all addresses in the channel command words (CCW's) that comprise the channel program are virtual addresses. The I/O structure on the Model 67 is such that I/O tasks are not subject to dynamic address translation, i.e., the channels deal with real, not virtual, addresses. In constructing an equivalent channel program, CP-67 obtains a copy of the virtual machine's channel program and builds in its working storage a translated equivalent of the program. This process involves the following operations:

1. For each virtual data address, a real address must be obtained. All pages of virtual storage involved in the I/O operation must be determined, and any pages that are missing must be brought into main storage. Further, all pages involved in the I/O operation must be *locked* in storage until the I/O operation is completed.
2. Channel commands that indicate data areas crossing page boundaries must be translated into multiple data-chained commands. This is necessary because, in general, contiguous virtual pages are not contiguous in real storage.

One consequence of the translation of channel programs is that, because a virtual machine's entire channel program is fetched at the start I/O instruction time, CP-67's support of virtual machines is at variance with standard System/360 channel architecture, wherein channel command words are fetched only when needed. Thus, modifications made to the channel command words in a virtual machine's address space after a start I/O instruction has been issued can have no effect on the I/O operation. In practical use of CP-67, this variance has not been a major problem. Further, no such variance occurs in the case of simulated devices, e.g., card readers. Here CP-67's simulation of the data flow through the virtual machine's I/O structure permits conformity with the channel architecture.

Another consequence of channel program translation is that some devices are not, in general, available to virtual machines under CP-67. For example, the data transfer rate of an IBM 2301 drum unit is too great relative to the storage speed of a Model 67 to permit data chaining (except at a record gap). Thus, these drum units can be used by a virtual machine only if it can be assured that no data areas will span page boundaries. A final consequence of channel program translation is that the page replacement algorithm must be able to recognize and pass over locked pages in main storage.

To summarize, the lack of address translation hardware on I/O channels necessitates a software translation of channel programs. This translation may have considerable impact on system performance and may impose minor I/O programming or hardware restrictions. *Preferred virtual machines*, a technique discussed by Parmelee,[84] permits the elimination of channel program translation, hence, the achievement of substantial performance improvements and the relaxation of restrictions.

virtual storage exploitation

One interesting exploitation of virtual storage by CP-67 is seen in its ability to establish a program (or data) in an address space without actual I/O or paging operations. In particular, the initial program load (IPL) function, which on a System/360 is used to cause a sequence of I/O operations followed by CPU execution of the input data, has been enhanced in CP-67 in the following manner. In addition to supporting the standard System/360 initial program load function, CP-67 permits the permanent assignment on backing store of the page images of an operating system at a point late in the program loading process. On being requested to initially load such a system, e.g., to "IPL OS", CP-67 needs only to establish in the virtual machine's swap tables appropriate entries indicating the permanently assigned area on backing store. Thus the virtual address space for an entire operating system can be established without the expense of I/O simulation. Furthermore, during subsequent use of the operating system, only those portions of the system that are actually used will be paged-in. From the user's point of view, a particular advantage of this feature of CP-67, which is called "named system IPL", is that it is unnecessary to repeat the system initialization dialogue each time a system is run. In effect, the user can create a frozen "checkpoint" of the system that can be quickly and cheaply re-established.

shared virtual storage

A further exploitation of virtual storage is the sharing among several virtual machines of read-only pages of storage. This is effected by CP-67 on initially loading the program of a named system, which initializes a virtual machine's address space. As part of this initialization, CP-67 establishes in the virtual machine's page tables pointers to common or shared page frames of the host machine's main storage.

To conclude this discussion of virtual machines, it must be emphasized that CP-67, in providing virtual System/360's, makes very direct and simple use of the address translation hardware of the host Model 67. The more elaborate forms of virtual storage, e.g., general virtual access methods and segment sharing, being outside the definition of System/360, are not supported. That is, a virtual machine system, and in particular CP-67, provides the basic resources of the computer hardware, but does not otherwise support or effect high-level user functions.

Summary

In this paper, we have discussed some of the salient aspects of virtual storage systems. Further, in the discussions of virtual machines and CP-67, we have illustrated an implementation of a virtual storage system as well as the generalization of the virtual storage concept to virtual machines. Although a virtual machine system does not within its strictly defined limits admit to the more complex uses of virtual storage, it does serve to illustrate the major problems confronted by many virtual storage systems. The utility of the virtual storage and machine concepts is well-established, and in the future, the application and extension of these concepts is certain to increase.

BIBLIOGRAPHY

general discussions and surveys

1. B. W. Arden, B. A. Galler, T. C. O'Brien, and F. H. Westervelt, "Program and addressing structure in a time-sharing environment," *Journal of the ACM* **13,** No. 1, 1–16 (January 1966). Discusses hardware and system software features desirable for time-shared computing facilities, specifically the motivation for segmentation and paging mechanisms. Describes in some detail a scheme for segment sharing. A key paper in the evolution of the Michigan Terminal System (MTS) for the IBM System/360 Model 67.
2. J. B. Dennis, "Segmentation and the design of multi-programmed computer systems," *Journal of the ACM* **12,** No. 4, 589–602 (October 1965). A key paper in the evolution of the MULTICS system for the GE 645. Explains motivation behind segmentation and paging, vis-a-vis multiprogramming and time-sharing.
3. P. J. Denning, "Virtual memory," *Computing Surveys* **2,** No. 3, 153–189 (September 1970). Motivates and defines virtual-memory concepts and discusses the implementation of virtual memory via segmentation and/or paging. Discusses problem of fragmentation in systems without paging, contending that the internal fragmentation associated with paging is usually offset by program compaction and at any rate can be controlled by proper choice of page size. Discusses replacement algorithms, the working-set concept, the effect of program structure on performance in virtual memory, and various hardware mechanisms to improve the performance of virtual memory systems. Contains an extensive bibliography.
4. H. Katzan, "Storage hierarchy systems," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **38,** 325–336 (1971). Reviews storage hierarchy concepts and developments, considering (1) buffer/core systems and LCS, (2) overlay schemes, relocation methods, and virtual memory, and (3) hierarchical data management and data base organization.
5. D. J. Kuck and D. H. Lawrie, *The use and Performance of Memory Hierarchies: A Survey*, Univeristy of Illinois at Champaign-Urbana, Department of Computer Science Report No. 363 (December 4, 1969). Reviews literature discussing (1) effects of memory and page size on page fault rate, (2) effects of page size on memory fragmentation and program superfluity, (3) page replacement algorithms, (4) effects of program organization on paging rates and programming guidelines to reduce paging, (5) paging and CPU utilization in multiprogramming systems, and (6) factors affecting I/O times for paging. Includes a large bibliography.
6. W. C. McGee, "On dynamic program relocation," *IBM Systems Journal* **4,** No. 3, 184–199 (1965). Shows desirability of dynamic program relocation and reviews several methods of achieving same, namely: (1) relocation and limit registers, as in the multiprogramming package for the IBM 7090, (2) paging mechanisms, as in the Ferranti ATLAS machine, (3) segmentation schemes, as in the Burroughs B5000, (4) base registers (with appropriate programming restrictions), as provided by the standard addressing technique of the IBM System/360, and (5) two-level dynamic address translation, as on the System/360 Model 67.
7. B. Randell and C. J. Kuehner, "Dynamic storage allocation systems," *Communications of the ACM* **11,** No. 5, 297–306 (May 1968). Describes hardware techniques for dynamic storage allocation, introducing concepts such as segmentation, paging, replacement strategies, etc. Contains a brief survey of several computer systems with dynamic allocation facilities, viz., ATLAS, M44/44X, B5000, Rice University machine, B8500, MULTICS, and System/360 Model 67.

program behavior and memory management

The following articles are studies of program behavior and memory management strategies.

8. A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement," *Journal of the ACM* **18,** No. 1, 80–93 (January 1971). Con-

tains succinct definitions of virtual memory and paging concepts. Shows that the optimal paging policy is a *demand* policy when the cost C (n) of placing n pages in memory satisfies $C(n) \geq nC(1)$. Defines as *k-optimal* a page replacement algorithm $A(k)$ which minimizes page faults for a program whose reference string is generated by a kth-order Markov process. Shows how to implement $A(0)$, and shows that the LRU and working-set algorithms approximate $A(0)$ when the probability that a given page will be referenced varies slowly with time.

9. A. Batson, S.-M. Ju, and D. C. Wood, "Measurements of segment size," *Second ACM Symposium on Operating System Principles*, Princeton University, 25–29 (October 20–22, 1969). Presents segment size distributions measured on a Burroughs B5500 under normal "production conditions" at a university. As ALGOL was the predominant programming language, and as ALGOL program blocks and data array rows are represented as distinct segments on the B5500, segments tended to be small, with about 60 percent containing fewer than 40 words.
10. M. H. J. Baylis, D. G. Fletcher, and D. J. Howarth, "Paging studies made on the I.C.T. ATLAS computer," *IFIP Proceedings of the 1968 Congress* **2,** 831–837 (1968). Describes performance measurements of several paging algorithms, storage access patterns of 300 jobs, and a simulation study showing the effects of varying page size. The paging algorithm measurements showed that the replacement policy, which is actually used on the ATLAS, and which assumes a strictly cyclic page usage pattern, produces about the same number of page swaps as an LRU policy, and about one-half as many swaps as a purely random policy. The simulation studies showed that, for all programs considered, substantial advantages could be gained by using smaller pages, provided system overheads could be reduced. Page size on the ATLAS is 512, 48-bit words.
11. L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal* **5,** No. 2, 78–101 (1966). Develops optimal replacement algorithm MIN as a basis for evaluation of random, FIFO, ATLAS, and several variants/approximations to LRU. Concludes from simulation experiments based on two problem programs that LRU-type algorithms have best overall performance. Nonoptimal algorithms in general caused two to three times as many page faults as the MIN algorithm.
12. L. A. Belady and C. J. Kuehner, "Dynamic space-sharing in computer systems," *Communications of the ACM* **12,** No. 5, 282–288 (May 1969). Defines a storage cost function, program lifetime function, storage value function, and value per unit cost function to show formally how program behavior, processor efficiency, space allocation, and hardware factors are interrelated in multiprogramming systems.
13. L. A. Belady, R. A. Nelson, and G. S. Shedler, "An anomaly in space-time characterstics of certain programs running in a paging machine," *Communications of the ACM* **12,** No. 6, 349–353 (June 1969). Shows how increasing memory availability may in certain cases *increase* the number of page exceptions a program incurs. Considers FIFO replacement primarily. Cf. Mattson, et al. (1970).
14. D. P. Bovet and G. Estrin, "A dynamic memory allocation algorithm," *IEEE Transactions on Computers* **C-19,** No. 5, 403–411 (May 1970). Discusses memory management in multiprocessing environments, explicitly excluding "time-shared systems in which the effects of time quanta associated with multiplexing-independent processes completely swamp out the effects of interaction factors within a given program." Proposes a replacement policy for a system with variable-length segments based on a graph model of program behavior in which branching probabilities and cycle factors are known.
15. B. S. Brawn and F. G. Gustavson, "Program behavior in a paging environment," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **33,** 1019–1032 (1968). Presents run times under paging of "casually written" and corresponding virtual memory-oriented programs for three dissimi-

lar "real" problems. The three problems involved (1) inverting a 100 × 100 matrix, (2) performing a large data-correlation calculation, and (3) sorting a large data set. Results indicated "that, if reasonable programming techniques are employed, the automatic paging facility compares reasonably well (even favorably in some instances) with programmer-controlled methods." Further, that "the basically external consideration of programming style can be considerably more important than the internal systems design consideration of replacement algorithms."

16. B. S. Brawn, F. G. Gustavson, and E. S. Mankin, "Sorting in a paging environment," *Communications of the ACM* **13,** No. 8, 483–494 (August 1970). Discusses the performance of various sorting programs in virtual memory, again showing the importance of intelligent program/data organization [see Brawn and Gustavson (1968)]. Presents specific experimental results in which 100,000-word data sets were arranged in virtual memory and sorted in several ways with varying real core sizes. When data and programs were "properly" organized, performance was "comparable to that achieved by conventional methods . . . even when run in a very limited core space environment." Rules of thumb for proper program/data organization are given.
17. E. G. Coffman and L. C. Varian, "Further experimental data on the behavior of programs in a paging environment," *Communications of the ACM* **11,** No. 7, 471–474 (July 1968). Presents trace/simulation results showing behavior under paging of a SNOBOL compiler, a program for computing Fourier transforms, a WATFOR compiler, and a differential equation solver. Effects on paging rates and page residence times of page size, core size, and paging algorithm are indicated. Conclusions: "(1) with the possible exception of carefully designed programs, page turning . . . appears excessive in light of current or proposed paging system designs; (2) a least-recently-used page replacement algorithm yields a performance within about 30 to 40 percent of that of the optimum page replacement sequence" (i.e., the Belady MIN algorithm); "and (3) for page residence confined primarily to small areas within the page size, performance is improved substantially more by increasing the number of pages held in core than by increasing the page size."
18. L. W. Comeau, "A study of the effect of user program optimization in a paging system," *ACM Symposium on Operating System Principles*, Gatlinburg, Tennessee, (October 1–4, 1967). Discusses the effects of deck-ordering of routines in the CMS nucleus for CP-40 on the paging performance of an assembler and a FORTRAN compiler. In experiments with the assembler, page transfers numbered approximately 6500, 4200, 2400, and 1200 when alphabetic, random, programmer-devised (intuition), and programmer-revised (trace-assisted) orderings, respectively, were employed. It was concluded that "user optimization is not only easily achieved, but absolutely necessary for frequent operation in a paging environment."
19. F. J. Corbato, *A Paging Experiment with the MULTICS System*, Massachusetts Institute of Technology Project MAC Memorandum MAC-M-384 (July 8, 1968). Describes the paging algorithm used in MULTICS and presents results of experiments showing the effects of varying a parameter which, at one extreme, yields a FIFO algorithm and, at the other, LRU. Concludes that efficient performance is obtained with a parameter setting corresponding to a particularly simple case of the algorithm, viz., when a single "used bit" determines whether a page is to be replaced.
20. P. J. Denning, "The working set model for program behavior," *Communications of the ACM* **11,** No. 5, 323–333 (May 1968). Attempts to make the concept of "memory demand" more precise by defining the working set $W(t,T)$ of a program at time t as the set of pages referenced during the time interval $(t\text{-}T,t)$. Discusses the problem of determining/estimating a program's working set and the use of the working set as a basis for a paging policy.
21. P. J. Denning, "Thrashing: Its causes and prevention," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **33,** 915–922 (1968). Shows how inefficiencies in multiprogramming systems due to thrashing, i.e., ex-

cessive paging, increase with increasing auxiliary storage access time and with the probability that a program will reference a missing page. Asserts that the latter probability cannot be adequately controlled when paging algorithms are applied globally to all programs, but can be controlled by using a working set policy in which a program is allowed to be active only if there is enough uncommitted momory space to contain its working set. Also recommends the use of three-level memory systems, with bulk core as an intermediate level between main memory and rotating storage.

22. P. J. Denning, "On the management of multilevel memories," *Proceedings of the 3rd Princeton Conference on Information Science and Systems,* Princeton University, 162–165 (1969). Describes a method for estimating page reference densities (i.e., frequencies) and for using such estimates to determine how to distribute pages in an addressable multilevel memory.
23. R. R. Fenichel and J. C. Yochelson, "A LISP garbage-collector for virtual-memory computer systems," *Communications of the ACM* **12,** No. 11, 611–612 (November 1969). Describes an algorithm for compacting list structures in a virtual memory computer system, noting that compaction is necessary, even though virtual memory may be essentially infinite, in order to avoid performance degradation when lists become spread over a large region.
24. G. H. Fine, C. W. Jackson, and P. V. McIsaac, "Dynamic program behavior under paging," *Proceedings of the 21st National Conference of the ACM* **P-66,** 223–228 (1966). Reports on paging behavior of several large programs (e.g., a LISP system) as determined by simulation experiments based on an AN/FSQ-32 computer with 48-bit words and 1024-word pages. One set of experiments showed an average page residency of 109.4 instructions. Concludes that the programs tested "will require considerable reorganization to operate efficiently in a demand-paging environment."
25. I. F. Freibergs, "The dynamic behavior of programs," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **33,** 1163–1167 (1968). Presents results of program-trace experiments run at McGill University on an IBM 7044 computer. Shows (1) various data-reflecting memory utilization assuming 1024-word page organization, (2) percentages of instructions by class (e.g., branch, register-only, etc.) for a FORTRAN compilation, GPSS simulation, and several other jobs, and (3) frequency of SVC's for various jobs.
26. M. N. Greenfield, "FACT segmentation," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **21,** 307–315 (1962). Describes the software mechanism used on the Honeywell 800 to dynamically relocate program segments created by the FACT (commercial) compiler.
27. R. L. Guertin, "Programming in a paging environment," *Datamation* **18,** No. 2, 48–55 (February 1972). Discusses techniques for taking advantage of the parallelism in many program operations so as to increase the locality of executed code. The areas of array manipulation and expression evaluation are considered. The effects of ordering the sequence of operations on multiply dimensioned arrays and ordering the sequence of elements in an expression are examined in detail.
28. D. J. Hatfield and J. Gerald, "Program restructuring for virtual memory," *IBM Systems Journal* **10,** No. 3, 168–192 (1971). Describes a method for increasing program localization in virtual memory by rearranging relocatable sectors on the basis of traces of actual program operation. Reports paging reductions of from 2:1 to 6:1 over both alphabetic and programmer-devised ordering of the modules of an assembler and of an AED-0 compiler for the IBM System/360.
29. D. J. Hatfield, "Experiments on page size, program access patterns, and virtual memory performance," *IBM Journal of Research and Development* **16,** No. 1, 58–66 (1972). Discusses the problem of page size selection, presenting experimental results showing that, contrary to common belief, smaller pages do not necessarily mean fewer paging I/O operations. Experiments were based on full instruction traces of real System/360 programs. From

these traces, page request sequences for page sizes of 1024 to 16,384 bytes were obtained. These sequences were processed by programs simulating several single-user page replacement algorithms. It was observed that, for a given program and real memory size and for replacement algorithms in use today, halving the page size often resulted in more than twice as many page exceptions, hence, actually more paging I/O operations. Furthermore, it was shown that address sequences are possible that can cause three or four times as many exceptions when the page size is halved.

30. H. Hellerman, "Complementary replacement—A meta scheduling principle," *Second ACM Symposium on Operating System Principles*, Princeton University, 43–46 (October 20–22, 1969). Notes that computer resource scheduling typically involves applications of two types of rules: admission and replacement. Introduces a notation for expressing admission rules and defines a procedure—the complementary replacement meta principle—for deriving a replacement rule from a given admission rule. Asserts that a variety of known schedulers are encompassed by this formalism, giving MIN and LRU replacement algorithms as examples.

31. R. M. Jones, "Factors affecting the efficiency of a virtual memory," *IEEE Transactions on Computers* **C-18**, No. 11, 1004–1008 (November 1969). Describes paging algorithms for the U. S. Seventh Army tested during development of the Tactical Operating System for the CDC 3300, a machine which offers relocation hardware permitting partial page allocation in units of a quarter page. It was found that an algorithm that proceeded in round-robin fashion through *virtual memory*, selecting the first available page for replacement, gave better performance than one which cycled through *physical memory*—a result which presumably reflects some bias in the way pages were allocated. A least-frequently-used (LFU) policy gave better performance than either of the round-robin algorithms, while the best performance was obtained by replacing pages belonging to programs lowest in the scheduling queue.

32. M. Joseph, "An analysis of paging and program behaviour," *Computer Journal* **13**, No. 1, 48–54 (February 1970). Discusses the dependence of paging behavior on page size, available memory, and paging algorithm. Presents simulation results, based on program traces showing: (1) the percentage of accesses to the *n*th last-used page, as a function of *n;* (2) page exception rates versus number of available pages for various page sizes; (3) exception rates versus page size for various memory sizes; (4) amount of storage referenced versus time for two page sizes; (5) space-time integrals versus page size for different programs and paging algorithms; and (6) program-halt counts versus page size for different paging algorithms, including, in particular, algorithms in which adjacent pages were prepaged.

33. B. W. Kernighan, "Optimal segmentation points for programs," *Second ACM Symposium on Operating System Principles*, Princeton University, 47–52 (October 20–22, 1969). Describes a method of partitioning a program so as to minimize the number of page transitions. Envisions programs as directed graphs whose nodes are indivisible groups of instructions, data areas, etc. "The nodes are assumed to have a given ordering which may not be changed . . . nodes on any page must be contiguous, so the only degree of freedom is in selecting 'break points' between the pages." Node sizes and transition probabilities are assumed to be given. Cf. Hatfield and Gerald (1971).

34. W. F. King, *Analysis of Paging Algorithms*, IBM Thomas J. Watson Research Center Report RC-3288, Yorktown Heights, New York (March 17, 1971). Models a program reference string as a sequence of independent, identically distributed random variables and obtains expressions for the expected page fault rate F for the LRU and FIFO algorithms and for the A(0) algorithm of Aho, et al. (1971). Finds that $F(\text{FIFO}) \geq F(\text{LRU}) \geq F(\text{A}(0))$ for several distributions.

35. C. J. Kuehner and B. Randell, "Demand paging in perspective," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **33**, 1011–1018

(1968). Discusses causes of poor performance of paging systems and possible remedies, viz., module repacking, recoding, and prepaging.

36. P. A. W. Lewis and P. C. Yue, "Statistical analysis of program reference patterns in a paging environment," *IEEE Computer Society Conference*, Boston, Massachusetts, 133-134 (September 22-24, 1971). Presents statistical data based on distance-string representations of reference patterns of three dissimilar programs. Shows how loop structures within these programs are revealed by cumulative periodograms derived from finite Fourier transforms of their distance strings.
37. A. C. McKellar and E. G. Coffman, Jr., "Organizing matrices and matrix operations for paged memory systems," *Communications of the ACM* **12,** No. 3, 153-165 (March 1969). "Matrix representations and operations are examined for the purpose of minimizing the page faulting occurring in a paged memory system . . . Examination of addition, multiplication, and inversion algorithms shows that a partitioned matrix representation (i.e., one submatrix or partition per page) in most cases induced fewer page faults than a row-by-row representation." (Authors' abstract)
38. J. R. Martinson, *Utilization of Virtual Memory in Time Sharing System/360*, IBM Corporation Systems Development Division, Yorktown Heights, New York, Technical Report TR 53.0001 (October 28, 1968). "This report explores the TSS/360 definition of virtual memory and the program structure imposed upon it. Guidelines are presented to describe how programs should and should not be constructed for effective utilization of virtual memory." (Author's abstract)
39. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal* **9,** No. 2, 78-117 (1970). Defines as *stack algorithms* replacement policies which, for a given reference string, produce a memory state with an m-page memory which is a subset of the state produced with an $(m + 1)$-page memory. Shows that LRU, WS, and a variant of the MIN algorithm are stack algorithms; also any other algorithm that replaces pages on the basis of a priority list which is independent of memory size. Notes that FIFO is not a stack algorithm. Defines the *success function* $F(m)$ of a given algorithm for a given memory size m as the fraction of references in a given reference string that do not cause exceptions, and notes that for any stack algorithm $F(m + 1) \geq F(m)$ for any reference string. Shows how $F(m)$ can be evaluated for stack algorithms. Obtains results for hierarchies with *congruence mapping*, where each page is restricted to occupy a member of a subset of the available page frames, and for hierarchies of more than two levels.
40. B. Randell, "A note on storage fragmentation and program segmentation," *Communications of the ACM* **12,** No. 7, 365-372 (June 1969). Presents results of simulation experiments showing the dependence of memory utilization on page size and indicating, in particular, that as page size increases "the loss of utilization due to increased internal fragmentation distinctly outweighs the gain due to decreased external fragmentation." To reduce internal fragmentation, recommends *partitioned segmentation*, wherein segments are subdivided into pages which are subdivided further into smaller storage units to permit the allocation of partial pages.
41. D. Sayre, "Is automatic 'folding' of programs efficient enough to displace manual?" *Communications of the ACM* **12,** No. 12, 656-660 (December 1969). Compares performance of automatically and manually folded (i.e., overlaid) programs, citing experimental results of Brawn, et al. (1968). Answers title question affirmatively, asserting that automatic folding can reduce programming costs by 25 to 45 percent, with a penalty in average program performance of no more than 25 percent.
42. J. E. Shemer and S. C. Gupta, "On the design of Bayesian storage allocation algorithms for paging and segmentation," *IEEE Transactions on Computers* **C-18,** No. 7, 644-651 (July 1969). Recommends basing storage allocation decisions on usage history and demands, giving as an example a page replacement strategy that is a somewhat elaborated version of the one used in

MULTICS (Cf. Corbato, 1968). Suggests that, in addition to "referenced," "altered," and other bits, systems should keep "pattern bits" distinguishing between random and sequential reference patterns.

43. S. S. Sisson and M. J. Flynn, "Addressing patterns and memory-handling algorithms," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **33,** 957–967 (1968). Presents results of program address trace analyses showing performance of systems with look-ahead, high-speed buffering, and memory interleaving.
44. J. M. Thorington and J. D. Irwin, "A new philosophy in dynamic memory allocation," *Proceedings of the 4th Hawaii Conference on Systems Science*, University of Hawaii, 341–343 (January 12–14, 1971). Examines the performance of three "dynamically adaptive" replacement algorithms using simulator-generated pseudo reference strings. Reports average performance improvements with one algorithm of 398%, 140%, and 2528% over FIFO, LRU, and LFU, respectively.
45. E. W. Ver Hoef, "Automatic program segmentation based on Boolean connectivity," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **38,** 491–496 (1971). Describes a method of partitioning programs into pages so as to reduce the number of interpage references. Using only connectivity information, the method is oriented primarily toward *a priori* partitioning in conjunction with program compilation. Cf. Hatfield and Gerald (1971).
46. J. W. Weil, "A heuristic for page turning in a multiprogrammed computer," *Communications of the ACM* **5,** No. 9, 480–481 (September 1962). Suggests a page replacement policy in which each page has a figure of merit that is incremented whenever the page is brought into memory and decreased in an exponential manner as other pages are brought into memory. The page with the lowest figure of merit is chosen for replacement.

virtual storage systems

The following references are discussions of particular systems with virtual storage.

47. R. J. Adair, R. U. Bayles, L. W. Comeau, and R. J. Creasy, *A Virtual Machine System for the 360/40*, IBM Corporation, Cambridge Scientific Center, Report No. 320-2007 (May 1966). One of the first papers describing the implementation of the virtual machine concept. An IBM System/360 Model 40 was modified with an associative memory to provide dynamic translation of addresses, and a control program was developed to allocate resources of the host machine to the virtual machines. Multiprogramming and multiprocessing tasks could then be studied in terms of machine utilization.
48. M. T. Alexander, *Time-Sharing Supervisor Programs*, In notes for University of Michigan Engineering Summer Conference, "Advanced Topics in Systems Programming" (June 21–July 21, 1971). Describes the structure of the supervisor programs of four time-sharing systems employing relocation: UMMPS, CP-67, TSS/360, and MULTICS. Concentrates on the functions of memory allocation, processor scheduling, and I/O processing. Discusses the degrees to which segmentation and sharing are supported by the systems considered.
49. A. Auroux and C. Hans, "Le Concept de Machines Virtuelles," *Revue Francaise d'Informatique et de Recherche Operationelle* **15,** No. B3, 45–51 (December 1968). Discusses the virtual machine concept and CP-67/CMS applications at the University of Grenoble, France.
50. J. N. Bairstow, "Many from one: The 'virtual machine' arrives," *Computer Decisions* **2,** No. 1, 29–31 (January 1970). An easy-to-read synopsis of the development of CP-67 and the concept of virtual machines, as represented by that system.
51. A. Bensoussan, C. T. Clingen, and R. C. Daley, "The MULTICS virtual memory," *Second ACM Symposium on Operating System Principles*, Princeton University, 30–42 (October 20–22, 1969). Describes the MULTICS virtual memory from a more practical, implementation-oriented point

of view than that of the earlier papers by Dennis, Corbato, and Vyssotsky, et al. in 1965, and by Daley and Dennis in 1968. Presents arguments for segmentation convincingly and explains the MULTICS segment linkage, protection, and attribute mechanisms in some detail.

52. F. J. Corbato and V. A. Vyssotsky, "Introduction and overview of the MULTICS system," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **27,** Part I, 185–196 (1965) One of several key papers on MULTICS presented at the 1965 FJCC.
53. *CP-67/CMS*, Program 360D-05.2.005, International Business Machines Corporation, Program Information Department, Hawthorne, New York (June 1969).
54. R. C. Daley and J. B. Dennis, "Virtual memory, processes, and sharing in MULTICS," *Communications of the ACM* **11,** No. 5, 306–312 (May 1968). Discusses concepts of paging and segmentation as implemented and used in MULTICS. Mechanisms for intersegment linking and addressing are explained in detail.
55. M. S. Field, *Multi-Access Systems—The Virtual Machine Approach*, IBM Corporation, Cambridge Scientific Center, Report No. 320–2033 (September 1968). A first paper on the implementation of CP-67 on the IBM System/360 Model 67. Emphasis is on the concept, applications, and the software implementation.
56. *Emulating DOS under OS for IBM System/360*, Systems Reference Library, GC26-3777, IBM Corporation, Data Processing Division, White Plains, New York.
57. R. E. Fikes, H. C. Lauer, and A. L. Vareha, Jr., "Steps toward a general-purpose time-sharing system using large capacity core storage and TSS/360," *Proceedings of the 23rd National Conference of the ACM* **P-68,** 7–18 (1968). Describes modifications to TSS/360 made at Carnegie-Mellon University to support LCS. Discusses problem of deciding when to execute pages in LCS versus moving them to main memory. See also Vareha, et al. (1969).
58. J. Fotheringham, "Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store," *Communications of the ACM* **4,** No. 10, 435–436 (October 1961). A key paper describing the address interpretation mechanism of the ATLAS.
59. E. L. Glaser, J. F. Couleur, and G. A. Oliver, "System design of a computer for time-sharing applications," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **27,** Part I, 197–202 (1965). One of several key papers on MULTICS presented at the 1965 FJCC.
60. S. E. Gluck, "Impact of scratchpads in design: Multifunctional scratchpad memories in the Burroughs B8500," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **27,** Part I, 661–666 (1965). Describes B8500 architecture, emphasizing the thin-film scratchpad memories used for data buffering, storage of temporary CPU results, counter storage, instruction look-ahead, addressing operations, etc. Also described is the 28-word associative memory used to speed address translation.
61. G. E. Hoernes and L. Hellerman. "An experimental 360/40 for time-sharing," *Datamation* **1,** No. 4, 39–42 (April 1968). A complementary paper to Adair, et al. (1966), describing associative-memory modifications to the IBM System/360 Model 40.
62. J. G. Jodeit, "Storage organization in programming systems," *Communications of the ACM* **11,** No. 11, 741–746 (November 1968). Describes the segmented storage allocation system used on the Rice University computer, noting possible extensions to the areas of multiprogramming and multilevel storage control.
63. O. W. Johnson and J. R. Martinson, *Virtual Memory in Time-Sharing System/360*, TSS/360 Compendium, IBM Corporation, Data Processing Division, White Plains, New York (1969). Discusses virtual memory advantages and implementation in general, then outlines features of virtual memory in TSS/360, e.g., segment sharing, variable-length segments, virtual access

method, dynamic loading, and protection. See also Lett and Konigsford (1968).

64. T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One-level storage system," *IRE Transactions* **EC-11,** No. 2, 223–235 (April 1962). A key paper describing the overall architecture of the Ferranti ATLAS and, in particular, its dynamic memory allocation mechanism.

65. A. S. Lett and W. L. Konigsford, "TSS/360: A time-shared operating system," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **33,** Part I, 15–28 (1968). Describes TSS/360 control system organization, user services, and task structure, noting several design changes that have been made to improve performance. Includes discussions of the TSS dynamic-loading and segment-sharing mechanisms, and of its page-oriented data management facilities. See also Johnson and Martinson (1969).

66. A. B. Lindquist, R. R. Seeber, and L. W. Comeau, "A time-sharing system using an associative memory," *Proceedings of the IEEE* **54,** No. 12, 1774–1779 (December 1966). Describes the IBM System/360 Model 40 virtual machine system discussed also by Adair, et al. (1966), and by Hoernes and Hellerman (1968). Emphasis is on the associative memory feature used for dynamic address translation and for page replacement operations.

67. F. B. MacKenzie, "Automated secondary storage management," *Datamation* **11,** No. 11, 24–28 (November 1965). Describes the dynamic memory allocation mechanism of the Burroughs B5500, reporting running times of several jobs (compilations, matrix operations) with varying core sizes and both single- and dual-processor configurations.

68. R. A. Meyer and L. H. Seawright, "A virtual machine time-sharing system," *IBM Systems Journal* **9,** No. 3, 199–218 (1970). Describes the CP-67/CMS system, outlining features and applications.

69. G. Oppenheimer and N. Weizer, "Resource management for a medium scale time-sharing system," *Communications of the ACM* **11,** No. 5, 313–322 (May 1968). Describes the RCA Spectra 70/46 Time-Sharing Operating System (TSOS), particularly the task scheduling and paging algorithms. Notes simulation results leading to final design decisions.

70. D. Sayre, *On Virtual Systems*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York (April 15, 1966). An early paper emphasizing the multiprogramming aspects of virtual machines. Results are shown for multiprogramming with the IBM M44/44X system.

71. D. Sayre, "Adding computers virtually," *Computing Report* **III,** No. 2, 12–15 (March 1967). A more colloquial presentation of the material in Sayre (1966).

72. A. L. Vareha, R. M. Rutledge, and M. M. Gold, "Strategies for structuring two-level memories in a paging environment," *Second ACM Symposium on Operating System Principles*, Princeton University, 54–59 (October 20–22, 1969). Discusses memory-management strategies used on the Carnegie-Mellon TSS/360 system with LCS [see Fikes, et al. (1968)]. A central problem with this configuration has been deciding what programs should execute from high-speed memory (HSM), and what ones should execute from LCS. Initially, only shared system pages were allowed to run in HSM (and these were swapped to LCS when more HSM space was needed), while private user pages were run in LCS and swapped to disk. A more recent policy calls for "executing the entire system from LCS and using the high-speed memory for only the paging tables, the system work areas, and the resident supervisor." Ultimately a mechanism is desired for dynamically determining the appropriate memory level for a page, according to its actual activity. It is noted that implementing such a mechanism may be impractical without additional data-gathering hardware.

73. N. Weizer and G. Oppenheimer, "Virtual memory management in a paging environment," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **34,** 249–256 (1969). Discusses hardware and software aspects of virtual memory management in TSOS, giving justifications for important design decisions, e.g., the partitioning of virtual memory to concurrently

accommodate the system and a single user. Describes the management of shared code and the allocation of backing store. See also Oppenheimer and Weizer (1968) and DeMeis and Weizer (1969).

In this group are system performance studies.

performance

74. R. Adair and Y. Bard, *CP-67 Measurement Method*, IBM Corporation, Cambridge Scientific Center, Report No. G320-2072 (May 1971). Describes a software measurement method employed on a virtual machine running under CP-67 to measure activities of the host machine. The method involves having CP-67 maintain various time and event counters which a virtual machine of priviledged class can sample and record at appropriate intervals. It is thus possible to determine, e.g., the identity of the currently running user, the number of logged-on users, the percentage of CPU time spent in various states, the rate of virtual and real start I/O's, and the rate of page reads, writes, and steals.
75. W. Anacker and C. P. Wang, "Performance Evaluation of Computing Systems with Memory Hierarchies." *IEEE Transactions on Computers* **EC-16,** No. 6, 764–773 (December 1967). Shows how upper and lower bounds on the performance of systems with memory hierarchies can be obtained, for specific computational loads, from program address traces and hardware data.
76. Y. Bard, "Performance criteria and measurement for a time-sharing system," *IBM Systems Journal* **10,** No. 3, 193–216 (1971). Describes software measurements taken on CP-67 at the IBM Cambridge Scientific Center with two different versions of CP-67 and two different real storage configurations. CP-67 overhead, i.e., CPU time spent by CP-67 in servicing users' requests for system resources, was analyzed by a linear regression model which relates overhead to various types of requests. Significantly improved performance resulted when additional real storage was provided. This marked effect was also noted in examining average throughput, approximately defined as the amount of work performed per unit time. Still another measure of performance, saturation, obtained from Pareto-maximal points representing maximal throughput, again confirmed this improvement, and also the improvements made in software. Evaluation of a particular software change in the module that manages free storage areas indicated that most of the software improvement was attributable to this change.
77. Y. Bard, B. H. Margolin, T. I. Peterson, and M. Schatzoff, *CP-67 Measurement and Analysis, I: Regression Studies*, IBM Corporation, Cambridge Scientific Center, Report No. G320-2061 (June 1970). An earlier report of results subsequently incorporated into Bard (1971).
78. J. Buzen, "Optimizing the degree of multiprogramming in demand paging systems," *IEEE Computer Society Conference*, Boston, Massachusetts, 141–142 (September 22–24, 1971). Discusses the calculation and optimization of throughput estimates using a central server model of a multiprogramming system with paging.
79. P. Callaway, *Performance Considerations for the Use of the Virtual Machine Capability*, IBM Corporation, Thomas J. Watson Research Center, Yorktown Heights, New York, Report RC-3360 (May 12, 1971). Discusses software measurements taken on CP-67 by activating and deactivating hooks for taking such measurements. Principal output was: elapsed time and CPU wait time; total virtual CPU time and virtual memory time; distribution of various counts, including working set size; and counts of page exceptions, privileged instructions, and other interrupts. The great difference on resources imposed by OS and CMS is graphically shown: e.g., one OS four-task virtual machine having multiprogramming with a variable number of tasks is equivalent to 54 editing CMS virtual machines when measured in terms of core-time product per minute of elapsed time.
80. W. M. DeMeis and N. Weizer, "Measurement and analysis of a demand paging time-sharing system," *Proceedings of the 24th National Conference*

of the ACM **P-69**, 201–216 (1969). Describes performance measurements of TSOS (RCA Spectra 70/46) based on controlled load tests in which a master program generated and ran pseudo-programs and simulated user interactions. Shows response time as a function of number of tasks, CPU utilization, and drum utilization, and shows how thrashing is avoided by a scheduling algorithm utilizing the working-set principle.

81. D. P. Gaver and G. S. Shedler, "Approximate models for multiprogramming computer systems," *IEEE Computer Society Conference*, Boston, Massachusetts, 135–136 (September 22–24, 1971). Describes a multiprogramming system model based on a continuous state approximation, namely, one-dimensional diffusion (Fokker-Plank equation) with two reflecting barriers. A simple explicit formula for CPU utilization is derived which compares favorably with more involved semi-Markov results.
82. S. J. Morganstein, S. Winograd, and R. Herman, "SIM/61: A simulation measurement tool for a time-shared, demand paging operating system," *ACM SIGOPS Workshop on System Performance Evaluation*, Harvard University, 142–172 (April 5–7, 1971). Describes a program for simulating the steady-state performance of the RCA Spectra/70 Virtual Machine Operating System. Using statistical models of task paging behavior, compute times, etc., the program predicts response times, overhead percentages, page rates, etc. for alternate system configurations and decision algorithms. Results were found to be highly sensitive to changes in load-characterizing parameters; hence "the authors are extremely skeptical of quantifying results of predictive runs (where precise information about the load characteristics is impossible to obtain) without qualifying them with a nominal 15 percent margin for error."
83. R. W. O'Neill, "Experience using a time-shared multiprogramming system with dynamic address relocation hardware," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **30,** 611–621 (1967). Describes the experimental IBM M44/44X system, reporting performance measurements taken to establish dependence of core requirements on page size and running time on core size (parachor curve). Also discussed are effects of multiprogramming and time-sharing on performance.
84. R. P. Parmelee, *Preferred Virtual Machines for CP-67*, IBM Corporation, Cambridge Scientific Center, Report No. G320-2068 (to appear). Discusses experimental studies in which CP-67 was modified to give preferential treatment to a virtual machine running OS having multiprogramming with a variable number of tasks. Specifically, all pages of the OS machine were locked in memory, and all pages except page zero were assigned real addresses identical to their virtual addresses. Thus paging and channel program translation were eliminated, and dynamic channel program modification became possible. As a result, execution stretchout was reduced by 65 percent and overhead by 63 percent in benchmark tests in which a single OS machine processed a commercial job stream.
85. G. S. Shedler and S. C. Yang, "Simulation of a model of paging system performance," *IBM Systems Journal* **10,** No. 2, 113–128 (1971). Describes the simulation of a probabilistic model of a multiprogrammed single-processor system with a fixed number of tasks operating under demand paging. Assumes that the running times of tasks between page exceptions and other CPU service times are exponentially distributed, whereas paging service times are constant. Presents means and variances of CPU and paging system utilization, and of various overhead service times, obtained by straightforward sampling, by the method of antithetic variables, by the method of stratification, and by a method combining antithetic variables and stratification. Observes that variances can generally be reduced by using the method of antithetic variables, although not necessarily in all response variables. Further variance reductions may be obtained by combining antithetics with stratification.
86. J. L. Smith, "Multiprogramming under a page on demand strategy," *Communications of the ACM* **10,** No. 10, 636–646 (October 1967). Analyzes a

probabilistic model of a multiprogramming system with demand paging, obtaining performance estimates for user programs typical of those arising in an interactive time-sharing environment. Concludes that "a conservative outlook . . . must be maintained" for such a system, although with sufficient high-speed memory "it does seem that there is some advantage (dependent on system overhead) to be gained from multiprogramming."

87. V. L. Wallace and D. L. Mason, "Degree of multiprogramming in page-on-demand systems," *Communications of the ACM* **12,** No. 6, 305–308 (June 1969). Describes a simple Markov model of a multiprogrammed time-sharing system using page-demand statistics that imply a burst of page demands at the beginning of each job. Shows CPU utilization as a function of the degree of multiprogramming, the average number of page demands per job, the average execution time per job, and the average page fetch time during burst paging. Shows how the optimum degree of multiprogramming is determined, given the relationship between the average number of page demands per job and the degree of multiprogramming. A linear relationship is treated.

related material

Here included are other articles related to virtual memory and machines.

88. K. Fuchi, H. Tanaka, Y. Manago, and T. Yuba, "A program simulator by partial interpretation," *Second ACM Symposium on Operating System Principles*, Princeton University, 97–104 (October 20–22, 1969). Describes a program for the HITAC-8400, a machine very much like the RCA Spectra 70/45, by which a single virtual HITAC-8400 is simulated. As in CP-67 and other virtual machine control programs, nonprivileged virtual CPU instructions are executed directly by the hardware, and privileged instructions are interpreted by software.

89. K. Fuchel and S. Heller, "Considerations in the design of a multiple computer system with extended core storage," *Communications of the ACM* **11,** No. 5, 334–340 (May 1968). Discusses the (proposed) usage of a one million-word ECS for a dual CDC 6600 system at Brookhaven National Laboratory. Presents analytic performance estimates.

90. E. Gelenbe, "Optimum choice of page sizes in a virtual memory with a hardware executive and a rapid-access secondary storage medium," *ACM SIGOPS Workshop on System Performance Evaluation*, Harvard University, 321–336 (April 5–7, 1971). Determines, for a system with multiple page sizes, the set of sizes minimizing the expected amount of storage needed for page tables. The smallest page size is assumed to be fixed at a value such that the cost of storage wasted because of internal fragmentation is negligible compared to that used for page tables. Results are obtained for uniform and exponential segment-sized distributions.

91. R. P. Goldberg, *Virtual Machine Systems*, Massachusetts Institute of Technology Lincoln Laboratory Report MS-2687 (September 4, 1969). Hardware characteristics are categorized for a virtual machine system. The notion of "sensitive" instructions is introduced to provide integrity of the supervisor state. Implementation of a virtual CP-67 under CP-67 is then discussed and shown to be feasible. With some restrictions, CP-67 as a virtual machine system running on a IBM System/360 Model 65 is also shown to be practicable.

92. R. P. Goldberg, "Hardware requirements for virtual machine systems," *Proceedings of the 4th Hawaii Conference on Systems Science*, University of Hawaii, 449–451 (January 12–14, 1971). Defines a virtual machine in terms of minimal criteria: (1) the method of execution of nonprivileged instructions in supervisor and problem state must be roughly equivalent for a large subset of the instruction repertoire; (2) a method of protecting the supervisor from the active virtual machine must be available; and (3) a method automatically signaling the supervisor when a virtual machine attempts to execute a sensitive instruction must be available. On this basis, it is shown that certain machines are suitable for a virtual machine system, while others cannot serve this purpose.

93. R. P. Goldberg, "Virtual machines: Semantics and examples," *IEEE Computer Society Conference*, Boston, Massachusetts, 141-142 (September 22-24, 1971). Defines a virtual machine as a "duplicate of a real existing machine, in which a nontrivial subset of the virtual machine's instructions execute directly on the host machine in native mode." Compares this definition with the notions of "extended" and "emulated" machines, virtual memory, and virtual machine time-sharing systems. Outlines certain types of virtual machine architectures and gives examples of these types.

94. D. D. Keefe, "Hierarchical control programs for systems evaluation," *IBM Systems Journal* **7**, No. 2, 123-133 (1968).

95. J. S. Liptay, "Structural aspects of the System/360 Model 85, Part II, The cache," *IBM Systems Journal* **7**, No. 1, 15-21 (1968). Discusses the organization of the high-speed buffer, or cache, used on the IBM System/360 Model 85 and the studies by which its parameters were selected.

96. S. E. Madnick, "Time-sharing systems: Virtual machine concept vs. conventional approach," *Modern Data* **2**, No. 3, 34-36 (March 1969). A discourse on time-sharing systems belonging to two categories: conventional and virtual machines. Features of both are discussed, with the following guidelines given. Conventional time-sharing systems: (1) computer utility, providing basic functions to many users; (2) pool of resources; and (3) efficiency. Virtual-machine time-sharing systems: (1) modular development; (2) medium-scale time-sharing systems; (3) development of systems programs; and (4) program evaluation and measurement.