DEPARTMENT OF COMMUNICATIONS

CANADA

MINISTÈRE DES COMMUNICATIONS

VOLUME 2

TIME SHARED SYSTEMS HARDWARE

- Memory Hierarchy Management -

by

John deMercado

Terrestrial Planning Branch

June 1972

CANADA

VOLUME 2

TIME SHARED SYSTEMS HARDWARE

- Memory Hierarchy Management -

by

John deMercado

Terrestrial Planning Branch

June 1972

Acknowledgements

The purpose of these notes is to promote dialogue within the Terrestrial Planning Branch and serve as a basis for our computer-communication systems implementation program.

The notes are only in first draft form and borrow heavily from the references. They should be read in conjunction with the attached reference papers.

As a revised version is planned the author would appreciate any corrections or omissions in the text that were brought to his attention. He also wishes to thank Messrs. John Harris, S. Mahmoud and Kalman Toth for their valuable contributions.

Miss Gail Widdicombe and Miss Yollande Chartrand typed them in record time from an almost unreadable handwritten manuscript.

CONTENTS

Introduction

The intent of these notes is to briefly review the features of the hardware required for effective memory hierarchy management in time sharing systems. The time shared systems will have the general architectures as shown in Figure 1 below.

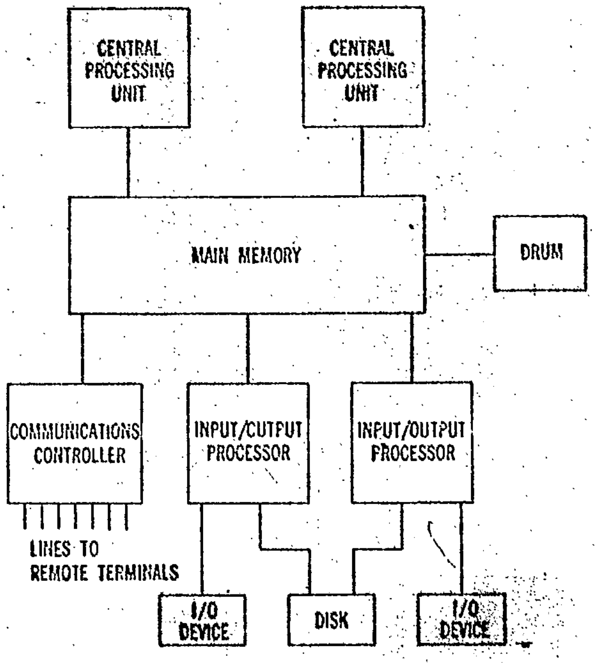


Figure 1

"Time Shared System"

The major hardware features required by such systems are:

- protection mechanisms to help safeguard one process from another and the system from itself and user processes and:
- mechanisms which contribute to efficient dynamic allocation of resources.
- high reliability.

Memory-system Design Problems

The central resource in current systems is the main memory. This main memory holds the instructions for the arithmetic-logic processors (CPU's) and for the I/O processors (IOP's).

It also serves as the buffer for information passing over communication lines and between various I/O and secondary storage devices, and stores the code for the resident operating system. It goes without saying that the proper design of the memory system is critical to the success of a large scale time-sharing system. Figure 2 shows the memory centered model of a computer system which shows the memory as the control resource.

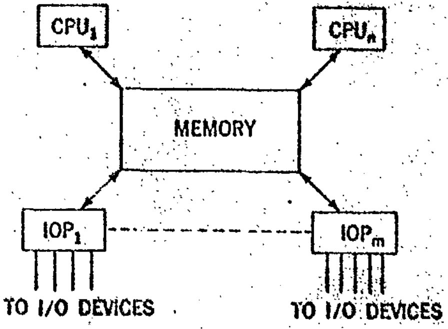


Figure 2

Addressing and Allocation

Main-memory devices have multiplexing properties that must be considered in order to specify an appropriate addressing and allocation scheme for a timeshared computer. While any portion of main memory can be dedicated to a process. Processors themselves can be allocated only as units. Processors, however, can be multiplexed rapidly, while main memory cannot. This time is required to move information between main and auxiliary memory. This moving of processes between main memory and auxiliary storage in order to multiplex main memory is called swapping.

A requirement in the design of an addressing scheme for a timeshared computer is that it should maximize the allocation advantages of memory and minimize the multiplexing disadvantages. For example, it is preferable to have only one copy of a particular procedure, say a compiler, in main memory that can be shared by several processes rather than have each process obtain a separate copy. Programs designed to be shared by several processes are called reentrant programs or pure procedures. A reentrant program has two characteristics:

- none of its instructions or addresses can be modified during its execution,

- temporary storage and data areas are maintained outside the procedure itself, usually in the memory space of the calling programs.

Although re-entrant programs can be written for machines with a wide variety of addressing techniques, certain addressing techniques can make the writing and protection of these programs simpler.

Memory can effectively be utilized by achieving flexibility with respect to where processes can be placed in physical memory. This ability to relocate processes dynamically in physical memory is by a variety of addressing and allocation techniques.

The cost of designing and implementing application systems, as well as the treatment of certain classes of problems - is to be affected by the properties of the addressing and allocation scheme. The various tradeoffs in the design of an addressing and allocation system must take into account both user needs and system considerations. A designer must decide whether the logical-address space is going to be smaller, the same size or larger than the physical-address space. The structure of the logical-address space must also be determined. Many structures are possible, e.g., the large linear array commonly used, a set of linkable linear arrays, as found in Multics, or a tree structure. It must be decided how much of this structuring to perform in hardware and how much in software. The technique of translating or mapping the logical addresses to physical addresses must be determined. Present systems perform this mapping at three points, namely

> When the procedure is prepared as an operable computer program; the result is an absolute program, which, in effect, is assigned the same resources each time it is run.
>
> When the program is loaded; this is known as static relocation.
>
> When the program is in execution; this is called dynamic relocation.

Usually only linear arrays or sets of linear arrays are considered as forms of hardware memory structures, because more specialized structures, such as trees, lists, or rings, are usually left for implementation by software processors.

Static Relocation

The translation of data references to physical addresses is easily accomplished during program preparation but suffers from the severe problems which arise when one attempts to share or modify programs. For example, if one inserts an instruction into a program, all references to instructions and data beyond the point of insertion must be updated.

Similarly when one constructs a program out of routines prepared independently, the address references must be modified to reflect the locations into which the routines are loaded. Further, translation at that time restricts the size of the logical-address space to that of the physical-address space.

The process of static relocation involves a fair amount of computation. In systems using static relocation, programs are usually assembled as if they were to be loaded with the first instruction at location zero, with succeeding instruction and data words being placed in contiguous cells from this point. The location of the first word of the program is called the base address. All instructions or data words with address references are marked by the assembler. Then at load time, a program called the loader adjusts all address references to reflect the actual base address at which the program was loaded. If several programs assembled independently are to be loaded as a unit, the loader, using information supplied by the assembler, adjusts the interprogram address references to reflect the actual locations of the different programs. This process is called linking.

With static relocation, a user can be initiaély loaded anywhere in memory. However, when the process is removed to auxiliary storage and then returned during swapping, it must be placed in the same locations as before, to avoid the loading process. (Furthermore, to go through the loading process again implies that the program must be separable into a pure procedure part and a data part and that the data part must contain no absolute-memory addresses.) The major gain of static relocation is that during the loading process independtly written programs and data can be combined into a computation with proper linking of parts. The proper mapping to the physical-address space is performed by the loader. Each program can be written in a logical space of its own, but no duplication of symbolic location names is allowed, although programming techniques can be developed to resolve such duplication.

The ability to load programs anywhere in physical memory is useful in the linking process above but of little value in achieving effective memory utilization in a timeshared system. For example, when a new process is to be started, the system can attempt to find a process which would fit in an available block of cells. If such a process can be found and it can remain in main memory until completion, static relocation is sufficient to enable several processes to share main memory. (The assumption of some sort of memory-protection scheme is implicit.) A more usual situation will be that the total number of free cells available is sufficient for the number required by a new process but that these cells are not in a contiguous block.

If swapping is required, then even if a contiguous block were available on initial loading, the same contiguous block will not necessarily be available each time the process is run, without moving some information to another aspot in main memory or moving it to secondary storage. For these reasons, systems without dynamic-relocation hardware, when used for timesharing, generally have allowed only one complete process to reside in memory at a given time. Thus, during the swapping operation, the system must remain idle. It is this situation which motivated the development of dynamic-relocation methods.

Dynamic Relocation Using Base Registers

One of the simplest and most common dynamic-relocation techniques uses base registers, which are registers that can have their contents added to the address of each memory operation. By adding the contents of a base register to all addresses, one can load a program anywhere in memory in a block of contiguous cells and then set the appropriate base address of the program into the base register. Using base registers, programs are initially loaded using static-relocation techniques but can be dynamically relocated as a unit later without going through the loading process. This flexibility results because the loading is to logical space not physical space. The base registers form a hardware map which maps logical space to physical space. Further flexibility is gained if there is more than one base register, which facilitates sharing programs and makes it possible to split a program for loading into noncontiguous storage areas.

There are many possible variations of the base-register technique. In fact, techniques such as segmentation are implemented using some hardware registers called base registers. Here, we are only interested in the concept of base registers in its simplest form as defined above and illustrated in Figure 3 below. There are two common ways of specifying which base register to use in forming an address. One technique, represented by the IBM System 360, requires the base registers to be directly addressed by the program and allows the program to access the base registers. The second technique, represented by the UNIVAC 1108, does not allow programs to access the base registers and implicitly addresses the base registers depending on the type of memory operation being executed. For example, all instruction fetches use one base register and all data fetches and stores use another base register.

Program sharing is performed in a system using base registers by writing the reentrant programs to make memory references to themselves through one base register and to make memory references to data in the calling process through a second base register.

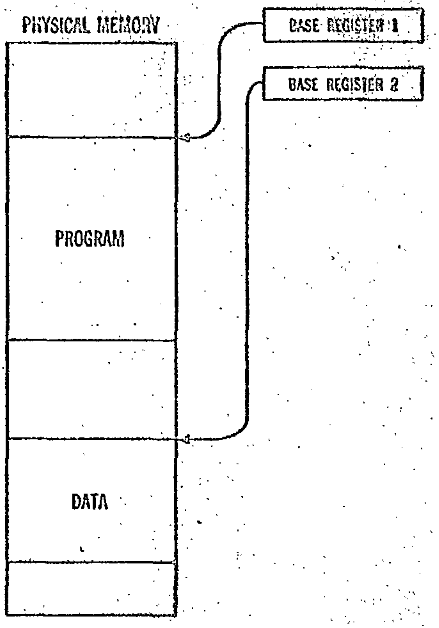


Figure 3

Two base registers used for dynamic relocation of program and data.

Size of Logical Space

The size of the logical-address space using static relocation or dynamic relocation with base registers is usually equal to or less then the size of the physical-address size. A larger physical space can be simulated by the user by explicitly overwriting a portion of his computation not immediately required with another part brought in from auxiliary storage. This process is called overlaying. Overlaying is closely related to the concept of swapping except that overlaying is a user responsibility whereas swapping is a system responsibility.

Memory Utilization

One of the problems uncovered by static relocation is the fact that, once loaded, a process's address references are bound to a certain contiguous area of memory and that during swapping the process must be returned to the same area of main memory each time it is given control of the physical processor. When base registers are used, this restriction no longer holds. When the processor is to be switched to a process not in main memory, a free contiguous block of main memory must be found for it to reside in. If such a block exists, no information need be saved on auxiliary memory in order to make room for the incoming process. The more usual situation results when although enough free cells are available in main memory for the process, they are not in a large enough contiguous block. In this case, a system designed to use base registers can do three things:

- search for a process which will fit into one of the available contiguous blocks,
- swap out part of some process presently in main memory bordering on a free area in order to make a large enough contiguous area, or
- perform a compacting operation on main memory. Figure 4 illustrates the last two ideas.

Figure 4a shows memory at a given point in time. There are two programs entirely residing in memory and three free-space areas (holes). It is desired to bring into memory a third program C which is larger than individual holes but smaller than total space available in holes 1 and 2. Figure 4b shows one approach to making enough space available to fit in program C. Program A is moved entirely to start at the beginning of memory, thus creating enough free space for program C. Figure 4c shows another way of making enough space available to fit in program C. Enough of program A bordering on hole 1 is removed to auxiliary storage to make room for program C.

One solution to the problem of finding a large enough contiguous area might be to use multiple base registers so that smaller pieces of the process could be loaded into existing free spaces. This approach seems to be impractical because the instructions of a given piece must refer to the correct base register. Thus, the programmer or compiler must decide how to split up the process and which base registers to assign which pieces. Binding instructions to base-register addresses at load time means binding the process to a portion of logical space.

The system could not easily perform this base-register assignment function dynamically because it would be very time-consuming and complicated to determine which instructions to modify.

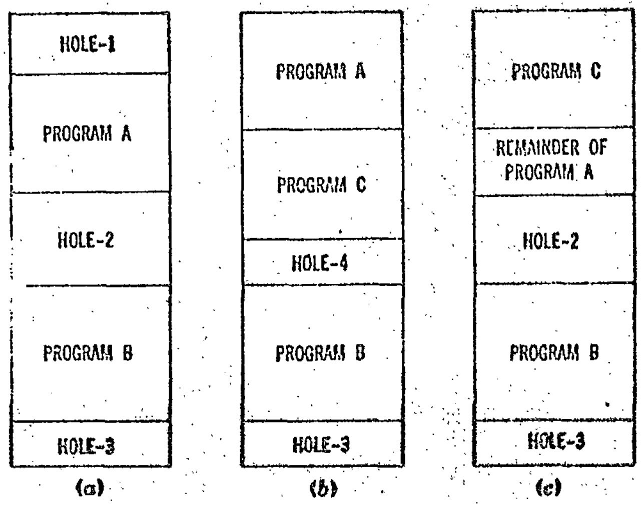


Figure 4: Memory allocation using base registers: (a) typical memory snapshot at a point in time; (b) making room for program C by compaction; (c) making room for program C by partial removal of program A.

Dynamic Relocation Using Paging

Dynamic relocation using base registers, which requires program to be located in contiguous areas of main memory, leads to difficulties in fully utilizing main memory because free areas develop which are not large enough to be used. If, however, programs and main memory could be broken into small units and the program pieces could be located in corresponding sized blocks anywhere in main memory, then the possibility exists of utilizing main memory more éffectively. Paging is the name given to a set of techniques which enable such a uniform memory ragmentation to be implemented. Paging techniques can also allow economic implementation of a logical-memory space larger than the physical-memory space.

In a paged system, physical memory is considered to be broken up into blocks of a fixed size, usually 512, 1,024, or 2,048 words. The term page refers to units of logical space, while equal-sized units of physical space are called blocks. The programs are also considered to be split into pages of a size equal to the block size of physical memory. Thus, the address in such a system is considered to be represented by two numbers: (1) a page address or number and (2) a line-within-page address. For a machine with an n-bit address field, the high-order p bits are considered the page address and the remaining n - p bits are the line address. The operating system may occupy less memory than a multiple of a larger page size. In newer systems the page size can be changed dynamically by the system. The memory can be more fully utilized by the system if smaller page sizes are available (64, 128, or 256 words). More effective utilization of memory results from using smaller page sizes for the following reason. Since a given process is not usually going to require an amount of memory space which is an even multiple of a page size, the last page of a process will not utilize all the block assigned to it. It seems reasonable to assume that on the average the last page of a process will use half of its assigned block. The larger the page size, the more potential waste space there is going to be. A paging mechanism requires a table, called a page table, or map with one entry for each page in order to perform address translation from logical to physical space. The smaller the page size, the larger the table required for a given logical-address space. Thus, there is a tradeoff between waster space related to page size and resources used to store and manipulate large page tables. The total amount of waste space due to unused block locations depends on the number of processes expected to reside in main memory.

E.g. Paging on the XDS-940

The address space of a process in the XDS-940 can be as large as 64K, and thus the logical-address space is smaller than the physical-address space. It should be noted that there are general cases of a paged system yielding a virtual memory larger than the physical-address space. A process in the XDS-940 is broken up into 2K word pages, and memory is similarly broken into 2K word blocks. There are 14 bits in the address field of a 940-instruction word. The address field is considered to contain two parts, a 3-bit page number and an 11-bit line-within-page number. The relocation mechanism (Figure 5) uses eight 6-bit bytes called a memory map. The memory map in the XDS-940 is organized as two 24-bit registers. Each register contains four map bytes. These registers are called the real relabeling registers, because they relabel (map) the page number into a physical-memory block number. These map bytes are considered by the hardware numbered 0 to 7 and correspond to logical pages. A given map byte is addressed by the page number contained in the memory address. Within a given map byte is a number for the actual physical block containing the code for the logical page. For example, in Figure 5 logical page 0 is in physical block 32, logical page 1 is in physical block 3, and so forth. The numbers in the physical blocks of the figure indicate which logical pages they contain.

The logical address is converted to a physical address as shown in Figure 6.

The 3-bit page number indicates which map register contains the physical-block number where the page actually resides. The map register is 6 bits long and is shown in Figure 7; note 5 bits contain the physical-block number, and 1 bit is for memory protection. The physical address is simply formed by concatenating the physical-block number with the line number to form a 16-bit address. With 16 bits, 64K of memory can be addressed.

This hardware mechanism is quite simple, but to work as part of the of the total system it requires additional software tables, which keep track of the memory space of each process. The basic idea is that when a process is to be brought into main storage, the software monitor examines the state of main storage and swaps out only as many pages as are required in conjunction with free pages to meet the needs of the incoming process. The monitor then assigns the available physical blocks to the logical pages of the incoming process and swaps its pages into these blocks. The memory map is updated. Then after restoring the processor registers and program counter to the values they had when the process was last executing, the process is restarted.

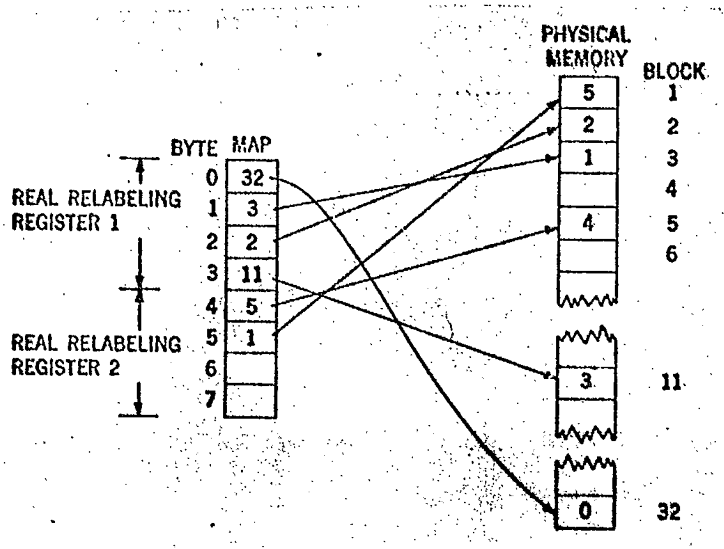


Figure 5: Paging in the XDS-940

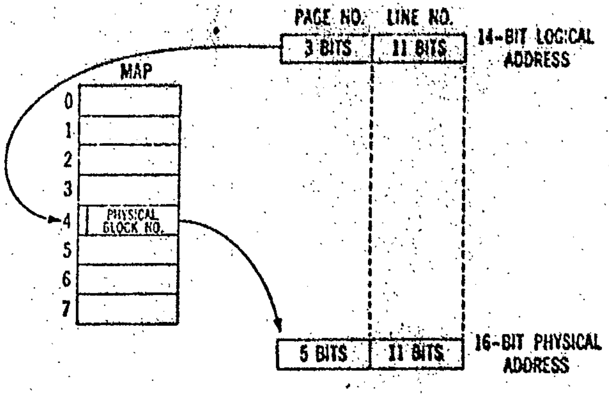


Figure 6: Mapping from logical to physical address in the XDS-940.

Memory Map

The most important general concept introduced above is that of a memory map. A map translates the logical-address space into the physical-address space. In the dynamic-relocation techniques, the map is a set of tables in memory or a set of hardware registers. In the static-relocation technique the map is a program. In the dynamic-relocation method using base registers, the base registers are the map. In the dynamic-relocation method using paging, the page map can be looked at as a way of efficiently implementing multiple base registers. The paging process is completely invisible to the users and to the compilers, which function as if they were working with one contiguous logical chunk. The ability to fragment memory uniformly, made possible by splitting main memory into blocks, means that all blocks of main memory can be used, although assure that no two shared procedures which might be used concurrently occupied the same position in logical space. If the page table were organized and addressed as an actual or simulated associative memory, then it could be reduced in size because no gaps need result. The practical problem of implementing in hardware and software such a large associative map for efficient execution may still create difficulties, although further study may be fruitful.

In summary, then the difficulty of using paging for sharing single copies of procedures and data in full generality and for allowing for data-structure growth results:

- Because of the large number of address bits required to ensure unique page numbers in a large logical space.

- Because of the large, possibly sparsely filled, map required using an indexed page table (with an efficient associative map this argument is reduced, although duplicate entries for each page of shared procedures and data must exist in the map of each process using the shared procedures or data).

- Because of the careful bookkeeping required by the installation and the system to be certain that procedures used concurrently do not occupy the same position in logical space (i.e., have the same page numbers), and to properly position data which contain address references.

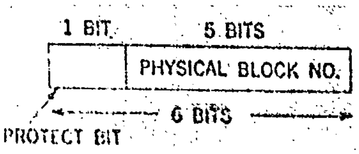


Figure 7: Map byte in the XDS-940.

Segmentation Concept

The problems with physical-space allocation using static relocation resulted because address references were bound to positions in physical space when procedures and data were loaded into the system. Once loading was accomplished, all addresses were absolute physical locations. This restriction was removed in the base-register and paged systems by introducing mechanisms which allowed physical-address references to be made relative to either a base register or block number, the contents of which did not have to be set until execution time. However, the particular base register or map entry to be used was bound into the instructions at load time. In other words, once loading was accomplished, all addresses were to absolute logical locations.

The problem which segmentation sets out to solve is that of allowing relative addressing within the logical-memory space. This means that logical space must be broken up into chunks of contiguous locations and all addresses within a given chunk are to be relative to the start of the chunk.

We then need a hardware or software base register which points to the base location for each chunk. Interchunk references must refer to the proper base register and give a relative address within the referenced chunk. The trick is to develop an efficient mechanism which allows these base registers to be assigned at execution time. The chunks of contiguous logical locations are commonly called *segments*. The basic idea of segmentation is thus quite simple, but the mechanisms for allowing assignment of base registers at execution time are more involved.

A *segment* is an ordered set of data elements (usually computer words) having a name. A particular data element within a segment is referenced by the symbolic segment name and the symbolic data-element name with the segment, (S)/[α]. The notation (S) indicates a symbolic segment named α. The symbolic segment name (S) is eventually (at run time) translated into a base-register number, and the symbolic data-element name in the segment [α] is going to be translated into a relative location within the segment. In other words, a segment is a one-dimensional array, and the segment name is related to the address in logical space of this array (its base address); the symbolic element name within the segment is related to the address of the referenced element relative to start of the segment, as shown in Figure 8.

Segmentationis often referred to as *two-dimensional logical-address space* because particular elements within the logical space are explicitly referenced by a pair of names. A paging system is not considered two-dimensional, even though the address has a page-number and a line-number pair, because these conventions are invisible to the user. To be general one could consider base-register and paged systems as segmented systems allowing one segment, and thus the segment name is implicit. In a general segmented system, the user programs his addresses using a pair notation, (S)/[α]. A segment is a self-contained logical entity of related information defined and named by the programmer, such as a procedure, data array, symbol table, or pushdown stack. There is no logical restriction on the length of a segment, although in any given implementation there will be an upper bound on segment length. Segments can grow and contract as needed.

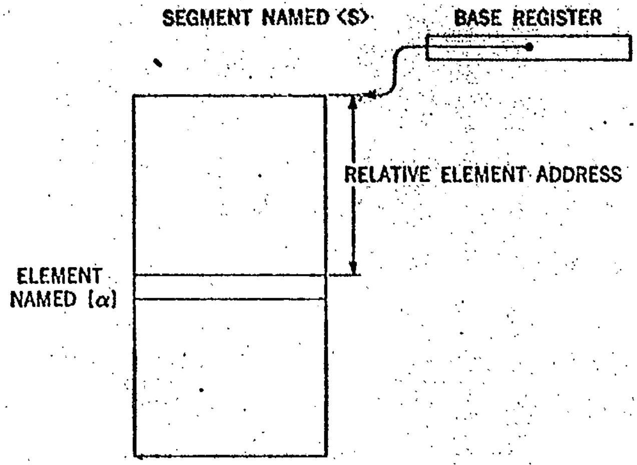


Figure 8: A Segment.

Communications Within the Time Shared Computer

The purpose of a timeshared computer system is rapid time multiplexing of computer-system resources on behalf of user requirements. The system attempts to perform this multiplexing so as to satisfy user completion and response-time needs and to utilize system resources efficiently. These time shared systems are usefully viewed as large communication-switching centers which control the transmission and transformation of information as it moves between the large number and variety of devices (terminals, discs, etc) that are attached.

Communication with Main Memory

The central point through which the information passes in present organizations is main memory (with a possible side journey to the CPU for transformation) as it moves from one device to another. Main memory is a prime system resource and consequently, a potential source of communications problems. In timesharing systems, multiple CPUs, high-transfer-rate secondary storage devices, and numerous I/O devices share access to main memory. The processors which control the secondary storage and I/O devices and communication with memory are usually referred to as channels, I/O controllers, or I/O processors.

Communication with Auxiliary Storage and I/O Devices

A basic communication problem with auxiliary storage and I/O devices is gaining access to a direct-transfer path to main memory. A timesharing system contains a variety of devices attached to it. Associated with these devices is a range of data-transfer rates. Direct-transfer paths to main memory require logic to resolve conflicts for access to a memory module and require sending an receiving circuits at each end of the path; therefore, it is usually uneconomical to provide a separate path for each device. It is possible, however, using the fact that the attached devices have a range of transfer-rate requirements, to design I/O processors which enable many devices to share one direct-transfer path to main memory concurrently.

Communication With Remote Devices

Three major communication problems are associated with remote devices such as terminals, printers, etc. these are

- the transmission of information between the central facility and the remote devices

- the interface between transmission lines and the central facility

- the interface between transmission lines and the remote devices

Along with the transmission of information techniques must be considered for utilizing standard telephone lines for digital information, sharing lines among several devices, and synchronizing communication between remote points. Associated with the interface between transmission lines and the central facility are the problems of identifying, controlling, and addressing communicating devices and converting the transmitted information to a form usable by the central machine and vice versa. Associated with the interface between transmission lines and the remote devices are problems of encoding information and providing identification.

Communication With the Main Memory

Multiple Memory Box and Bus Organization:The technological problem to be solved in the design of a memory communication system is to provide adequate transfer capability between main memory and all processors requiring access. In practical systems, the rate at which data can be transferred between processors and main memory is limited by the transfer capabilities of the memory itself and the memory busses. The rate at which the memory can transfer information is often referred to as the memory bandwidth, usually measured in words per second. Bandwidth limitations also exist for the busses. Because the memory system is shared by several processors, care must be taken in the design to keep performance from being seriously degraded due to interference caused by simultaneous attempts on the part of the several processors to utilize a facility such as a memory bus or portion of memory itself. Figure 9 shows a common method for organizing the memory structure in a resource-sharing system.

The maximum memory-system bandwidth for the system shown in Figure 9 is p X R, where p is the smaller of the number of memory modules m and the number of access paths n, and R is the maximum transfer rate of each box. In other words, the maximum transfer rate is achieved when each path requests access to a separate module.

The minimum transfer rate is just R and occurs when all paths request access to the same module. There is interference in this case.

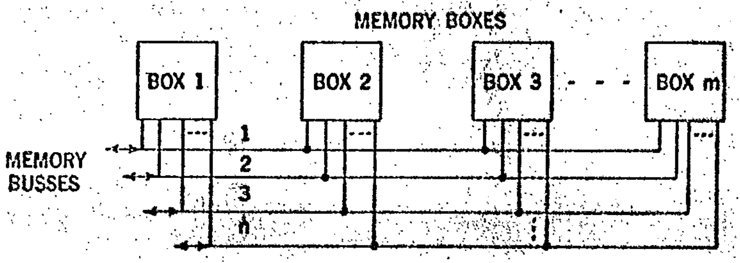


Figure 9: Memory organization in a resource-sharing system.

The scheme shown in Figure 9 cuts interference by allowing simultaneous access to more than one box. That is, if bus 1 requests access to box 2 at the same time buts 2 requests access to box 3, both accesses are granted because each box has its own addressing and read/write circuitry. Even given the scheme shown in Figure 9, serious interference can result when memory addresses are contiguous in the boxes, e.g., box 1 having addresses 1 to 16K 1 to 32K. Consider the case of a high-speed drum processor which transfers at the memory rate. If this device has a higher priority for memory access than the arithmetic unit, then during a block transfer the arithmetic unit could be denied memory access for a prolonged period if it tried to access the memory box bieng used by the drum processor. To get around this problem designers have developed the technique called <u>interleaving</u>.(Analagous to multiplexing)

In an interleaved memory, consecutive addresses are in different memory boxes. For example, in a two-memory-box system all the even addresses might be in one box and all the odd addresses in the other. With an interleaved memory, the probability of one processor's tying up the memory for a significant time is greatly decreased. The design problem is to determine the size of each box and whether or not interleaving is to occur over all boxes or over groupings of boxes.

E.g. The IBM 360/85 Memory Organization. A schematic of the model 85 memory system is given in Figure 10. Main storage in this system has a cycle time of about 1 microsecond. For storage configurations of 500K and 1,000K words (32-bit), storage is interleaved four ways. For smaller storage configurations, storage is interleaved two ways. Note that the buffer storage is available only to the CPU and not to the

I/O or other processors. The buffer storage has a cycle time of 80 nanoseconds. The buffer storage is either 4K, 6K, or 8K words. The design of this system was oriented toward increasing the effective speed of memory as seen from the CPU. The importance of high data-transfer rate between all processors and memory has not been highly developed in this machine. The memory bus is four words wide in order to achieve the bandwidth required for the main applications envisioned. For I/O oriented systems, this organization offers little advantage, but the basic ideas can be extended.

Main memory and the buffer storage are organized into sectors of 256 words. During operation, a correspondence is set up between buffer-storage sectors and main-storage sectors, in which each buffer-storage sector is assigned to a single different main-storage sectors. Because of the limited number of buffer storage sectors, most main-storage sectors do not have any buffer-storage sectors assigned to them. Each of the buffer-storage sectors has a 14-bit sector address register, which holds the address of the main-storage sector to which it is assigned.

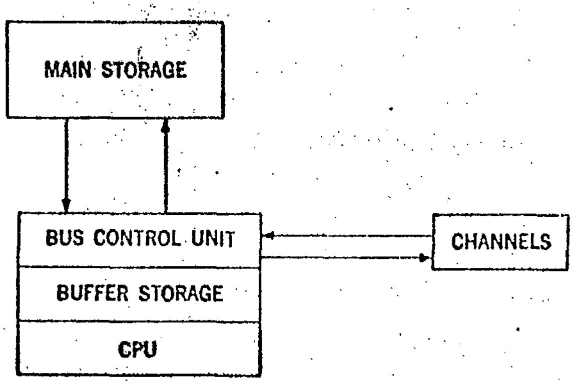


Figure 10:
IBM 360/85 memory system.

The assignment of buffer-storage sectors is dynamically adjusted during operation so that they are assigned to the main-storage sectors that are currently being used by programs. If the program causes a fetch from a main-storage sector that does not have a buffer-storage sector assigned to it, one of the buffer-storage sectors is then reassigned to that main-storage sector. To make a good selection of a buffer-storage sector to reassign, enough information is maintained to order the buffer-storage sectors into an activity list.

When a buffer-storage sector is assigned to a different main-storage sector, the entire 256 words located in that main-storage sector are not loaded into the buffer at once but each sector is divided into 16 blocks of 16 words each, which are located on demand.

Storage operations always cause main storage to be updated. If the main-storage sector being changed has a buffer-storage sector assigned to it, the buffer is also updated; otherwise no activity related to the buffer takes place. Since all the data in the buffer are also in main storage, it is not necessary on a buffer-storage-sector reassignment to move any data from the buffer to main storage.

Two 80-nanosecond cycles are required to fetch data that are in the buffer. The first cycle is used to examine the sector address and the validity bits to determine if the data are in the buffer. The second cycle is then used to read the data out of the buffer. If the data are not in the buffer, additional cycles are required while the block is loaded into the buffer from main storage.

Simulation was used extensively during the design of this memory system. There are many important parameters, such as choice of a replacement algorithm, buffer size, sector and block sizes, which must be determined.

With the simulation running a representative scientific-oriented job mix, it was found that mean performance of this system as compared to an ideal system consisting of only 80-nanosecond memory was 81 percent. That is, on average, the CPU obtained information from the buffer storage on 81 percent of its references.

Memory Management Software - Storage Hierarchies

The purpose of storage system is to hold information and to associate the information with a logical address space known to the remainder of the computer system. For example, the CPU may present a logical address to the storage system with instructions to either retrieve or modify the information associated with that address. If the storage system consists of a single device, then the logical address space corresponds directly to the physical address space of the device. Alternatively, a storage system with the same address space can be realized by a hierarchy of storage ranging from fast but expensive to slower but relatively inexpensive devices. In such storage hierarchies, the logical address space is often partitioned into equal size pages (or unequal size segments) that represent the blocks of information being moved between devices in the hierarchy.

A hierarchy management facility is included to control the movement of pages and to effect the (generally dynamic) association between the logical address space and the physical address space of the hierarchy. When the CPU references a logical address, the hierarchy management facility first determines the physical location of the corresponding logical page and may then move the page to a fast storage device where the reference is effected. The goal of the hierarchy management facility is to maximize the number of times logical information is in the faster devices when being referenced. As this goal is approached, most references are directed to the fast, small stores whereas most of the logical address space is distributed over the slower, large stores.

Memory (hierarchy) management becomes a severe problem in multiprogramming and critical memory systems. In a multiprogramming system, many programs are concurrently executed by the processor. Thus the main memory is shared by many programs. Since the total size of all the programs far exceeds the size of the main memory, in order to keep information that will be used in the near future in the main memory, the system constantly moves information between several levels of storage media. Here, for example, we shall consider the case of paged memory system; that is, the address spaces are partitioned into equal size blocks of contiguous addresses. The page replacement problem is defined as the problem of deciding which page should be kept in memory and which should be removed when additional space is needed. Obviously, the page removed should be a page with the least probability of being needed in the near future. However, this should be done without incurring difficult implementation problems at the same time.

Many replacement algorithms have been proposed and studied, examples:

1. Least Recently Used (LRU)
2. Stack Replacement Algorithms
3. Random Replacement
4. Working Set Replacement Algorithm

for an excellent introduction to those algorithms, see the paper given by R.L. Mattson, et. al.

We shall illustrate briefly as an example the Working Set Replacement Algorithm. (see the paper by W.W. Chu)

Model (Working Set Replacement Algorithm)

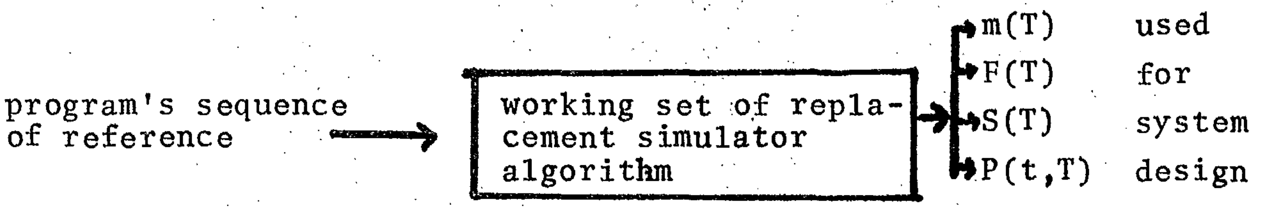
The working set W(t,t) at a given time **t** is the set of distinct pages referenced in the time interval (t-(T-1),t) where T is called the working set parameter. The working set size w(t,T) is the number of pages in W(t,T). The average working set size S(T) is defined as $S(T)= \lim_{k\to\infty} \frac{1}{k} \sum_{t=1}^{k} W(t,T)$.

For systems employing working set replacement algorithm, several parameters are of interest:

1. page inter-reference - internal distribution F(t) which describes the fraction of the page inter-referenced intervals less than T.

2. Average page fault freq. m(T) which describes the average number of page faults per page reference for working set parameter T.

3. Average working set size S(T).


program's sequence of reference
working set of replacement simulator algorithm
m(T)
F(T)
S(T)
P(t,T)
used for system design

(An example including the results is given in Chu's paper).

Examples of how to use the parameters of the working set replacement algorithm.

1. Suppose we would like the system to operate at an average page fault level of about 10^{-4} page faults/reference; that is one page fault in every 10^{-4} page reference, then from the graph representing m(T) versus T for different programs,

$m(T^o) = 10^{-4}$ page faults/reference

$T^o = 22$ m.sec FORTRAN

$T^o = 45$ m.sec DCDL

$T^o = 54$ m.sec META-7

and from the graph representing the average working set size S(T) we find:

$S(T^o) = 15$ page FORTRAN

$S(T^o) = 36$ page DCDL

$S(T^o) = 39$ page META-7

4. Inter-page-fault-time (time between page fault) distribution P(t,T) which describes the fraction of the inter-page-fault times less than or equal to t for a given T.

If we assume that page reference rate is one page/unit time, we immediately obtain the following relationships:

$m(T) = 1-F(T)$

$1/m(T)$ = average running time between page faults

$$1/m(T) = \sum_{t=1}^{\infty} \left\{ t.\ P(t\ 1,T) - P(t,T) \right\}$$

To employ measurement techniques for estimating these parameters, we collect data bout the pattern of references to all the pages which comprise the executed program and measure these parameters experimentally via interpretive execution (steps are shown in the following representation).

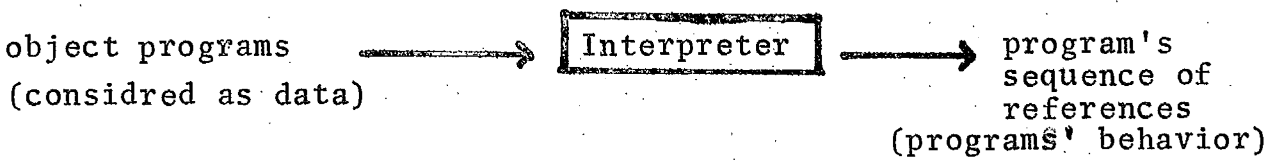


Figure Captions

Figure 11: Average page fault frequency $m(\tau)$ as a function of working set parameter τ.

Figure 12: Average working set size $S(\tau)$ as a function of working set parameter τ.

Figure 13: Inter-Page-Fault-Time Distribution

a) FORTRAN Compiler

b) DCDL

c) META-7 Compiler

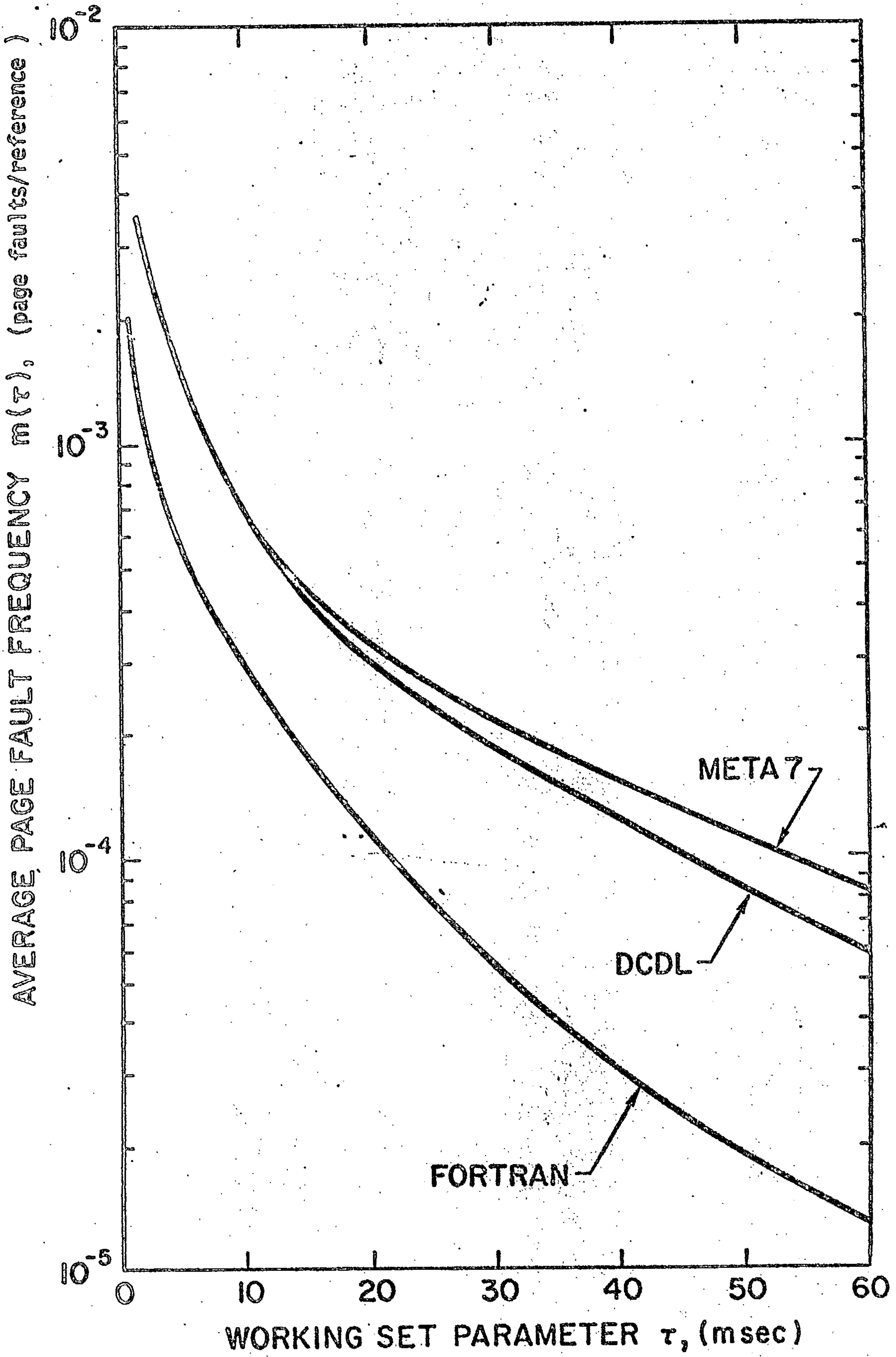

AVERAGE PAGE FAULT FREQUENCY m(τ), (page faults/reference)
10⁻²
10⁻³
10⁻⁴
10⁻⁵
0
10
20
30
40
50
60
WORKING SET PARAMETER τ, (msec)
META 7
DCDL
FORTRAN

Figure 11

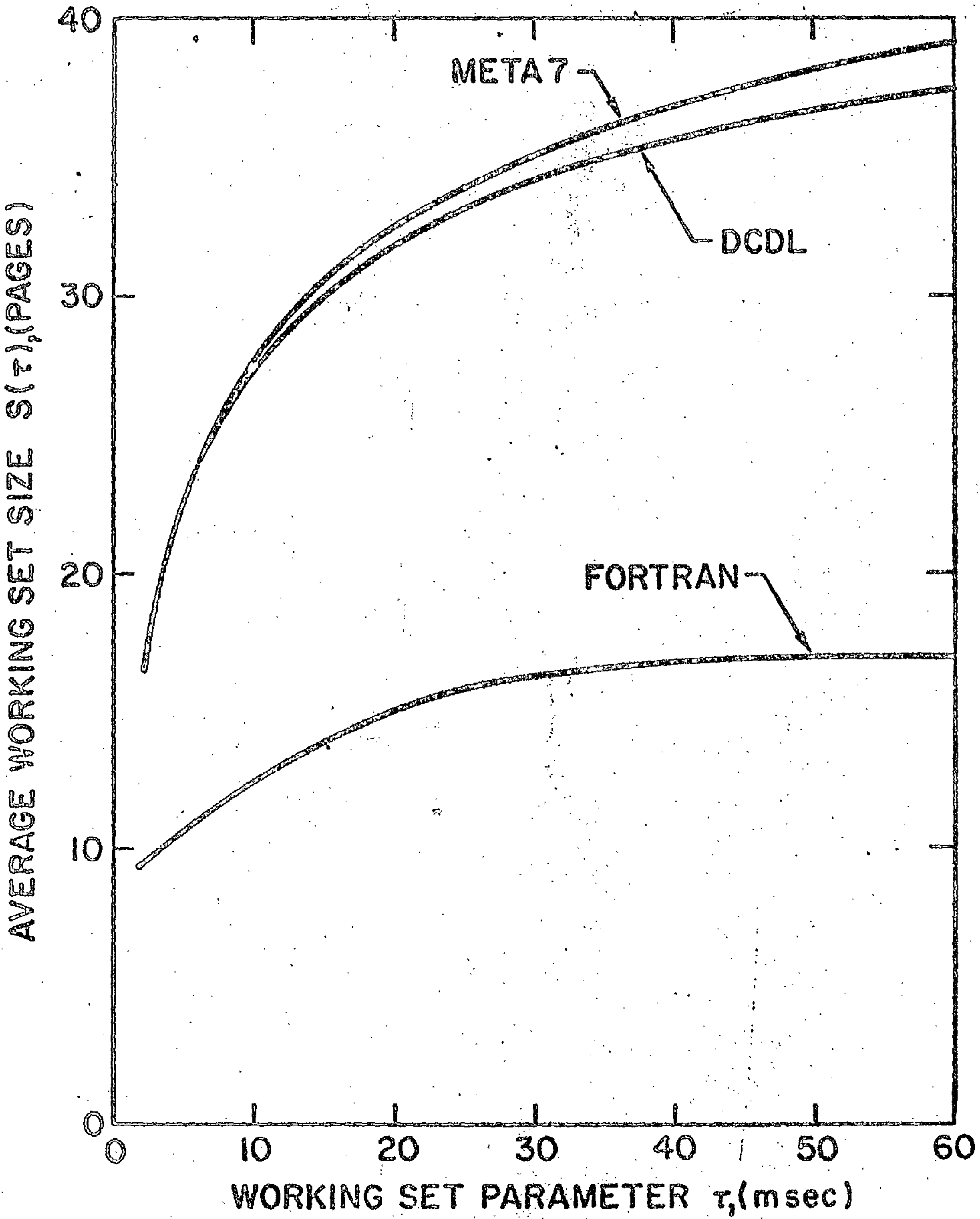

META7
DCDL
FORTRAN
40
30
20
10
0
0
10
20
30
40
50
60
AVERAGE WORKING SET SIZE S(τ),(PAGES)
WORKING SET PARAMETER τ,(msec)

Figure 12

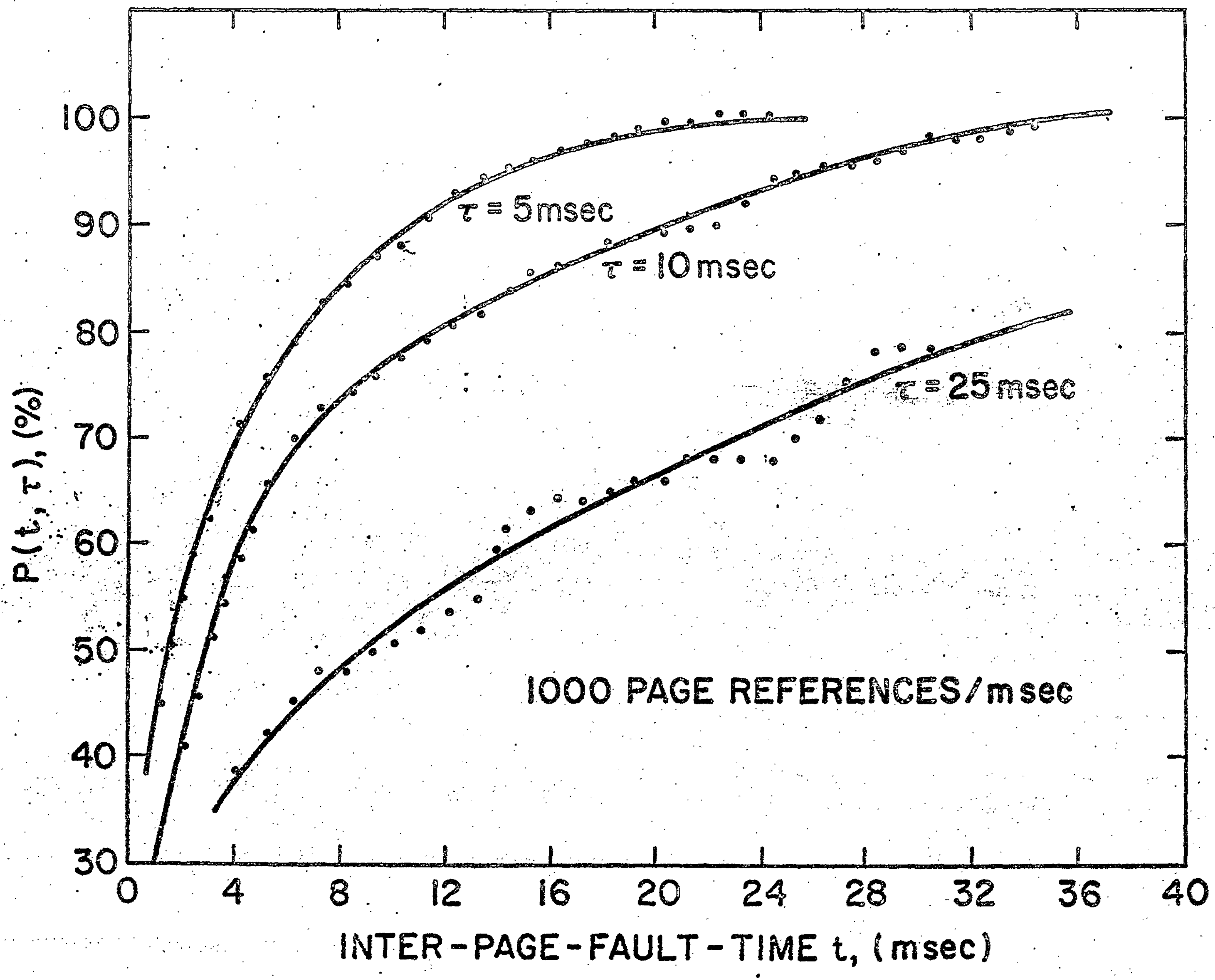

100
90
80
70
60
50
40
30
P(t, τ), (%)
0
4
8
12
16
20
24
28
32
36
40
INTER-PAGE-FAULT-TIME t, (msec)
τ = 5msec
τ = 10msec
τ = 25msec
1000 PAGE REFERENCES/msec

Figure 13

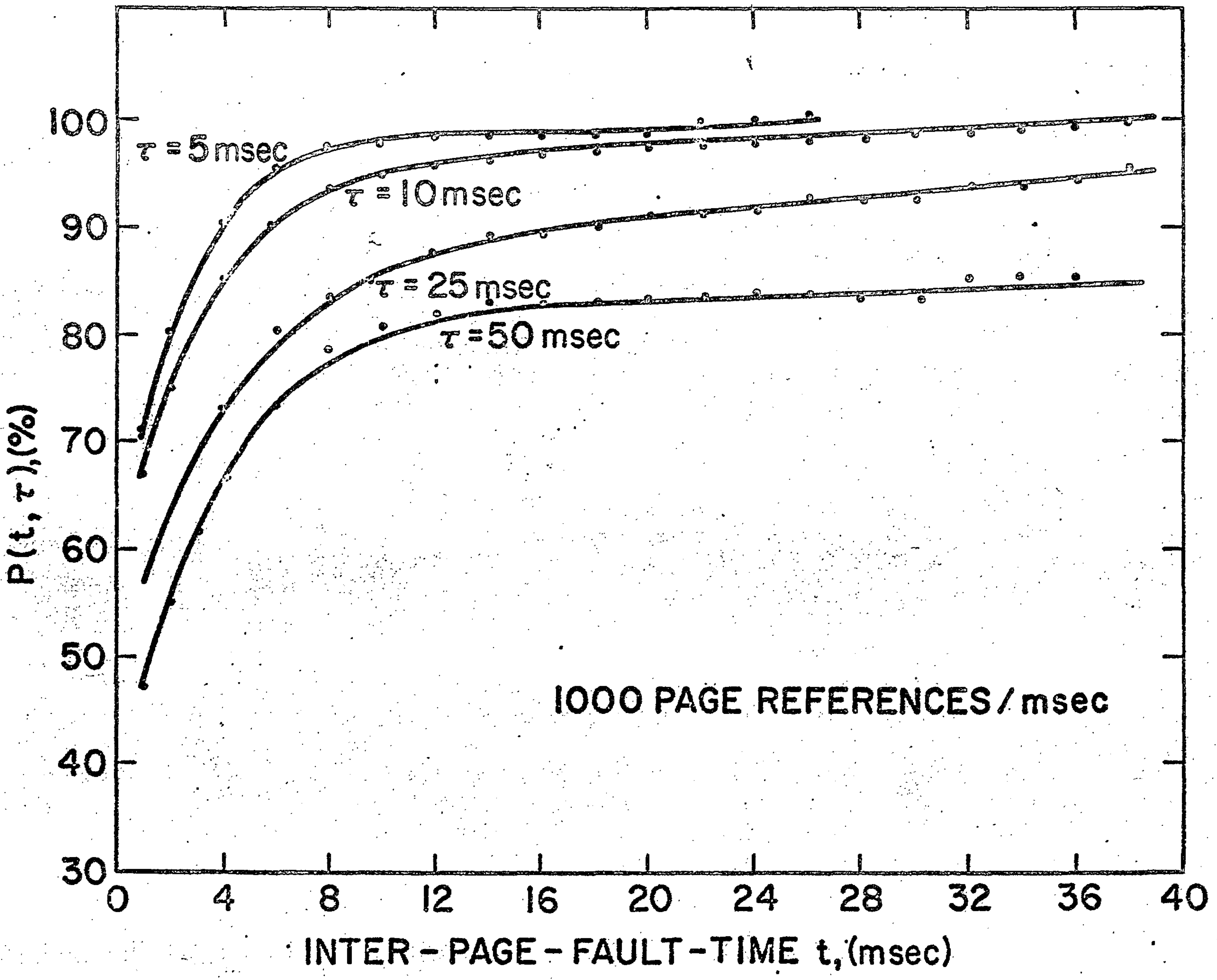


Figure 13 continued

References - Memory Hierarchy Management

Presented at the PBI International Symposium XXI on Computer-Communications Network and Teletraffic, April 1972.

Measurement Data on the Working Set Replacement Algorithm and Their Applications*

by

W.W. Chu, N. Oliver[†] and H. Opderbeck

Computer Science Department
University of California
Los Angeles, California 90024

(Revised March 31, 1972)

ABSTRACT

Page inter-reference interval distribution, average page fault frequency (the frequency of those instances at which an executing program requires a page of data or instructions not in the main memory) average working set size and inter-page fault-time (time between page fault) distribution for a simulated Working Set Replacement Algorithm for three typical programs with different sizes were measured on the UCLA Sigma Executive (SEX) time-sharing system via page reference strings. These measured results are reported in this paper. The average page fault frequency relationships between working set parameters and process scheduling are discussed. These relationships are useful in planning the working set size and process scheduling which optimize system efficiency.

*This research was supported by the U.S. Office of Naval Research, Mathematical and Information Sciences Division, Contract No. N00014-69-A-0200-4027, NR 048-129 and the Advanced Research Projects Agency of the Department of Defense, Contract No. DAHC 15-69-C-0285

[†] Formerly of UCLA, now at General Motors Research Technical Center, Warren, Michigan.

I. Introduction

Memory management becomes a severe problem in multiprogramming and virtual memory systems. In a multiprogramming system, many programs are concurrently executed by the processor. Thus the main memory is shared by many programs. Since the total size of all of the programs far exceeds the size of the main memory, in order to keep information that will be used in the near future in the main memory, the system constantly moves information between several levels of storage media.

In this paper, we consider the case of paged memory systems: that is, the address spaces are partitioned into equal size blocks of contiguous addresses. The paged memory system has been used by many computer systems. However, the basic page replacement problem of deciding which page should be kept in main memory and which should be removed when additional space is needed is still little understood and has been of considerable interest. Obviously, the page removed should be a page with the least probability of being needed in the near future. The difficulty lies in trying to determine which page this will be without incurring difficult implementation problems at the same time.

Many replacement algorithms have been proposed and studied in the past: such as Random, First-in First-out, Stack Replacement Algorithms[1] (for example, Least Recently Used (LRU)), and the Working Set Replacement Algorithm.[2] The first three replacement algorithms require a fixed size memory space for each process. The Working Set Replacement Algorithm, however, requires a variable size storage space for each process and the size varies with program demands. This variable storage space provides an adaptive capability in the replacement algorithm which is quite appealing. The working set principle of memory management states that a program may use a processor only if its working set (set of pages) is in the main memory, and no working set pages of an active program may be considered for removal from the main memory. Properties of the working set replacement algorithm, the relationships among page inter-reference interval, average page fault frequency and average working set size for the Working Set Replacement Algorithm are described in a recent paper by Denning and Schwartz.[3]

Because of the complex nature of program behavior, analytical estimation of the above mentioned parameters of program behavior becomes very difficult. Yet this information is important in the planning of an efficient replacement algorithm that optimize system performance. Therefore we employ measurement techniques for such estimations. We collect data about the pattern of references to all the pages which comprise the executed program, and measure these parameters experimentally via interpretive execution. This technique has been used previously to measure dynamic program behavior[4] and also to measure the performance of Belady's Optimal Replacement Algorithm[5] and LRU replacement algorithms.[6,7]

Here we report the measured program behavior of the Working Set Replacement Algorithm. We shall first report measurement results such as page inter-reference interval distribution, average page fault frequency, average working set size and inter-page-fault-time distribution. We then discuss the use of average page fault frequency to determine the working set parameter, and propose a page fault scheduling algorithm for process scheduling which improves system efficiency.

II. Measurements and Results

The working set $W(t,\tau)$ at a given time t is the set of distinct pages referenced in the time interval $((t-\tau+1), t)$, where τ is called the working set parameter. The working set size $w(t,\tau)$ is the number of pages in $W(t,\tau)$. The average working set size $S(\tau)$ defines as $S(\tau) = \lim_{k\to\infty} \left\{ \frac{1}{k} \sum_{t=1}^{k} w(t,\tau) \right\}$. For systems employing working set replacement algorithms, several parameters of interest are: 1) page inter-reference interval distribution $F(\tau)$, which describes the fraction of the page inter-reference intervals less than τ; 2) average page fault frequency $m(\tau)$ which describes the average number of page faults per page reference for working set parameter τ; 3) average working set size $S(\tau)$, and 4) inter-page-fault-time (time between page fault) distribution $P(t,\tau)$ which describes the fraction of the inter-page-fault-times less than or equal to t for a given τ.

$F(\tau)$ is a fundamental distribution; it closely relates to the other three parameters. When we assume that the page reference rate is one page per unit time, we know that the page references that result in page faults are those references whose inter-reference intervals exceed τ. Thus, $m(\tau) = 1-F(\tau)$. It can be shown[3] that $S(\tau) = \sum_{Z=0}^{\tau-1} m(Z)$. Thus, $S(\tau)$ is closely related to $m(\tau)$. $1/m(\tau)$ is the average running time between page faults. Since $P(t,\tau)$ is the fraction of inter-page-fault-time less than or equal to t, $1/m(\tau)$ is the time average of the density function $P(t+1,\tau) - P(t,\tau)$; that is, $1/m(\tau) = \sum_{t=1}^{\infty} t\cdot[P(t+1,\tau) - P(t,\tau)]$.

To employ measurement techniques for estimating these parameters, we collect data about the pattern of references to all the pages which comprise the executed program and measure these parameters experimentally via interpretive execution. For this purpose an interpreter for the UCLA Sigma-7 time-sharing system has been developed. This interpreter is capable of executing Sigma-7 object programs by handling the latter as data and reproducing a program's sequence of references. This sequence, in turn, can then be used as input to programs which simulate the Working Set Replacement Algorithm.

Three different programs with different sizes were interpretively executed, and their behavior was investigated under the Working Set Replacement Algorithm. A FORTRAN Compiler was chosen as the representative for a small program. META-7 was chosen as the representative for a large program. It translates programs written in META-7 to the assembly language of the Sigma-7. A DCDL (Digital Control Design Language) compiler was chosen as a representative for a medium size program. This compiler is written in META-7. DCDL translates specifications of digital hardware and microprogram control sequences into interpretive code.

Table 1 shows some characteristic properties of these programs. The column 'size' is divided into two parts. 'Static' refers to the number of pages necessary to store the program as an executable file on a disk where one page consists of 512 32-bit words. 'Dynamic' indicates the number of different pages actually referenced while processing the given input data. The difference between the number of pages in static and dynamic results from the fact that programs creat new pages during execution for working storage areas and that not all pages of programs are reference during executing a specific set of input data.

Table 1. Program sizes of the three measured programs

		Size		Number of page references
		Static	Dynamic	
FORTRAN	small	24	34	1,000,000
DCDL	medium	44	58	1,000,000
META-7	large-size	84	153	1,000,000

Figure 1 shows the average page fault frequency m(τ) for the three programs. We note that all three programs exhibit similar page fault characteristics. The average page fault frequency decreases rapidly with τ. Large programs tend to have a slower rate of decrease. The reason for such characteristics is mainly the locality of the program; that is, during any interval of execution, a program favors a subset of its pages, and this set of favored pages changes its membership slowly. Further, the locality for large programs is usually larger than that of small programs. The page inter-reference interval distribution F(τ) = 1-m(τ) can be obtained easily from m(τ). The average working set sizes as a function of τ are shown in Figure 2. Measurement data support the premise that average working set size increases as program size increases and reaches a constant level as τ reaches a certain value. The P(t,τ)'s of the three programs for selected τ's are shown in Figure 3. We note that P(t,τ) is very sensitive to τ and program size. For a given program, the average inter-page-fault-time increases as τ increases. This occurs because for the small τ case, many of the pages to be referenced in the near future are in the secondary memory; thus the average working set size is very small and yields a high page fault rate. For the large τ case, most of the pages are in the main memory which yields a large average working set size and a small page fault rate. For a given τ, large size programs have a higher page fault rate than that of a small size program. In the next section we shall discuss the applications of these parameters to determine the working set parameters and process scheduling which improve system efficiency.

III. Applications of Measurement Data

A. Working Set Parameter τ is an important parameter which affects page fault rate, memory utilization, and thus system efficiency. The measurement data support the fact that τ should be chosen according to the executing program (e.g., size) and system organization (e.g., available memory size and the speed ratio between main and secondary memory). If τ is not properly chosen, for example if τ is too short, then pages are removed from the main memory while still potentially useful. This results in high page traffic between the different levels of memory. If τ is too long, then pages that are not needed may remain in the main memory, which is an inefficient use of memory space. Instead of choosing τ arbitrarily, we propose to determine τ from the measured $m(\tau)$ and designate it as τ^0. As a result, τ^0 is now closely related to program behavior as well as to system organization.

The efficiency of a program is defined as the ratio of total virtual running time to total real running time (total virtual time and total page waiting time); that is,

$$\text{Eff} = \frac{\text{total virtual running time}}{\text{total real running time}} \tag{1}$$

$$= \frac{1}{1+m(\tau)R}$$

where $R = A/T$

A = Access time of the main memory

T = Access time of the secondary memory

Since R is fixed for a given system, from (1) we know a fixed average page fault frequency $m(\tau)$ insures a certain level of efficiency.

Suppose we would like the system to operate at an average page fault level of about 10^{-4} page faults/reference; that is, one page fault in every 10^4 page references. Then from Figure 1, τ^o for Fortcomp, DCDL and META-7 are 22, 45, and 54 m sec (1 μsec per page reference) respectively. From Figure 2, the corresponding average working set size is 15, 36, and 39 pages.

Usually in a multiprogramming environment several types of programs may be concurrently operated by the operating system. The working set parameter of such a system may either be variable of fixed. In the variable τ case, the τ^o should change from one program to another; while in the fixed τ case, the τ^o remains fixed for all types of programs. Because of the simplicity of a fixed τ scheme, it requires less overhead to implement than the variable τ scheme. However, the efficiency may not be as high as that of the variable τ case.

One way to determine the value of a fixed τ is to use the weighted average working set parameters of each program; that is,

$$\tau^o = \frac{1}{n} \sum_{i=1}^{n} u_i \tau_i^o \qquad (2)$$

where τ_i^o = working set parameter for the i^{th} program that selected from its $m(\tau)$

u_i = relative usage frequency of the i^{th} program

n = total number of distinct programs used in the system

The decision as to which scheme should be used for a given system should be based on program behavior, relative usage frequency of all the distinct programs used by the system, and the overhead in implementing these schemes.

B. Process Scheduling

In a multiprogramming system, to increase system efficiency and to reduce response time for short jobs, the job queues for CPU processing usually have several priority levels. Let us consider a system having two levels of queues: Short Quantum Queue (SQQ) and Long Quantum Queue (LQQ). SQQ has a higher priority than LQQ. All jobs enter the SQQ. Processes in the SQQ are given one time slice at a time. The process is put at the back of the SQQ after the process either incurred a page fault or used up the time slice; that is, the process is serviced in a round-robin fashion. A process stays in the SQQ until its short quantum time runs out. It is then put on the front of the LQQ. The LQQ will not be serviced until the SQQ is empty. A process in the LQQ receives service until its long quantum time runs out. It is then put at the end of the LQQ.

When a system is properly designed, such scheduling algorithms yield: 1) fast response time to short jobs, and 2) most of the short jobs are run in the SQQ and long jobs (compute-bound processes) will run in the

LQQ. Since LQQ provides more memory space for each process than that of SQQ, such scheduling yields less page swapping.

If the quantum time of the SQQ is too short, then many of the short jobs will be in the LQQ; if the quantum time is too long, then many computational jobs will be in the SQQ. The system is designed such that most of the short jobs finish their processing in the SQQ and only the compute-bound processes enter into the LQQ. The short quantum time should be larger than the average real process time of short jobs. However, the process time varies from one process to another. In addition, the processing time is further complicated by page faults occurring during its execution.

The real processing time of a process is the sum of the virtual process time and the total time wasted* due to page faults of that process. For example, two processes requiring the same amount of virtual CPU processing time could have very different page fault frequences, and thus yield very different real processing time. Therefore the real processing time is extremely difficult to estimate.

We know that page fault frequency has great influence on system efficiency and the response time of the short jobs. We propose to use a page fault as a measure in process scheduling; that is, when a process exceeds a certain number of page faults or exceeds the quantum time of the SQQ (whichever occurs first), then the process switches from the SQQ to the LQQ. We shall call such a scheme a page fault scheduling algorithm. In a multiprogramming environment, the CPU idle times due to page swapping between main and secondary memories are directly affected by the page fault frequency. The page fault scheduling algorithm should be effective in reducing CPU idle time And improve system efficiency. (See Appendix).

*For a system operating in a multiprogramming environment, we should also include the time spent in waiting for the availability of CPU.

Processes with high page fault rates occupied in the main memory greatly reduce the efficient utilization of main memory. The page fault scheduling algorithm adaptively allocates the low page fault rate processes in the main memory and higher page fault rate processes in the secondary memory. Thus such scheduling improves the utilization of main memory. As a result, this will improve the average response time of the system. An analogy to the above scheduling algorithm is the well known "serving the shortest job first" algorithm in queueing theory that results in improvements in average waiting time; except in our case we have further improved the memory utilization efficiency.

The number of page faults occurring during processing before switching a process from a SQQ to a LQQ depends on the response time required, the number of processes operating concurrently, the replacement algorithm used, and page fault frequency characteristics. Further study in this area is needed.

In order to reduce response time, the quantum time of the SQQ and LQQ are further divided into many time slices. The optimal size of time slices is another important parameter that affects system efficiency. The time slice should be selected such that most of the processes either page fault or become inactive before running out of the time slice. Since $P(t,\tau)$ describes the inter-page-fault-time distribution of a process for a given τ, the time slice for the Quantum Queues can be determined from $P(t,\tau)$. For example, if we wish 95% of the time that the process will page fault before running out of the time slice -- that is, only 5% of the time the process will run to the end of the time slice -- then from Figure 3 we know the time slices of the LQQ* for τ = 10 m sec are: 28 m sec for the FORTRAN Compiler, 13 m sec for DCDL, and 12 m sec for META-7. Time slices for τ = 25 m sec are: 58 m sec for the FORTRAN Compiler, 38 m sec for DCDL, and 35 m sec for META-7. Thus, the measured inter-page-fault-time distribution provides a good way to determine the optimal time slices for the Quantum Queues which avoids excessive unnatural interrupts that degrade response times.

The page fault scheduling algorithm, as well as the selection of the time slice form inter-page-fault-time distribution, are quite general and can be applied to other types of replacement algorithms.

V. Conclusions

Page inter-reference interval distribution, average working set size, average page fault frequency, and inter-page-fault-time distribution for three typical programs with working set replacement algorithms are measured and reported. Measurement results support program locality and the following working set properties: the average page fault frequency decreases rapidly as τ increases and increases as program size increases. Based on these measured data, working set parameter and process scheduling may be selected from and based on the average page fault frequency. The time slices for the Quantum Queues may be determined from inter-page-fault-time distributions. A page fault scheduling algorithm is proposed for process scheduling in a multiprogramming environment. Such an algorithm is effective in reducing CPU idle time and improve system efficiency.

*The three measured programs are not short jobs; they should be run in LQQ. Therefore, these measured $P(t,\tau)$'s provide the estimate of time slices for the Long Quantum Queue.

Although the Working Set Algorithm provides an upper bound on replacement algorithm performance, the high cost of implementation prevents it from being widely used. Therefore future research should be in developing low cost hardware devices for economically implementing the Working Set Algorithm or, perhaps even more fruitful, in developing new replacement algorithms that have performance comparable to that of the Working Set Algorithm but are much easier to implement. For example, we have recently studied a Page Fault Frequency Replacement Algorithm. Such an algorithm adjusts the LRU (Least Recently Used) stack according to page fault frequency. Preliminary results already indicate it has excellent performance.

Acknowledgement

The authors wish to thank P.E. Denning of Princeton University for his critical comments on this paper.

APPENDIX

A Cyclic Queueing Model to Study CPU and I/O Operations

To illustrate the relationships among CPU idle time, average page fault frequency and swapping time (time to bring in a new page from the auxiliary memory) T, a cyclic queueing model[8] is used to study CPU and I/O operations. The system in Figure 4 consists of two classes of service facilities. Service facility class I represents a single CPU; its service rate is directly determined by the average page fault rate* λ. Service facility class II represents k parallel I/O servers with each having an average service rate $\mu = \frac{1}{T}$. The k parallel servers represent, for example, a paging drum with k different sectors. Using such I/O facilities, a high degree of overlap of I/O requests can be achieved in a multiprogramming system with relatively low page fault frequency.

Let P_{ij} be the probability that a job leaving server i will proceed to server j. We assume that the job leaves CPU (server 0) and goes randomly to the k I/O servers for service; thus $P_{0j} = \frac{1}{k}$, for $j = 1, 2, \ldots, k$. Since jobs which have finished their I/O operations always return for CPU operations, $P_{i0} = 1$ for $i = 1, 2, \ldots, k$; and all the other P_{ij}'s are equal to zero.

Let N be the total number of jobs in the system, and let n_i denote the number of jobs in service plus the number in queue at the i^{th} server. The state of the system can then be determined by the k + 1 tuple $(n_0, n_1, \ldots, n_k)$ in which $\sum_{i=0}^{k} n_i = N$. The number of distinguishable states of the system--equal to the number of partitions of N customers among k + 1 servers--is $\binom{N+k}{k}$.

*For a system using Working Set Replacement Algorithm with parameter τ, then $\lambda = m(\tau)$.

Let $P(n_0,n_1,\ldots,n_k)$ be the stationary probability that the system is in state $(n_0,n_1,\ldots,n_k)$, and let all the service times be assumed to be exponentially distributed. Then the steady state equations can be written in the form:

$$\left\{\varepsilon(n_0)\ \lambda + \sum_{j=1}^{k} \varepsilon(n_j)\ \mu\right\} P(n_0,n_1,\ldots,n_k)$$

$$= \sum_{j=1}^{k} \varepsilon(n_j)\ \lambda\ P_{0j}\ P(n_0+1,n_1,\ldots,n_j-1,\ldots,n_k)$$

$$+ \sum_{i=1}^{k} \varepsilon(n_0)\ \mu\ P_{i0}\ P(n_0-1,n_1,\ldots,n_i+1,\ldots,n_k) \qquad \text{(A1)}$$

where the indicating function

$$\varepsilon(n_j) = \begin{cases} 0 & \text{if } n_j = 0 \\ 1 & \text{if } n_j \neq 0 \end{cases}$$

accounts for the impossibility of any customer leaving the j^{th} server if that server is empty.

The left hand side of (A1) represents the rate of transition out of state $(n_0,n_1,\ldots,n_k)$; and the right hand side is the rate of transition into this state. Solving (A1) by a method of separation of variables[8], we have

$$P(n_0,n_1,\ldots,n_k) = \frac{1}{G(N)} \prod_{i=1}^{k} \left(\frac{P_{0i}\lambda}{\mu}\right)^{n_i}$$

$$= \frac{1}{G(N)} \left(\frac{\alpha}{k}\right)^{N-n_0} \qquad \text{(A2)}$$

where $\alpha = \lambda/\mu$ and the normalizing function $G(N)$ is determined from the fact that the sum of all the $P(n_0,n_1,\ldots,n_k)$ is equal to 1. Thus

$$G(N) = \sum_{\sum_{i=0}^{k} n_i = N} \prod_{i=1}^{k} \left(\frac{\alpha}{k}\right)^{n_i}$$

$$= \sum_{n_0=0}^{N} \binom{N-n_0+k-1}{k-1} \left(\frac{\alpha}{k}\right)^{N-n_0} \qquad \text{(A3)}$$

where $\binom{N-n_0+k-1}{k-1}$ is the number of distinguishable partitions of $N-n_0$ jobs among k I/O servers.

The probability that the CPU is idle is

$$P_0 = \sum_{\sum_{i=1}^{k} n_i = N} P(0,n_1,n_2,\ldots,n_k)$$

$$= \frac{1}{G(N)} \binom{N+k-1}{k-1} \left(\frac{\alpha}{k}\right)^{N} \qquad \text{(A4)}$$

For the case k = 1, then (A4) reduces to $P_0 = \frac{\alpha^N}{\sum_{i=0}^{N} \alpha^i}$

For the case N = 3 and k = 6, the values of P_0's for selected α's are shown in Table II.

Table II P_0 vs. α

α	P_0
0.25	0.003
0.50	0.019
1.00	0.091
1.50	0.187
2.00	0.278
2.50	0.362
3.00	0.431
3.50	0.488
4.00	0.537
4.50	0.577
5.00	0.612

We note that α is the ratio of average page swapping time (from secondary memory) to average inter-page-fault-time. A large α implies large page swapping time or small inter-page-fault-time (high page fault frequency), or both. Thus the probability of CPU idle time increases as α increases. Hence, the page fault scheduling algorithm should be effective in reducing CPU idle time and should thus improve system efficiency.

REFERENCES

1. Mattson, R.L. et al., "Evaluation Techniques for Storage Hierarchies," IBM System Journal, Vol. 9, No. 2, pp. 78-117, 1970.

2. Denning, P.J., "The Working-Set Model for Program Behavior," Communications of the ACM, Vol. 11, No. 5, pp. 323-333, May 1968.

3. Denning, P.J. and S.C. Schwartz, "Properties of the Working Set Model," Proceedings of the 3rd ACM Symposium on Operating System Principles, October 1971.

4. Fine, G.H., C.W. Jackson and P.V. McIsaac, "Dynamic Program Behavior Under Paging," Proceedings of the 21st National Conference on ACM, pp. 223-228, 1966.

5. Belady, L.A., "A Study of Replacement Algorithm for a Virtual-Storage Computer," IBM System Journal, Vol. 8, No. 2, 1966.

6. Coffman, E.G. and L.C. Varian, "Further Experimental Data on the Behavior of Programs in a Paging Environment, "Communications of the ACM, Vol. 11, No. 7, pp. 471-474, July 1968.

7. Joseph, M., "An Analysis of Paging and Program Behavior," Computer Journal, Vol. 13, No. 1, February 1970.

8. Gordon, W.T. and G. F. Newell, "Closed Queueing Systems with Exponential Service," Operations Research, Vol. 15, No. 2, April 1967, pp. 245-265.

Figure Captions

Figure 1: Average page fault frequency m(τ) as a function of working set parameter τ.

Figure 2: Average working set size S(τ) as a function of working set parameter τ.

Figure 3. Inter-Page-Fault-Time Distribution
a) Fortran Compiler
b) DCDL
c) Meta 7 Compiler

Figure 4: A Cyclic Queueing system for modeling CPU and I/O operations.

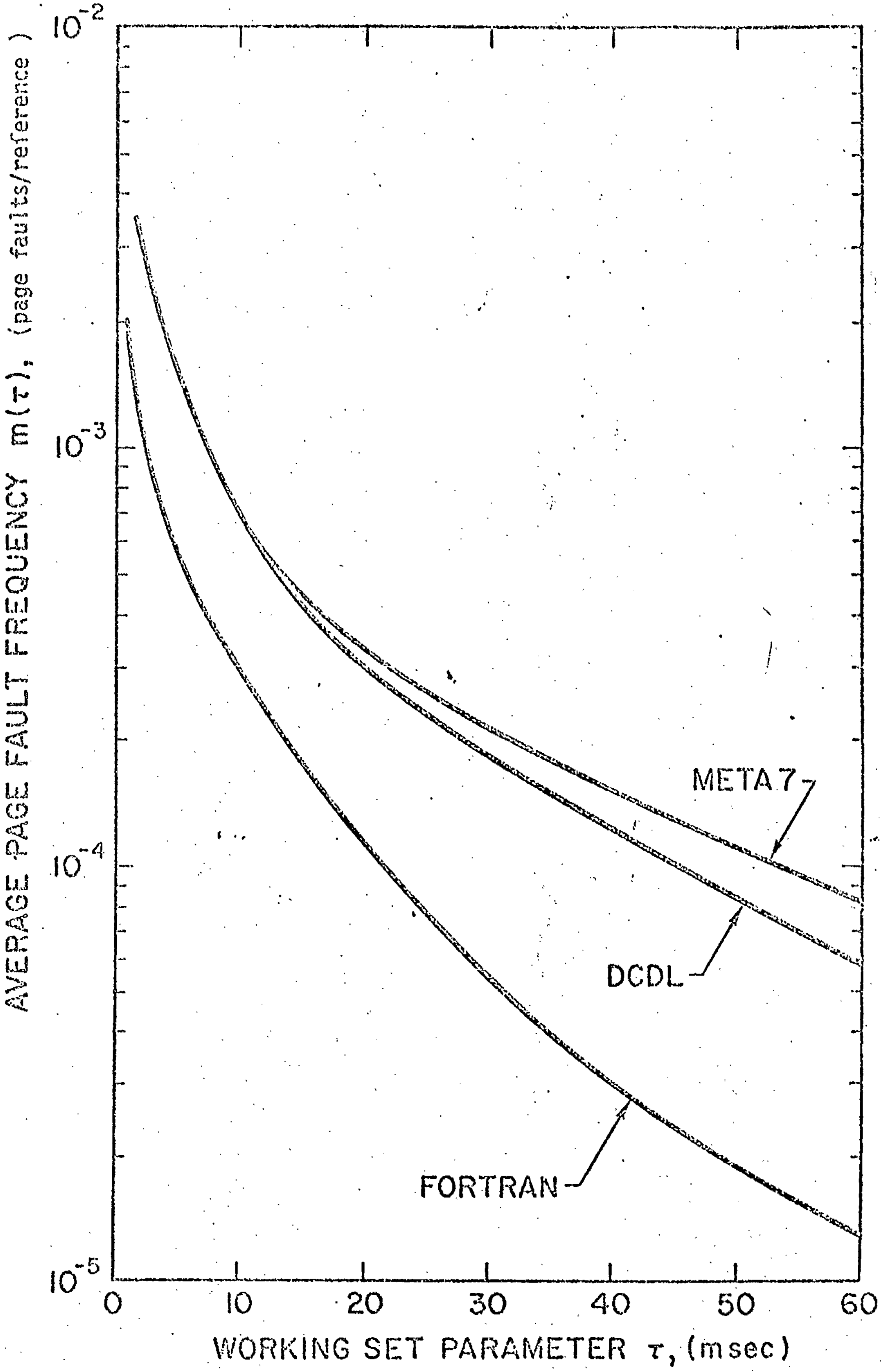

10^-2
10^-3
10^-4
10^-5
AVERAGE PAGE FAULT FREQUENCY m(τ), (page faults/reference)
0
10
20
30
40
50
60
WORKING SET PARAMETER τ, (msec)
META 7
DCDL
FORTRAN

FiG 1

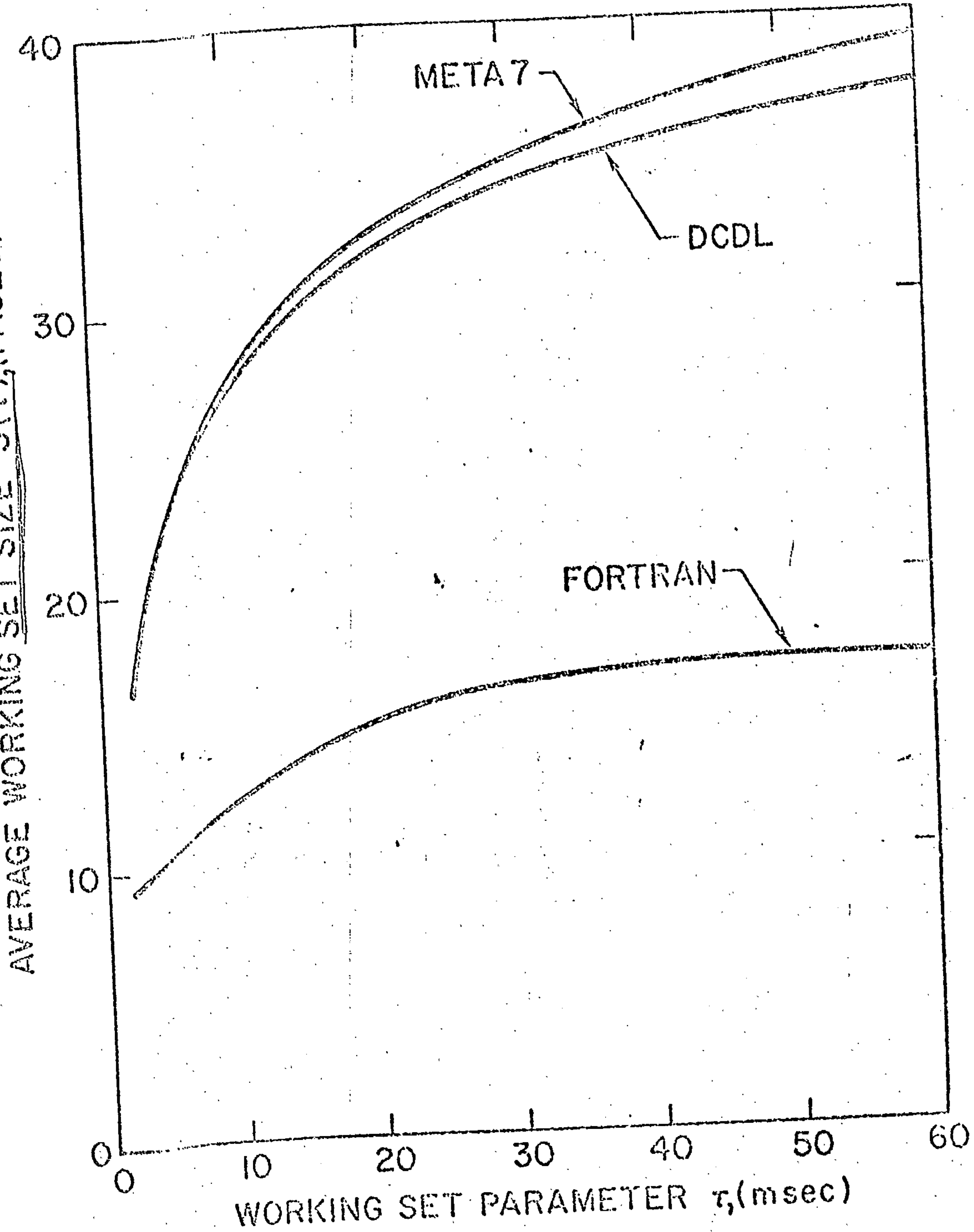

40
30
20
10
0
0
10
20
30
40
50
60
META 7
DCDL
FORTRAN
AVERAGE WORKING SET SIZE S(τ),(PAGES)
WORKING SET PARAMETER τ,(msec)

Fig 2

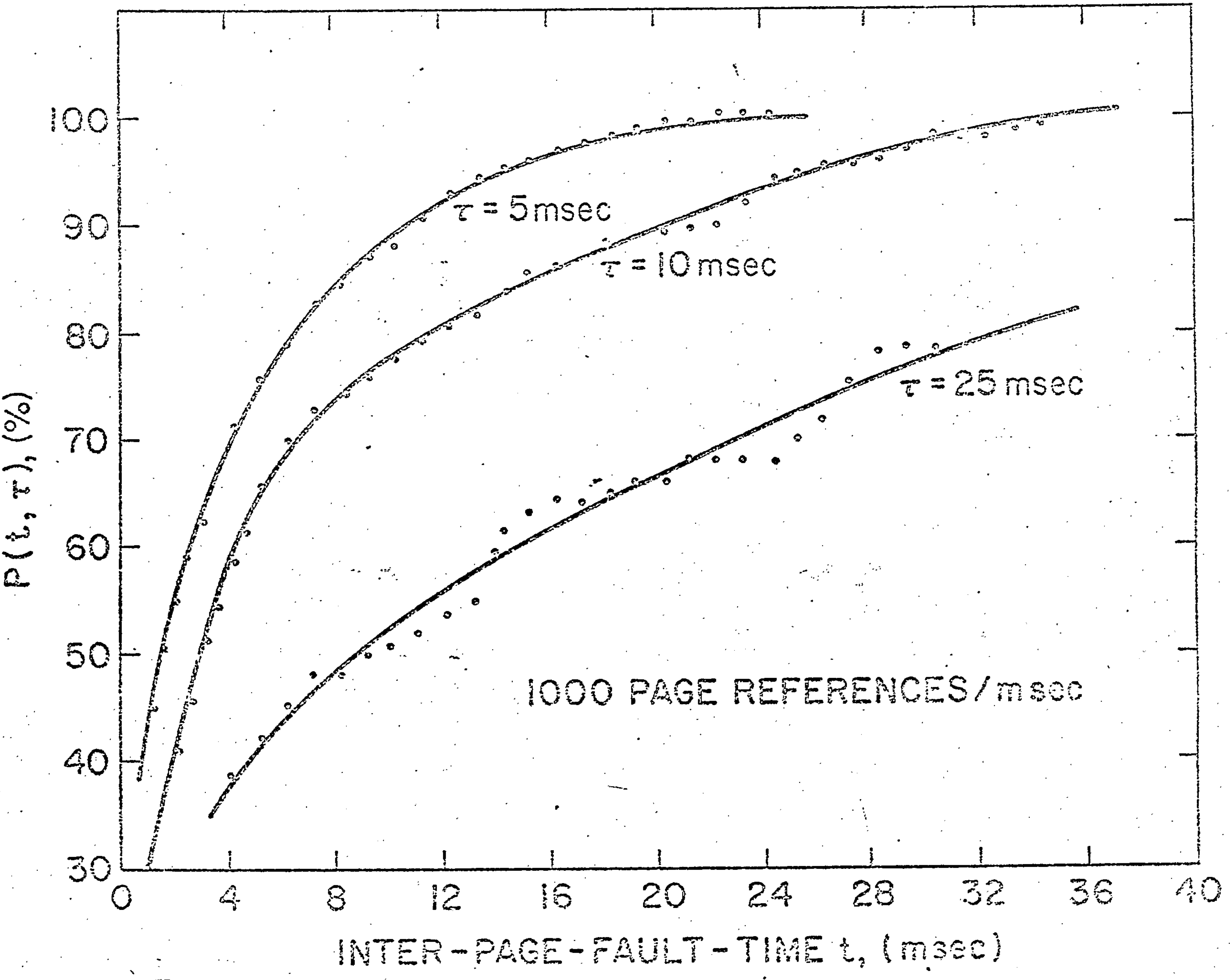

P (t, τ), (%)
100
90
80
70
60
50
40
30
0
4
8
12
16
20
24
28
32
36
40
INTER - PAGE - FAULT - TIME t, (msec)
τ = 5 msec
τ = 10 msec
τ = 25 msec
1000 PAGE REFERENCES / msec

FIG 3A

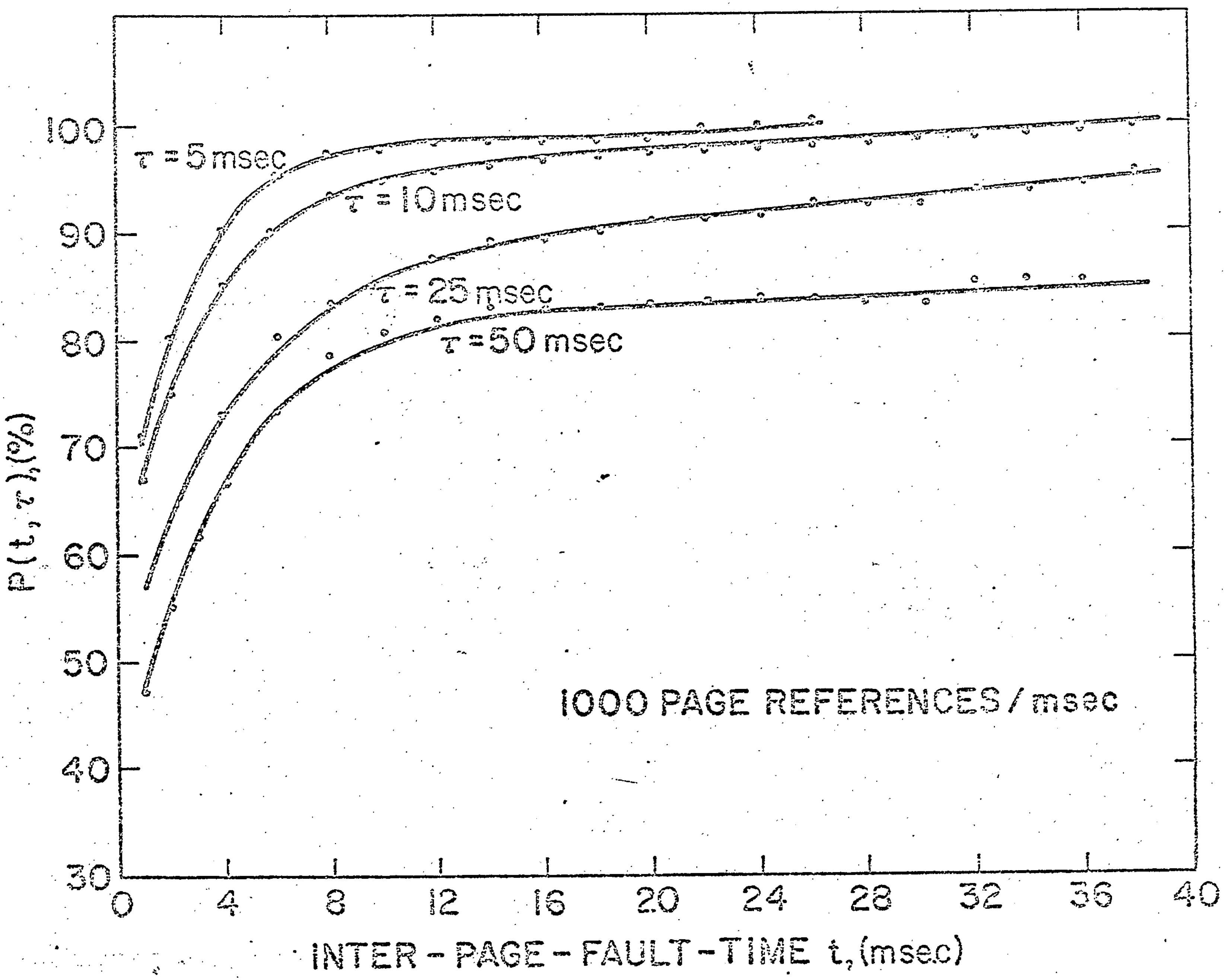

τ = 5 msec
τ = 10 msec
τ = 25 msec
τ = 50 msec
1000 PAGE REFERENCES / msec
P(t, τ), (%)
30
40
50
60
70
80
90
100
0
4
8
12
16
20
24
28
32
36
40
INTER - PAGE - FAULT - TIME t, (msec)

Fig 35

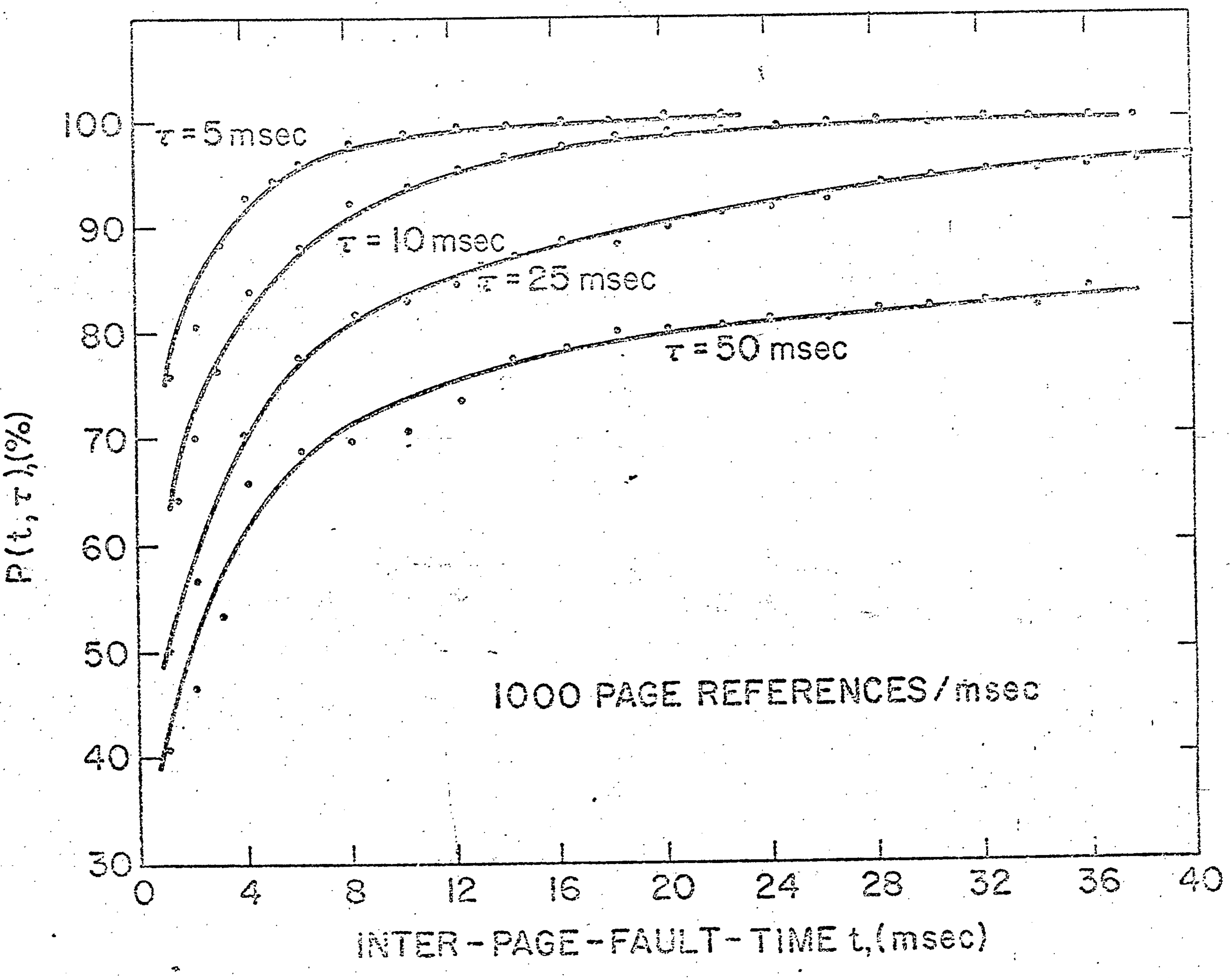

τ = 5 msec
τ = 10 msec
τ = 25 msec
τ = 50 msec
1000 PAGE REFERENCES/msec
P(t, τ),(%)
100
90
80
70
60
50
40
30
0
4
8
12
16
20
24
28
32
36
40
INTER-PAGE-FAULT-TIME t,(msec)

Fig 3c

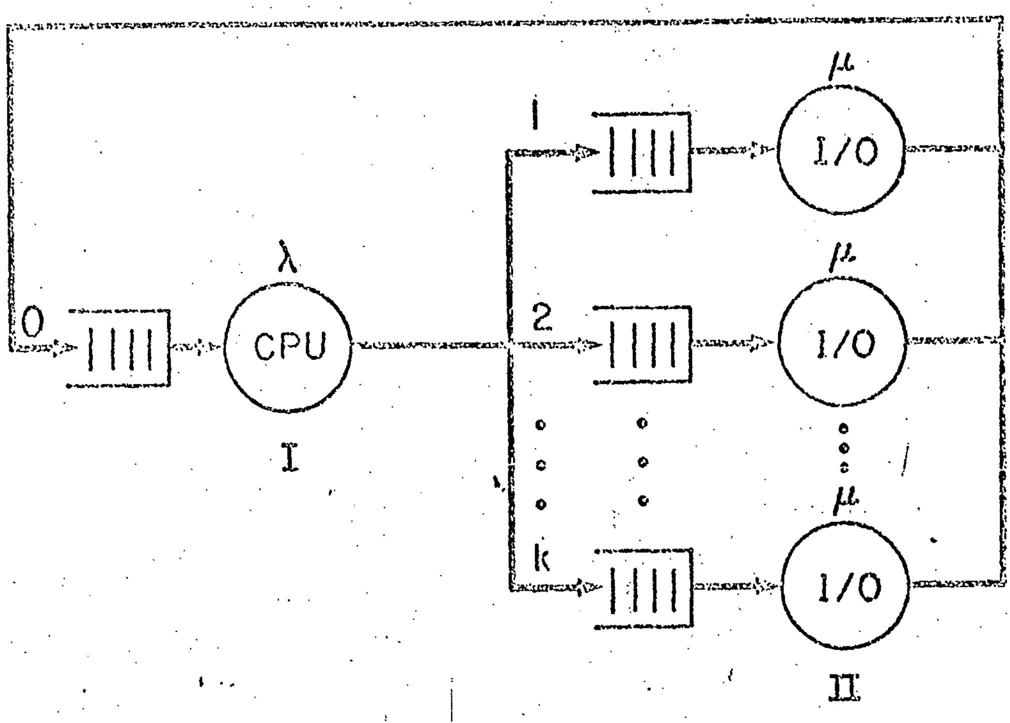


FIG 4

IBM Systems Journal
Volume Nine | Number Two | 1970

The design of efficient storage hierarchies generally involves the repeated running of "typical" program address traces through a simulated storage system while various hierarchy design parameters are adjusted.

This paper describes a new and efficient method of determining, in one pass of an address trace, performance measures for a large class of demand-paged, multilevel storage systems utilizing a variety of mapping schemes and replacement algorithms.

The technique depends on an algorithm classification, called "stack algorithms," examples of which are "least frequently used," "least recently used," "optimal," and "random replacement" algorithms. The techniques yield the exact access frequency to each storage device, which can be used to estimate the overall performance of actual storage hierarchies.

Evaluation techniques for storage hierarchies

J. Gecsei, D. R. Slutz, and I. L. Traiger

Increasing speed and size demands on computer systems have resulted in corresponding demands on storage systems. Since it has been generally recognized that the speed and capacity requirements of storage systems cannot be fulfilled at an acceptable cost-performance level within any single technology, storage hierarchies that use a variety of technologies have been investigated.

Several previous papers describe the general concepts of hierarchy design[1-3] and evaluation,[4-6] whereas others deal with specific hierarchy systems, such as the core-drum combination on the ICT Atlas computer[7-9] and the cache-core combination on the IBM System/360, Model 85.[10,11]

This paper introduces an efficient technique called "stack processing" that can be used in the cost-performance evaluation of a large class of storage hierarchies. The technique depends on a classification of page replacement algorithms as "stack algorithms" for which various properties are derived. These properties may be of use in the general areas of program modeling and system analysis, as well as in the evaluation of storage hierarchies. For a better understanding of storage hierarchies, we briefly review some basic concepts of their design.

The purpose of a s
associate the inform
the remainder of th
Processing Unit (CPU
system with instructi
tion associated with
a single device, then
to the physical addre
system with the same
of storage devices ra
relatively inexpensiv
logical address spac
(or unequal-size segn
being moved between

A hierarchy manage
ment of pages and
between the logical
of the hierarchy. Wh
hierarchy manageme
tion of the correspo
page to a fast storag
these actions are "tr
system (except for ti
is indistinguishable fr

The goal of the hier
number of times log
being referenced. As
directed to the fast, s
space is distributed
system then acquire
while maintaining th
less expensive store
primary justification

Clearly, many facto
hierarchy. On the pe
and characteristics o
of the hierarchy, the
hierarchy manageme
references. On the co
to find and move lo
as the cost-per-bit a
factors, it is quite dif

The typical approach
designers has been to
at various levels of
large number of rel

hierarchy concepts

The purpose of a storage system is to hold information and to associate the information with a logical address space known to the remainder of the computer system. For example, the Central Processing Unit (CPU) may present a logical address to the storage system with instructions to either retrieve or modify the information associated with that address. If the storage system consists of a single device, then the logical address space corresponds directly to the physical address space of the device. Alternatively, a storage system with the same address space can be realized by a hierarchy of storage devices ranging from fast but expensive to slower but relatively inexpensive devices. In such storage hierarchies, the logical address space is often partitioned into equal-size pages (or unequal-size segments) that represent the blocks of information being moved between devices in the hierarchy.

A hierarchy management facility is included to control the movement of pages and to effect the (generally dynamic) association between the logical address space and the physical address space of the hierarchy. When the CPU references a logical address, the hierarchy management facility first determines the physical location of the corresponding logical page and may then move the page to a fast storage device where the reference is effected. Since these actions are "transparent" to the remainder of the computer system (except for timing), the logical operation of the hierarchy is indistinguishable from that of a single-device system.

The goal of the hierarchy management facility is to maximize the number of times logical information is in the faster devices when being referenced. As this goal is approached, most references are directed to the fast, small stores whereas most of the logical address space is distributed over the slower, large stores. The storage system then acquires the approximate speed of the fast stores while maintaining the approximate cost-per-bit of the slower and less expensive stores. This increase in cost-performance is the primary justification for storage hierarchies.

Clearly, many factors can affect the cost-performance of a storage hierarchy. On the performance side, one must consider the capacity and characteristics of each storage device, the physical structure of the hierarchy, the way in which information is moved by the hierarchy management facility, and the expected pattern of storage references. On the cost side, the hardware and/or software required to find and move logical information must be considered, as well as the cost-per-bit and capacity of each device. Because of these factors, it is quite difficult to design an "optimal" hierarchy.

The typical approach to hierarchy evaluation employed by computer designers has been to simulate as many hierarchy systems as possible, at various levels of detail.[9–12] During the first stages of design, a large number of relatively simple simulations may be run with

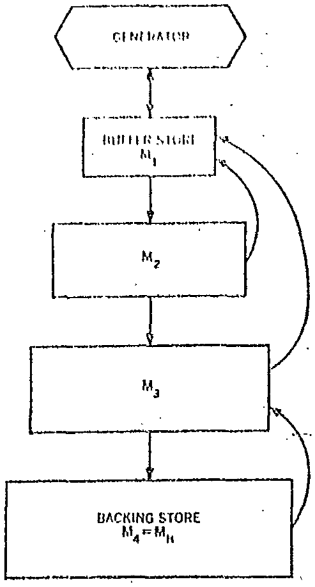


Figure 1 Linear storage hierarchy

fixed, standard address traces. These traces are assumed to be "typical" sequences of storage references obtained from existing computer systems, and they are used to approximate the reference behavior of future systems. The purpose of these simulations is to measure such statistics as data flow and frequency of access to each device in order to estimate the overall performance of an actual system. The resulting performance estimates can then be used to narrow the field of possible designs, which then receive more detailed examination.

Alternatively, one may try to develop analytical techniques that avoid point-by-point simulation but still yield accurate statistics for data flow and access frequencies. Several papers deal with such techniques for hierarchy evaluation.[4–6] In general, the approach here is to run a relatively small number of simulations and extrapolate the measured statistics to a larger class of hierarchies. The difficulty with this approach is the need for various assumptions about the statistical properties of address traces and data flows required to formulate the analytical equations. Moreover, it is difficult to include a quantitative dependence on such factors as data path structure, page replacement algorithm,[13] and address mapping scheme,[3] so that many simulations may still be necessary.

objectives of the paper

This paper presents a technique that can be used to circumvent much of the simulation effort required in hierarchy evaluation. Specifically, we present an efficient procedure that determines, for a given address trace, the exact frequency of access to each level of a hierarchy as a function of page size, replacement algorithm, number of levels, and capacity at each level. In the following, we consider a class of multilevel, demand-paging hierarchies[14] with the same replacement algorithm at every level. The procedures developed here are applicable to a large class of well-known replacement algorithms having certain inclusion properties defined later. These algorithms—which we call stack algorithms—include "least frequently used," "least recently used," "optimal," and a "random" replacement algorithm.

The system model

basic model concepts

An H-level paged storage hierarchy consists of a collection of storage devices M_1, M_2, $\cdots$, M_H, a network of data paths connecting the devices, and a hierarchy management facility. Each device is partitioned into physical blocks called *page frames.* For convenience, the highest-level store M_1 is called the *local store* and the lowest-level store M_H is the *backing store* as shown in Figure 1. The hierarchy management facility controls page movement between the devices and associates each logical page with a physical page frame. Special storage and processing hardware may be required, but they are not included in our model.

References to the
called the *genera*
in which they a
may represent th
the channel. In
address reference
where each addr
set of 2^n possibl
logical addresses
resent the numb
low-order $n -$
the address withi
hierarchy is acc
we can analyze s
by considering a
where each x_t^i is
When we conside
and denote pages

A reference fron
local store M_1.
device M_i, i.e. w
must bring that p
a path for bringi
staging through i
for bringing a pa
ment hardware,
In this paper we
in which the onl
direct ones from
$H - 1$. The rea
paper. Note tha
hierarchy.

The capacity of t
frames, and all
At any time, ea
of the hierarchy
erarchical level,
it may occupy in
as:

- *Unconstrained*
storage devic
- *Fully constra*
frame.
- *Partially cons*

In a later section,
that generates a

References to the storage hierarchy are presented by a single device called the *generator*, and they are sequentially serviced in the order in which they are presented. References from the generator may may represent the requests of several devices, such as the CPU and the channel, in an actual system. The time sequence of logical-address references $X = x_1, x_2, \cdots, x_L$ is called an *address trace*, where each address consists of n bits as shown in Figure 2. The set of 2^n possible addresses is partitioned into 2^k pages of 2^{n-k} logical addresses each. The high-order k bits of each address represent the number of the page containing the address, and the low-order $n - k$ bits represent the location or displacement of the address within the page. Since information movement on the hierarchy is accomplished by transferring pages between levels, we can analyze space allocation and data movement for a trace X by considering a corresponding *page trace* $X^k = x_1^k, x_2^k, \cdots, x_L^k$—where each x_t^k is the number of the page containing address x_t. When we consider a given fixed page size, we omit the superscript k, and denote pages by x_t.

A reference from the generator can be serviced only from the local store M_1. Thus if the desired page resides in a lower level device M_i, i.e. where $i > 1$, the hierarchy management facility must bring that page up to M_1 for servicing. The hierarchy provides a path for bringing pages up to M_1, which may or may not require staging through intermediate levels. Any temporary storage required for bringing a page up to M_1 is included in the hierarchy management hardware, and is therefore not represented in our model. In this paper we restrict our attention to *linear storage hierarchies* in which the only paths for moving pages down the hierarchy are direct ones from each level M_i to level M_{i+1}, where $i = 1, 2, \cdots, H - 1$. The reasons for this restriction are discussed later in this paper. Note that the four-level hierarchy in Figure 1 is a linear hierarchy.

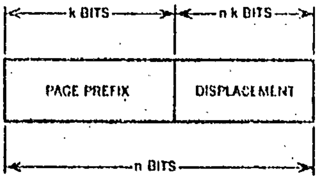


Figure 2 Logical address

The capacity of the backing store is assumed to be at least 2^k page frames, and all logical pages initially reside in the backing store. At any time, each logical page resides in exactly one page frame of the hierarchy. A *mapping function* is associated with each hierarchical level, and specifies for each logical page the page frames it may occupy in that level. The mapping function is further defined as:

- *Unconstrained* if any page/[illegible] can occupy any page frame of the storage device.
- *Fully constrained* if each page can occupy only a single page frame.
- *Partially constrained* in all other cases.

In a later section, we define a technique called "congruence mapping" that generates a whole spectrum of mapping functions.

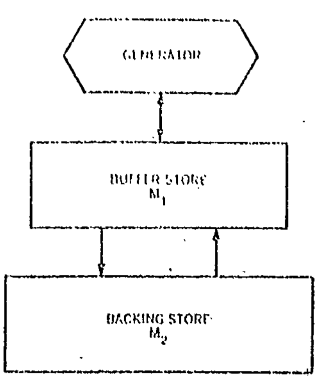


Figure 3 Two-level hierarchy

For simplicity in developing techniques for analyzing storage hierarchies, we first consider a two-level, demand-paged hierarchy with unconstrained mapping. Later, our results are extended to certain classes of multilevel linear hierarchies employing the three types of mapping functions. The local store or buffer has a capacity of C pages, and is directly connected to the backing store as shown in Figure 3. At time t, the generator presents a request for page x_t to the hierarchy. Under *demand paging*, if x_t is in the buffer, the reference proceeds and no page movement occurs. Otherwise, x_t is brought to the buffer from the backing store. If the buffer is already full, x_t replaces some page y_t in the buffer. The selection of the particular page y_t is performed by the buffer *replacement algorithm*. This operation is a key element of storage management.

In the two-level hierarchy shown in Figure 3, a reference to a page residing either at level M_1 or at M_2 is called an access to that level.

For a given hierarchy and page trace, we define the *access frequencies* F_1 and F_2 where F_i is the relative number of accesses to level M_i during the processing of the trace. Thus, if N_1 accesses are made to level M_1, and $N_2 = L - N_1$ accesses are made to level M_2, we obtain $F_1 = N_1/L$ and $F_2 = N_2/L$.

Some important measures of storage hierarchy performance can be obtained from these access frequencies. For example, one can combine access frequencies with a set of effective access times $\{T_i\}$ to obtain an effective (or average) hierarchy access time

$$\bar{T} = F_1T_1 + F_2T_2$$

In general, access times depend on the access paths, device access times, and characteristics of the hierarchy management facility. The access frequencies depend only on the page trace, capacity of the buffer, and replacement algorithm.

For a two-level hierarchy, accesses to the buffer are called *successes*; the relative frequency of successes as a function of capacity is given by the *success function* $F(C)$. For a given capacity C, page trace $X = x_1, x_2, \cdots x_L$, replacement algorithm, and arbitrary time t (where $1 \leq t \leq L$), the set of pages in the buffer just after the completed reference to x_t is denoted by $B_t(C)$. The initial buffer contents is represented by $B_0(C)$. By convention

$$B_0(C) = \phi$$

for all C where ϕ is the empty set. The set of distinct pages referenced in $x_1, x_2, \cdots, x_t$ is denoted by Γ_t, and the number of pages in Γ_t is denoted by (working set)

$$\gamma_t = |\Gamma_t|$$

Demand paging in the two-level hierarchy is formally defined by the following requirements, wherein the operator "+" denotes the union of disjoint sets:

1. If $x_t \in B_{t-1}(C)$ then $B_t(C) = B_{t-1}(C)$
2. If $x_t \notin B_{t-1}(C)$ and $|B_{t-1}(C)| < C$ then
   $B_t(C) = B_{t-1}(C) + \{x_t\}$
3. If $x_t \notin B_{t-1}(C)$ and $|B_{t-1}(C)| = C$ then
   $B_t(C) = B_{t-1}(C) - \{y_t\} + \{x_t\}$

where $y_t \in B_{t-1}(C)$ is determined by the replacement algorithm. Under demand paging, a buffer of capacity C simply fills as required by 1 and 2, while the first C distinct pages are referenced. Subsequently, referenced pages are swapped in, as required by 1 and 3.

Least recently used replacement

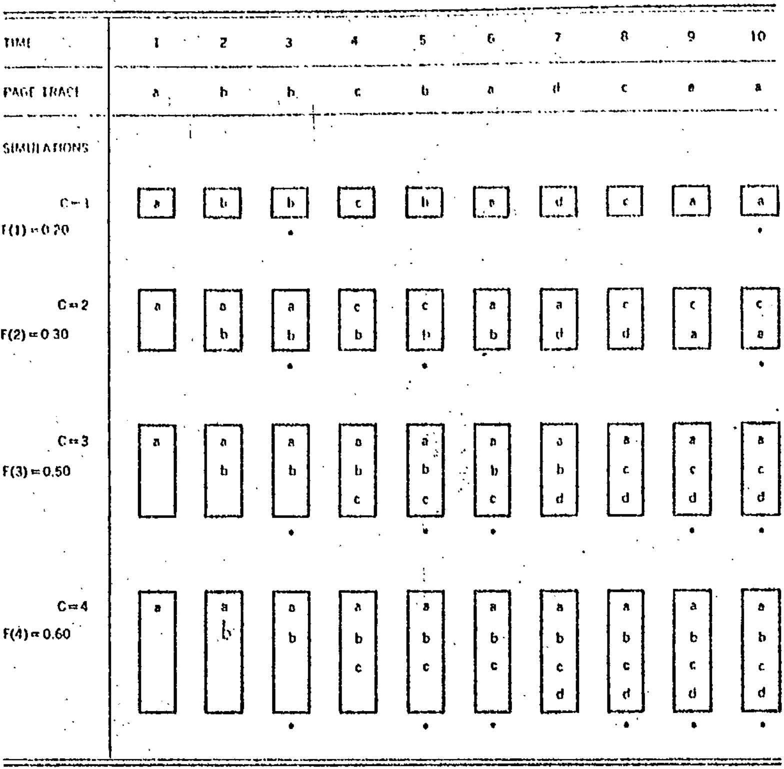
We now consider a particular replacement algorithm called "least recently used" (LRU), and show that the entire success function can be obtained by stack processing in a single pass of the address trace. Briefly, the single-pass technique requires the maintaining of a list of pages, called an LRU stack, and measuring a distance on this stack for every page reference. Frequencies of these stack distances are used to calculate the success function. The existence of the LRU stack follows from an inclusion property satisfied by LRU replacement, whereas the use of distance frequencies hinges on the related concept of critical capacity.

success function

Under LRU, the page selected for replacement is the one that has not been referenced for the longest time (i.e., the least recently used page). One way to obtain the success function for a given trace is to simulate the two-level hierarchy system for each buffer capacity. Such a simulation determines the buffer contents at every time t, and counts the number of times the current reference x_t is found in the buffer. In Figure 4, we show an example of this simulation procedure for a given page trace and buffer capacities $C = 1, 2, 3, 4$. Pages are denoted by lower-case letters, and page successes are marked by asterisks.

A greatly simplified method for obtaining the success function under LRU replacement can be derived from certain properties of that replacement algorithm. For any page trace and buffer capacity C, the buffer is initially empty, and in say τ time units, it fills up with the first C distinct pages referenced by the trace. At time τ, the buffer contains the C pages most recently referenced through time τ. When a new page is referenced at a later time ($t > \tau$), this page replaces the least recently used page in the buffer.

Figure 4 Determining success function by buffer simulation



TIME	1	2	3	4	5	6	7	8	9	10
PAGE TRACE	a	b	b	c	b	a	d	c	a	a
SIMULATIONS										
C=1 F(1)=0.20	a	b	b ●	c	b	a	d	c	a	a ●
C=2 F(2)=0.30	a	a b	a b ●	c b	c b ●	a b	a d	c d	c a	c a ●
C=3 F(3)=0.50	a	a b	a b ●	a b c	a b c ●	a b c ●	a b d	a c d	a c d ●	a c d ●
C=4 F(4)=0.60	a	a b	a b ●	a b c	a b c ●	a b c ●	a b c d	a b c d ●	a b c d ●	a b c d ●

Thus at time t, the buffer still contains the C most recently referenced pages. It is easy to see that under LRU the buffer contains the C most recently referenced pages for all subsequent times, and that this property holds for all page traces and buffer capacities. One can generate the buffer contents $B_t(C)$ for any time t on a trace and any capacity by scanning backward from point t and collecting the first C distinct pages encountered.

Since the set of C most recently referenced pages is always contained in the set of $C + 1$ most recently referenced pages, the buffer contents $B_t(C)$ at any time must be a subset of $B_t(C + 1)$. In fact, $B_t(C)$ is a proper subset of $B_t(C + 1)$ if at least $C + 1$ distinct pages have been referenced through time t. More formally, under LRU replacement, the buffer contents for any page trace $X = x_1, x_2, \cdots, x_L$ and any time t (where $1 \leq t \leq L$) satisfy the following *inclusion property*:

$$B_t(1) \subset B_t(2) \subset \cdots \subset B_t(\gamma_t) = B_t(\gamma_t + 1) = \cdots \qquad (1)$$

where

$$|B_t(C)| = C \qquad \text{for } 1 \leq C \leq \gamma_t$$

and

$$|B_t(C)| = \gamma_t \qquad \text{for}$$

The inclusion prope
$t = 5$, for example

$B_t(1) = \{b\}$

$B_t(2) = \{c, b\}$

$B_t(3) = \{a, b, c\}$

and

$B_t(4) = \{a, b, c\}$

Because of the inclu
and for all capacities
and useful way. We
$s_t(2), \cdots s_t(\gamma_t)$, where

$s_t(i) = B_t(i) - B_t(i$

Hence

$$B_t(C) = \begin{cases} \{s_t(1), s_t(2 \\ \{s_t(1), s_t(2 \end{cases}$$

The list S_t is referre
entry and $s_t(\gamma_t)$ as the
for $t = 5$ in Figure 4

$S_5 = [b, c, a]$

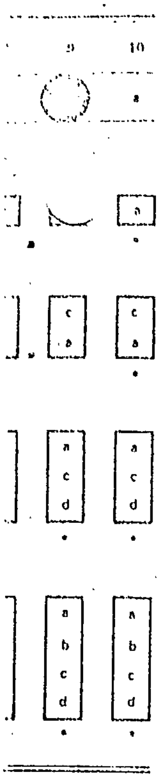
The stack S_0 at time
null stack, that is,
LRU stacks correspo

Besides representing
stack can be used
$F(C)$. Let us suppos
referenced and thus
$1 \leq C \leq \gamma_{t-1}$. Let

$x_t \in B_{t-1}(C)$

We call C_t the *critic*
given in Equation 1,
not been previously
contained in a buffer

From the definition
that C_t is simply the



ly r...nced
ntains the C
es, ... that
pac... One
t on a trace
nd collecting

ys contained
, the buffer
- 1). In fact,
+ 1 distinct
nally, under
trace X =
isfy the fol-

(1)

and

$$|B_t(C)| = \gamma_t \qquad \text{for } C \geq \gamma_t$$

The inclusion property can be observed in Figure 4 where at time $t = 5$, for example

$$B_t(1) = \{b\}$$

$$B_t(2) = \{c, b\}$$

$$B_t(3) = \{a, b, c\}$$

and

$$B_t(4) = \{a, b, c\}$$

Because of the inclusion property, the buffer contents at any time and for all capacities can be represented in the following compact and useful way. We order the set of pages Γ_t into a list $S_t = s_t(1)$, $s_t(2), \cdots s_t(\gamma_t)$, where

$$s_t(i) = B_t(i) - B_t(i - 1) \qquad \text{for } i = 1, 2, \cdots, \gamma_t \tag{2}$$

Hence

$$B_t(C) = \begin{cases} \{s_t(1), s_t(2), \cdots, s_t(C)\} & \text{for } C \leq \gamma_t \\ \{s_t(1), s_t(2), \cdots, s_t(\gamma_t)\} & \text{for } C \geq \gamma_t \end{cases} \tag{3}$$

The list S_t is referred to as the *LRU stack*, with $s_t(1)$ as the top entry and $s_t(\gamma_t)$ as the bottom entry. As an example, the LRU stack for $t = 5$ in Figure 4 is

$$S_5 = [b, c, a]$$

The stack S_0 at time $t = 0$ has no entries and is therefore called a null stack, that is, one with no entries. The entire sequence of LRU stacks corresponding to Figure 4 is included in Figure 5.

Besides representing the buffer contents for all capacities, the LRU stack can be used to efficiently determine the success function $F(C)$. Let us suppose that at time t, page x_t has been previously referenced and thus is a member of at least one set $B_{t-1}(C)$, where $1 \leq C \leq \gamma_{t-1}$. Let C_t denote the least buffer capacity such that

$$x_t \in B_{t-1}(C)$$

We call C_t the *critical capacity* since, from the inclusion property given in Equation 1, $x_t \in B_{t-1}(C)$ if and only if $C \geq C_t$. If x_t has not been previously referenced, we set $C_t = \infty$ because x_t is not contained in a buffer of any finite capacity.

From the definition of LRU stacks in Equation 2, it may be seen that C_t is simply the position of page x_t in the stack S_{t-1}, so that

Figure 5 Sequence of LRU stacks

TIME	1	2	3	4	5	6	7	8	9	10
PAGE TRACE	a	b	b	c	b	a	d	c	a	a
LRU STACK	a	b a	b a	c b a	b c a	a b c	d a b c	c d a b	a c d b	a c d b
STACK DISTANCE	∞	∞	1	∞	2	3	∞	4	3	1
DISTANCE COUNTERS $n(\Delta)$										
1	0	0	1	1	1	1	1	1	1	②
2	0	0	0	0	1	1	1	1	1	①
3	0	0	0	0	0	1	1	1	2	②
4	0	0	0	0	0	0	0	1	1	①
∞	1	2	2	3	3	3	4	4	4	④

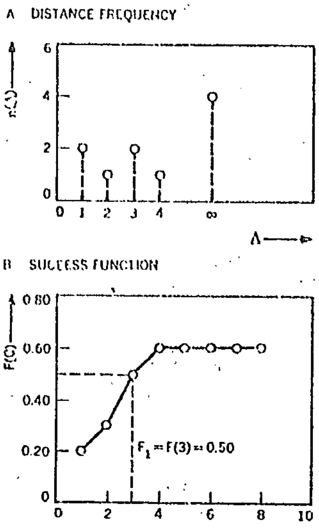


Figure 6 Obtaining success function from distance frequencies

$x_t = s_{t-1}(C_t)$

We call this page position the *stack distance* Δ_t, since Δ_t is essentially the "distance" from the top of the stack to

$x_t = s_{t-1}(\Delta_t)$

(Note that here $\Delta_t = C_t$. When constrained mapping functions are considered, the stack distance may not always equal the critical capacity.) If x_t has not been previously referenced, then Δ_t is set to infinity. The sequence of stack distances for our example is included in Figure 5.

The significance of stack distances is that they lead directly to the success function. To see this, let $n(\Delta)$ be the number of times the stack distance Δ is observed in processing a trace. Since the stack distance equals the critical capacity, the number of times that the referenced page is found in the buffer is

$$N(C) = \sum_{\Delta=1}^{C} n(\Delta) \tag{4}$$

and the success function is given by the expression

$$F(C) = N(C)/L \tag{5}$$

In practice, the set $\{n(\Delta)\}$ can be determined from a set of distance counters, as shown in Figure 5. All counters are set initially to zero, and the counter for each distance Δ is incremented whenever that distance occurs. For k-bit page numbers, we need at most $2^k + 1$ counters, corresponding to $1 \le \Delta \le 2^k$ and $\Delta = \infty$. At the conclusion of a page trace, the final values of the distance counters are the values $\{n(\Delta)\}$, and $F(C)$ is obtained from Equations 4 and 5.

numerical example

We now calculate the value of the success function in a numerical example. For Δ's of 1, 2, 3, 4, and ∞, the corresponding final counter values in Figure 5 are 2, 1, 2, 1, and 4. This distribution is shown in Figure 6A. Dividing by L equals 10 in Figure 5, and summing cumulatively, we obtain the success function shown in Figure 6B. One can verify that the $F(C)$ values for the curve in Figure 6B agree with those obtained in the simulations of Figure 4.

To find the access frequencies F_1 and F_2, for a given buffer capacity C, we take $F_1 = F(C_1)$ and $F_2 = 1 - F_1$. As an example, for $C = 3$ pages, $F_1 = F(3) = 0.50$ as indicated in Figure 6B, $F_2 = 1 - 0.50 = 0.50$, and the average access time $\bar{T}$ of the hierarchy is $0.50T_1 + 0.50T_2$.

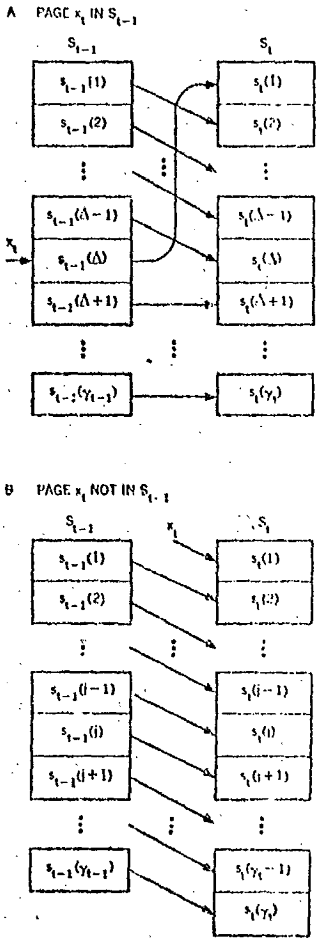
Note that $F(C)$ is always a monotonic, non-decreasing function of C for LRU replacement, since $F(C)$ is obtained by cumulative summation as shown in Equation 4. Also, $F(C)$ never exceeds $(L - \gamma_L)/L$ for any capacity, because all pages initially reside in the backing store.

To avoid constructing each LRU stack separately, we now give an iterative construction of S_t from S_{t-1} and x_t. Observe that at every time t, the stack S_t is simply the list of pages in Γ_t, according to their most recent reference. The most recently referenced page is $s_t(1)$ since $s_t(1) = x_t$. The second most recently referenced page is $s_t(2)$, and $s_t(\gamma_t)$ is the least recently referenced page in Γ_t.

Let us suppose that page x_t has been previously referenced and appears at position Δ on stack S_{t-1}. For time t, we know that x_t must be the top entry in S_t, because it is the most recently referenced page. Consider now a page b at some position j on S_{t-1} where $1 \le j < \Delta$. At time $t - 1$, page b is the jth most recently referenced page, and the intervening pages do not include x_t. At time t, page x_t is added to this set so that page b must now be at position $j + 1$ on stack S_t. If j is greater than Δ, page b must remain at position j at time t, since the set of more recently referenced pages is unchanged from time $t - 1$.

The net effect of this page motion is shown in Figure 7A. Page x_t is moved to the top of the stack, pages previously above x_t are down-shifted one position, and all other pages retain the same position. If x_t were not previously referenced, x_t would be placed on the top and all other pages would be down-shifted one position as shown in Figure 7B.

Figure 7 Constructing LRU stacks



This iterative procedure can be used to generate the sequence of stacks in Figure 5. In an actual evaluation, it is not necessary to store the entire sequence of stacks. Rather, only the current stack must be maintained as the trace is scanned. When a page reference occurs, that page is put on the top of the stack, and entries in the stack are down-shifted one-by-one starting from the top. If page x_t is encountered, its distance Δ_t is recorded, and x_t is erased because it has already been placed on top. The position vacated by x_t is filled by the page downshifted from position $\Delta_t - 1$. If x_t is not encountered, then the downshifting proceeds to the bottom of the stack, and distance $\Delta_t = \infty$ is recorded.

Stack algorithms

We now examine the general class of replacement algorithms that satisfy the inclusion property. Such algorithms are called "stack algorithms." It is shown that stacks can be iteratively maintained for any stack algorithm, and that stack distance frequencies for a given trace can be used to obtain the corresponding success function. The main problems considered are (1) to efficiently generate stacks $\{S_t\}$ for an arbitrary stack algorithm, and (2) to identify those algorithms that are stack algorithms. Several examples of stack algorithms are described, along with one replacement algorithm that is not a stack algorithm.

A replacement algorithm is called a *stack algorithm* if the buffer contents in a demand-paged, two-level hierarchy satisfy the inclusion property given in Equation 1, for every page trace and every point in time. As shown for LRU replacement, a stack can be defined according to Equation 2 in such a way that the buffer contents for all capacities are given by Equation 3. Furthermore, since the stack distance Δ_t is a critical capacity, the success function for any page trace can be obtained by summing the stack distance frequencies $\{n(\Delta)\}$ according to Equation 4. This summation implies that the success function is a monotonic and nondecreasing function of the capacity C for every stack algorithm.

stack generation

Let us now consider a replacement algorithm R as a collection of mappings

$$R_C : B_{t-1}(C) \to y_t(C) \qquad \text{where } y_t(C) \in B_{t-1}(C)$$

is the page replaced by x_t in a buffer of capacity C. From the constraints of demand paging, we know that R is applied only when the following conditions are true: $x_t \notin B_{t-1}(C)$ and $|B_{t-1}(C)| = C$. If the inclusion property is satisfied up to and including time $t - 1$, then R must satisfy certain restrictions at time t to maintain the inclusion property. Specifically, if a replacement is required for some capacity $C + 1$ (and therefore for C), then $y_t(C + 1)$ must be either $y_t(C)$ or $s_{t-1}(C + 1)$. To prove this, let us assume the following:

$$B_{t-1}(C) \subset B_{t-1}(C+1)$$

$$|B_{t-1}(C)| = C$$

$$|B_{t-1}(C+1)| = C+1$$

and

$$x_t \notin B_{t-1}(C+1)$$

Note that from Equation 2, page $s_{t-1}(C+1)$ is contained in $B_{t-1}(C+1)$ but not in $B_{t-1}(C)$. If page $y_t(C+1)$ is neither $s_{t-1}(C+1)$ nor $y_t(C)$, then $y_t(C+1)$ is some other page $z \in B_{t-1}(C)$. However, page z is included in $B_t(C)$, but not in $B_t(C+1)$, which would violate the inclusion property.

We have given a necessary condition for stack algorithms. The same condition is also sufficient, because if $y_t(C+1)$ is either $y_t(C)$ or $s_{t-1}(C+1)$, then $B_t(C)$ is a subset of $B_t(C+1)$. Therefore, we conclude that a replacement algorithm is a stack algorithm if and only if for every time t

$$y_t(C+1) = s_{t-1}(C+1) \qquad \text{or} \qquad y_t(C+1) = y_t(C) \tag{6}$$

for

$$1 \leq C < \gamma_{t-1} \qquad \text{and} \qquad C+1 < \Delta_t$$

stack algorithm identification

Important replacement algorithms that satisfy Equation 6 are those that induce a total ordering on all previously referenced pages and use this ordering to make replacement decisions. The ordering can be represented in the form of a *priority list*

$$P_t = p_t(1), p_t(2), \cdots, p_t(\gamma_{t-1})$$

where $p_t(i)$ has a higher priority than $p_t(i+1)$ for $1 \leq i < \gamma_{t-1}$. The algorithm then selects for replacement the page in $B_{t-1}(C)$ that has the lowest priority.

A convenient notation for working with priorities is $\min(A)$, where A is an arbitrary set of pages in Γ_{t-1}, and $\min(A)$ is the unique page in A having lowest priority on the list P_t. If $B_{t-1}(C) \subset B_{t-1}(C+1)$ and $x_t \notin B_{t-1}(C+1)$, we can express the replaced pages $y_t(C)$ and $y_t(C+1)$ as follow:

$$y_t(C) = \min\,[B_{t-1}(C)] \tag{7}$$

and

$$\begin{aligned} y_t(C+1) &= \min\,[B_{t-1}(C+1)] && (8)\\ &= \min\,[B_{t-1}(C), s_{t-1}(C+1)] && (9)\\ &= \min\{\min\,[B_{t-1}(C)], s_{t-1}(C+1)\} && (10)\\ &= \min\,[y_t(C), s_{t-1}(C+1)] && (11) \end{aligned}$$

Equations 7–9 are based on the definition of the replacement algorithm, whereas Equation 10 is based on the properties of minimization.

We conclude from Equation 11 that any replacement algorithm that induces a priority list P_t for every time t satisfies Equation 6 and is therefore a stack algorithm. For example, the priority list for LRU is just the ordering of pages in Γ_{t-1} by most recent reference. The priority list for "least frequently used" (LFU) replacement is the ordering of referenced pages by most frequent reference together with a scheme to break ties.

stack updating

Before describing other examples of stack algorithms, let us develop a stack updating procedure for algorithms inducing a priority list. For any page trace $X = x_1, x_2, \cdots, x_L$ and any time t, where $1 \leq t \leq L$, suppose that stack S_{t-1} is available. Also, for any two pages $a, b \in \Gamma_{t-1}$, let max (a, b) denote the page having higher priority. If x_t has been previously referenced and appears at position Δ_t on stack S_{t-1}, the stack at time t is given by

$$s_t(1) = x_t \qquad (12)$$

$$s_t(i) = \max\,[y_t(i-1), s_{t-1}(i)] \qquad \text{for } 2 \leq i < \Delta_t \qquad (13)$$

$$s_t(\Delta_t) = y_t(\Delta_t - 1) \qquad (14)$$

$$s_t(i) = s_{t-1}(i) \qquad \text{for } \Delta_t < i \leq \gamma_{t-1} \qquad (15)$$

Equations 12, 14, and 15 are based on the constraints of demand paging, whereas Equation 13 is derived from Equation 11.

If x_t has not been previously referenced, the defining equations for stack S_t are the following:

$$s_t(1) = x_t \qquad (16)$$

$$s_t(i) = \max\,[y_t(i-1), s_{t-1}(i)] \qquad \text{for } 2 \leq i \leq \gamma_{t-1} \qquad (17)$$

$$s_t(\gamma_t) = y_t(\gamma_{t-1}) \qquad (18)$$

In this case, Equations 16 and 17 express the fact that replacements are required for all buffer capacities in the range $1 \leq C \leq \gamma_{t-1}$. Equation 18 corresponds to the new page x_t being added to the stack, with the result that a buffer of capacity

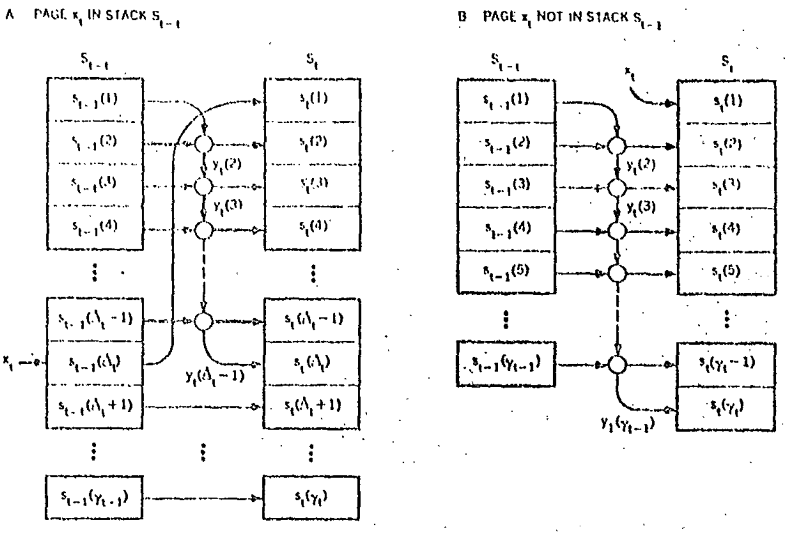
$$\gamma_t = \gamma_{t-1} + 1$$

is now full.

Figure 8 illustrates the stack updating procedure as given in Equations 12–18. The top entry $s_t(1)$ is always x_t, and the first page replaced is

$$y_t(1) = s_{t-1}(1) \qquad \text{for } \Delta_t > 1$$

Figure 8 Stack updating



A PAGE x_t IN STACK S_{t-1}

B PAGE x_t NOT IN STACK S_{t-1}

Each subsequent entry $s_t(i)$ is then determined iteratively from $s_{t-1}(i)$ and $y_t(i-1)$ according to Equation 13 or 17. If x_t is found on stack S_{t-1} as shown in Figure 8A, we use Equation 14 to determine $s_t(\Delta_t)$. All lower entries are unchanged from time $t - 1$. If x_t is not found on stack S_{t-1}, as shown in Figure 8B, then $\Delta_t = \infty$, and we use Equation 18. In either case, the replacement algorithm does not have to be applied to all the pages for stack updating. Only a sequence of pairwise decisions between pages $s_{t-1}(i)$ and $y_t(i-1)$ is required.

Comparing our stack updating procedure with the one for LRU shown in Figure 7, we see that page $y_t(C)$ under LRU is always $s_{t-1}(C)$. In fact, the priority list P_t is exactly equal to stack S_{t-1}, since both lists give the order of pages in Γ_{t-1} by most recent reference. Thus

$$y_t(C) = s_{t-1}(C)$$

and Equations 13 and 17 then reduce to

$$\begin{aligned} s_t(i) &= \max[s_{t-1}(i-1), s_{t-1}(i)] \\ &= s_{t-1}(i-1) \end{aligned}$$

For an arbitrary stack algorithm, the stack updating is more complex than for LRU, and the order of stack elements at time $t - 1$ may be very different from that at time t.

examples of stack algorithms

Let us now examine several examples of stack algorithms. In general any replacement algorithm that bases its decisions on some page usage quantity, whether measured or predicted, naturally induces a priority list and is, therefore, a stack algorithm. One example, of course, is LRU, and another example previously mentioned is least frequently used (LFU) replacement.

Under LFU, the page replaced from a buffer at time t is that page that has been referenced the fewest number of times over the interval $1 \leq \tau \leq t$, or perhaps over some "backward window" interval $t - h \leq \tau \leq t$, where $0 < h \leq t$. If two or more pages are tied for least frequency of use, then some arbitrary rule is used to break the tie. As long as the rule is consistent for all pages and all capacities (e.g., if the tied pages are numerically ordered) a priority list P_t is induced, and LFU is a stack algorithm.

Other examples of stack algorithms may arise in analytical studies of program behavior. If an address trace is generated from some random process, it may be desirable to study the behavior of replacement algorithms that base their decisions on the parameters of the random process. One such process is a time-invariant, first-order Markov chain,[15,16] where any page c is referenced immediately after page b with a fixed transition probability π_{bc}. The process is completely described by the matrix $\Pi = \{\pi_{bc}\}$, (where b and c range over all referenced pages) and by the page referenced at time $t = 1$.

One possible replacement algorithm is to remove the page least likely to be referenced next. We call this strategy "least transition probability" (LTP) since, for page x_t equal to page b, the page c chosen for removal is the one that minimizes π_{bc} over those pages in the buffer. Supplying an appropriate rule for breaking ties, we see that LTP induces a priority list and is a stack algorithm.

Another replacement algorithm is to remove the page with the largest expected time until next reference. We call this strategy LNR for "longest next reference." The expected times until next reference can be obtained from the Π-matrix by standard techniques.[17] As with LTP, LNR induces a priority list if we supply an appropriate tie-breaking rule.

To analyze an actual program trace under LTP or LNR (perhaps for testing a Markov model of the program), page reference statistics may be used to estimate the matrix Π. For example, the observed transition frequencies over some interval $t - h$ to t can be used to generate a time-varying estimator matrix $\hat{\Pi}_t$. A priority list P_t can then be constructed for each time t, according to the probabilities in $\hat{\Pi}_t$, with the result that the overall strategy for replacement remains a stack algorithm.

Other stack algorithms may base their decisions on information from the programmer or compiler, or on properties of the computer system. For example, the programmer or compiler may supply to the system[14] special "program directives" that indicate which pages should be given high priorities in the immediate future. Another case is where the operating system assigns priorities to program pages in a multiprogrammed system, based perhaps on the position of the program in a task queue. If all the pages in the address space can be ordered in a priority list P_t for each time t, the resulting replacement algorithm is a stack algorithm.

first-in/ first-out

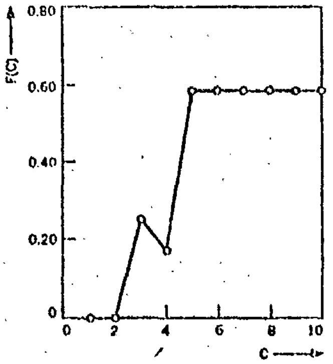
In the examples given, we see that priority lists can arise in a variety of ways. We now consider a replacement algorithm called "first-in/first-out" (FIFO) that is not a stack algorithm. Under FIFO, the page that has remained in the buffer for the longest (continuous) time up to time t is removed.

A peculiarity of FIFO is illustrated by the following page trace

$X = a\ b\ c\ d\ a\ b\ e\ a\ b\ c\ d\ e$

As shown in Reference 18, the success function for this trace is not monotonic, and takes the form shown in Figure 9. Since stack algorithms have monotonic success functions, we conclude that FIFO is not a stack algorithm and does not induce a priority list P_t at every time t. In amplifying this conclusion, we note that the relative priorities between pages in Γ_{t-1} may depend on the buffer capacity C. Thus in the example, one can verify that page d has lowest priority of all pages in $B_6(3)$ in the sense that d has been in the buffer longest. However, page d has highest priority in $B_6(4)$, since it was brought into the buffer latest.

Figure 9 Success function for FIFO replacement



Whenever the priorities among pages depend on the capacity of the buffer, we cannot define a single priority list that applies to every capacity. One instance of this is when priorities depend on the frequency of reference to pages after their entering the buffer. Another case is when priorities depend on total time spent in the buffer.

As long as priorities are independent of capacity, and as long as one can order the referenced pages to reflect these priorities, then stack-processing techniques can be used to find the success function.

An optimum replacement algorithm

We now discuss a replacement algorithm that yields the maximum value for the success frequency over the space of all replacement algorithms—for every page trace and every buffer capacity. Such an algorithm is said to be an *optimum replacement algorithm*. Belady[13] describes an optimum replacement algorithm called MIN, and shows how to evaluate the success frequency for a given page trace and a given buffer capacity. In the following discussion, we describe a stack algorithm called OPT and prove that it is also an optimum replacement algorithm. Using certain properties of LRU and OPT, the entire success function for OPT can be determined in two passes of a page trace.

OPT

The replacement algorithm OPT has the following characteristics. Whenever a page must be pushed from the buffer, the chosen page is the one whose next reference is farthest in the future. If a tie results because two or more buffer pages are never referenced again, the tie is broken by an arbitrary rule Ω that pushes the page with the latest alphabetical or numerical order. An example of OPT replacement is shown in Figure 10, for the buffer capacity $C = 3$. As an illustration, notice that at time $t = 5$ page c is pushed from the buffer, since the other buffer pages a and b are referenced sooner. At time $t = 9$, page b is pushed from the buffer, because page d is referenced again (at time $t = 10$), and page a has priority over page b by our rule Ω.

Figure 10 Example of OPT replacement

TIME	1	2	3	4	5	6	7	8	9	10
PAGE TRACE	a	b	c	a	d	b	a	d	c	d
BUFFER CONTENTS FOR C=3	a	a	a	a	a	a	a	a	a	a
		b	b	b	b	b	b	b	c	c
			c	c	d	d	d	d	d	d
				•		•	•	•		•

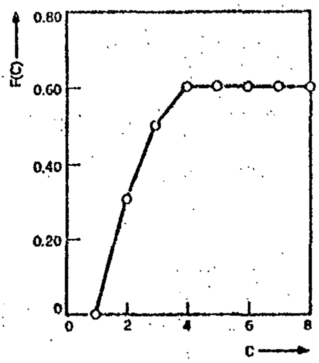
A formal proof that OPT is an optimal replacement algorithm is given in the Appendix. We note here that OPT is not realizable in an actual computer system because it requires knowledge of future page references. However, OPT does serve as a useful benchmark for any replacement algorithm, including stack-type algorithms. To show that OPT is a stack algorithm, observe that a priority list P_t can be constructed for OPT at each time t. Specifically, P_t is the list of the pages referenced again, ordered by their time of next reference, followed by the list of the pages not referenced again, as ordered by the tie-breaking rule Ω.

stack processing example

The stack processing technique for OPT is illustrated in Figure 11. Priority lists are ordered as described above, and curly brackets denote the pages ordered under the rule Ω. For example, at time $t = 8$ the priority list is $P_8 = c, d, a, b$, because c is the next page

Figure 11 Stack processing and success function for OPT replacement

TIME	1	2	3	4	5	6	7	8	9	10
PAGE TRACE	a	b	c	a	d	b	a	d	c	d
PRIORITY LIST	a	a	a	b	b	a	d	c	d	{a
		b	b	a	a	d	c	d	{a	b
			c	c	d	{c	{a	{a	b	c
					c	b	b	b	c	d
OPT STACK	a	b	c	a	d	b	a	d	c	d
		a	a	c	a	a	b	a	d	c
			b	b	b	d	d	b	a	a
					c	c	c	c	b	b
STACK DISTANCE	∞	∞	∞	2	∞	3	2	3	4	2



referenced (at $t = 9$) and d is the second page referenced (at $t = 10$). Pages a and b are not referenced again, and thus are ordered according to rule Ω. The sequence of OPT stacks is constructed using the priority lists, and the success function is obtained from the stack distance frequencies. A major difficulty with the technique is the amount of forward scanning required to construct the priority lists.

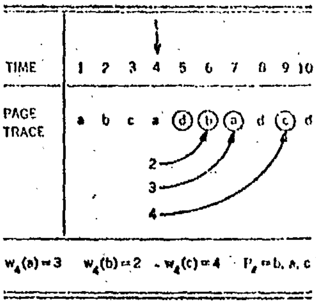
forward distance

Fortunately, a more efficient procedure exists for obtaining the priority lists. For a given page trace X, we define the *forward distance* $w_t(a)$ to a page a at time t as the number of distinct pages referenced in $x_{t+1}, \cdots, x_{t'}$, (where $x_{t'}$ is the first reference to page a after time t). If page a is not referenced again, the forward distance is defined as infinity. Note that the priority list under OPT is a listing of the pages in Γ_{t-1} according to their increasing forward distances. An illustrative example of forward distance determination is given in Figure 12.

If the forward distances to all pages in Γ_{t-1} are known at time $t - 1$, the new forward distances at time t can be determined iteratively from the single forward distance $w_t(x_t)$. Specifically, for page $a \neq x_t$ and $w_t \triangleq w_t(x_t)$, we have

$$w_t(a) = \begin{cases} w_{t-1}(a) - 1 & \text{for} \quad w_{t-1}(a) \leq w_t \quad \text{and} \quad w_{t-1}(a) \neq \infty \\ w_{t-1}(a) & \text{for} \quad w_{t-1}(a) > w_t \quad \text{or} \quad w_{t-1}(a) = \infty \end{cases} \tag{19}$$

Figure 12 Determination of forward distances at time $t = 4$



To determine the sequence of forward distances $\{w_t\}$ for a page trace X, consider the *reverse trace* $X^R = x_L, x_{L-1}, \cdots, x_2, x_1$. Suppose that X^R is analyzed according to LRU replacement and that x_i and x_j denote two successive references to page a in the reverse trace. Thus $X^R = x_L, \cdots, x_i = a, \cdots, x_j = a, \cdots, x_1$. At time j, the stack distance Δ_j is the number of distinct pages referenced in $x_i, \cdots, x_{j+1}$. (Note that x_{j+1} precedes x_j in X^R.) However, this number of distinct pages is precisely the forward distance w_j for page trace X. Thus the sequence of LRU stack distances for trace X^R, namely, $\Delta_L, \Delta_{L-1}, \cdots, \Delta_2, \Delta_1$, is the reverse of the sequence of forward distances $w_1, w_2, \cdots, w_{L-1}, w_L$ for trace X.

maximum success function

These results form the basis of a two-pass stack processing technique for determining the success function for OPT replacement. The technique is illustrated by Figure 13. The first pass is a backward scan of the page trace X using LRU replacement, denoted by the left-pointing arrow. The LRU stack distances are stored, in reverse order, on a "distance tape." The second pass is a forward scan using OPT replacement, as shown by the right-pointing arrow. Forward distances read from the distance tape are used to maintain the OPT priority lists according to Equation 19.

The LRU stack distances gathered from the reverse page trace yield important information about the forward page trace. Specifically,

Figure 13 Two-pass technique for LRU and OPT replacement

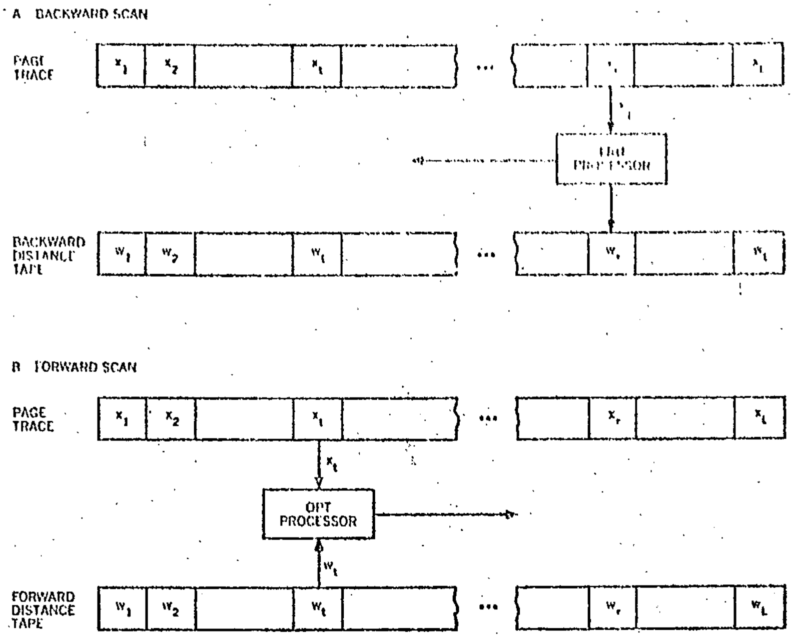
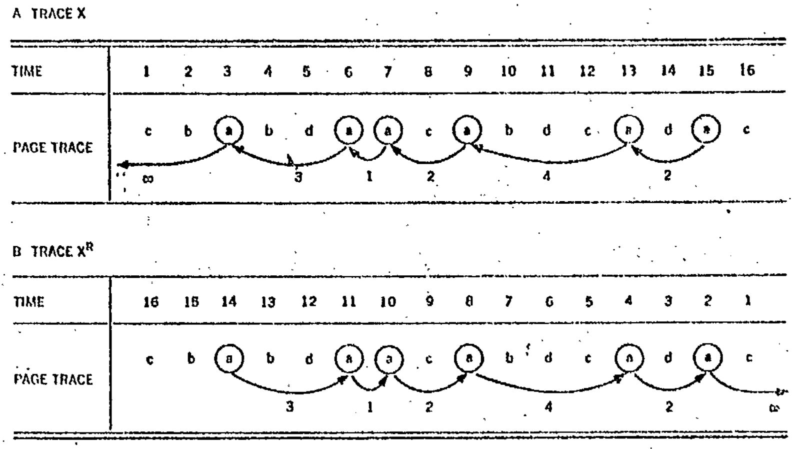


Figure 14 Sequence of LRU distances for page a



we claim that the success function for the reverse trace X^R under LRU replacement is equal to the success function for the forward trace X under LRU replacement. Thus one can use the backward scan of X, not only to generate the distance tape for OPT, but also to generate the success function for LRU.

To prove this result, let $F_{\text{LRU}}(C, X)$ denote the LRU success function for trace X, and consider the set of LRU stack distances measured for a given page a in X and X^R. As the example in Figure 14 illustrates, these sets are always identical. Since this holds for every

distinct pa
identical.
are equal.

Another re
is equal to i
for trace X.
forward-ba
During the
the distance
function for

Random r

In the stack
tion is asso
techniques
that does n
if the buffer
ment with
perform a
obtain a RA
yield a set
sample succ
success func

A question
generate a s
that bases
variable. W
there certai
property fai

Our approa
stack algori
each capaci
time t, the
of pages in
likely to be
induces a p

To establis
that a repla
Since $y_t(C)$
bility that a

One difficul
priority list
actually con

distinct page in the trace, the distance frequencies for X and X^R are identical, so that the success functions $F_{\text{LRU}}(C, X^R)$ and $F_{\text{LRU}}(C, X)$ are equal.

Another result, which is proved in the Appendix, is that $F_{\text{OPT}}(C, X)$ is equal to $F_{\text{OPT}}(C, X^R)$, where $F_{\text{OPT}}(C, X)$ is the OPT success function for trace X. Thus, our two-pass technique can be implemented with forward-backward scans as well as with backward-forward scans. During the first scan, the success function for LRU is obtained, and the distance tape generated. During the second scan the success function for OPT is obtained.

Random replacement

In the stack algorithms considered thus far, a unique success function is associated with each trace. We now extend stack-processing techniques to cover a "random replacement" algorithm (RAND) that does not always yield a unique success function. With RAND, if the buffer has a capacity of C, any given page is chosen for replacement with a probability of $1/C$. In analyzing RAND, one might perform a Monte Carlo simulation for each buffer capacity to obtain a RAND success function. Repeating these simulations would yield a set of sample success functions to characterize RAND. The sample success functions could then be used to estimate an "average" success function.

A question that arises is whether stack processing can be used to generate a sample success function for RAND or any other algorithm that bases a replacement choice on the value of some random variable. We observe that RAND is not a stack algorithm, because there certainly exists a trace and a time t for which the inclusion property fails to hold with a nonzero probability.

Our approach is to define a replacement algorithm RR, which is a stack algorithm having the same statistical properties as RAND for each capacity C. The algorithm RR is defined as follows: at each time t, the priority list P_t is obtained by randomly ordering the set of pages in Γ_{t-1} (each of the $\gamma_{t-1}!$ possible orderings is equally likely to be chosen). Observe that RR is a stack algorithm, since it induces a priority list.

To establish that RR is statistically equivalent to RAND, assume that a replacement is necessary in a buffer of capacity C at time t. Since $y_t(C) = \min\,[B_{t-1}(C)]$, and P_t is randomly chosen, the probability that any given page is $y_t(C)$ is $1/C$—the same as for RAND.

One difficulty in implementing RR is the generation of the random priority list P_t. Fortunately, it is possible to update the stack without actually constructing the entire priority list. Assuming that $\Delta_t > j$, let $q_t(t)$ denote the probability that page $s_{t-1}(j)$ has priority over page $y_t(j-1)$ at time t. If $s_{t-1}(j)$ does not have priority over $y_t(j-1)$, we know that $s_{t-1}(j) = \min [B_{t-1}(j)]$. Since this occurs with probability $1/j$, we obtain

$$1 - q_t(t) = 1/j$$

or

$$q_t(t) = (j-1)/j \tag{20}$$

Using Equation 20, the stack can be updated at time t for RR replacement by choosing page $s_t(j) = s_{t-1}(j)$ with probability $(j-1)/j$, for $2 \leq j < \Delta_t$ and $j < \gamma_{t-1}$. As a check, let us compute the probability Q that an arbitrary page b is pushed from a buffer of capacity C at time t. Assuming that page b occurs at some position k on stack S_{t-1} where $1 \leq k \leq C$, then Q is given by the following expression:

$$\begin{aligned} Q &= P_r\{y_t(C) = b\} \\ &= P_r\{s_t(k) = y_t(k-1),\ s_t(k+1) = s_{t-1}(k+1), \\ &\qquad s_t(k+2) = s_{t-1}(k+2),\ \cdots,\ s_t(C) = s_{t-1}(C)\} \end{aligned} \tag{21}$$

The events in the joint probability in Equation 21 are independent, so that we obtain

$$\begin{aligned} Q &= P_r\{s_t(k) = y_t(k-1)\} \cdot P_r\{s_t(k+1) = s_{t-1}(k+1)\} \\ &\qquad \cdot P_r\{s_t(k+2) = s_{t-1}(k+2)\} \cdot \cdots \cdot P_r\{s_t(C) = s_{t-1}(C)\} \\ &= \left(\frac{1}{k}\right)\left(\frac{k}{k+1}\right)\left(\frac{k+1}{k+2}\right) \cdots \left(\frac{C-1}{C}\right) \\ &= \frac{1}{C} \end{aligned}$$

Since $Q = 1/C$ holds for any page b and capacity C, we have verified that the stack updating for RR can be accomplished using Equation 20, and that RR has the same statistical properties as RAND for each buffer capacity. Note that although a particular value of a point on the success function, for example $F(4) = 0.3$, is equally likely to occur under both RAND and RR, the occurrence of a particular success function is not equally likely.

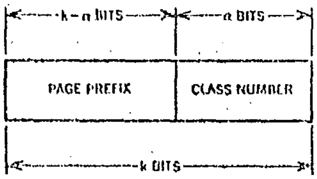
As the example with RR illustrates, stack processing techniques can be extended to cover probabilistic replacement algorithms. In fact, a replacement algorithm can have a mixture of probabilistic and nonprobabilistic aspects. For instance, the arbitrary rule used to break ties in LFU and other algorithms may choose a page at random. Another possibility is for a replacement algorithm to favor some pages probabilistically in the construction of the priority list, thereby realizing a so-called "biased replacement" algorithm.[12] In any case, the only requirement is that the priority list be constructed to reflect the probabilistic properties of the desired replacement algorithm for every capacity C.

Congruence mapping

Up to now, we have restricted our attention to two-level storage hierarchies with unconstrainted mapping at the first level. Under this type of mapping, any page in the buffer may be replaced by the referenced page. The advantages of unconstrained mapping are that all available page frames in the buffer can be used, and also that seldom used pages cannot become "locked" into the buffer by mapping constraints. A disadvantage with unconstrained mapping is that extensive associative searches may be necessary to locate pages in the buffer. Moreover, the implementation overhead of the replacement algorithm may be excessive, since relative priority information must be maintained for all pages in the buffer. To offset these disadvantages, a constrained mapping scheme can be employed whereby each page is restricted to occupy a member of only a subset of the buffer page frames.

One such mapping technique is called *congruence mapping*, by which the 2^k distinct pages in the address space are partitioned into 2^α disjoint *congruence classes*, where $0 \leq \alpha \leq k$, and each class contains $2^{k-\alpha}$ pages. The classes are numbered consecutively from 0 to $2^\alpha - 1$, and class membership is determined from the α low-order bits of the page number. In this case, the α low-order bits constitute the *class number* $[x]$ of a page, and the remaining $k - \alpha$ bits are called the *page prefix* as shown in Figure 15. The quantity α is called the *class length*. For a class length equal to zero, we set $[x] = 0$ for all pages.

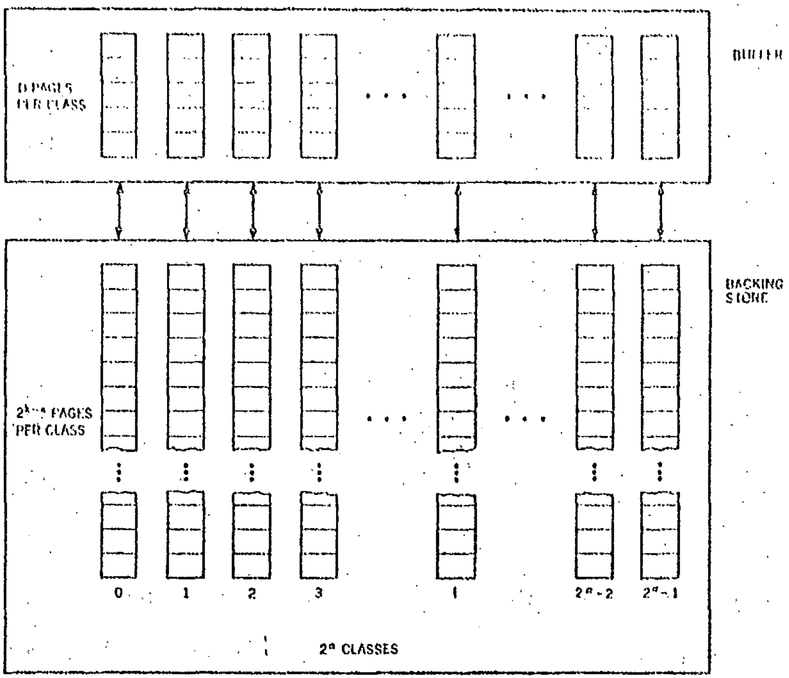
Figure 15 Page number



In a two-level hierarchy with congruence mapping, every congruence class is assigned an equal number of page frames in the buffer—to be used exclusively by members of that class. This number is called the *class capacity* and is denoted by D. (The total capacity of the buffer in pages is thus $C = 2^\alpha \cdot D$.) When a page x is referenced, it may appear in any of the D page frames reserved for class $[x]$. If the reference page is not in the buffer, and if the D page frames are all occupied by other members of class $[x]$, a replacement algorithm selects one of these pages for removal. We assume that the same replacement algorithm is used separately for each of the classes.

Note that when the class length α is zero, all pages are in the same class, and the mapping is unconstrained. When the buffer capacity C is a power of 2, and when $C = 2^\alpha$, only one page is allocated to each class, and the mapping function is fully constrained. Thus for a fixed buffer capacity $C = 2^h$, where $0 \leq h \leq k$, we can vary the mapping function from unconstrained to partially and fully constrained simply by varying the value of α from 0 to h.

Figure 16 Two-level hierarchy with congruence mapping



Since the congruence classes are disjoint, and since the same number of buffer page frames are allocated to each class, it is possible to treat a buffer as a collection of 2^a distinct buffers—one for each class $[x]$. If we also view the backing store as 2^a individual backing stores, as shown in Figure 16, the two-level hierarchy partitions into a collection of 2^a distinct subhierarchies, each with a buffer capacity of D page frames. When the replacement algorithm is a stack algorithm, these subhierarchies can be evaluated separately using stack processing techniques. In practice, 2^a stacks (one for each subhierarchy) can be maintained as the trace is processed. Each page reference x causes only the stack for class $[x]$ to be updated, and a stack distance Δ to be determined from that stack.

In congruence mapping, to calculate the success function for a given trace and given class length α, the stack distances must be carefully interpreted. Whenever a stack distance Δ is measured, the corresponding critical capacity of the entire buffer is $2^\alpha \cdot \Delta$, since this is the minimum buffer capacity necessary to contain the referenced page. Therefore, the success function $F^\alpha(C)$ for the set of capacities $C = 2^\alpha \cdot D$ where $D = 1, 2, \cdots$, is given by

$$F^\alpha(C) = F^\alpha(2^\alpha \cdot D) = \sum_{\Delta=1}^{D} \frac{n(\Delta)}{L}$$

where $n(\Delta)$ is the total number of times the distance Δ occurs for any of the stacks.

Generally, stack processing techniques must be used separately for each value of the class length α. However, for LRU replacement, only a single stack need be maintained in order to determine the success functions for all values of α in the interval $0 \leqq \alpha \leqq k$. Recall that under LRU, the stack S_{t-1} is the list of all the pages in Γ_{t-1} ordered according to most recent reference. To form the stack $S_{t-1}(i, \alpha)$ corresponding to congruence class i and class length α, one would list the pages in class i according to their most recent reference. However, this ordering is preserved in the stack S_{t-1} for any i and any α. Therefore, $S_{t-1}(i, \alpha)$ can be determined by listing in order all the stack entries of S_{t-1} belonging to class i. In practice, it is not necessary to actually construct each stack $S_{t-1}([x_t], \alpha)$ in order to find the distance Δ_t^α. One can determine all the stack distances $\{\Delta_t^\alpha\}$ in one scan of the LRU stack S_{t-1}. To do this, we first define the *right match function* $\text{RM}(x, y)$ for two page numbers x and y as the number of consecutive low-order bits that match. For example, $\text{RM}(01101,00101) = 3$, and $\text{RM}(0000,0001) = 0$. Note that the class numbers of two pages are equal ($[x] = [y]$) if and only if the class length satisfies the inequality $\alpha \leq \text{RM}(x, y)$. Now suppose that the current reference is to page x, and consider the jth entry on stack S_{t-1}, which is $y = s_{t-1}(j)$. The occurrence of page y on the stack will contribute to the distance Δ_t^α if and only if $\text{RM}(x, y) \geq \alpha$. Therefore, Δ_t^α can be determined by counting the number of stack entries y above (and including) page x that satisfy $\text{RM}(x, y) \geq \alpha$.

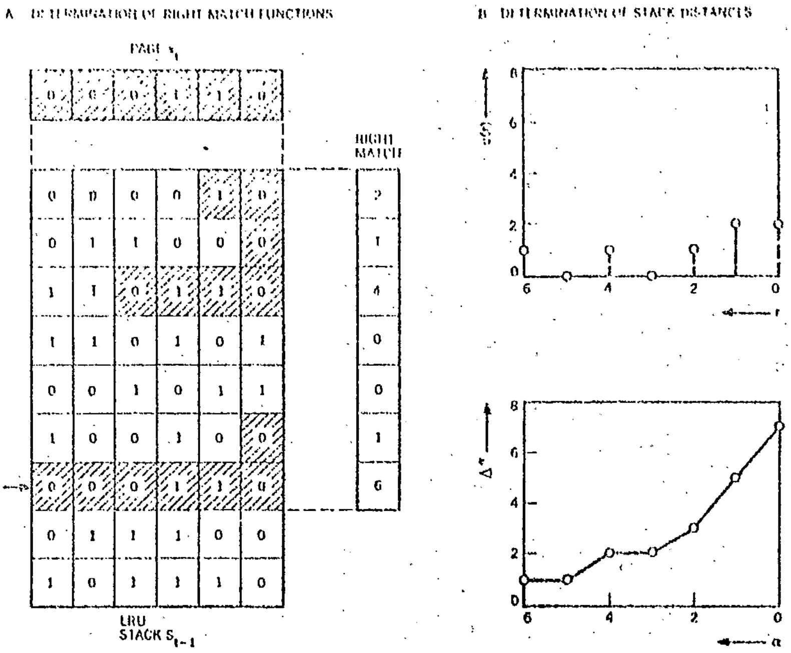
A simple procedure for determining Δ_t^α for all α is to scan down the stack, and maintain a set of right match frequency counters $\{\mu(r)\}$ for $0 \leq r \leq k$. Counter $\mu(r)$ is incremented whenever $\text{RM}(x, y)$ is equal to r. If page x has been previously referenced, we eventually find $\text{RM}(x, y) = k$ (corresponding to $x = y$), and each distance Δ_t^α is given by

$$\Delta_t^\alpha = \sum_{r=\alpha}^{k} \mu(r) \qquad \text{where} \quad 0 \leq \alpha \leq k \tag{23}$$

However, if page x has not been previously referenced, the bottom of stack S_{t-1} is reached and Δ_t^α is set equal to infinity for all class lengths α. In either case, each distance Δ_t^α is used to increment the appropriate distance counter for class length α.

An example of this procedure is indicated in Figure 17. In Figure 17A, the right match functions are found by scanning down the stack. In Figure 17B, the right match frequencies $\{\mu(r)\}$ are plotted in reverse order as a function of r. Cumulative summation, according to Equation 23, then yields the desired LRU stack distances $\{\Delta_t^\alpha\}$. Note that the stack distance for class length zero is the same stack distance Δ as obtained for LRU replacement with unconstrained mapping.

Figure 17 Right match function for LRU replacement



A DETERMINATION OF RIGHT MATCH FUNCTIONS

B DETERMINATION OF STACK DISTANCES

0	0	0	1	1	0	
0	0	0	0	1	0	2
0	1	1	0	0	0	1
1	1	0	1	1	0	4
1	1	0	1	0	1	0
0	0	1	0	1	1	0
1	0	0	1	0	0	1
0	0	0	1	1	0	6
0	1	1	1	0	0	
1	0	1	1	1	0	

Multilevel hierarchies

In previous sections of this paper, stack processing techniques are developed to obtain the success function for a two-level hierarchy. For each buffer capacity, this success function represents the relative number of accesses to the buffer for a given page trace.

We now show that the same success function can be used to find the access frequencies for all levels of a multilevel, linear hierarchy for any number of levels, and any capacity at each level. Recall that in a linear hierarchy, the only downward data path from each level M_i is to the next level M_{i+1}, for $1 \leq i < H$. Also a path or sequence of paths is available from each level M_i, for $1 < i \leq H$, to the local store. Furthermore, no replacement decisions are required when a page moves upward through intermediate levels. We now assume that the same replacement algorithm is used at all levels, and that the mapping function is unconstrained at every level. (Hierarchies with constrained mapping functions are considered later in this paper.) At time $t = 0$, the backing store contains all pages, and these pages are moved to the local store M_1 on demand. When M_1 is full, pages replaced in M_1 are pushed down to the next lower level in the hierarchy, M_2. As each successively lower level M_i fills, the pages replaced in M_i are pushed to the next level M_{i+1}. At level M_1, the replacement algorithm is applied to the

set of pages
reference of pa
the replacem
to the page p

When page
a page is repl
page replaced
since a page f
the backing s
$M_1, M_2, \cdots$,
of pages in t
do not depen
store, but de
capacity at ea

We have sho
for a two-lev
contents of a
now assume t
archy induces
the replaceme
true, then for
$C_2, \cdots, C_H$, t
from the sta
let $B_t^i(C_i)$ den
the sum $C_1 +$

$B_t^i(C_i) = B_t(\sigma$

or equivalentl
stack S_t, and
result is illust

The main elen
that Equation
$s_{t-1}(\Delta_t)$ is an
stack S_{t-1} is
the top C_1 ele
$s_t(C_1)$ represe
removed fron
hierarchy, we
since the hiera
selects a page
page $y_t(C_1)$ h
removal has b
is $y_t(C_1 + C_2$
$B_t^2(C_2)$, and th

A similar argu

set of pages already present, thereby making room for the currently referenced page x_t. At the intermediate levels M_i, for $2 \leq i < H$, the replacement algorithm is applied to the set of pages in M_i and to the page pushed from level M_{i-1}.

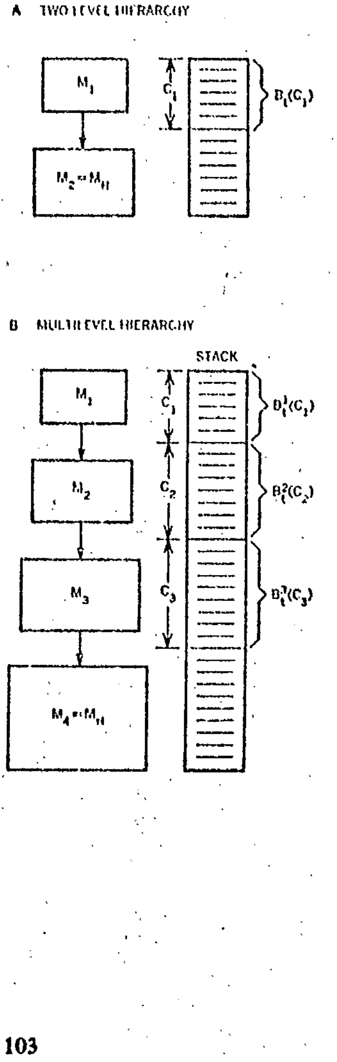
When page x_t is accessed from some level M_i (for $2 \leq i \leq H - 1$), a page is replaced from each of the levels $M_1, M_2, \cdots, M_{i-1}$. The page replaced from level M_{i-1} is guaranteed to find space at level M_i, since a page frame was vacated by x_t. When page x_t is accessed from the backing store M_H, a page is displaced from each of the levels $M_1, M_2, \cdots$, until a vacant page frame is found. Note that positions of pages in the hierarchy—and therefore the access frequencies—do not depend on the structure of upward data paths to the local store, but depend only on the replacement algorithm and the capacity at each level.

We have shown that when a stack replacement algorithm is used for a two-level hierarchy, the top C_1 pages of the stack are the contents of a buffer of capacity C_1 as shown in Figure 18A. Let us now assume that the replacement algorithm for a multilevel hierarchy induces a priority list at every time and that this list determines the replacement decisions at every level of the hierarchy. If this is true, then for any number of levels and any set of capacities C_1, $C_2, \cdots, C_H$, the contents of each level at any time can be determined from the stack for this replacement algorithm. More precisely, let $B_t^i(C_i)$ denote the contents of level M_i at time t, and let σ_i denote the sum $C_1 + C_2 + \cdots + C_i$. We then claim that

$$B_t^i(C_i) = B_t(\sigma_i) - B_t(\sigma_{i-1}) \qquad \text{for } i = 1, 2, \cdots, H - 1 \qquad (24)$$

or equivalently that $B_t^1(C_1)$ can be identified as the first C_1 entries of stack S_t, and B_t^2 can be identified as the next C_2 entries, etc. This result is illustrated for a four-level hierarchy in Figure 18B.

Figure 18 Relationship between stack and hierarchy levels


A TWO LEVEL HIERARCHY

B MULTILEVEL HIERARCHY

The main elements of the proof of this result are as follows. Assume that Equation 24 is satisfied at time $t - 1$, and that page $x_t = s_{t-1}(\Delta_t)$ is an element of $B_{t-1}^g(C_g)$ (i.e., level M_g is accessed.) As stack S_{t-1} is updated to stack S_t, page $y_t(C_1)$ is removed from the top C_1 elements of S_{t-1}, with the result that pages $s_t(1), \cdots$, $s_t(C_1)$ represent $B_t^1(C_1)$. Now observe that page $y_t(C_1 + C_2)$ is removed from the top $C_1 + C_2$ elements of S_{t-1}. In terms of the hierarchy, we know that $y_t(C_1)$ is pushed to the next lower level M_2, since the hierarchy is a linear one. The replacement algorithm then selects a page from $y_t(C_1) + B_{t-1}^2(C_2)$ for removal from M_2. Since page $y_t(C_1)$ has lowest priority in $B_{t-1}^1(C_1)$, the page selected for removal has lowest priority in $B_{t-1}^1(C_1) + B_{t-1}^2(C_2)$. But this page is $y_t(C_1 + C_2)$, so that $s_t(1), \cdots, s_t(C_1 + C_2)$ represent $B_t^1(C_1) + B_t^2(C_2)$, and thus $s_t(C_1 + 1), \cdots, s_t(C_1 + C_2)$ represent $B_t^2(C_2)$.

A similar argument applies to subsequent levels M_i where $2 < i \leq$

$g - 1$. Page $y_t(\sigma_{i-1})$ is pushed from level M_{i-1} of the hierarchy, and competes with the pages in $B^i_{t-1}(C_i)$. The replacement algorithm selects for replacement the page

$$\min [y_t(\sigma_{i-1}), B^i_{t-1}(C_i)] = \min [B_{t-1}(\sigma_i)] = y_t(\sigma_i)$$

with the result that

$$B_t(\sigma_i) = B^1_t(C_1) + B^2_t(C_2) + \cdots + B^i_t(C_i)$$

and

$$B^i_t(C_i) = B_t(\sigma_i) - B_t(\sigma_{i-1})$$

At level M_g, the page $y_t(\sigma_{g-1})$ that has been pushed from M_{g-1} finds a vacant page frame, and all lower levels remain unchanged. Then

$$B^g_t(C_g) = B^g_{t-1}(C_g) + y_t(\sigma_{g-1}) - x_t = B_t(\sigma_g) - B_t(\sigma_{g-1})$$

and

$$B^j_t(C_j) = B^j_{t-1}(C_j) = B_t(\sigma_j) - B_t(\sigma_{j-1}) \text{ for } j > g$$

Thus we have shown that Equation 24 is satisfied at time t.

The significance of this result is that a stack distance Δ, where $C_1 + \cdots + C_{g-1} < \Delta \leq C_1 + \cdots + C_g$, corresponds to an access to hierarchy level M_g, and the relative number of such Δ's is simply the access frequency F_g to that level. Thus

$$F_g = \sum_{\Delta=\sigma_{g-1}+1}^{\sigma_g} \frac{n(\Delta)}{L} = F(\sigma_g) - F(\sigma_{g-1}) \qquad \text{for} \quad 1 \leq g \leq H - 1 \tag{25}$$

As with two-level hierarchies, all other accesses are directed to the backing store so that

$$F_H = 1 - \sum_{i=1}^{H-1} F_i$$

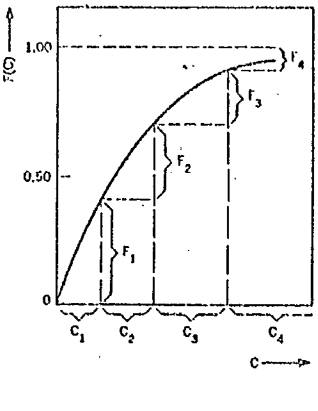


Figure 19 Obtaining access frequencies from success function

The determination of access frequencies is illustrated graphically in Figure 19 for a four-level hierarchy. Note that the technique illustrated in the figure cannot be used for an arbitrary hierarchy or success function. However, the technique can be used for any linear hierarchy as long as the replacement algorithm always induces a single priority list for all hierarchy levels.

Our treatment of multilevel linear hierarchies can be extended to include hierarchies with congruence mapping functions. We assume that the same class length α is used for every level and that D_i page frames are allocated to each congruence class at level M_i. The total capacity of level M_i is then

$$C_i = 2^\alpha \cdot D_i \qquad \text{where } 1 \leq i \leq H. \tag{26}$$

Using the success function $F^\alpha(C)$ and Equations 25 and 26, we obtain the access frequency F^α_i for each level as follows:

$$F_i^m = \begin{cases} F^m(a_i) - F^m(a_{i-1}) & \text{for } 1 \leq i \leq H - 1 \\ 1 - \sum_{i=1}^{H-1} F_i^m & \text{for } i = H \end{cases} \qquad (27)$$

When using Equation 27 or the graphic technique shown in Figure 19, it is important to remember that the success function for multilevel hierarchies with congruence mapping is defined only when the storage capacity is a multiple of 2^n.

Possible extensions

It is possible to extend stack processing techniques to account for various changes in the hierarchy model. For example, with appropriate encoding of the n-bit address, systems with page sizes that are not a power of two can be evaluated. Similarly, other encodings of the n-bit address can be used to evaluate systems with congruence mapping functions for any number of congruence classes with equal or unequal class sizes. Indicative of other changes of the hierarchy model that can be handled by stack processing techniques are the following:

- Pre-loading program pages into the buffer for starting execution
- Loading a working set[10] of pages into the buffer when resuming program execution
- Returning all pages to the backing store upon program interruption
- Maintaining copies of pages in several levels of the storage hierarchy
- Bringing pages to the local store only for fetch operations
- Returning pages to the backing store for references such as stores from an I/O channel
- Moving unequal size pages or segments between levels

To illustrate how stack processing techniques can be adapted to these variations in hierarchy design, we describe two extensions in some detail. In our original model, the generator does not distinguish fetch operations from store operations. In some computer systems, however, pages are brought to the local store only for fetch operations, and usage statistics for page replacement algorithms refer only to references for fetches. Stores to pages in lower levels of the hierarchy are broadcast to these levels by the hierarchy management facility, and no pages are moved. The justification for fetch-store hierarchies is that fetches or additional stores usually do not immediately follow stores to a page.

The evaluation of fetch-store hierarchies requires that the generator tag each reference as either a fetch or a store. For fetches, the priority list and the stack are updated, and a fetch distance Δ^f is recorded. For stores, neither the priority list nor the stack is updated, but a store distance Δ^s is recorded. The distributions $\{n^f(\Delta^f)\}$ and $\{n^s(\Delta^s)\}$ can then be used to determine the fetch and store access frequencies to each level of the hierarchy. It should be clear that this technique also works if congruence mapping is included. We can also consider a modified fetch-store design where the page usage statistics are updated for a store operation even though no page motion occurs. This change is incorporated by updating the priority list for both fetches and stores. Thus, for modified fetch-stores, the net change in our model is that the stack is not updated for store operations.

Besides distinguishing fetches from stores, a computer system may also distinguish the various sources of store requests. For example, a "call-back" feature can be used by which a page in the buffer is moved to the backing store if the page is stored into by an I/O device. The motivation here is to free the buffer of pages not needed by the CPU, and to service all I/O stores from the backing store.

For a call-back hierarchy, the generator must specify at least two kinds of references—CPU references, and stores from the I/O channel. Stack processing techniques can then be modified as follows. When a CPU store or fetch occurs, the stack is updated in the normal way (except for special entries to be described later), and a distance counter $n^{\text{CPU}}(\Delta)$ is incremented. When an I/O store occurs, say at time t, a counter $n^{\text{I/O}}(\Delta)$ is incremented. If page x_t does not occur on stack S_{t-1}, then S_t is equal to S_{t-1}. If page x_t does occur on stack S_{t-1}, then $S_t = S_{t-1}$ except that x_t is replaced by the special entry "#." This entry, counted for all stack distance measurements, represents the empty page frame caused by page x_t returning to the backing store. To ensure that empty page frames are filled as soon as possible, all #-entries are assigned the lowest priority in replacement decisions.

The call-back feature can be used in conjunction with the fetch-store or modified fetch-store schemes. In all cases, the correctness of the modified stack processing techniques can be established.

Since stack processing allows a large sample of "typical" address tapes to be analyzed, for many hierarchy models, the efficiency gained at the early stages of hierarchy design may be great enough to impact the whole design process. More of these traces can be processed in a given time, and more hierarchy designs can be evaluated for a given number of traces. The availability of this data may help justify the "typical"-trace approach to design, or may help in the development of other models for system requirements. As an example, program models can be more deeply investigated by evaluating both a program and its model under a very large number of address traces. Improvement in program modeling, in turn, may enhance the success of analytical disciplines that use these models, such as storage interference studies for multiprogrammed systems.

Concluding remarks

The concepts presented in this paper have been used to develop a variety of stack processing techniques that are useful in the evaluation of storage hierarchies. Using the inclusion property, we define a class of page replacement algorithms, called stack algorithms, and show that replacement algorithms that induce priority lists—such as least recently used, least frequently used, and random replacement—belong to this class.

For any stack algorithm, the frequency of stack distances can be obtained from an address trace by stack processing and used to calculate the success functions. The success function can then be used to determine the relative frequency of access to all levels of a multilevel, linear storage hierarchy, with any number of levels and any capacity at each level.

For least recently used replacement (LRU), the access frequencies of hierarchies with congruence mapping functions can be determined in a single pass of the address trace—for any number of congruence classes, any number of levels, and any capacity per class at each level.

Some special results are presented concerning an optimal replacement algorithm (OPT). It is shown that OPT is a stack algorithm and that OPT minimizes the number of page swaps for any address trace and buffer capacity. Also, both OPT and LRU can be evaluated with a forward pass of the address trace followed by a backward pass of the same address trace.

We conclude that stack processing techniques can eliminate much of the simulation effort required in storage hierarchy evaluation. Furthermore, we believe that the classification of stack algorithms and the various extensions to stack processing techniques may provide insight into the areas of program modeling, system analysis, and computer design.

ACKNOWLEDGMENT

The authors wish to acknowledge J. H. Eaton for his helpful comments and criticism, and T. W. MacDowell for his help in the proof of Theorem 4.

Appendix

Two results mentioned in the paper concerning the OPT replacement algorithm are proved here. To do this, it is first shown that given any trace and replacement algorithm (not necessarily using demand paging) another replacement algorithm exists that uses demand paging and causes the same or a fewer total number of pages to be loaded into the buffer. This result is used to show that OPT is an optimal replacement algorithm and, in fact, that OPT causes the minimum total number of pages to be loaded into the buffer. Finally, it is shown that the success function under OPT for any trace is identical to the success function under OPT for the reverse of the trace.

Definition

- $|S|$ denotes the number of elements in a set S.
- $|a|_X$ denotes the number of occurrences of a symbol a in a sequence X.
- $A = \{a, b, \cdots\}$ is a finite set of N *page addresses* or *pages*.
- $X = x_1, x_2, \cdots, x_L$ is a finite sequence of L elements from A, and is called a *trace*.
- $B_t(C) \subseteq A$ denotes the contents of a buffer of capacity C at time t, and is called a *state*.

Throughout this appendix, we consider a two-level storage hierarchy with fixed buffer capacity C. Consequently, we use B_t instead of $B_t(C)$. The term B_t denotes the contents of the buffer immediately after reference x_t is made; B_0 is called the *initial buffer* state; and ϕ, the empty set, denotes an empty buffer state.

Definition

- $P = p_1, p_2, \cdots, p_L$ is a finite sequence of L sets, $p_t \subseteq A$, called an *O-policy*.
- $Q = q_1, q_2, \cdots, q_L$ is a finite sequence of L sets, $q_t \subseteq A$, called an *I-policy*.

A policy is a particular realization of a replacement algorithm for a given trace. For such a trace and initial buffer state B_0, an *I*-policy and an *O*-policy together determine the sequence of buffer states that will occur during the trace. An *I*-policy gives the set of pages loaded into the buffer, and an *O*-policy gives the set removed. If $p_t = \phi$, no page is removed, and if $q_t = \phi$, no page is loaded in. Note that only certain pairs of *O*- and *I*-policies are meaningful. For example, a page cannot be removed if it is not in the buffer. We consider only meaningful policies, where $q_{t+1} \not\subseteq B_t$ and $p_{t+1} \subseteq B_t + q_{t+1}$, for $0 \leq t \leq L - 1$. In this case, B_{t+1} is obtained from B_t by

$$B_{t+1} = [B_t + q_{t+1}] - p_{t+1}$$

Definition

Let X be a trace and B_0 (where $|B_0| \leq C$) an initial state. A sequence of states $B = B_0, B_1, \cdots, B_L$ is a *valid sequence* if $x_t \in B_t$, for $1 \leq t \leq L$. A policy pair P and Q is a *valid pair* for X and B_0 if application of the pair results in a valid sequence.

Note that valid policy pairs are quite general in that any number of pages may be moved into or out of the buffer. However, most of our attention is directed toward *demand paging* where

$$\left.\begin{array}{l} \circ \quad |p_t| \leq 1 \quad \text{and} \quad |q_t| \leq 1 \\ \circ \quad x_t \in B_{t-1} \Rightarrow p_t = q_t = \phi \\ \circ \quad p_t \neq \phi \Rightarrow q_t \neq \phi \quad \text{and} \quad |B_{t-1}| = C \end{array}\right\} \qquad \text{(A1)}$$

for all t, $1 \leq t \leq L$.

Under demand paging, single pages are loaded when necessary until the buffer fills; subsequently, page swaps occur only when necessary.

One measure of goodness for a policy pair P and Q is the total number of pages loaded into the buffer $\sum_{t=1}^{L} |q_t|$ under the policy pair. The following theorem supports the usefulness of demand paging.

Theorem 1

Let P and Q be a valid policy pair for X and B_0. There exists a valid demand policy pair P^D and Q^D for X and B_0 such that

$$\sum_{t=1}^{L} |q^D_t| \leq \sum_{t=1}^{L} |q_t|$$

Proof. P^D and Q^D will be constructed by forming a sequence of valid policy pairs $(P^0, Q^0), (P^1, Q^1), (P^2, Q^2), \cdots, (P^K, Q^K)$, where $P^0 = P$, $Q^0 = Q$, $P^K = P^D$, $Q^K = Q^D$, and $\sum_{t=1}^{L} |q_t^j| \leq \sum_{t=1}^{L} |q_t^{j-1}|$ for $1 \leq j \leq K$. Informally, P^j and Q^j are constructed from P^{j-1} and Q^{j-1} by altering p_t^{j-1} and q_t^{j-1} to satisfy the demand paging constraints where p_t^{j-1} and/or q_t^{j-1} are the first occurrences of nondemand paging in P^{j-1} and Q^{j-1}. This is done by "sliding" offending elements of p_t^{j-1} and/or q_t^{j-1} to a later time in P^j and Q^j. If $a \in p_t^j$ and $a \in q_t^j$ ever occurs then we trivially remove page a from both p_t^j and q_t^j. Clearly, this does not disturb the validity of P^j and Q^j and only decreases the value of $\sum_{t=1}^{L} |q_t^j|$.

To construct P^j and Q^j from P^{j-1} and Q^{j-1}, $1 \leq j \leq K$, let t be the smallest time such that p_t^{j-1} and/or q_t^{j-1} do not satisfy Equation A1. Set $P^j = P^{j-1}$ and $Q^j = Q^{j-1}$, except as noted below. Suppose that $x_t = a$ and that q_t^{j-1}, for $t < L$, does not satisfy Equation A1. If $a \notin q_t^{j-1}$, then set $q_t^j = \phi$ and $q_{t+1}^j = q_{t+1}^{j-1} + q_t^{j-1}$. (Note that "+" is defined here since $q_t^{j-1} \cap p_t^{j-1} = \phi$). If $a \in q_t^{j-1}$, then set $q_t^j = a$, and $q_{t+1}^j = q_{t+1}^{j-1} + [q_t^{j-1} - a]$. If $t = L$, then set $q_L^j = \phi$ if $a \notin q_L^{j-1}$, or $q_L^j = a$ if $a \in q_L^{j-1}$. In all cases, note that Q^j is valid, since $q_t^j \notin B_{t-1}^j$ for $1 \leq t \leq L$, and that $\sum_{t=1}^{L} |q_t^j| \leq \sum_{t=1}^{L} |q_t^{j-1}|$.

Now suppose that p_t^{i-1}, for $t < L$, does not satisfy Equation A1. We observe first that $|q_t^i| \leq 1$ and $q_t^i = a$, if $a \notin B_{t-1}^{i-1}$. If $q_t^i = \phi$ or $|B_{t-1}^{i-1}| < C$, then set $p_t^i = \phi$ and $p_{t+1}^i = p_{t+1}^{i-1} + p_t^{i-1}$. If $q_t^i = a$ and $|B_{t-1}^{i-1}| = C$, set $p_t^i = b$ for some $b \in p_t^{i-1}$ and $p_{t+1}^i = p_{t+1}^{i-1} + [p_t^{i-1} - b]$. (Note that $p_t^{i-1} \neq \phi$, since $|B_{t-1}^{i-1}| = C$ and $q_t^{i-1} \neq \phi$.) For $t = L$, set $p_L^i = b \in p_L^{i-1}$ if $q_L^i = a$ and $|B_{L-1}^{i-1}| = C$, or $p_L^i = \phi$ otherwise. In all cases, we observe that P^i is valid, since $p_t^i \subseteq B_{t-1}^i$ for $1 \leq t \leq L$. Since P^i and Q^i satisfy demand paging at least up through time t, the desired demand policies must eventually be obtained. Thus the theorem is proved.

Before considering an optimum replacement algorithm we make two observations. First, under demand paging, a valid policy pair P and Q can be completely represented by specifying just the O-policy P. This follows from Equation A1 because $q_t \neq \phi$ can only occur when $x_t = a$ and $a \notin B_{t-1}$ (in which case we know that $q_t = a$). Second, for demand policies P and Q, we can use $|\phi|_P$ as an alternative criterion of goodness. To see this let u be the smallest integer such that $|B_t| = C, t \geq u$. Then $|\phi|_P$ is given by the following expression:

$$|\phi|_P = u + (L - u) - \sum_{t=u+1}^{L} |q_t| \qquad \text{(A2)}$$

Since u in Equation A2 is not a function of the policies, $\sum_{t=1}^{u} |q_t|$ is a constant and

$$|\phi|_P = \left(L + \sum_{t=1}^{u} |q_t|\right) - \sum_{t=1}^{L} |q_t| = \text{constant} - \sum_{t=1}^{L} |q_t| \qquad \text{(A3)}$$

optimum replacement algorithm

For a given trace X and initial state B_0 let us define an optimum policy pair P and Q as a pair that is valid and minimizes $\sum_{t=1}^{L} |q_t|$ over the class of valid policies. From Theorem 1 there always exists an optimum policy pair which is also a demand policy pair. Since (A3) holds for all demand policies we can find an optimum demand policy pair if we can find a demand policy P^D such that $|\phi|_{P^D} \geq |\phi|_P$ where P is any demand policy.

Definition

Let X be a trace, and let $a \in A$ be a page. The *forward distance* $d(a, x_t)$ to page a from page x_t is the number of distinct pages occurring in $x_{t+1}, \cdots, x_e$, where e is the smallest integer satisfying $e > t$ and $x_e = a$. If no such e exists then $d(a, x_t) = \infty$.

Definition

Let X be a trace and B_0 an initial state. A valid demand policy P^O, called an OPT *policy*, for X and B_0 is defined as follows. For $t = 1, 2, \cdots, L$, whenever $p_t \neq \phi$ is required then $p_t = a$ where

$$(\forall b \in B_{t-1})(d(a, x_t) \geq d(b, x_t))$$

The forward distance to a page is just the number of distinct pages referenced before that page is referenced again. An OPT policy requires that the page removed from the buffer be one with the greatest forward distance. Note that an OPT policy is a particular realization of the OPT replacement algorithm discussed in the paper. We observe that, at time t, all pages with finite forward distances have distinct forward distances. However, more than one page may have an infinite forward distance. This means that there may exist more than one OPT policy for a given X and B_0. It should be clear that all such policies P^o have the same value of $|\phi|_{P^o}$.

To show that any P^o maximizes $|\phi|_{P^o}$ over the class of demand policies we use the following lemma.

Lemma 1

Let X be a trace and B_0 and B'_0 initial states where

$$\left.\begin{aligned} B'_0 &= T_0 + \{a\} \\ B_0 &= T_0 + \{b\} \end{aligned}\right\} \quad \text{for} \quad T_0 \subseteq A \quad \text{and} \quad a, b \notin T_0 \tag{A4}$$

and $d(a, x_1) \leq d(b, x_1)$. For any demand policy P, corresponding to X and B_0, there exists a demand policy P', corresponding to X and B'_0, such that

$$|\phi|_{P'} \geq |\phi|_P$$

Proof. Given P, we construct P'. Suppose page a first occurs in X at x_{i_a} and b at x_{i_b}. Thus, $i_a < i_b \leq L$ is assumed. If either b or a does not occur in X, then set i_b or i_a equal to $L + 1$. We consider three cases.

Case 1. $p_j = b$ where p_j is the first occurrence of b in P, and $1 \leq j < i_a$. Here we set $p'_k = p_k$, $1 \leq k \leq L$ and $k \neq j$, and $p'_j = a$. This results in $B_t = T_t + \{b\}$ and $B'_t = T_t + \{a\}, 0 \leq t \leq j - 1$ and $B_t = B'_t, j \leq t \leq L$. Since pages a and b are both not referenced up to time j, it should be clear that P' is a valid demand policy (because P is) and that $|\phi|_{P'} = |\phi|_P$.

Case 2. $p_{i_a} = b$ where p_{i_a} is the first occurrence of b in P. In this case we set $p'_k = p_k$, $1 \leq k \leq L$ and $k \neq j$, and $p'_{i_a} = \phi$. As in Case 1, P' is a valid demand policy and $|\phi|_{P'} = |\phi|_P + 1 \geq |\phi|_P$.

Case 3. $p_j \neq b$, $1 \leq j \leq i_a$. Here we must consider two subcases.

Case 3A. $p_{i_a} = c$. At time $t = i_a$ the states of the buffer are given by

$$B'_{i_a} = T_{i_a} + \{a\}$$

$B_{i_a} = T_{i_a} + \{b\} + \{a\} - \{c\}$ for $c \in T_{i_a}$

which can also be written as follows:

$B'_{i_a} = [T_{i_a} + \{a\} - \{c\}] + \{c\}$

$B_{i_a} = [T_{i_a} + \{a\} - \{c\}] + \{b\}$

Note this is the same form as Equation A4 with T_0 replaced by $[T_{i_a} + \{a\} - \{c\}]$ and a replaced by c. If $d(c, x_{i_a+1}) \leq d(b, x_{i_a+1})$ then we have a situation identical to that in the statement of Lemma 1 where X now is $x_{i_a+1}, \cdots, x_L$. Setting $p'_k = p_k$ for $1 \leq k \leq i_a - 1$ and $p'_{i_a} = \phi$, we again consider Cases 1, 2, and 3. Since the "new" X is strictly shorter than the original X, this situation can only occur a finite number of times. Note that P' is valid as far as it is specified and that $p'_1, \cdots, p'_{i_a}$ contains one more ϕ than $p_1, \cdots, p_{i_a}$.

If $d(c, x_{i_a+1}) > d(b, x_{i_a+1})$, we set $p'_k = p_k$ for $1 \leq k \leq i_a - 1$ and $p'_{i_a} = \phi$, and consider two more cases. First, if $p_\ell = b$, where p_ℓ is the first occurrence of b in X and $\ell < i_b$, we set $p'_k = p_k$, for $i_a + 1 \leq k \leq L$, and $k \neq \ell$ and $p'_\ell = c$. Here $B'_t = B_t$ for $\ell \leq t \leq L$, and as in Case 1, we see that $|\phi|_{P'} \geq |\phi|_P$ still holds. Second, if $p_\ell \neq b$, for $\ell < i_b$, we set $p'_k = p_k$, $i_a + 1 \leq k \leq L$, and $k \neq i_b$ and $p'_{i_b} = c$. Again we have $B'_t = B_t$ for $i_b \leq t \leq L$, but we note that $p_{i_b} = \phi$, whereas $p'_{i_b} = c \neq \phi$. However, since $p_{i_a} \neq \phi$ and $p'_{i_a} = \phi$, the relation $|\phi|_{P'} \geq |\phi|_P$ still holds.

Case 3B. $p_{i_a} = \phi$. Since $q_{i_a} = a$ we observe that $|B_{i_a-1}| < C$. Let ℓ be the smallest integer such that $p_\ell \neq \phi$. If no such integer exists, then let $\ell = L + 1$. We set $p'_k = p_k$ for $1 \leq k \leq i_a$ and consider two cases. First, if $i_b \nless \ell$ then we set $p'_k = p_k$ for $i_a + 1 \leq k \leq L$. Note that $Q' = Q$ except at times i_a and i_b. Since $|B'_t| = |B_t|$ for $i_b \leq t \leq L$, we see that P' is valid, and $|\phi|_{P'} = |\phi|_P$, since $P' = P$. Second, for the case $i_b > \ell$, note that $x_\ell = c$, where $c \neq a$ and $c \neq b$. We set $p'_k = p_k$ for $i_a + 1 \leq k \leq L$ and $k \neq \ell$, and $p'_\ell = \phi$. If $p_\ell = b$, then $|B'_t| = |B_t|$ for $\ell \leq t \leq L$, and $|\phi|_{P'} = |\phi|_P + 1 \geq |\phi|_P$. If $p_\ell = a$, then the buffer states at times $\ell - 1$ and ℓ are:

$B'_{\ell-1} = T_{\ell-1} + \{a\}$ $\qquad B'_\ell = T_{\ell-1} + \{a\} + \{c\}$

$B_{\ell-1} = T_{\ell-1} + \{a\} + \{b\}$ $\qquad B_\ell = T_{\ell-1} + \{b\} + \{c\}$

Rewriting the buffer states at time ℓ as

$B'_\ell = [T_{\ell-1} + \{c\}] + \{a\}$

$B_\ell = [T_{\ell-1} + \{c\}] + \{b\}$

we arrive at a case similar to Case 3A. As in Case 3A, P' contains one more ϕ than P in the interval $t = 1, \cdots, \ell$. Therefore, we treat this case in the same way, with the result $|\phi|_{P'} \geq |\phi|_P$. Finally, if $p_\ell = d$ where $d \neq a$ and $d \neq b$ the buffer states at time ℓ can be written as

$B'_\ell = [T_{\ell-1} + \{a\} + \{c\} - \{d\}] + \{d\}$

$B_\ell = [T_{\ell-1} +$

which again ca

Note that the si $b \in B_{i_b-1}$. We cases, and Lem

Theorem 2

Let X be a trac for X and B_0. $|\phi|_{P^O} \geq |\phi|_P$.

Proof. We rec exactly the sam only find any O will construct a is an OPT policy

P^1 is constructe $p_i \neq p^O_i$, where $p_i = a$ and p^O_i demand policies

$B_i = T_i + \{b\}$

$B^O_i = T_i + \{a\}$

where $d(a, x_i) \leq$ $d(a, x_{i+1}) \leq d(b$ as X, we can use as least as many p^1_L as

$p^1_k \begin{cases} p_k, & 1 \leq k \\ b, & k = i \\ p'_k, & i + 1 \end{cases}$

Note that P^1 is $1 \leq k \leq \ell_1$ for

Policy P^2 is con that $p^2_k = p^O_k$, 1 finite, constructi $p^j_k = p^O_k$, $1 \leq k$ that $|\phi|_P \leq |\phi$ proved.

Combining the Theorems 1 and

$$B_\ell = [T_{\ell-1} + \{a\} + \{c\} - \{d\}] + \{b\}$$

which again can be treated as in Case 3A.

Note that the situation where $i_b = \ell$ can not arise in Case 3B, since $b \in B_{i_b-1}$. We have therefore successfully exhausted the possible cases, and Lemma 1 is proved.

Theorem 2

OPT is an optimal replacement algorithm

Let X be a trace, B_0 an initial state, and P a valid demand policy for X and B_0. If P^O is any valid OPT policy for X and B_0, then $|\phi|_{P^O} \geq |\phi|_P$.

Proof. We recall first that every OPT policy for X and B_0 has exactly the same number of ϕ's. To prove the theorem, we need only find any OPT policy P^O such that $|\phi|_{P^O} \geq |\phi|_P$. To do this we will construct a finite sequence of policies $P^1, P^2, \cdots, P^j$, where P^j is an OPT policy and $|\phi|_P \leq |\phi|_{P^1} \leq \cdots \leq |\phi|_{P^j}$.

P^1 is constructed as follows. Let i be the smallest integer such that $p_i \neq p_i^O$, where p_i^O is an element of an OPT policy. Suppose that $p_i = a$ and $p_i^O = b$. (Neither p_i nor p_i^O can be ϕ, since both are demand policies.) We observe that

$$\left.\begin{aligned} B_i &= T_i + \{b\} \\ B_i^O &= T_i + \{a\} \end{aligned}\right\} \qquad \text{for} \quad a, b \notin T_i$$

where $d(a, x_i) \leq d(b, x_i)$. Since $x_i \neq$ a and $x_i \neq b$, it follows that $d(a, x_{i+1}) \leq d(b, x_{i+1})$. Treating B_i as B_0, B_i^O as B_0^y, and $x_{i+1}, \cdots, x_L$ as X, we can use Lemma 1 to find a policy $p'_{i+1}, \cdots, p'_L$ that contains as least as many ϕ's as $p_{i+1}, \cdots, p_L$. We then define $P^1 = p_1^1, \cdots, p_L^1$ as

$$p_k^1 \begin{cases} p_k, & 1 \leq k \leq i - 1 \\ b, & k = i \\ p'_k, & i + 1 \leq k \leq L \end{cases}$$

Note that P^1 is valid and that $|\phi|_P \leq |\phi|_{P^1}$. Furthermore, $p_k^1 = p_k^O$, $1 \leq k \leq \ell_1$ for some $\ell_1 \geq i$.

Policy P^2 is constructed from P^1 in a similar manner with the results that $p_k^2 = p_k^O$, $1 \leq k \leq \ell_2$ where $\ell_2 > \ell_1$ and $|\phi|_{P^1} \leq |\phi|_{P^2}$. Since X is finite, construction of $P^1, P^2, \cdots$ must result in P^j, for finite j, where $p_k^j = p_k^O$, $1 \leq k \leq L$. It follows from $|\phi|_P \leq |\phi|_{P^1} \leq \cdots \leq |\phi|_{P^j}$ that $|\phi|_P \leq |\phi|_{P^j}$ where P^j is an OPT policy and the theorem is proved.

Combining the relation in Equation A3 for demand paging with Theorems 1 and 2, we have the following theorem.

Theorem 3

OPT minimizes page loading

Let X be a trace, B_0 an initial state, and P^O a valid OPT policy. (Also, let Q^O be the corresponding I-policy.) For any valid policy pair P and Q,

$$\sum_{t=1}^{L} |q_t| \geq \sum_{t=1}^{L} |q_t^O|$$

Thus we see that an OPT policy results in a minimum number of pages being loaded into the buffer over the class of all valid policies. After giving preliminary Lemmas 2 and 3, we present a final theorem concerning OPT policies.

Lemma 2

For a trace X, let the set B_C represent the first C distinct pages referenced in X. For a buffer of capacity C, if P is a valid demand policy for X and some $B_0' \subseteq B_C$, then P is a valid demand policy for X and any $B_0' \subseteq B_C$.

Proof. Let i be the smallest integer such that $x_1, \cdots, x_i$ contains C distinct pages. If $B_0 \subseteq B_C$ then, for any valid demand policy P, we have $B_i = B_C$, since $p_1 = p_2 = \cdots = p_i = \phi$. For $B_0' \subseteq B_C$ this also holds, so P is a valid demand policy for X and B_0'. (Note that for different initial states, $B_0 \subseteq B_C$, the Q policies will not be the same.)

Lemma 3

For a trace X, let the set E_C represent the last C distinct pages referenced in X. For a buffer of capacity C, if P is a valid demand policy for X and B_0, there exists a valid demand policy P' with a state sequence $B_0, B_1', B_2', \cdots, B_L'$ such that $B_L' = E_C$ and $|\phi|_{P'} \geq |\phi|_P$.

Proof. Let i be the smallest integer such that $x_i, \cdots, x_L$ contains C distinct pages. Suppose, under policy P, that B_{i-1} contains n elements of E_C, i.e. $|B_{i-1} \cap E_C| = n$. It follows that at least $C - n$ pages will be loaded into the buffer following time $i - 1$. Setting $p_k' = p_k$ for $1 \leq k \leq i - 1$, we will specify the remainder of P' in such a way that exactly $C - n$ pages are loaded into the buffer following time $t - 1$. We observe that, since at most C distinct pages are referenced following time $i - 1$, we never need remove a page b from the buffer where $b \in E_C$. Thus, if a page must be removed at time ℓ for $i \leq \ell \leq L$, there always exists a page c, where $c \notin E_C$, in the buffer, and we set $p_\ell' = c$. If P' is constructed in this manner,

$$\sum_{t=1}^{L} |q_t'| \leq \sum_{t=1}^{L} |q_t|$$

and from Equation A3 we have $|\phi|_{P'} \geq |\phi|_{P}$. Furthermore, since no page in E_C is ever removed from the buffer following time $t = i$ and $|E_C| = C$, we see that $B_L' = E_C$.

Theorem 4

forward/ backward OPT

Let $X = x_1, \cdots, x_L$ be a trace and ${}^rX = x_L, \cdots, x_1$ its *reverse*. If P^O is an OPT policy for X and $B_0 = \phi$, and ${}^rP^O$ is an OPT policy for ${}^rX$ and ${}^rB_0 = \phi$, then $|\phi|_{P^O} = |\phi|_{{}^rP^O}$.

Proof. Let us assume that the theorem does not hold. Thus, without loss of generality, suppose that $|\phi|_{{}^rP^O} = |\phi|_{P^O} + k$ where k is an integer and $k > 0$. If D distinct pages are referenced in X (and in ${}^rX$) and if $D \leq C$, the buffer capacity, then we have an immediate contradiction, since $|\phi|_{P^O} = |\phi|_{{}^rP^O} = L$. We therefore assume $D > C$.

Let us denote the state sequence under P^O as $B_0, B_1, \cdots, B_L$. From Lemma 2 we can set $B_0 = B_C$ without disturbing the validity of P^O. From Lemma 3 we can alter P^O such that $B_L = E_C$. Note that the altered policy contains the same number of ϕ's as P^O, since P^O is an OPT policy. (We subsequently refer to the altered policy as P^O.) Similarly, if ${}^rB_0, {}^rB_1, \cdots, {}^rB_L$ is the state sequence under ${}^rP^O$ we can assume that ${}^rB_0 = {}^rB_C$ and ${}^rB_L = {}^rE_C$.

Consider now the state sequence ${}^rB_L, {}^rB_L, {}^rB_{L-1}, \cdots, {}^rB_2, {}^rB_1$. Since $x_L \in {}^rB_1, x_{L-1} \in {}^rB_2, \cdots, x_2 \in {}^rB_{L-1}, x_1 \in {}^rB_L$, we see that this sequence is a valid (not necessarily demand) sequence for the trace X. Let us denote the corresponding valid policy pair as P' and Q'. We observe first that, since ${}^rE_C = B_C$, we have ${}^rB_L = B_C = B_0$. Thus P' and Q' (as well as P^O) are valid policies for X and B_0. Next we observe that ${}^rB_L = {}^rB_{L-1} + \{{}^rq_L^O\} - \{{}^rp_L^O\}$ can be written as ${}^rB_{L-1} = {}^rB_L + \{{}^rp_L^O\} - \{{}^rq_L^O\}$. But we also have ${}^rB_{L-1} = {}^rB_L + \{q_2'\} - \{p_2'\}$, which yields $q_2' = {}^rp_L^O$ and $p_2' = {}^rq_L^O$, since ${}^rp_L^O \cap {}^rq_L^O = \phi$. Similarly, since ${}^rB_{L-1} = {}^rB_{L-2} + \{{}^rq_{L-1}^O\} - \{{}^rp_{L-1}^O\}$, we have $q_3' = {}^rp_{L-1}^O$ and $p_3' = {}^rq_{L-1}^O$. Continuing in this manner we can show that

$$\left.\begin{aligned} q_t' &= {}^rp_{L+2-t}^O \\ p_t' &= {}^rq_{L+2-t}^O \end{aligned}\right\} \quad \text{for} \quad 2 \leq t \leq L \tag{A5}$$

Now, since $x_L \in {}^rB_0$ (recall that ${}^rB_0 = {}^rB_C$), it follows that ${}^rp_1^O = {}^rq_1^O = \phi$. Similarly, since $x_1 \in B_0$ (recall that $B_0 = B_C$), it follows that $p_1' = q_1' = \phi$. We can then trivially assume that $p_1' = {}^rq_1^O$ and $q_1' = {}^rp_1^O$. The significance of this is that, using Equation A5, we have established a one-to-one correspondence between P' and ${}^rQ^O$, and between Q' and ${}^rP^O$. In particular, $|\phi|_{P'} = |\phi|_{{}^rQ^O}$ and $|\phi|_{Q'} = |\phi|_{{}^rP^O}$. We now observe that $|\phi|_{{}^rQ^O} = |\phi|_{{}^rP^O}$, since $|{}^rB_0| = |{}^rB_1| = \cdots = |{}^rB_L| = C$. In other words, ${}^rp_t^O = \phi$ if and only if

${}^{r}q_t^0 = \phi$, since the buffer is always full. We thus have shown that $|\phi|_{P'} = |\phi|_{rQ0} = |\phi|_{rP0}$.

Recall that P' and Q' are not necessarily demand policies. From Theorem 1 we can find a demand policy pair P'' and Q'' such that

$$\sum_{t=1}^{L} |q_t''| \leq \sum_{t=1}^{L} |q_t'|$$

From Equation A5 and the discussion that follows, we know that $|p_t'| = |q_t'|$ for $1 \leq t \leq L$. Since P'' and Q''
are demand policies, and since $|B_0| = |B_1''| = \cdots = |B_L''| = C$, we have
$|p_t''| = |q_t''|$ for $1 \leq t \leq L$. Combining these results yields

$$\sum_{t=1}^{L} |p_t''| \leq \sum_{t=1}^{L} |p_t'| \quad \text{or} \quad |\phi|_{P''} \geq |\phi|_{P'}$$

But then we have $|\phi|_{P''} \geq |\phi_{P'}| = |\phi|_{rPO} = |\phi|_{PO} + k$. Since P^O was given as an OPT policy, we have from Theorem 2 a contradiction with $|\phi|_{P''} > |\phi|_{PO}$ for the demand policy P''. Thus our original assumption is false, and it must be the case that $|\phi|_{rPO} = |\phi|_{PO}$.

CITED REFERENCES

1. A. Opler, "Dynamic flow of programs and data through hierarchical storage," *Information Processing 1965, Proceedings of IFIP Congress* 1, 273–276 (1965).
2. E. Morenoff and J. B. McLean, "Application of level changing to a multilevel storage organization," *Communications of the Association for Computing Machinery* 10, 3, 149–154 (1967).
3. C. J. Conti, "Concepts for buffer storage," *IEEE Computer Group News* 2, 8, 9–13 (1969).
4. W. Anacker and C. P. Wang, "Performance evaluation of computing systems with memory hierarchies," *IEEE Transactions on Electronic Computers* EC-16, 6, 764–773 (1967).
5. R. L. Mattson and J.-P. Jacob, "Optimization studies for computer systems with virtual memory," *Information Processing 1968, IFIP Congress Booklet I*, 47–54 (1968).
6. J. E. Shemer and G. A. Shippey, "Statistical analysis of paged and segmented computer systems," *IEEE Transactions on Electronic Computers* EC-15, 6, 855–863 (1966).
7. J. Fotheringham, "Dynamic storage allocation in the ATLAS computer, including an automatic use of a backing store," *Communications of the Association for Computing Machinery* 4, 10, 435–436 (1961).
8. T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One-level storage system," *IEEE Transactions on Electronic Computers* EC-11, 2, 223–235 (1962).
9. M. H. J. Baylis, D. G. Fletcher, and D. J. Howarth, "Paging studies made on the I.C.T. ATLAS computer," *Information Processing 1968, IFIP Congress Booklet D*, 113–118 (1968).
10. D. H. Gibson, "Considerations in block-oriented systems design," *AFIPS Conference Proceedings, Spring Joint Computer Conference* 30, Academic Press, New York, New York, 75–80 (1967).
11. S. J. Liptay, "Structural aspects of the System/360 Model 85: II The cache," *IBM Systems Journal* 7, 1, 15–21 (1968).

12. R. W. O'N
tem with
Proceeding
New York,
13. L. A. Bela
computer,
14. C. J. Kuch
Conference
1018 (1968
15. C. V. Ran
and progr
Proceeding
Computing
229–239 (
16. J. Kral, "C
Communic
7, 475–480
17. J. G. Kem
Company,
18. L. A. Bela
time chara
Communic
6, 349–353
19. P. J. Denn
Communic
5, 323–333

12. R. W. O'Neill, "Experience using a time-sharing multiprogramming system with dynamic address relocation hardware," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **30**, Academic Press, New York, New York, 611–621 (1967).
13. L. A. Belady, "A study of replacement algorithms for a virtual-storage computer, *IBM Systems Journal* **5**, 2, 78–101 (1966).
14. C. J. Kuehner and B. Randell, "Demand paging in perspective," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **33**, 1011–1018 (1968).
15. C. V. Ramamoorthy, "The analytic design of a dynamic look ahead and program segmenting system for multiprogrammed computers," *Proceedings of the 21st National Conference of the Association for Computing Machinery*, Thompson Book Company, Washington, D. C., 229–239 (1966).
16. J. Kral, "One way of estimating frequencies of jumps in a program," *Communications of the Association for Computing Machinery* **11**, 7, 475–480 (1968).
17. J. G. Kemeny and J. L. Snell, *Finite Markov Chains*, D. van Nostrand Company, Inc., Princeton, New Jersey (1960).
18. L. A. Belady, R. A. Nelson, and G. S. Shedler, "An anomaly in spare-time characteristics of certain programs running in a paging machine," *Communications of the Association for Computing Machinery* **12**, 6, 349–353 (1969).
19. P. J. Denning, "The working set model for programming behavior," *Communications of the Association for Computing Machinery* **11**, 5, 323–333 (1968).

IBM

Atlas Computer

B.H. Journal Computer

1966

Dynamic program behavior under paging*

by GERALD H. FINE, CALVIN W. JACKSON,** and PAUL V. MC ISAAC
System Development Corporation
Santa Monica, California

828 2506

INTRODUCTION

In May, 1965, System Development Corporation (SDC) proposed to do some research to study program organization with respect to dynamic program behavior. Further, the proposal suggested that simulation techniques might be used to study the problem of resource allocation in a multiprocessor time-sharing system. Some of the reasons for the proposal related to the prospective utilization of the time-sharing hardware features of the GE and IBM time-sharing computers. At the time, there was considerable interest in investigating the concepts of program segmentation and page turning, both at SDC and in the time-sharing community at large. The concept of fixed-size paging on demand particularly, raised some questions of practicality. One of the early papers on the subject by Dennis and Glaser[1] states that the concept of page-turning can be either useful or disastrous, depending on the class of information to which it is applied. However, the theory appeared to be both advantageous and elegant, so that the future of time-sharing seemed to be committed to the concept.

As a result, an independent activity was initiated to investigate some of the problems outlined in the proposal: this paper reports the results of this effort, and points out some of the implications of the data obtained.

Discussion of the problem

A large high-speed memory is not being used efficiently if a large portion of it is occupied by portions of programs that are never used. Avoidance of fetching unnecessary instructions and data thus appears desirable; there are obvious gains if processing can be accomplished in parallel with pertinent fetching. However, attempts to achieve the above by an arbitrary division of programs into fixed-sized pages that are brought to memory only on actual reference (demand paging) presuppose a program organization scheme which minimizes interpage references with respect to Galler, et al[2] that "the 'single page' loading strategy incurs, each time, the overhead of discovering why a processing sequences. It has been suggested by Arden, storage reference failed, finding the needed page in secondary storage, and switching to another user during transmission of the needed page to high-speed storage." One should possibly add, "if there is another user." Fetching can be overlapped with processing only if there is some processing to be done at the time; it is possible that many user programs desiring processing may be simultaneously held in an unexecutable state while waiting for pages. Further, these pauses for page fetching may delay completion of user service requests and result in a generally high user demand. This high user demand might be useful for a batch-processing system, but for time sharing it probably means congestion and poor response for at least some of the users.

Method of investigation

The approach taken by the project was to obtain empirical information about the actual memory requirements and page demand rates of existing programs operating under the Q-32 Time-Sharing System.[3] Such programs, of course, have not been specifically organized to operate in a paging environment. Since it is not obvious how to accomplish this organization nor even that programs are susceptible to such organization, it was felt that such empirical data would provide a starting point, perhaps would give some clues to automatic methods of structuring, and in any event, would be useful as input to a simulation model.

To obtain an accurate picture of a program's dynamic behavior, it was decided to execute the program in an interpretive manner. In this way recordings could be made to show memory utilization as a function of time (instruction count). An interpretive routine was written that performed this function on the AN/FSQ-32 computer, a high-speed 48-bit word computer.[3] Memory was considered as 46 pages of 1024 words each. Every memory reference made by the object program was checked to obtain the instructions themselves and the

*The research reported in this paper was sponsored by the Advanced Research Projects Agency Information Processing Techniques Office and was monitored by the Electronic Systems Division, Air Force Systems Command under contract AF 19(628)-5166 with the System Development Corporation.

**Currently with *Abacus Programming Company*, Santa Monica, California.

data references (including all levels of indirect addressing). These references were examined in terms of page addresses and then were recorded along with the instruction count at the time of occurrence.

Results

In the initial runs, the instruction count was reset to zero whenever the object program branched or fell through to a new instruction page; all pages were considered inactive at this point. As each inactive page was referenced, the page number and instruction count were recorded. The page was then considered to be active and available for the remainder of the sequence, that is, until the instruction count was again reset. Thus, a count of the instructions actually executed on each page was obtained, followed by a list of data pages referenced by that instruction sequence. Both the last and the first instruction referencing each data page were also recorded as well as an indicator as to whether the data page was "set" (written) or "used only" (read only) during the sequence.

The first runs on various popular programs all exhibited pretty much the same pattern:

1. Short instruction sequences — relatively few instructions executed on any particular page before a branch or fall-through to another instruction page.
2. Considerable data page reference per sequence.
3. Early and late reference to data pages.
4. Rather rare occurrences of "used only" data pages.

For example, in a small sample (200 instruction sequences taken from the JOVIAL compiler in a normal card-processing stage) the mean instruction sequence was only 109.4 instructions. During each sequence, 3.5 data pages were referenced on the average. Only one data page (of 11 referenced) was "used only" during the entire 200 page sequence (21881 instructions). Further, data pages tended to be required quite early in each sequence and usually were needed until nearly the end of the sequence.

In later runs, the recording was modified slightly to examine multi-page sequences corresponding to what used to be defined as a service interval on the Q-32 Time-Sharing System. Such a service interval was terminated by a call to the system or by the execution of 80,000 instructions, whichever occurred first. The 80,000 instruction figure was used to approximate a system-imposed quantum interrupt of about 400 ms of Q-32 time. The instruction count was accordingly reset to zero at the beginning of each such interval, and again all pages were considered inactive at this point. As each inactive page was referenced, the page number and instruction count were recorded as before; in addition an indicator was recorded if the page was referenced for instructions to show that the program was operating in that page. Once activated, pages were considered to be available for the remainder of the entire interval. This approach provided a picture of the page call rate and total storage requirements for each service interval. (One or more such intervals constituted a complete user service request or action.)

The following five programs were examined in this manner:

1. LISP — A programming system providing for the generation, editing, compilation, and execution of programs written in the list-processing language, LISP 1.5. (44 pages)
2. META5 — A syntax-directed meta compiler which translates an object language to a target language interpretively. (14 pages)
3. GPDS — An interpretive display generation system that is first interactive while acquiring a data base and then computational while generating the display. (41 pages)
4. TINT — A conversational, on-line, algebraic JOVIAL interpreter. (23 pages)
5. SURE — A JOVIAL source language programming tool that "launders" JOVIAL source language, providing a reformatted and concordance listing of the program. (30 pages)

These programs were operated for short periods of time because the cost of interpretive execution was high. For the most part, they were performing tasks that might be selected for demonstration purposes. Some effort was made to choose typical actions covering the range of time-sharing requests, though in the sense of frequency of occurrence of various request types, the sample is not quite representative of actual time-sharing operations. One hundred and eighty-two service intervals ranging from three to 80,000 instructions were examined; these intervals comprise 35 service requests ranging from seven to 1,281,504 instructions in length.

The results of recording the dynamic behavior of these programs in the manner described are summarized in Figures 1, 2, 3, and 4.

Figure 1 shows the cumulative relative frequency of the number of instructions executed between consecutive calls for new pages. In nearly 59% of 1737 cases less than 20 instructions were executed; in about 80% of the cases, less than 200 instructions. In only 2.3% of the cases, 10,000 or more instructions were executed between calls; these longer sequences occurred usually only after the program had accumulated a majority of the pages it required.

This effect is illustrated more clearly in Figure 2 which shows page demand as a function of time. The time scale is logorithmic in milliseconds, derived from the instruction counts by assuming a processor speed of 1.6 μs per instruction. The initial call rate for pages is

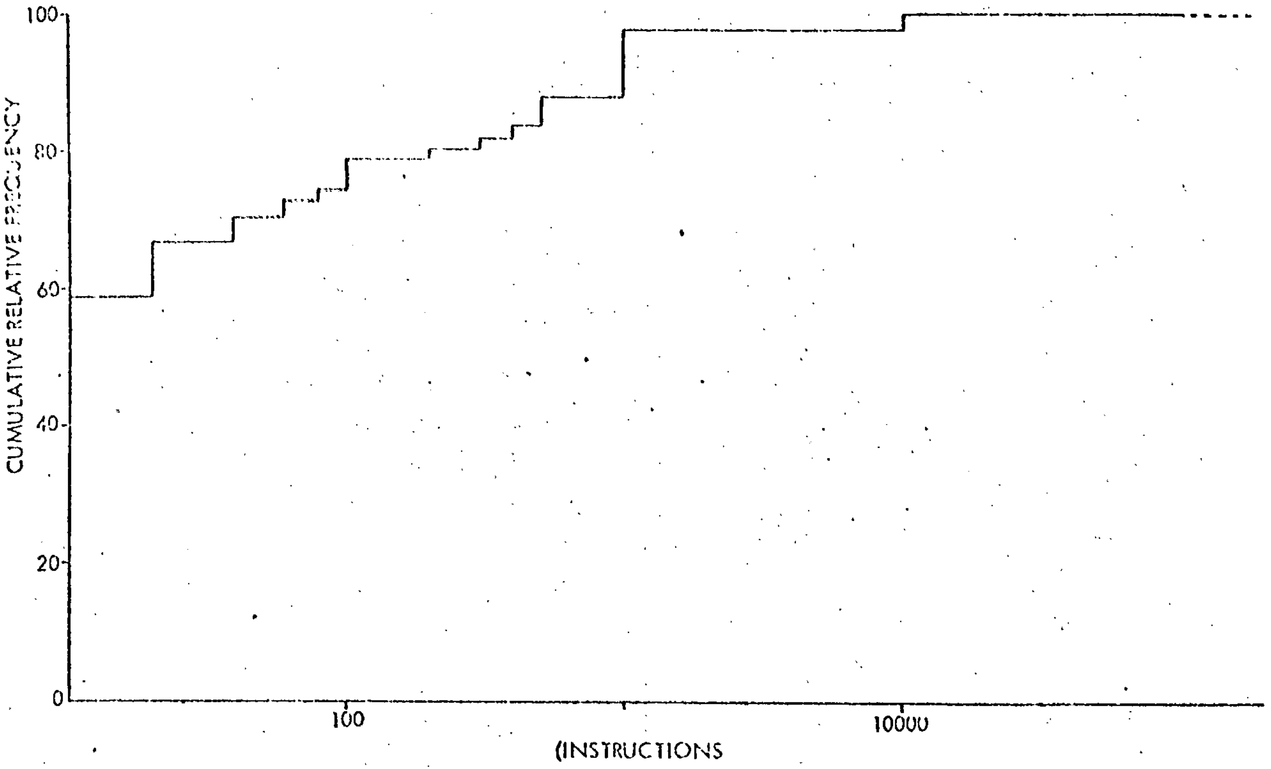


Figure 1 — Cumulative relative frequency of number of instructions executed between page calls

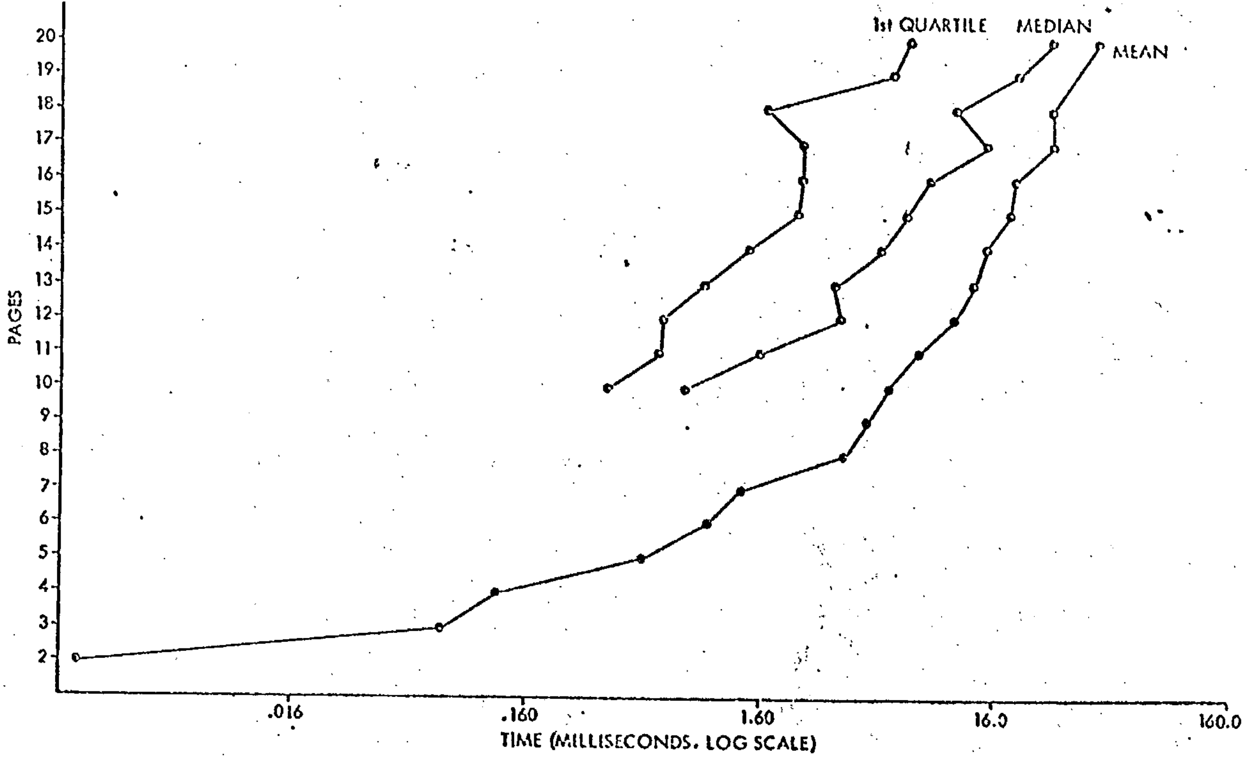


Figure 2 — Page demand (all programs)

extremely high; the first ten pages, on the average, were required within about 5.6 ms; in half of the cases, these first ten pages were required in less than .8 ms. In 25% of the cases where 20 or more pages were required, the first 20 pages were needed within about 7.0 ms.

Figure 3 also shows the (mean) page demand by individual program. The over-all pattern seems to be fairly consistent in spite of the distinct dissimilarity of function of the various programs.

A plot of total execution time per request versus percentage of pages required is shown in Figure 4. The general trend appears to be what one would expect; the longer the service request the more pages required. The two points in the upper left portion of the plot illustrate the occasional occurrence of requests with rather heavy page needs even for very small amounts of processing service.

The dynamic behavior of the examined programs may be briefly generalized here:

1. The programs tend to demand pages at very rapid rates until they have acquired a sufficiency of pages.
2. The programs frequently do not run very long even after having acquired a sufficiency of pages.
3. For those program requests which do run for a while, a sufficiency of pages means a considerable fraction of their total declared page requirements.

Discussion and speculations

It is difficult to assess with any certainty the benefits of a demand paging strategy in a time sharing system. Computer configuration, work load environment, and other system characteristics such as scheduling and priority schemes all strongly influence system performance; performance itself means different things to different people. For a general-purpose system such as MAC or SDC's, required to service with reasonable responsiveness a heavy load of programs similar to those examined, the data obtained in this study seem to indicate that such programs will require considerable reorganization to operate efficiently in a demand-paging environment.

The usual conception of a high-speed memory filled with a page or two from each of many programs desiring processing does not look as though it will stand up subject to the page call rates observed in this study. The page-fetching mechanism seems likely to congest within a few milliseconds; until some of the programs have acquired a sufficiency of pages there would be little chance of processing-fetching overlap; and a sufficiency

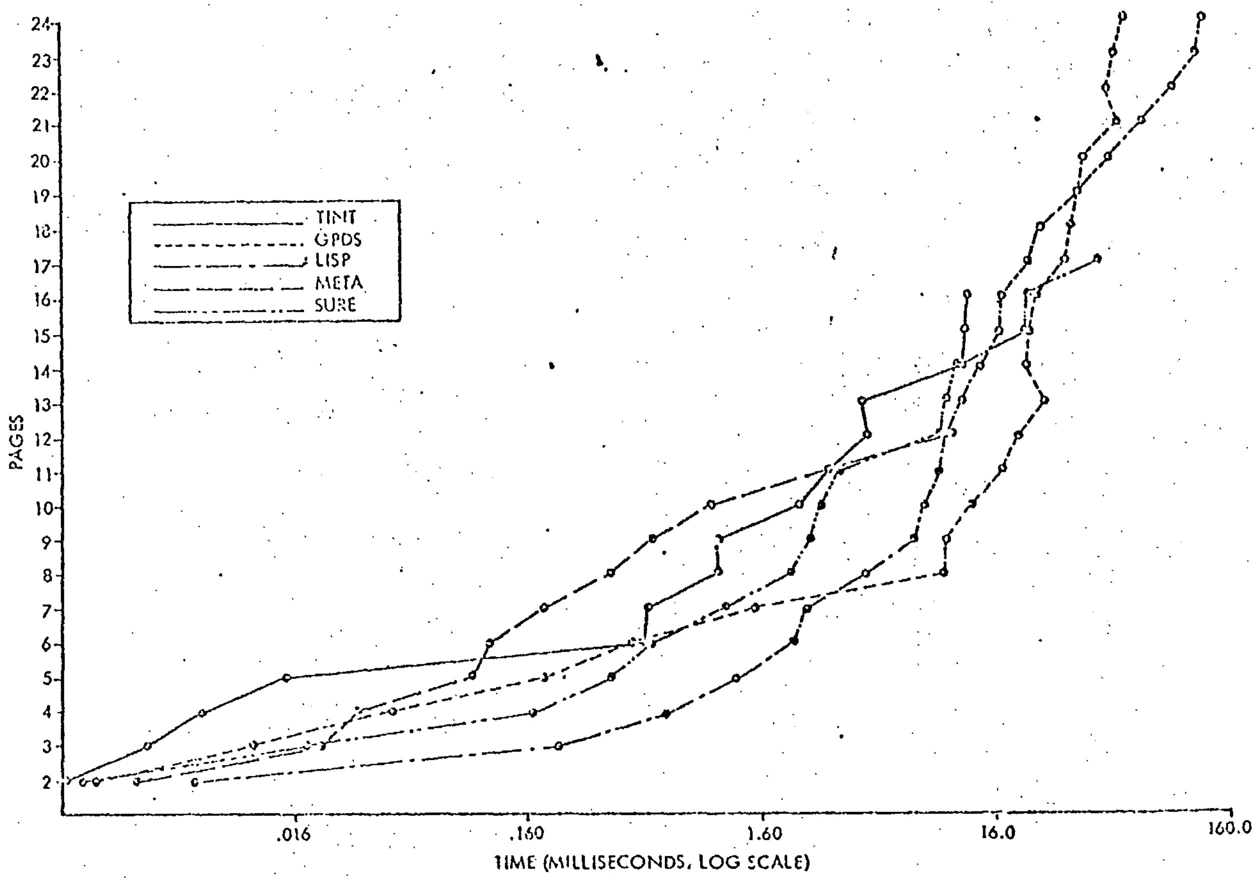


Figure 3 — Page demand (by program)

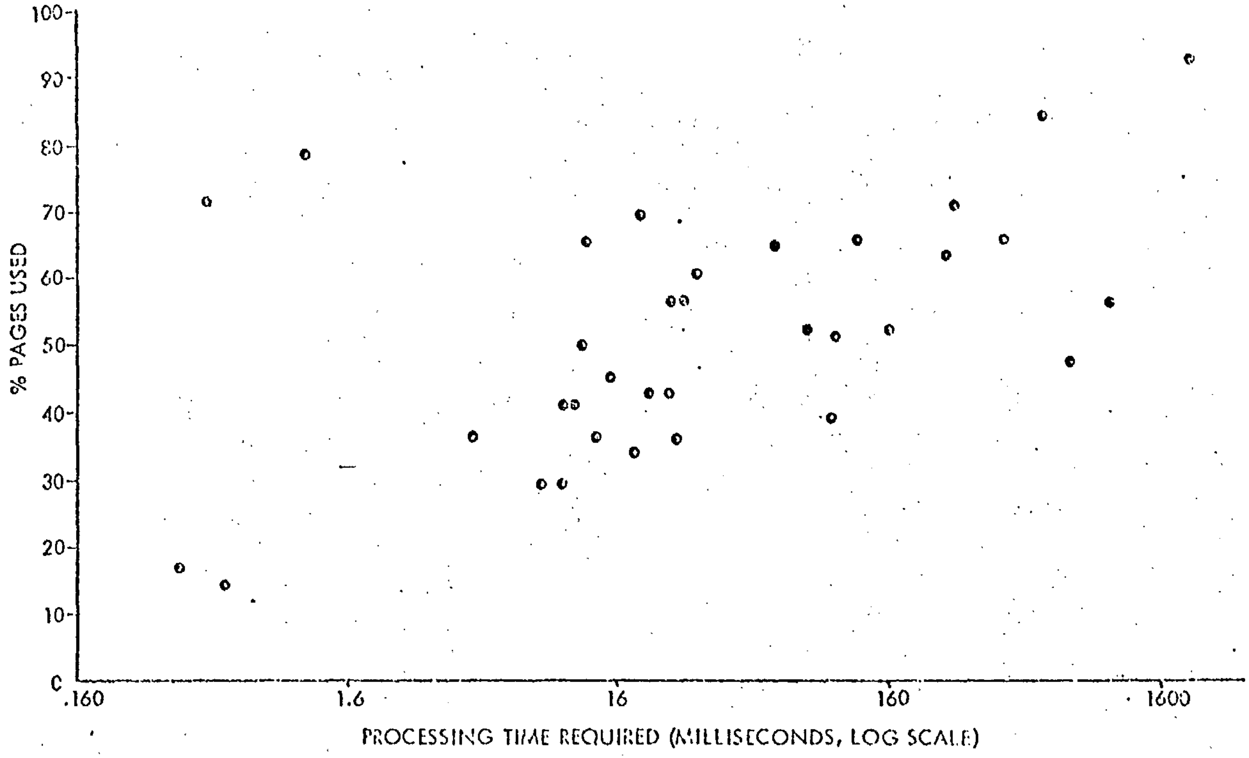


Figure 4 — Page usage vs. processing time

of pages for some programs means that others must be squeezed out of core and deferred.

Reorganization or structuring of the programs for paging is usually proposed as a solution to this problem. Just how much structuring is needed or can be done or how this is to be accomplished is a matter of speculation. Ideally, every program, during both checkout and running phases, for each possible actibn that it handles, should be somehow arranged so that it preferably uses very few pages per action and that it processes long enough between page calls on the average to overlap the time to fetch a page. Further, still speaking ideally, the arrangement of programs to behave in this manner should be accomplished automatically, perhaps by the compiler or a special optimizing routine, without burdening the programmer.

The authors confess that they do not know how to achieve this ideal or even an approximation to it. The following suggestions for structuring have been culled from various sources:

1. Put data in the instruction pages referring to them.
2. Somehow rearrange data structures to reduce data page flow without causing an appreciable increase in instruction page flow.
3. Duplicate subroutines and constants within pages referring to them frequently.
4. Make considerable use of "common routines."

In the authors' opinions, none of these seems likely to have sufficient pay-off, if any. The last suggestion needs some comment perhaps; it is not clear just what is meant by "common routines." If one means common subroutines such as I/O conversions, log, exponential, and trigonometric functions, etc., the whole set of them hardly constitutes more than a page or two of code and a frequently used majority of them might more simply be offered to programs as system services. If on the other hand, "common routines" means larger functional entities such as matrix-manipulation routines or packages of multi-function routines such as an on-line alegbraic interpreter, file search routines, etc, there undoubtedly would be considerable common usage of these. The problem here is simultaneity; a time-sharing system is usually unable, without serious degradation in response, to withhold service to requests until they can be "batched" to use a particular routine in common. In a heavily loaded general-purpose system at least, the chances seem small that the user request will find the particular routine requested remaining in core from some previous request. The on-line frequency of requests for a particular package is probably somewhat proportional to the variety of service offered by the package; the more variety the larger the package and therefore the less likelihood that it can reside in core for any period of time.

An alternative is to abandon the demand-paging strategy and try something else. One idea that has been advanced is to structure programs into functional segments and to bring in "sets of pages" by having the program give "advice" to the time-sharing Executive or monitor in advance of its needs. With this in mind, the TINT program was examined in some detail to determine if there are enough clues in the source program to provide a better organizational scheme. A teletype communicator, a compiler, an interpreter, explanation routine, and data area are used in TINT. These program regions are functionally independent and vary in size. If the program refers to any one page in any of these regions, the entire region is likely to be required. The data area is dynamic in its storage requirements. Some better utilization of the main store might be realized if this kind of segment information could be made available to the time-sharing Executive.

Realistically, it does not seem likely that programmers will supply such information; it is still less likely that a compiler could abstract such information from static code and automatically pass it on to the Executive. It is probably optimistic to assume that programs in general are susceptible to automatic segmentation beyond the nonfunctional division into instructions, data, and read-only data. For these programs which do exhibit functional patterns of behavior, the amount of information required to describe these patterns and the processing required to detect the currently requested pattern might prove prohibitive. In programs which are primarily data driven, for example, any achievable functional segmentation seems likely to be gross. The benefits of inaccurate segmentation may become marginal considering that, in addition to the facility for handling segment information, one must retain the mechanism to discover and fetch, on demand, odd missing pages. This leads to program segments waiting, dead in core, for such pages and can lose back in occupancy time the savings which may have been achieved in occupancy space.

SUMMARY and CONCLUSIONS

The results of examining the dynamic behavior under paging of certain existing time-sharing programs have been presented here. The data obtained in this study seem to indicate that the handling of programs similar to these may be difficult in a time-sharing environment utilizing a paging on-demand strategy. The problem of trying to alleviate these difficulties by reorganization of the programs has been discussed and some speculations on the problems involved in employing an alternative "sets of pages" or segmentation strategy have been presented.

The difficulty with both the demand paging and "sets of pages" strategies is that system performance seems strongly dependent on assumptions that something can and will be done to the programs to be handled by the system. In the opinion of the authors, this approach of trying to fit the work to the system instead of vice versa, seems unrealistic. It may not perhaps be entirely valid to assume that the work load characteristics of future systems can be extrapolated from those of existing systems, but there is no reason to believe they will differ greatly. In view of the fact that existing load characteristics are measurable and have been measured, it would appear more fruitful to base system design criteria on these known parameters than on optimistic hypothetical assumptions.

REFERENCES

1 J B DENNIS E L GLASER
The structure of on-line information processing systems
Preceedings of the Second Congress on the Information System Sciences p 6 1965

2 B W ARDEN B A GALLER T C O'BRIEN F H WESTERVELT
Program and addressing structure in a time-sharing environment
Journal of the ACM vol 13 pp 1-17 January 1966

3 J I SCHWARTZ E G COFFMAN C WEISSMAN
A general-purpose time-sharing system
SDC document SP-1499 31 pp 29 April 1964

TEXTBOOK REFERENCES

1.	R. Watson	Time Sharing System Design Concepts McGraw Hill 1970 ($12.50)
2.	Harry Katzan Jr.	Advanced Programming Van Nostrand Reinhold 1970 ($15.00)
3.	J. Martin	Teleprocessing Network Organization Prentice Hall 1970
4.	DATAMATION	A Catalogue of EDP Products of Services (1971) Available from Datamation, 1301 South Grove Ave., Barrington, Illinois 60010. ($35.00)
5.	AUERBACH	On Time Sharing (1967) available from Auerbach Info. Inc., Philadelphia, Pa 19109. ($14.00)
6.	James Ziegler	Time Sharing Data Processing Systems Prentice Hall 1967 ($13.00)
7.	Douglas Parkhill	The Challenge of the Computer Utility. Addison Wesley ($8.00)