1971 INTERMAG PAPERS

IEEE TRANSACTIONS ON
MAGNETICS

VOL. MAG-7 NO. 4 DECEMBER 1971

IEEE TRANSACTIONS ON

MAGNETICS

DECEMBER 1971 VOLUME MAG-7 NUMBER 4

Published Quarterly

1971 INTERMAG PAPERS

Session 13—Storage Hierarchies

Storage Hierarchies: Gaps, Cliffs, and Trends (*Invited*) ... *E. W. Pugh* 810
Evaluation of Multilevel Memories (*Invited*) ... *R. L. Mattson* 814
Virtual Memory: A Combined Hardware–Software Memory System (*Invited*) (*Digest*) ... *N. Weizer* 819
The Use of Hierarchical Storage in Language-Based Multiprocessing Systems (*Invited*) (*Digest*) ... *R. V. Bock* 819
Data Base Management Systems: The Influence of Hardware (*Invited*) ... *A. H. Vorhaus* 820
The Storage Complex Considered as a System Component (*Invited*) ... *G. G. Scarrott* 824

Session 23—Bulk Storage Systems

Bulk Storage Requirements in Large-Scale Scientific Calculations (*Invited*) ... *W. J. Worlton* 830
Bulk Storage Requirements at the Social Security Administration (*Invited*) ... *J. J. Fraher* 833
Information Retrieval: United Air Lines Operations System (*Invited*) ... *J. T. Kirkpatrick* 835
Bulk Storage Applications in the Illiac IV System (*Invited*) ... *M. A. Knapp and D. E. McIntyre* 838
National Archives Mass Storage Requirements—1975 to 1980 (*Invited*) ... *G. J. Rosenkrantz* 843
Mass Storage Utility: Considerations for Shared Storage Applications (*Invited*) ... *R. B. Gentile and R. W. Grove* 848

PAPERS

Annealing Effects in Plated-Wire Memory Elements, Part I: Interdiffusion of Copper and Permalloy ... *C. I. Knudson and J. R. Kench* 852
Annealing Effects in Plated-Wire Memory Elements, Part II: Recrystallization in Permalloy Films ... *S. J. Marquardt and J. R. Kench* 858
Magnetic Stress Anisotropy Field in Plated Cylindrical Permalloy Films ... *O. S. Lutes* 861
A High-Density Coupled-Magnetic-Film Memory Array ... *F. S. Lee* 868
Self-Consistently Computed Magnetization Patterns in Thin Magnetic Recording Media ... *R. I. Potter and R. J. Schmulian* 873
The WRITE Process in Magnetic Recording ... *B. Kostyshyn* 880
Thermomagnetic Writing in Homogenous MnBi Films ... *W. K. Unger and R. Räth* 885
An Unsymmetrical Mode of Ferroresonance ... *O. M. Baycura and J. C. Donovan* 890
Magnetization, Anomalous Barkhausen Effect, and Core Loss of Supermendur Under High Temperature Cycling ... *J. M. Niedra and G. E. Schwarze* 896
Measurement of Flux Distribution in Toroidal and Multiaperture Cores ... *Y. Ohbuchi, T. Urabe, and Y. Sakurai* 899
Influence of Construction Data on the Dynamics of Dry-Reed Relays ... *W. Gruszczyński, J. Hryńczuk, and H. J. Wierzba* 905
Long-Term Stability of a Ferrite Core Flux-Gate Magnetometer in High Field (*Letter*) ... *F. Primdahl and W. R. Darker* 909
A Simple Method for Determining the Uniaxial Anisotropy Direction and Skew in Magnetic Thin Films (*Letter*) ... *E. J. Hsieh and L. J. Schwee* 910

1971 INDEX ... *Follows page* 914

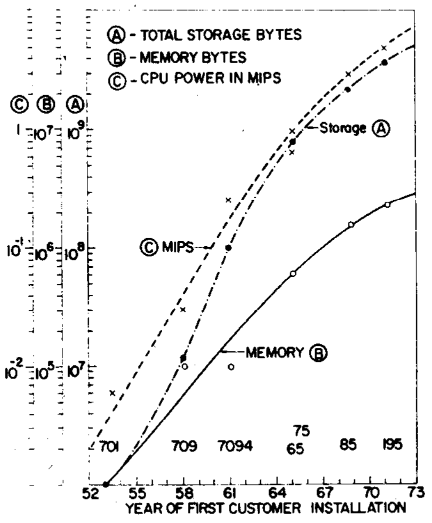


Fig. 1. Computer processing power in MIPS compared to main memory and total on-line storage capacity for large IBM computers.

prepared to tackle this problem along with their applications software development.

Compared with their predecessors, these new operating systems tended to be complex. OS/360, for example, was measured in millions of bytes of instructions, a small part of which was permanently resident in main memory while the rest was held on disk storage. Operating systems managed most system resources, established priorities, handled I/O requests, allocated space in memory, etc. In order to keep the CPU and main memory busy while waiting for information from electromechanical storage, support of multiprogramming was often provided. With this support, when a task makes reference to information not in main memory, the operating system starts the CPU working on another task already resident in main memory and simultaneously begins searching for the requested information and preparing to transfer it into main memory.

To the extent that the user can write his problem program and get his job executed without being aware of the hardware characteristics of the storage system, the storage system is said to be *transparent* to the user. Many clever schemes have been devised to permit nearly complete systems management of the storage system and transparency to the user. However, the cost/performance and certain unique characteristics of the available hardware largely determine the extent to which transparency can be achieved.

The most commonly identified hardware problem is that of the *access gap* between electronically addressable semiconductor, plated-wire, or ferrite-core memories, and electromechanically addressable storage such as magnetic drums, disks, and tapes. To distinguish between the hardware cost/performance gap and the system and software problems it presents to the user, we will refer to the latter as the *cliff* in the hierarchy.

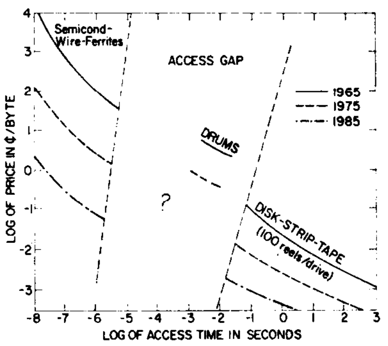


Fig. 2. Access gap in 1965 with projections to future.

Access Gap

The large difference between access time to electronically and electromechanically addressable storage has been with us about as long as computers themselves. Furthermore, projections of future technology developments suggest that this access gap will continue for some time.

The access time in seconds (scale given in powers of ten) is shown in Fig. 2 versus price per byte for storage technologies available during the mid to late 1960's. A byte is eight logical bits and is usually implemented with nine or more hardware bits for error detection and correction. Price is to the user and includes prorated cost of channel and control units (if any) required to attach the storage to the system. Prices to 1985 assume that projected technology advances are achieved and that cost savings are passed on to the user.

Dominant electronically addressable technologies of this era were ferrite-core, plated-wire, and semiconductor memories, while electromechanical technologies were magnetic tapes, disks, and drums or fixed-head files. The projection to 1975 assumes that the same technologies will continue to dominate the market, but that semiconductor memories will make significant inroads into the plated-wire and ferrite-core market. By 1985 most of the electronically addressable storage will be semiconductor, while magnetic disks and tapes will continue to be the primary form of electromechanical storage. The lack of a curve for drums or fixed-head files beyond 1975 presumes that this technology will no longer be competitive with low-cost semiconductor technology, while the large question mark suggests the possibility that a successful gap-filling technology will be achieved using magnetic bubbles, low-cost semiconductor technology, beam-addressable memory, or some approach as yet not conceived.

The better price/performance technologies in Fig. 2 are found to the lower left, while the poorer ones are to the upper right. On this basis, drums or fixed-head files are seen to provide poor price/performance compared to the other technologies available during the same era—they do, however, provide something within the access gap and therefore find a limited market where a customer requires this function. The price/performance projections of future technologies indicate that the size of the performance gap will remain nearly constant, spanning the range from 10^{-6}

to 10^{-2} s. The gap, when measured in cents per bit, may narrow from three orders of magnitude to less than two orders of magnitude if pressent cost projections of semiconductor technology are met. This would make successful introduction of a gap-filling technology more difficult and would have profound implications on future system design.

Hierarchy Cliff

The size of the cliff presented to the user by the access gap varies from system to system. In general, when a user's application is containable in main memory, his job is relatively straightforward; however, when his application must make use of electromechanical storage, he becomes concerned with the representation of data structures on storage devices whose addressing rules differ markedly from those of main memory. Frequency of access to storage devices and contention with other programs for devices must be minimized. He must select a method for structuring the data in his file, select one of the software access methods provided by the operating system, and allocate sufficient space within main memory to buffer the records transmitted between main memory and the channel attached storage. Improper handling of these functions will result in failure of the program to execute.

A clever programmer, who understands the peculiarities of the system, can dramatically improve the performance of the system for his program. For example, if he arranges accesses to disk storage so as to minimize arm motion, he may more than double the speed at which his job is completed. In a multiprogramming environment, however, his efforts may be thwarted by repositioning of the arm by an intervening task. In this case, global optimization of the resources by the operating system is in direct conflict with local optimization attempted by the user.

A revealing example of a user's concern with this cliff is that of an application which operates on an array of data too large to fit into main memory. The programmer must decide how to divide the array into parts which will be separately brought into memory. If the application requires logical operations among all (or many) parts of the array, then each of the several parts of the array may have to be loaded into memory many times before all of the logical operations can be completed. Devising an efficient algorithm for doing this, and writing the code to implement it, may be far more work than writing the basic applications program itself.

It is therefore reasonable to ask if it is possible for the user to write his program as if he had an infinitely large address space, and then let the system select the algorithms for moving parts of the program and data into and out of main memory. Techniques for doing this have in fact been devised and implemented. The very large, and physically nonexistent, memory assumed by the application program is referred to as *virtual memory*. It is worth noting that such techniques may be even more important for terminal-oriented systems in which many users are simultaneously requesting large amounts of main memory.

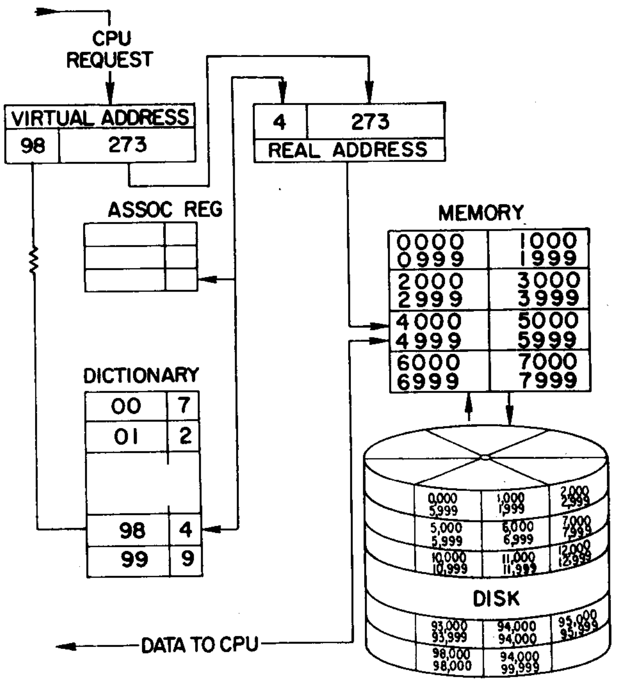


Fig. 3. Virtual addressing implemented by demand paging from disk to memory.

Demand Paging

Demand paging in conjunction with an address translation and mapping scheme is frequently used to provide large virtual memory for users on a system with relatively small real memory. A simplified version of this is depicted in Fig. 3 for illustrative purposes. The disk storage space, which represents the total virtual address space available to the users, in this example is divided into 100 pages of 1000 words each. Directly addressable memory is illustrated with eight pages of 1000 words each. The purpose of the addressing and paging scheme is to provide (as nearly as possible) the function and performance of 1000 words of main memory with 1000 words of inexpensive disk storage and only eight words of the more expensive main memory.

A CPU memory request, e.g., virtual address 98 273, is divided into its high-order (98) and low-order (273) terms. The latter are immediately placed in a register for the real address, while the high-order terms are used to address a dictionary stored in main memory. In the example given, virtual page number 98 was assumed to be already resident in page 4 of the main memory, and this information was therefore found in the dictionary. The number 98 is replaced by 4 in the real address, which is now transmitted to main memory in the usual manner to find the information stored in location 4273

If the page is not in memory, then the dictionary will so indicate. In the example, a request for a virtual address 99XXX would result in translation information 99-9, where 9 exceeds the number of pages in real memory and is used to symbolize a request for the system to move page 99 into main memory from the disk. If the main memory is full, then some page already there will have to be selected for replacement. A very common practice is to replace the page which has been least recently used, namely, the LRU algorithm. In addition to replacing the page in memory, the system must also update the entry in the dictionary.

The time required to make two memory references to locate each word can be shortened if a high-performance associative memory technology is available. In Fig. 3 three words of associative storage are assumed. When the address translation information 98-4 is found, it is placed in one of the three associative words. A subsequent virtual address anywhere in page 98XXX will quickly be translated into the real address 4XXX without reference to the full dictionary stored in main memory. Because of the small size of the associative register, some algorithm must be selected for replacing one address with another.

The performance actually achieved by demand paging depends strongly on the effectiveness of the paging algorithm for the job stream on which it is used. If the time to access and transfer data from memory is T_M and from disk is T_D, then the average time to access data from virtual memory is given by

$$T_A = T_M + (1-h)T_D$$

where h is the *hit ratio*, i.e., the fraction of address requests which are already in main memory. The term T_M as opposed to hT_M is used because it is presumed that a request not found in memory results in an access and transfer from disk to memory followed by an access and transfer from memory. For a system in which $T_D \gg T_M$, the access time is simply $T_A \cong (1-h)T_D$, and a 90-percent hit ratio reduces the average access time T_D to 0.1 T_D, and an average access time of 0.01 T_D would result for $h = 0.99$.

While demand paging was originally proposed for use across the access gap, its most successful implementation has been with the *buffer-backing store*, or *cache* memory concept, first introduced on the IBM 360/85 computer. In this computer a 0.4-μs access ferrite-core backing store of 512 000 byte or larger is teamed with 16 000 byte of semiconductor memory with about ten times faster access. Since hit ratios well in excess of 0.9 are typically achieved, the average access time of the system is

$$T_A = T_M + 0.1\ T_D \cong T_M$$

where T_M in this case is the access time to the high-speed cache and T_D is the access time to the lower speed main memory. The cost per bit of the buffer-backing store combination is approximately that of the lower cost ferrite-core memory, especially on systems having large amounts of core storage.

Three factors account for the greater success of demand paging between cache and main memory than between main memory and disk storage:

1) higher hit ratios are achieved closer to the CPU due to the greater sequentiality of the job streams at this level;

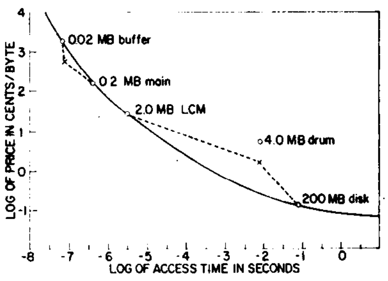


Fig. 4. Price versus access time achieved by demand paging between buffer and main memory and between large capacity memory (LCM) and disk storage using 1965–1970 technologies and assuming 90-percent hit ratio to higher performance device.

2) the performance difference to be spanned is smaller, i.e., one instead of four orders of magnitude;
3) the slope of the price-to-access time curve is steeper between cache and main memory than between main memory and disk.

The impact of factors 2) and 3) is shown in Fig. 4. Two cases are illustrated: a 0.2-Mbyte main memory buffered by a 0.02-Mbyte semiconductor cache, and a 200 Mbyte disk buffered by a 2-Mbyte ferrite-core memory. In order to eliminate sensitivity to differing hit ratios, a hit ratio of 0.9 was assumed in both cases. The X connected by dotted lines to the two parent technologies represents the resultant price/performance of the combined systems. While the price/performance of the memory-disk system for the numbers selected is superior to that of the drum, it lies well above the price/performance trend line drawn through the other storage technologies. The combined cache-memory system, in contrast, has a price/performance better than the trend line.

Off-Line Information Storage

The typical computer in the 0.05- to 5.0-MIPS range has 100 to 1000 times as much information stored in its tape library as it has on-line. By looking at Table I we can see that the low cost of off-line tape storage is the prime reason for the use of magnetic tapes. It is about 40 times cheaper than off-line disk storage and 1000 times cheaper than on-line disk storage. The relatively large cost of a tape drive combined with the relatively small capacity of the tape reel (10^7 byte) result in a cost of on-line tape storage which is about equal to that of on-line disk storage, even though the access time and flexibility of the tape system is inferior to that of the on-line disk storage.

Because the time to locate a reel and mount it on a drive is measured in minutes, accesses to off-line data is handled very differently than to on-line data. The operator is advised to mount the tapes that will be needed *before* the job is initiated, so that the access times actually experienced by the system are measured in milliseconds, or seconds at worst. The list of tapes to be mounted for each job is frequently given on the schedule which advises the operator which jobs are to be run that day. In a larger installation the schedule may be stored in the computer and even some provision made for automatic scheduling by the computer as additional jobs are placed in the queue. In such cases the operator is advised to mount tapes by the computer before execution is undertaken.

Such conventions have evolved because of the physical limitations of tape storage and have proved to be reasonably satisfactory. However, the number of tapes which must be mounted per day on a given system is clearly related to the rate at which the CPU can do work. A typical 1-MIPS processor may average 2 tape mounts per minute distributed over its 15 or so tape drives. Keeping track of this activity requires a fair amount of clerical activity, not to mention the physical activity of mounting, demounting, and storing tape reels. The time it takes to obtain and mount tapes makes a change in the order in which jobs are executed very time consuming and precludes dynamic load balancing that involves that part of the hierarchy served by magnetic tapes.

If a proportionally high mount rate is required for future computers capable of 10 to 100 MIPS, then 20 to 200 tape mounts per minute will be needed! Clearly, there is a growing importance of the gap between on-line and off-line storage in the hierarchy. Two approaches are available for solving this problem: one is to place high-usage information from the tape library onto disk storage, while the second is to devise a storage technology that permits easier access to the information now stored in tape libraries.

Evaluation of Multilevel Memories

RICHARD L. MATTSON MEMBER, IEEE

Abstract—Proposed memory hierarchy technologies and configurations are usually evaluated by repeated running of "typical" jobs through simulated hierarchies while various parameters are adjusted. Simulation is too slow to be a tool for selecting among the choices in 1) technologies to be included, 2) implementation of each technology, and 3) management of data flow in the hierarchy.

Four current hardware-managed hierarchies are described in a manner which parameterizes their design. The evaluation process is described in terms of address traces, hit ratios, and system cost performance. Stack processing is then described as a replacement for simulation that obtains hit-ratio data 1000 times faster than before. Finally, an example is given to illustrate how to select between two competing technologies, how to design the best hierarchy, and how to determine the information flow which optimizes the total cost performance of the system.

I. Introduction

Justification for Memory Hierarchies

OVER THE PAST several years there has been a substantial increase in the speed and capacity demands placed on computer memory systems. These demands cannot be fulfilled at an acceptable cost with any single current technology, but the problem can be overcome by a memory hierarchy which combines a variety of technologies with differing cost-performance characteristics. To design an optimal memory hierarchy for a given program load, one must determine the best combination of technologies to include in the hierarchy, the hierarchy parameters, and a policy for managing the flow of data in the hierarchy.

Manuscript received April 26, 1971. Paper 13.2, presented at the 1971 INTERMAG Conference, Denver, Colo., April 13–16.

The author is with the IBM Research Laboratory, San Jose, Calif. 95114.

Hierarchy Management

Management of a memory hierarchy involves determining where information is stored, how it is located, and when it should be moved. The objective is to maintain currently used data in fast and expensive devices in order to minimize the access time to the hierarchy. This requires recognizing when data is no longer needed so it can be moved to slower and cheaper devices. Typically, management policies involve partitioning data and programs into blocks and adopting some technique for moving these blocks in the hierarchy.

Hierarchy management may be performed by the user or automatically by the system. Management by the user allows him to incorporate his knowledge regarding data and program usage. However, this places an additional burden on him and reprogramming may be required if his program is run on a machine with a different hierarchy.

Automatic management relieves the user, but transfers the task to operating system programmers or hardware designers. Operating system control is easy to change when necessary, but each access to the memory system requires software intervention and serious performance degradation may result. Hardware control is inherently more efficient, but it is expensive and cannot be changed. Thus, a careful design tailored to the intended workload is required to insure that a hardware-managed hierarchy will have good cost-performance characteristics.

Hierarchy Evaluation

The problem of hierarchy evaluation is to determine the overall computing system cost performance when processing its workload. The cost-performance characteristics of the system depend on the technologies in the memory hierarchy, the hierarchy organization, and the given hierarchy management policy.

The usual approach has been to define a set of "typical" applications, run these on existing computers, trace the time sequence of memory access requests, and use this trace as input to a program which simulates the computing system with a memory hierarchy. The simulation produces the average access time to the hierarchy which can be used to obtain the processing speed of the system and, when combined with costs, the cost performance of the hierarchy. Repeated simulation with different memory technology parameters and hierarchy management policies leads to a hierarchy design with low cost performance. However, simulation is a slow and detailed process to use as a design tool. This paper discusses a new technique for the evaluation of memory hierarchies which allows for the rapid evaluation of technology and system design choices.

II. Hierarchy Management Techniques

Pages, Classes, and Data Movement

For purposes of illustration, a technique for the automatic management of two-level memory hierarchies is discussed in this section. The level closest to the central processing unit (CPU) will be called the buffer, and the other level will be called the memory. The memory and buffer are both partitioned into equal-size blocks called page frames, and are further partitioned into classes as indicated by the vertical columns in Fig. 1(a).

The CPU issues a request for data by specifying a memory address (see Fig. 1). This address specifies the desired page, the page class, and the byte within the page. All pages are initially in the memory and are moved to the buffer only when requested by the CPU. In this paper, movement is constrained so that a page cannot cross class boundaries, and when the buffer has no free space in a given class, the page removed from the buffer is the least recently used (LRU) page in that class

Management Techniques

To manage this type of hierarchy one needs 1) a directory for each class to indicate which page from the memory is in which page frame of the buffer, and 2) a priority list for each class that orders the buffer page frames in order of their availability of being emptied when space is needed. Examples of systems with four classes and one class are shown in Fig. 1(a) and (b), respectively.

Tradeoff Between Classes, Cost, and Speed

The number of classes in the memory (buffer) has conflicting effects on system cost and performance. With one class every page frame in the buffer can be fully utilized, but the directory and priority list are large. Because the directory must be scanned and the priority list updated for each CPU request, the scan and update will either take a long period of time, or require a great deal of hardware. On the other hand, with many classes the directories and priority lists for each class are small and can be searched and updated quickly and with little hardware, but, unless page requests by the CPU are uniformly distributed over all classes, the buffer space may be poorly utilized. Thus, the number of classes is a design choice that affects both cost and performance.

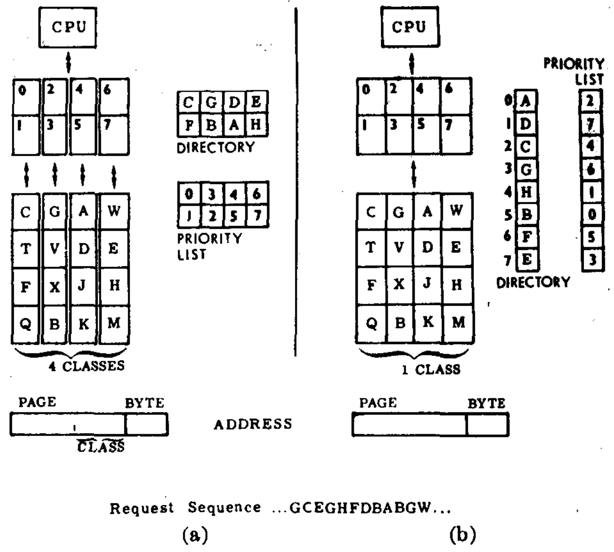


Fig. 1. Hierarchy management. (a) Four classes. (b) One class.

TABLE I
Design Parameters

System	360/85	360/195	370/165	370/155
Buffer Size (bytes)	16,384	32,768	8,192	8,192
Memory Size (bytes)	4,194,304	4,194,304	3,145,728	2,097,152
Page Size (bytes)	1024	64	32	32
Blocks per page	16	1	1	2
Classes	1	128	64	128
Buffer page Frames/Class	16	4	4	2
Memory Page Frames/Class	4096	65,536	49,152	32,768
Bits/Class for Priority List	64	5	5	1
Directory Storage (bits)	192	8,192	4,352	4,096
Priority List Storage (bits)	64	640	320	128
Block Loaded Indicators (bits)	256	0	0	512

Examples of Memory Hierarchies

In this section, four two-level memory hierarchy organizations are discussed that are hardware managed and are transparent to the user and system programs which use them. These correspond to the IBM computer systems 360/85, 360/195, 370/165, and 370/155, the parameters of which are listed in Table I.

The operation of these systems is similar in that each time the processor makes a memory access, the memory system identifies the page class, checks the directory for that class to see if space has been allocated for the desired page, and checks to see if the data is in the space allocated. If the data is in the buffer, processing continues. If space has been allocated, but the data is not there, data is moved to the buffer. If space has not been allocated, then space is allocated for the entire page, and part or all of the page is moved to the buffer.

Each time the processor stores information, it is stored in the memory. If the buffer also contains the data, it is stored in the buffer also. When all buffer page frames in a given class have been allocated and space is needed, the least recently accessed page frame is allocated to the new page (LRU replacement).

In the IBM 360/85 the operation is as illustrated in Fig. 1(b), with the parameters given in Table I. The 360/85 differs from the other system in that only 1/16 of the page is loaded on demand in order to make the page wait-time small. Space for the remainder of the page is reserved and other parts can be quickly loaded if needed.

In the IBM 360/195, 370/165, and 370/155 the operation is as illustrated in Fig. 1(a). In the 360/195 and 370/165 entire pages are loaded into the buffer on demand. In the 370/155 only 1/2 pages are loaded into the buffer in a manner similar to that in the 360/85.

III. Evaluation Methods

Hit Ratio

In two-level hierarchies, if the requested memory access is found in the buffer it is called a hit. The hit ratio is defined as the ratio of the total number of hits to the total number of address accesses. This hit ratio can be used to compute the average time between hierarchy accesses from the equation

$$T_{\text{ave}} = t_1 + (1 - f)t_2 \tag{1}$$

where f is the hit ratio, t_1 is the average time between memory requests, and t_2 is the time required to get the requested data into the buffer and ready to be used by the CPU. The times t_1 and t_2 must include directory search time and directory and priority list update times.

Cost Performance

A common cost-performance measure of a system is dollars times seconds per access; it can be computed by 1) computing the average time (in seconds) between hierarchy responses (from (1)), 2) computing the total system cost including the CPU, buffer, directory, priority list table, and main memory, and 3) multiplying the average seconds per access by the system cost to obtain the dollars times seconds per access for the system. If the buffer capacity is increased, the average time between hierarchy responses goes down, but the cost goes up, so that to balance the cost performance of the system one needs the hit ratio for all possible buffer capacities, page sizes, and numbers of classes.

Simulation

The memory hierarchy systems discussed in Section II represent only a few of the possible organizations. In the

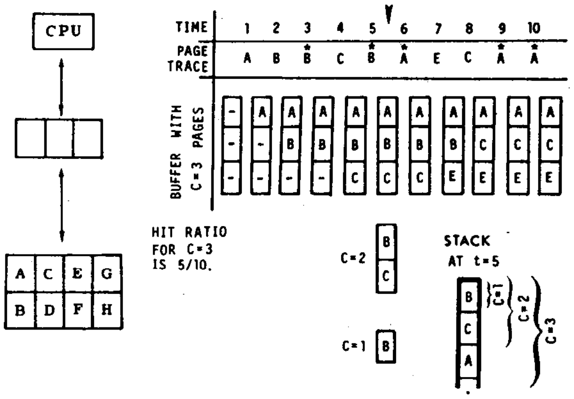

CPU
A C E G
B D F H
TIME 1 2 3 4 5 6 7 8 9 10
PAGE TRACE A B B* C B* A* E C A* A*
BUFFER WITH C=3 PAGES
HIT RATIO FOR C=3 IS 5/10.
C=2
C=1
STACK AT t=5
C=1
C=2
C=3

Fig. 2. Simulation versus stack for LRU.

systems shown the design variables were 1) the cost and speed of the CPU, 2) buffer technology, 3) memory technology, 4) buffer capacity, 5) page size, 6) number of blocks per page, 7) number of classes, 8) buffer page frames per class, 9) replacement algorithm (LRU in the examples), and 10) method of handling stores. If only two alternatives of the above ten variables are examined, then 1024 simulations per typical application would be required to determine the cost-performance tradeoff between the choices examined.

Stack Processing

The stack processing technique [1] discussed in this section has been measured to be over 1000 times faster than two-level hierarchy simulation, and thus makes hit-ratio data easy to obtain on a variety of hierarchy configurations. The technique is illustrated in Fig. 2, where pages are labeled with capital letters and the contents of a three-page frame buffer are shown as they would be after each page request (LRU replacement is used). In this example there are five hits (indicated by asterisks) in ten requests, so the hit ratio is 0.5.

The preceding process could be repeated for buffers with one or two page frames. If this were done, the contents of these buffers after the fifth time interval would be as shown in Fig. 2.

The observation that led to the development of stack processing is that for many replacement algorithms, at each instant of time and for any sequence of address requests, the contents of a buffer with k page frames is included in the contents of every buffer with more than k page frames. This allows one to define a list of pages called a "stack," where the top k entries of the stack represent the pages which would be included in a buffer with k page frames at that particular instant of time. A single stack can be used to represent the buffer contents for many different hierarchy configurations, as illustrated in Fig. 3.

The significance of the stack is that hit-ratio data can easily be obtained from it. The stack that would exist at each instant of time is illustrated in Fig. 4 for LRU replacement. For each reference a stack distance is defined as the distance from the top of the stack to the

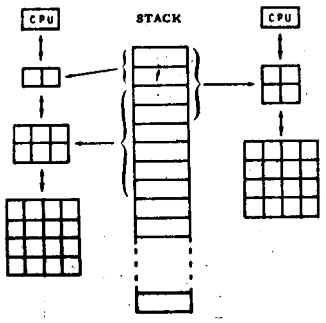
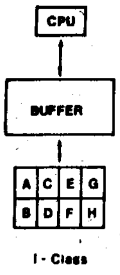


Fig. 3. Single stack represents buffer contents for many hierarchy configurations.



Time	1	2	3	4	5	6	7	8	9	10
TRACE	A	B	B	C	B	A	E	C	A	A
STACK	A	B	B	C	B	A	E	C	A	
	-	A	A	B	C	B	A	E	C	
	-	-	-	A	A	C	B	A	E	
	-	-	-	-	-	-	C	B	B	
Distance	-	1	-	2	3	-	4	3	1	

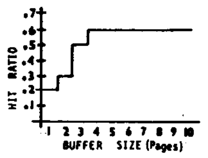


Fig. 4. Stack processing hit ratios.

TABLE II
HIT-RATIO DATA

Buffer Capacity (bytes)	Page Size (bytes): 16	32	64	128	256	512	1024	Classes
1024	0.894	0.915	0.928	0.924	0.884	0.792	0.495	1
	0.895	0.916	0.921	0.904	0.791	-	-	4
	0.891	0.903	0.860	-	-	-	-	16
	0.857	-	-	-	-	-	-	64
	-	-	-	-	-	-	-	256
2048	0.931	0.949	0.957	0.958	0.950	0.912	0.823	1
	0.931	0.948	0.954	0.955	0.933	0.808	-	4
	0.930	0.943	0.943	0.909	-	-	-	16
	0.921	0.913	-	-	-	-	-	64
	-	-	-	-	-	-	-	256
4096	0.951	0.969	0.973	0.978	0.977	0.966	0.939	1
	0.955	0.969	0.973	0.977	0.974	0.951	0.834	4
	0.955	0.968	0.972	0.970	0.933	-	-	16
	0.955	0.963	0.948	-	-	-	-	64
	0.934	-	-	-	-	-	-	256
8192	0.977	0.986	0.988	0.985	0.987	0.987	0.984	1
	0.981	0.986	0.988	0.986	0.987	0.985	0.965	4
	0.981	0.985	0.988	0.987	0.983	0.954	-	16
	0.979	0.984	0.985	0.974	-	-	-	64
	0.974	0.971	-	-	-	-	-	256
16384	0.985	0.993	0.994	0.996	0.993	0.992	0.994	1
	0.990	0.993	0.994	0.996	0.994	0.992	0.993	4
	0.990	0.994	0.995	0.997	0.995	0.991	0.957	16
	0.990	0.994	0.995	0.995	0.985	-	-	64
	0.989	0.992	0.986	-	-	-	-	256
32768	0.989	0.996	0.997	0.997	0.999	0.994	0.997	1
	0.994	0.996	0.997	0.998	0.998	0.996	0.997	4
	0.994	0.996	0.998	0.998	0.998	0.997	0.997	16
	0.994	0.996	0.998	0.998	0.998	0.988	-	64
	0.994	0.996	0.997	0.992	-	-	-	256

TABLE III
TECHNOLOGY ASSUMPTIONS

CPU Cost $100,000
Buffer Technology 60ns cycle @ 50¢/bit
Memory Technology (A) 1,000ns cycle @ 3¢/bit (B) 8,000ns cycle @ 0.5¢/bit

TABLE IV
TIMING AND COST ASSUMPTIONS

(A) Symbols

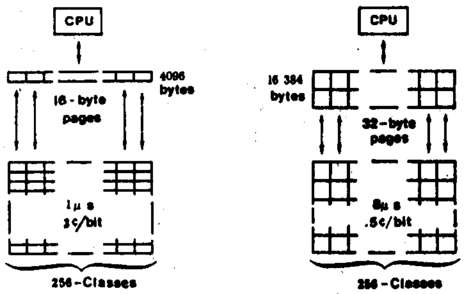
Page Size (bytes)	P	2^i, i = |4,...,10|
Buffer Size (bytes)	C =	2^j, j = |10,...,15|
Memory Size (bytes)	262,144	
Number of Classes	K	2^k, k = |0,2,4,6,8|
Number of Page Frames	PF	C/P
Number of Frames per Class	FPC	PF/K
Bits of Directory Storage	DS	(18 - i) PF
Bits of Priority List Storage	PS	0 if FPC = 0 K if FPC = 2 5K if FPC = 4 DS if FPC > 4

(B) Timing

Directory Search Time	DST	(30 ns) FPC
Priority List Update Time	PLU	60 ns if FPC ≤ 4 DST otherwise
Data Access Time	DAT	60 ns
Memory Cycle Time	MCT	1μs or 8μs
Page Move Time	PMT	(P/16) MCT
Priority List Check and Update	PLC	60 ns if FPC ≤ 4 FPC·60 ns otherwise
Directory Update Time	DUT	60 ns
Ave. Mem. System REsponse Time	T (ave)	$T_1 + (1 - f) T_2$
where T_1 = DST + PLU + DAT, and T_2 = PLC + PMT + DUT		

(C) System Cost (CPU, Buffer, Memory, Directory, Priority List)

CPU Cost	$100,000
Buffer Cost	$ 4·C
Directory Cost	$ 0.25 DS
Priority List Cost	$ 0.25 PS
Memory Cost	$ 60,000 with 1μs or $10,000 with 8μs



Hit Ratio	.934	Hit Ratio	.992
T_1	150 ns	T_1	180 ns
T_2	1120 ns	T_2	8120 ns
T(ave)	224 ns	T(ave)	244 ns
CPU	$100 000	CPU	$100 000
Buffer	16 384	Buffer	65 536
Memory	62 915	Memory	10 486
Directory	896	Directory	1 664
Priority List	-	Priority List	64
TOTAL	$180 195	TOTAL	$177 750

Fig. 5. Technology selection and system design.

TABLE V
COST-PERFORMANCE DATA[a]

Buffer Capacity (bytes)	PAGE SIZE (bytes)														
	16		32		64		128		256		512		1024		
	(1)	(8)	(1)	(8)	(1)	(8)	(1)	(8)	(1)	(8)	(1)	(8)	(1)	(8)	Tech
1024	0.74	0.59	0.39	0.40	0.23	0.39	0.20	0.62	0.35	1.73	1.15	6.13	5.43	29.66	1
	0.21	0.23	0.13	0.22	0.09	0.32	0.16	0.72	0.59	3.09	--	--	--	--	4
	0.06	0.13	0 06	0.20	0.12	0.53	--	--	--	--	--	--	--	--	16
	0.05	0.15	--	--	--	--	--	--	--	--	--	--	--	--	64
	--	--	--	--	--	--	--	--	--	--	--	--	--	--	256
2048	1.44	1.06	0.72	0.59	0.38	0.41	0.24	0.45	0.24	0.83	0.52	2.70	1.98	10.81	1
	0.38	0.32	0.20	0.23	0.13	0.24	0.10	0.37	0.22	1.04	1.08	5.86	--	--	4
	0.11	0.14	0.06	0.14	0.07	0.24	0.15	0.71	--	--	--	--	--	--	16
	0.05	0.10	0.06	0.19	--	--	--	--	--	--	--	--	--	--	64
	--	--	--	--	--	--	--	--	--	--	--	--	--	--	256
4096	2.94	2.13	1.45	1.08	0.74	0.62	0.39	0.44	0.25	0.50	0.29	1.17	0.75	4.01	1
	0.75	0.57	0.38	0.32	0.21	0.24	0.13	0.26	0.12	0.45	0.32	1.62	1.94	10.82	4
	0.20	0.18	0.11	0.14	0.06	0.14	0.08	0.27	0.22	1.11	--	--	--	--	16
	0.05	0.08	0.05	0.10	0.07	0.23	--	--	--	--	--	--	--	--	64
	0.04	0.09	--	--	--	--	--	--	--	--	--	--	--	--	256
8192	6.28	4.65	3.09	2.30	1.55	1.18	0.80	0.70	0.43	0.52	0.28	0.62	0.31	1.25	1
	1.58	1.18	0.79	0.61	0.40	0.34	0.22	0.28	0.15	0.31	0.14	0.60	0.47	2.56	4
	0.41	0.32	0.21	0.18	0.12	0.13	0.07	0.15	0.09	0.33	0.32	1.71	--	--	16
	0.11	0.11	0.05	0.09	0.07	0.09	0.07	0.26	--	--	--	--	--	--	64
	0.04	0.06	0.04	0.09	--	--	--	--	--	--	--	--	--	--	256
16384	14.71	11.45	7.20	5.59	3.58	2.79	1.79	1.42	0.92	0.85	0.51	0.70	0.32	0.71	1
	3.67	2.87	1.81	1.42	0.91	0.73	0.46	0.40	0.26	0.32	0.18	0.44	0.16	0.67	4
	0.93	0.74	0.47	0.38	0.24	0.21	0.13	0.13	0.07	0.16	0.11	0.46	0.67	3.94	16
	0.25	0.20	0.13	0.12	0.06	0.07	0.05	0.09	0.09	0.36	--	--	--	--	64
	0.06	0.06	0.05	0.06	0.05	0.10	--	--	--	--	--	--	--	--	256
32768	38.31	31.81	18.61	15.38	9.19	7.60	4.58	3.80	2.29	1.93	1.21	1.31	0.65	0.91	1
	9.55	7.94	4.67	3.87	2.32	1.93	1.16	0.99	0.59	0.53	0.34	0.51	0.22	0.51	4
	2.40	2.01	1.18	0.99	0.59	0.50	0.31	0.27	0.17	0.18	0.10	0.25	0.11	0.44	16
	0.62	0.52	0.31	0.27	0.16	0.15	0.07	0.08	0.06	0.10	0.16	0.78	--	--	64
	0.17	0.15	0.07	0.07	0.06	0.07	0.06	0.16	--	--	--	--	--	--	256

[a] Dollars times seconds per access.

requested page. The distance recorded is the minimum number of page frames required in a buffer so that a hit would occur. (If the page is not in the stack, no distance is recorded.) The complete technique for maintaining the stack is given in [1].

The hit ratio for a buffer of k page frames is simply the number of times that a distance k or less was recorded divided by the total number of accesses. Using this stack processing technique, hit-ratio data can be obtained for buffers with any number k of page frames in just one pass of the page trace. It can be shown [1] that for LRU replacement simple extensions of this procedure can be used to simultaneously obtain hit-ratio data for all page sizes and any number of classes in the same pass of the address request sequence. Such hit-ratio data can be tabulated as shown in Table II.

IV. System Design and Technology Selection

Stack processing offers the ability to obtain hit-ratio data on hierarchy configurations with different capacities, page sizes, and numbers of classes. The purpose of this section is to illustrate by example how the techniques described in Section III can be used for system des gn and technology selection.

The hit-ratio data given in Table II, the technology parameters given in Table III, and the timing and cost assumptions given in Table IV will be used to calculate the cost performance in dollars times seconds per access for each of the possible designs using the 1- and 8-μs memories. These results are tabulated in Table V, showing the dollars times seconds per access for each configuration. The "best" hardware-managed hierarchy for this workload and each memory technology is illustrated in Fig. 5.

V. Conclusions

Many new memory and storage technologies are currently in the early stages of development. Selection of the best technologies to pursue depends on determining their relative ability to fulfill a total system need. Exhaustive simulation, the only previously known evaluation method, is not a powerful enough tool to thoroughly examine two- or more level memory hierarchies. In addition, since simulation required long processing times, job selection was limited and had to be carefully done.

With the stack processing technique, system performance can be evaluated in detail for many more job types. Furthermore, it is shown in [1] that the same data as presented in Table II can be used to evaluate hierarchies with three or more levels containing different page sizes at each level. Thus technologies and system designs which fill the gap between electronically accessed memory and mechanically accessed storage can be evaluated as illustrated in Section IV.

References

[1] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, 1970, no. 2.

[2] A. Opler, "Dynamic flow of programs and data througl. hierarchical storage," in *1965 Information Processing, Proc. IFIP Congr.*, vol. 1. Washington, D. C.: Spartan, 1965, pp. 273–276.

[3] C. J. Conti, "Concepts for buffer storage," *IEEE Comput. Group News*, vol. 2, Aug. 1969, pp. 9–13.

[4] R. L. Mattson and J. P. Jacob, "Optimization studies for computer systems with virtual memory," in *1968 Information Processing, IFIP Congr. Booklet I*, pp. 47–54.

[5] J. Fotheringham, "Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store," *Commun. Ass. Comput. Mach.*, vol. 4, 1961, no. 10, pp. 435–436.

[6] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H.

Sumner, "One-level storage system," *IEEE Trans. Electron. Comput.*, vol. EC-11, Apr. 1962, pp. 223–235.
[7] M. H. J. Baylis, D. G. Fletcher, and D. J. Howarth, "Paging studies made on the I.C.T. Atlas computer," in *1968 Information Processing, IFIP Congr. Booklet D*, pp. 113–118.
[8] D. H. Gibson, "Considerations in block-oriented systems design," in *1967 Spring Joint Computer Conf., AFIPS Conf. Proc.*, vol. 30. New York: Academic Press, 1967, pp. 75–80.
[9] S. J. Lipta, "Structural aspects of the system/360 model 85: II the cache," *IBM Syst. J.*, vol. 7, 1968, no. 1, pp. 15–21.
[10] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, 1966, no. 2, pp. 78–101.
[11] D. J. Kuehner and B. Randell, "Demand paging in perspective," in *1968 Fall Joint Computer Conf., AFIPS Conf. Proc.*, vol. 33. New York: Academic Press, 1968, pp. 1011–1018.
[12] P. J. Denning, "The working set model for programming behavior," *Commun. Ass. Comput. Mach.*, vol. 11, 1968, no. 5, pp. 323–333.

Virtual Memory: A Combined Hardware–Software Memory System

NORMAN WEIZER

Virtual memory concepts as they relate to providing a device independent, seemingly single-level memory system are discussed. The hardware–software virtual memory approach to hierarchical memory systems is placed in perspective with the more familiar hardware-only techniques now in use or being discussed. The advantages and disadvantages of the various systems are discussed as well as the complexity of the hardware needed to implement the various techniques.

First, the basic concepts of a virtual memory system are described. Second, the current state of the virtual memory art including some of the implementations for the third and fourth generation systems are described. Then the virtual memory approach is compared with various hardware-only approaches of producing memory hierarchies. Finally, the future of the virtual memory technology in view of the memory advances which are currently being reported are discussed.

Paper 13.3, presented at the 1971 INTERMAG Conference, Denver, Colo., April 13–16.
The author is with Computer Systems Division, RCA Corporation, Camden, N.J.

The Use of Hierarchical Storage in Language-Based Multiprocessing Systems

R. V. BOCK, MEMBER, IEEE

In the design of large computer systems the use of virtual memory is a means of improving the system cost–performance ratio. Virtual memory is taken in this paper as any hierarchy of memories for which the system provides the mapping of information at each level, and controls the transfer of information between levels. The use of the word system is meant to imply either the hardware, firmware, or the operating system software. Within this framework the paper describes first how parts of the memory hierarchy can be simplified and reduced in size, and then how the design of the hierarchy is complicated by the existance of multiple processors in the system.

Processors that are designed to interpret structured higher level languages tend to restrict the range of addresses that can be generated by a processor at any given instance in time. This fact can be used to great advantage in virtual memory design. The increased knowledge about those items currently addressable and therefore most likely to be used by the processor allows the designer to use a smaller capacity buffer memory and simpler control algorithms. This paper will describe aspects of language-based processor design that can be used in simplifying virtual memory design.

When using virtual memory techniques the access time of the highest level in the hierarchy is generally assumed to match the logic speed of the processor. This usually means a very close coupling between the processor and the memory. In systems with more than one processor, each with its own high-speed buffer, problems can result if information is shared. The problem of shared data is considered and several solutions are described.

Paper 13.4, presented at the 1971 INTERMAG Conference, Denver, Colo., April 13–16.
The author is with the Burroughs Corporation, Paoli, Pa.

Data Base Management Systems: The Influence of Hardware

ALFRED H. VORHAUS

***Abstract*—The technology of software of data base management systems is described, and the influence that hardware has had on such systems in the past is discussed. In addition, an attempt is made to highlight some hardware advances that are awaited eagerly by software designers.**

Definitions

COMMUNICATION difficulties arise when a software designer presents his thoughts to those who are hardware oriented. In an attempt to circumvent these problems terms will be defined and used herein in accordance with the given definitions.

It is necessary to differentiate between logical data structure and storage structure. *Logical data structure* will be used to express the collection of data as viewed or interpreted by a system user. This is an external view. The *storage structure,* on the other hand, gives an internal view. It is the collection of data as it is physically stored within the computer. These terms are easily confused. In some systems the structures are identical, although the trend today is toward having two discrete ones. By having two separate structures, the user is permitted to interact with the data in terms that are independent of the manner in which the data are stored physically.

The terms used in discussing the logical data structure are as shown in Table I. The word schema might have been included after each of these terms because that is what is meant in these definitions—the associations or logical relationships—not the actual values. An *item* is the elementary data structure from which all other data structures are constructed. Items are generally associated with the attributes of an entity. For example, the entity ORGANIZATION might have attributes of organization code, organization name, and budget. Each of these could be an item. All data base management systems permit item schemas to have attributes and generally require that items have names and types. Item types can be classified into numeric (integer, fixed decimal, and floating), string (an ordered set of character symbols), and other (date, coordinate, Boolean). In the example the organization name item is called TITLE and would have a type of string; the item named BUDGET would be numeric in type. Other item attributes variously available in today's systems include synonyms, units of measure, usage, input/output conversion and editing, access lock, and value limitations such as length, character arrangements, uniqueness, ranges, and lists of discrete values.

Manuscript received April 30, 1971. Paper 13.5, presented at the 1971 INTERMAG Conference, Denver, Colo., April 13–16.
The author is with the MITRE Corporation, McLean, Va. 22101.

TABLE I
LOGICAL DATA STRUCTURE

ITEM	– ELEMENTARY DATA STRUCTURE
GROUP	– ASSOCIATION OF ITEMS AND GROUPS
ENTRY	– A SPECIAL ASSOCIATION OF GROUPS
FILE	– A SET OF ENTRIES
DATA BASE	– A SET OF FILES

TABLE II
STORAGE STRUCTURE

ITEM INSTANCE	– FIELD*
GROUP INSTANCE	
ENTRY INSTANCE	– RECORD*
FILE INSTANCE	– DATA SET*
DATA BASE INSTANCE	

*COMMONLY USED EQUIVALENT TERMS

A *group* is an associated set of items and possibly other groups. A *compound group* contains both items and groups; a *simple group* contains only items. The given example might have a simple group called JOBS with items job code, authorized quantity, and authorized salary and a compound group about the persons in the organization with items of NAME, SEX, SALARY, and groups of SKILLS and BIRTH. Having groups within groups provides a method of establishing hierarchic relationships to be discussed later. Groups may be *repeating* or *nonrepeating* depending upon whether or not more than one occurrence of the item values is permitted. For example, the BIRTH group containing items BIRTH YEAR, BIRTH MONTH, and BIRTH DAY would generally be considered a nonrepeating group since a person is only born once. However, if you really believe in reincarnation and could identify a person's previous birth dates, then BIRTH could become a repeating group like SKILLS which definitely can have more than one occurrence for each person.

Entry is similar to group except that an entry is not contained in or subordinate to any other group. The entry is used to represent the major entities of an application—organizational units in the given example.

A *file* is a set of entries. Many data base management systems do not permit more than one entry schema in a file schema. In that case the two terms are indistinguishable. Some systems do permit multiple entry types within a file and both file and entry are also useful in talking about storage structures. *Data base,* a set of files together with any interfile associations specified, completes the list of logical data structure term definition.

The terms for storage structure are derived by adding "instance" to the terms of logical data structure as shown in Table II. An *instance* (or "occurrence") refers to the actual value or appropriate collection of values as stored on some physical medium.

Logical Data Structures

To return to logical data structures, those in use today cover a wide range of complexity. The first of three to be described is the simplest. An example of this tabular array is shown in Fig. 1. The entry schema fits the definition for a simple nonrepeating group. Each organizational entity has one and only one ORG CODE, TITLE, BUDGET, and ORGANIZATION to which it reports. This type of structure is easy to comprehend, but it is limited in utility. Most applications happen to have much more complex data relationships.

The most popular logical structure in use today is the hierarchical tree structure. Fig. 2 shows most of the features of such a structure, although they can be more or less complicated than the one shown. The boxes with extra vertical lines at each end indicate that the item or group represented may repeat, i.e., it may have zero or more occurrences for each occurrence of the group (or entry) which contains its parent. The relationships among the groups and items in this structure are traditional for a tree type structure. They are all of the parent/dependent form such that a group may have any number of dependent groups and/or items, but each item or group must have one and only one parent (group or entry). This hierarchical structure often fits neatly into real life situations. Fig. 2 has shown three levels of hierarchy. Each organization has, among other attributes, a variable number of persons, and each of these persons may have a variable number of skills. Until recent years most of the logical data structures that were implemented were limited to two levels of hierarchy, and the relationships between actual data values were relatively easy to understand. Newer systems, however, allow for many levels of hierarchy. The complexity that such systems allow may make the actual data relationships extremely difficult to comprehend and communicate. For example, you could determine the number of years experience as a plating operator of those persons having any experience at all as plating operators who are in organizations that have authorized jobs for machine operators only if the system permits you to express it explicitly.

It gets worse. The most complex structure, capable of handling virtually anything, is the plex structure (see Fig. 3). Relationships are not restricted to parent/dependent but can be anything the user requires. Three plex type relationships are indicated in the figure. The circular associations "reports to" and "subordinates are" mean that an organizational unit could report to many other organizational units and even (if it is permitted) could report to itself. The "is filled by" association relates individuals to authorized jobs. Indiscriminate use of the plex structure with large data volume can lead to uncontrollably complex problems. However, there are situations where special uses of plex structure relationships are applicable and reasonable. The tabular array, hierarchical tree, and plex structures represent a continuum of sophistication in logical data structures.

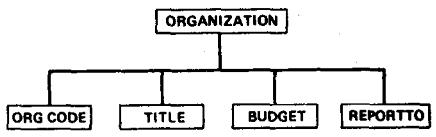


Fig. 1. Tabular array.

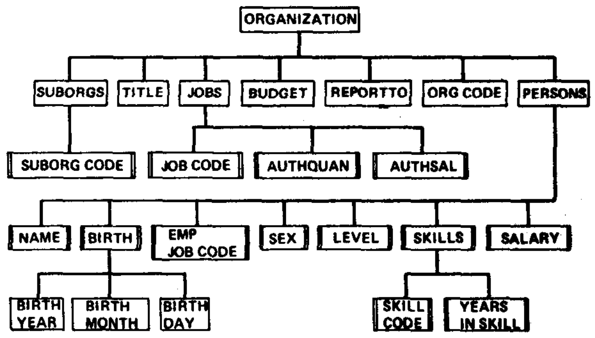


Fig. 2. Hierarchical tree.

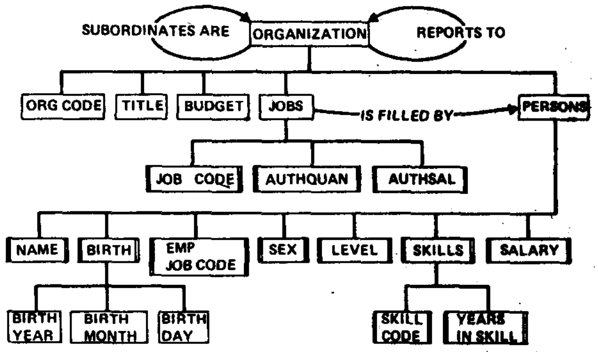


Fig. 3. Plex structure.

Storage Structures

Over time, system designers have implemented these logical data structures by using a variety of storage structures. The first techniques to be developed were worked out on card machines—sorters, collators, and accounting machines. Every item has a fixed set of columns on a card. This technique, described by the name "fixed field formatted file system," handles tabular array structures as shown in Fig. 4. The organization code is in columns 1–4, the title in columns 5–24, budget in 25–31, and columns 32–35 hold the code of the parent organization. This file may be ordered easily on the card sorter by ORG CODE, alphabetically on the first six characters of the title, by size of the budget, or into groups by parent organization. Changes are made merely by replacing a card with a new one. The accounting machine can be used to print reports, including totals.

1—4	5 —— 24	25—31	32—35
ORG CODE	TITLE	BUDGET	REPORTTO

Fig. 4. Fixed fields on cards.

1 4	5 24	25 31	32 36
ORG CODE	TITLE	BUDGET	REPORTTO

36—39	40—43	44—47	48—51	52	53-56	57-59	60-65
SUBORG1	SUBORG2	SUBORG3	SUBORG4	*	JOB CODE	AUTH QUAN	AUTH SAL

*JOBS CARD NUMBER

Fig. 5. Fixed fields with hierarchy.

Hierarchy presents a problem, but that can be handled in several ways. Fig. 5 indicates items added to the card. Four-column fields have been set aside for a maximum of four subordinate organization code numbers. If there are less than four, the columns are left blank. The JOBS repeating group is handled by setting aside columns 53–65 for the three items and adding a one-column card number so that each job code authorized would be on a separate card. The ORG CODE would have to be repeated to keep the cards in order properly, but the rest of the items could have blanks for a value on the extra cards. This limits the number of different job codes that could be authorized, but in this case the limit would never be reached. The literature is full of descriptions of methods of fitting what is essentially hierarchical data into 80-column cards. That 80-column limit and the fixed-field formatted file it spawned do present some nasty problems. Twenty columns are set aside for the title of the organization, and that becomes an absolute limit. Anything bigger than that is going to get chopped off. There are innumerable files that were designed with limits like the 999 placed on authorized quantity in the example. If that needs to be increased, all of the cards have to be completely redone. This happens—often.

Some of the limitations of cards were mitigated with the advent of magnetic tape units. No longer was there an 80-column restriction. Physical records could be any length desired. Items could be variable in length if an additional item giving the character count was included or the variable length item was followed by a special terminating symbol. The biggest change was speed. Tape READ/WRITE speeds are several hundred times as fast as card READ/PUNCH speeds. The main effect of this speedup was that much bigger data aggregates could be handled. The basic principles of formatted file systems changed little, but usage increased, leading to interest in generalized data base management systems. The possibility of considerable economy in the implementation of varied applications has maintained the interest in developing generalized systems.

Shortly after tape systems became popular, disk memories (or, more properly, rotating random access devices) appeared on the scene. It was easy to convert the tape oriented formatted file processing systems to disks, but this represented little value because disks used serially were no faster than tapes.

With the advent of random accessing, methods of indexing became a prime concern of software designers. Most files have one or a combination of items which provide unique key values for each entry or group. The problem of indexing is to make these unique keys translate into disk file addresses. If the indexes or keys are limited in number and spaced evenly without gaps and the entry or group instances are fixed in size, solution is easy. An algebraic one-to-one transformation can be found from index to disk address. This situation, however, is rare. Even a company that assigns employee numbers in a sequential fashion finds that the set is not limited. Entries must be maintained for departed employees or gaps develop. Software solutions include index tables and hash-coding. The index tables or directory method usually involves considerable storage since each index must be included together with the corresponding disk address. When entries are large in number and small in size, it is often helpful to arrange the directories in a hierarchy to minimize search time and core storage needs. In essence, this is what indexed sequential access is all about. Hash-coding is simply a method of transforming the index algebraically to a new number that can be used for direct accessing. The simplest method of hash-coding is to divide the index by the number of slots and use the remainder, as shown in the following example.

Example of Hash-Coding: We are given 10 records per track and 10 tracks per cylinder; therefore, 200 cylinders = 20 000 possible slots. An index 98 798 223 is divided by the possible solts as follows:

$$\frac{98\,798\,223}{20\,000} = 4939 \text{ plus } 18\,223 \text{ remainder.}$$

Thus the index 98 798 223 converts to

cylinder 182 track 2 record 3.

The problem, of course, is that there are multiple indexes that convert to the same disk address. Complicated provisions must be made to handle the overflow situations that can occur. In actual practice hash-coding algorithms are chosen very carefully to increase the probability of an even distribution of the codes and therefore minimize the overflow.

There have been hardware attempts to solve the indexing problem, too. Most successful has been the use of automatic hardware search for special disk records sometimes called "key parcels." This method has the advantage of permitting the central computer to continue processing while the disk is independently searching and positioning for data access. Total disk use time may increase, but if processing can be overlapped with input/output and transfers can be foreseen within the program, wasted input/output time and disk delays can be virtually eliminated.

All of these things have been attempts to convert old tape ideas to disk. New types of storage structures which would never work on tape are lists, multilists, ring structures, and inverted files. Fig. 6 shows the salient features of a list structure. Physical ordering of the data can be arbitrary as each entry instance contains a pointer with the address of the next entry instance in the desired order. Inserting an entry instance between B and C in the example simply involves changing the pointer in B and having the

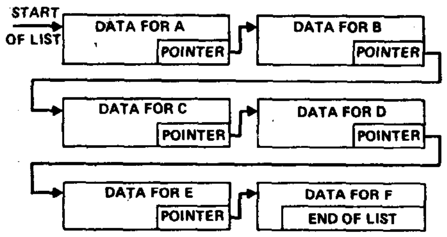


Fig. 6. List structures.

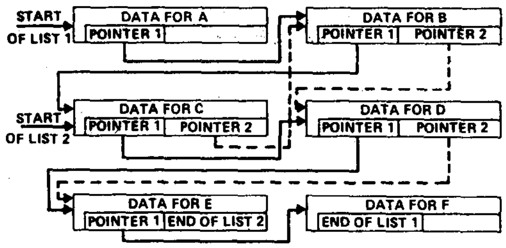


Fig. 7. Multilist structure.

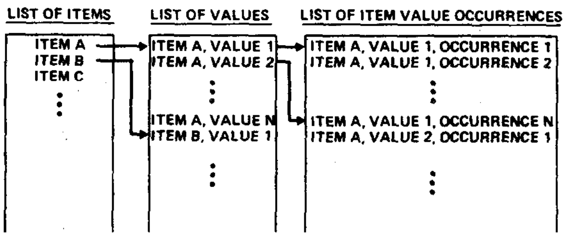

LIST OF ITEMS
LIST OF VALUES
LIST OF ITEM VALUE OCCURRENCES
ITEM A
ITEM B
ITEM C
ITEM A, VALUE 1
ITEM A, VALUE 2
ITEM A, VALUE N
ITEM B, VALUE 1
ITEM A, VALUE 1, OCCURRENCE 1
ITEM A, VALUE 1, OCCURRENCE 2
ITEM A, VALUE 1, OCCURRENCE N
ITEM A, VALUE 2, OCCURRENCE 1

Fig. 8. Inverted file structure.

new entry instance point to *C*. Deletions are equally easy to handle. Multilists differ only in that more than one list is possible (see Fig. 7). The lists need not contain the same entry instances and every entry instance is eligible to be in any list. This necessitates added care when deleting an entry instance from one list. Some list structure implementations permit backward pointers in addition to the forward pointers so that it is not always necessary to start at the beginning. If the last entry instance points back to the first entry instance in the list, a complete ring—or ring structure—is formed. These ring structures are extremely flexible and can be used to handle all types of logical data structures including hierarchical tree and plex. An advantage of list type structures is that updating is relatively easy. However, extra storage space is required for the pointers, and some retrievals can be very slow.

A simplified partial view of an inverted file structure is shown in Fig. 8. Essentially, the data are rearranged into a list of occurrences for each value for each item. The list of items and list of unique values per item permit rapid selection of the desired sections of the list of item value occurrences. Any or all items can be inverted. The advantage of this structure is the rapid retrieval that is possible for previously unplanned queries. The disadvantages are the extra storage required (for usually the data are maintained simultaneously in more conventional form) and the difficulties of keeping the lists correct when updating. Implementation of inverted file structures has demonstrated their feasibility for logical structures as complex as the hierarchical tree.

Challenges for Hardware Innovation

The variations of storage structures in use today are too numerous to mention. Nor is there space to discuss the special problems attendant with nondisk bulk storage devices like magnetic tape loops, magnetic cards, and optical stores. All of these have interesting properties of their own, but they really do not change the basic limitations that system designers are facing. Bigger, faster, and cheaper storage is needed. As long as mechanically aided rotating devices are used, little improvement can be anticipated. There are too many physical limitations. The ideal solution might be a plug-in cube, about the size of a child's building block, holding millions of characters each randomly addressable, transferring character streams in microseconds, and all at a cost of pennies per thousand characters. Some such device will be built by the hardware specialists someday.

Finally, most of the storage devices in use have fixed length tracks containing exactly so many characters each, no more, no less. Virtually all of the data in this world are variable in length. The city you live in could be ROME, CHICAGO, or LOS ANGELES, but the space allocated for that in most systems is not 4, 7, or 11 characters, respectively. It is either 3 for some code like LAX or perhaps 19 which is enough for almost any place, unless you happen to live in the little Texas town of DALWORTHINGTON GARDENS. There are innumerable software methods to fit variable length data into the fixed length tracks, but all those methods are costly in storage and time. Storage devices that will handle variable length data are a must for the future.

Two other hardware inventions that would send software designers back to the drawing boards to come up with new concepts in data base management systems are sorely needed. The first is a really big, fast, and good associative memory. Associative memory based systems could be used to make both updating and selective retrieval equally efficient. These have been simulated and real ones have been tried on a small scale. Moving the ideas from the laboratory to full scale systems, however, has proved impossible so far. The other invention needed is something that would eliminate much of the time devoted to sorting. Most of the commercial data processing centers around today use up about 30 percent of their computer time sorting data, an astronomical dollar cost. Managers want, and need, to obtain information ordered in ways that make the data more meaningful. Some of the best programmers in this country have spent years studying, developing, and refining sorting and merging techniques and have reached an impasse. Something new is needed.

This, then, is the message of this paper. There are many techniques to use with today's equipment, but somehow the finished systems are not completely satisfactory. The software designers of generalized data base management systems have caught up with the hardware innovations of the past. Something new is needed.

The Storage Complex Considered as a System Component

G. G. SCARROTT

Abstract—To permit further growth in the application of information systems, the lower level activities must be automated so that they cease to be a burden on the user. A synthesis of known hardware devices and software algorithms for dynamic storage allocation is advocated which can achieve acceptable efficiency at run time by using the explicitly defined data structure as a basis for store management and the prediction of store usage.

The resultant automated storage complex also provides security against the propagation of consequential software failures. Consideration of the function of the individual storage devices which would be components of such a storage complex leads to some conclusions which may be useful for guiding the development of advanced storage devices.

I. Introduction

THE TERM "storage hierarchy" is now firmly established. It is unfortunately a somewhat misleading phrase since in most information systems all the storage devices communicate directly with the main store, so that in the strict sense of the word the organization of the physical storage devices is not hierarchical. The information in the collection of stores, can, however, usefully be regarded as a hierarchical structure; this paper will be concerned with the organization of the storage complex considered as a macrocomponent of an information system.

II. What Would We Like a Storage Complex to Do?

We must start by taking a new look at the storage complex by regarding it as an autonomous subcontractor to the whole information processing system. Then ideally, we would like the complex to have the following properties.

1) The user must be able to specify the hierarchical structure of his information by creating appropriately organized reserved space in the storage complex.

2) The user must be able to store information in his reserved space in the complex and retrieve it.

3) It must operate autonomously without bothering the user with its internal workings.

4) It must achieve the shortest access time to information at the lowest cost with the largest total storage capacity.

5) It must be safe to use. A user, i.e., a program, may make a mistake at any time and try to retrieve the wrong item of information. If he does so he obviously cannot always be protected from his error at the time that he makes it, but the storage complex should be so organized that the mistake does not lead to a storage malfunction either in another part of the same user's information, or in any other user's information. In short, the storage complex should incorporate means for preventing, or at least restricting, consequential software failures.

These requirements can be regarded as primary in the sense that they are derived from the needs of the information processing system without any reference to the properties of individual storage devices. At first sight, property 1) has an unfamiliar look, but is logically necessary; 2) and 4) look like platitudes; and 3) and 5) appear to be counsel of perfection, but not really practicable. However, an attempt will be made to show that by the use of appropriate hardware and software techniques which have already been published, i.e., fast storage, cheap storage, hardware store management devices, and programmed store management algorithms, it is possible to satisfy these primary requirements, and thus to make available a very powerful macrocomponent to system designers.

III. "Apparent" Store Concept

The specification of the requirements for the automated store complex describes how it should appear to the user. It may, therefore, be useful to introduce the term "apparent" store to refer to the user's view of the storage complex. The contrast between the apparent store and the physical store can be illuminated by considering how the ideal secretary operates. She stores documents in filing cabinets and retrieves them as requested by her employer. He does not know or care where in the filing cabinet each document is stored; he identifies it for filing and recovery by name. For example, he says: "Put this paper in Mr. Bloggs' file." He may give extra information, such as Mr. Bloggs is a customer, or an employee, which implies that in his mind the papers are organized in sets, such as employees files, and customers files. It follows that in his view the information could be said to be functionally structured, i.e., organized in a way which arises naturally out of the data itself. This is how the store appears to him, and it is, therefore, the apparent store. In physical fact the papers are held in filing cabinets, and to economize space each cabinet may contain papers of many unrelated classes which are so organized autonomously by the secretary, so that she in effect translates between the apparent and the real store.

One can describe the secretary's operation in mathematical terms. She maps the apparent store onto the real store and continuously, by interpreting her employer's instructions, modifies the map to ensure that space is available and that papers which are likely to be required fre-

Manuscript received April 1, 1971; revised June 25, 1971. Paper 13.6, presented at the 1971 INTERMAG Conference, Denver, Colo., April 13-16.

The author is with the Research and Advanced Development Organization, International Computers Ltd., Stevenage, Hertfordshire, England.

quently are easily accessible, whereas papers which are less likely to be required are banished to less accessible corners. In computer terms, the filing cabinets represent the storage devices in an information system, the secretary represents the store management system, and the employer represents the program.

IV. Store Management Problem

Evidently, the task of our hypothetical secretary is no sinecure, and equally, the design of a store management system to yield the apparent store organization with adequate productivity is a task which can be accomplished only if it is tackled systematically starting from a clear statement and proper understanding of the requirements. To the user the information in the apparent store is organized in a hierarchical pattern specified by him when he requests storage space, and which is a natural property of the information itself. Moreover, the user identifies each item of information for storage and retrieval by its position in that hierarchy. In the physical storage, however, each item is identified by its physical location. It follows that means must be provided for recording the organization structure of the data and its current map on the physical storage, keeping this store management information up to date, and referring to it as required when information is stored and retrieved.

These considerations are concerned with the logic of the store mapping process. There is also the problem of doing the store management job efficiently. Obviously, the following requirements must be satisfied insofar as possible.

1) Information which is required most frequently should be stored in the fastest store in the complex.

2) Information which is temporarily inactive should be banished to a cheaper and slower store.

3) As far as possible, the transfers of information between stores to satisfy criteria 1) and 2) should be carried out immediately before the relevant situation arises.

4) Information which must be read frequently and fast, but written very rarely, may be kept permanently in a storage device with very primitive rewriting facilities (e.g., interconnection wiring, to quote an extreme case).

5) The transfers of information between one store and another are parasitic in the sense that they do not contribute directly to the information processing job in hand. These parasitic transfers should occur no more frequently than is strictly necessary.

6) The information transfers between storage levels should be made in optimum size packages, i.e., small packages of only a few words to the fastest storage level of the complex, and larger packages between the slower and cheaper stores.

The available means for satisfying these requirements are fast costly stores, slow cheaper stores, "read mostly" stores, hardware store management devices (e.g., datum and limit registers and page registers), and software (e.g., store management programs). In order to see how these can be used to create the desired apparent store, it is necessary to specify the available techniques in a little more detail.

V. Available Storage Devices

This is not the place to make a detailed list of available storage techniques since this paper is primarily concerned with storage organization. There are, however, certain basic types of information store whose reason for existence seems to have such firm foundations that they are likely to continue in the foreseeable future, and must therefore be taken into account in any proposed organization.

A. Digitally Selected Storage Devices

Such stores incorporate a discrete device for each bit of information, so that they are fast but costly. It is primarily for their speed that stores of this type are used, so that we can be sure that no matter what methods (e.g., MOS technology) are used to bring down their manufacturing cost, they will still be more expensive than other contemporary storage devices. Moreover, even within the class of digitally selected stores, there will be several different compromises between speed and cost ranging from the ultimate in speed (bipolar integrated circuits) to minimum cost (core stores, MOS, plated wires, and possibly MNOS).

B. Geometrically Selected Storage Devices

Such stores incorporate a continuous storage medium such as magnetic tape, magnetic disks, or a magnetic bubble-domain material. A piece of information is accessed by physically moving the recording device to the appropriate area of storage medium (or vice versa), so that stores of this class have a low cost per bit, but are inevitably much slower than the digitally selected storage devices. In this class of store we can be sure that technological development will be directed to achieve a shorter access time; nevertheless, no matter what technique is introduced to increase the speed of geometrically selected stores they will always be decisively slower and cheaper than digitally selected stores since the prime reason for their existence is their low cost.

It is possible to combine digital selection with geometrical selection as in fixed-head disk stores with one head per track. The introduction of such combinations offers a useful tactical maneuverability to designers, but does not invalidate the general statement that geometrically selected stores are cheap but slow, whereas digitally selected stores are fast but costly.

VI. Large-Scale Store Management—Approach of Macrosystems Designer

The problem of large-scale store management has been tackled by many system designers. It is a problem involving many difficult compromises since, for example, if the information is handled in large packets, storage space is wasted, whereas if it is handled in small packets, store management computer time overheads become excessive.

A store management system proposed as part of a complete system concept by Iliffe [2] goes very far toward satisfying the requirements for an autonomous storage complex while leaving adequate room for maneuvers to make the necessary compromises, and it may therefore be