J. Gecsei

Determining Hit Ratios for Multilevel Hierarchies

Abstract: The applicability of stack processing for evaluation of storage hierarchies has been limited to two-level systems and to a very special group of multilevel hierarchies. A generalization of stack processing, called joint stack processing, is introduced. This technique makes possible the efficient determination of hit ratios for a class of multilevel hierarchies—staging hierarchies. These hierarchies are rather realistic in the sense that they allow for multiple block sizes and multiple copies of data in the hierarchy. Properties of storage management schemes that lend themselves to joint stack processing are studied, and the notion of distributed hierarchy management is described and illustrated.

Introduction

As computing grows increasingly more data oriented, the speed of data handling, and especially of storage functions, becomes the limiting factor in the overall performance of modern computers. Storage systems typically use several technologies, which are linked together with the objective of effectively utilizing the advantages of each technology (high speed, low cost).

Designing optimal storage hierarchies is rather complicated, not only because of the high dimensionality of the mathematical problems associated with optimization, but also because of the ever changing technology and workload environments. Information available on technology, costs, workload, and performance requirements (the designer's input) is often of limited accuracy and representativeness.

For all these reasons, it is desirable to develop efficient and easily automated storage system design methods to allow the exploration of as many design options as possible. Stack processing introduced by Mattson, et al [1] has been frequently applied and has also generated some theoretical interest, e.g., in the areas of reference trace analysis and theory of replacement algorithms.

This paper presents an outgrowth of standard stack processing, extending its applicability to a realistic class of storage hierarchies, called staging hierarchies by Slutz and Traiger [2], who originally described them. Staging hierarchies allow for an arbitrary number of memory levels, using different block sizes at various levels, and for multiple copies of the same block in the system.

This paper reviews some earlier results, presenting them from a perspective intended to facilitate understanding of the new extensions to stack processing. These new contributions include the following main results.

- An extension to stack processing, called joint stack processing allows simultaneous (one-pass) determination of success functions for a staging hierarchy.
- Systems of replacement algorithms exist (called joint stack algorithms, one for each hierarchy level) such that the hit ratios for variable capacities at each level can be obtained from the results of joint stack processing.
- In a system of joint stack algorithms, the algorithm at the first level (the "principal" algorithm A_1) may be an arbitrarily chosen stack algorithm; algorithms at lower levels are "driven" (uniquely determined) by A_1.
- If the principal algorithm is the least recently used (LRU), then correct replacement decisions at each level are entirely derivable from the history of that level; no "broadcasting" of requests is necessary. This leads to a view of the hierarchy as a collection of memory devices with their associated autonomous controllers.

Real hierarchies, of course, are not likely at the present time to satisfy strictly the conditions set forth in abstract hierarchy models. Replacement algorithms, for example, are in practice designed as a compromise between their practicality and their ease and cost of implementation, as well as cleanness and compliance with theory.

Stack processing

Stacks and stack processing are notions originally created during the search for an efficient method to evaluate various aspects of storage hierarchies. This section reviews informally topics treated earlier by Mattson, et al [1], emphasizing notions directly applicable to joint stack processing and establishing notation used later in

the paper. (A summary of this notation is provided preceding the References).

Consider the two-level hierarchy in Fig. 1. In such a configuration, level M_1 is often called the *buffer* and level M_2 the *backing store.* The totality of information contained in this hierarchy is divided into blocks of equal size (each block containing the same number B_1 of bytes). The blocks are identified *logically* by their names, independently of their physical locations in either level. Only one copy of each block is kept in the system, and all blocks are initially supposed to reside in the backing store. The capacities of the levels are C_1 and C_2 bytes, or equivalently D_1 and D_2 blocks. The $D_i = C_i/B_1$ are assumed to be integers.

The hierarchy is always accessed at the upper level. That is, when a particular byte is requested, the (unique) block containing that byte must be either present in the buffer or it must be brought in from the backing store before the actual access can take place. If the buffer is full at the time of the request, some block from its current contents must be replaced (pushed out to M_2) to make space for the requested block. The rule for selecting the block to be pushed out is called the *replacement algorithm.* Some common replacement algorithms are FIFO (first-in, first-out), LRU (least recently used out), LFU (least frequently used out), and their variations.

A sequence of byte references is called the *reference string* and is denoted as

$$Z = z_1, z_2, \cdots, z_L,$$

where z_t is the name of the byte referenced at time t, and L is the length of the string. The *block reference string* for given block size B_i is

$$X(i) = x_1(i), x_2(i), \cdots, x_L(i),$$

where $x_t(i)$ is the name of the block that contains z_t. We use the simpler notation X and x_t when the block size is implied by the context.

Block reference string $X(i)$ is a string over the set M_2 of blocks contained initially in the backing store. (The symbol M_i is used to denote both the ith level of a hierarchy and the set of blocks contained in that level.) If $|X(i)|$ is the number of distinct blocks that occur in $X(i)$, the following relation holds:

$$|X(i)| \leq |M_2| \leq D_2.$$

Throughout this paper we assume that references are satisfied by the hierarchy management in *strict sequence:* x_t can be requested only after x_{t-1} has been successfully accessed. Thus, we do not consider overlapped or out-of-order referencing.

A reference string represents the environment of the hierarchy; the exact nature of the source of requests is irrelevant for the internal operation of the hierarchy.

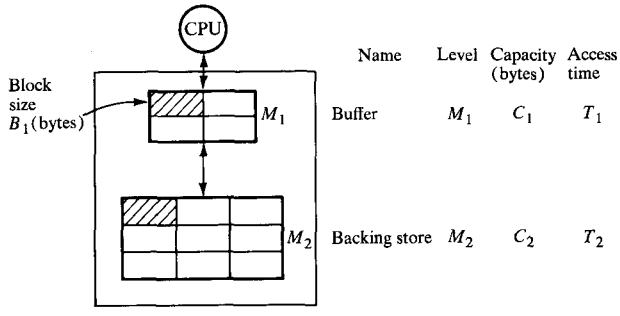

Figure 1 Two-level hierarchy.

The requests may come from a single processor in single or multiprogramming mode, from several processors, or in general from any data processing subsystem that needs access to a pool of data.

When a string of length L is applied to a two-level hierarchy with a given buffer size D_1 and a given replacement algorithm A_1, some references result in *hits* in the buffer (no transfer from M_2 is necessary). References that do require such a transfer are commonly called *misses.* If the number of hits and misses are L_H and L_M, respectively, then the *hit ratios* p_1 and p_2 are defined as

$$p_1 = \frac{L_H}{L}, \; p_2 = \frac{L_M}{L} = 1 - p_1.$$

Thus, hit ratios express the fraction of all memory references that result in accessing levels M_1 and M_2 of the two-level hierarchy in Fig. 1. This notion of hit ratio is meaningful also for properly defined n-level hierarchies. We use the buffer-backing store concept only as a vehicle for describing stacks and stack algorithms.

The most significant use of hit ratios for design and analysis of computer systems with memory hierarchies is as input values to various probabilistic models for evaluating system performance. A simple, and perhaps the first, application is the calculation of the expected access time T to the hierarchy

$$T = p_1 T_1 + p_2 T_2.$$

(Time T_2 is the total time to access a block in M_2, which is not always equal to the device access time.) In more sophisticated models [3] the CPU and individual memory levels become servers in a cyclic queuing system such as the one in Fig. 2, where hit ratios are seen as probabilities of an I/O request being directed to level 1 or 2. The aim of such models is to calculate the CPU and I/O utilization factors and the throughput rates.

There are many possible uses and interpretations of hit ratios. A common feature of all applications is that p_i's provide a convenient link between some kind of ab-

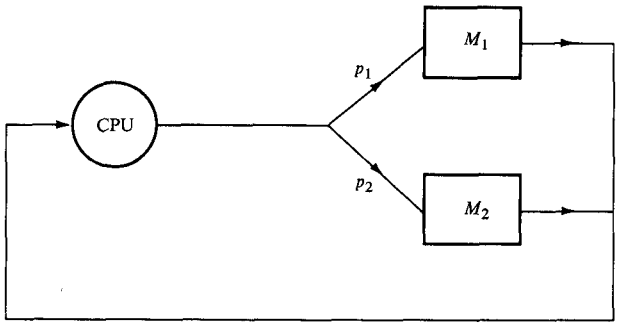

Figure 2 Cyclic queuing system.

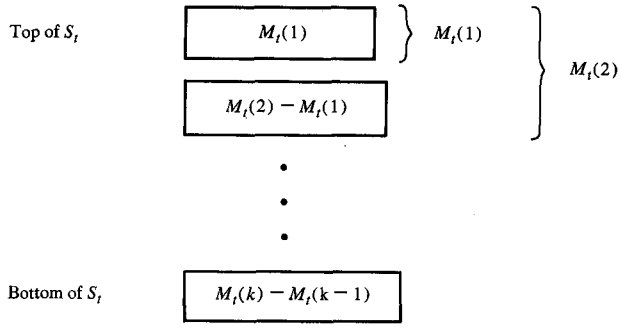

Figure 3 Ordering of blocks in stack S_t.

stract model and the actual workload on a computer system. When using such methods, however, a complex programming environment is represented by a few numbers, amounting to a considerable reduction of information; thus results from such an analysis are necessarily only approximate.

For a given block reference string $X(1)$ and a given replacement algorithm A_1, the hit ratio p_1 depends only on the buffer capacity. This function is sometimes called the *success function* $p(C_1)$. Knowledge of it permits optimization for various cost-performance combinations when designing systems with memory hierarchies. This is done by picking values of C_1 (which essentially gives the cost of the hierarchy) and using the associated value of $p(C_i)$ for calculating performance.

Early ways of constructing success functions consisted of a series of time-consuming simulation runs, each for a different value of C_1. In [1] a method called *stack processing* is introduced, which accomplishes the whole task of finding the success function in a single pass of the block reference string (equivalent to little more than one direct simulation run). Replacement algorithms suitable for stack processing are called *stack algorithms*. Fortunately, most commonly used replacement algorithms belong to this class. FIFO is an example of a nonstack algorithm.

At this point it is clear that the set of blocks $M_t(D)$ contained in a buffer of size D immediately before the occurrence of request x_t is determined by the block reference string and the replacement algorithm. Stack algorithms meet the following conditions:

1. $M_t(D) \subset M_t(D+1)$
2. $M_{t+1}(D) = M_t(D)$ if $x_t \in M_t(D)$
   $M_{t+1}(D) = \{M_t(D) - y_t(D)\} \cup x_t$ if $x_t \notin M_t(D)$,

where $y_t(D) \in M_t(D)$ is the block selected for replacement by A.

Both properties hold for $1 \leq t \leq L$, $0 \leq D \leq |X|$ and any string X. Condition 1) means that blocks contained in a buffer of size D form a subset of the blocks present in a buffer of size $D+1$. This is called the *inclusion property*, and it induces a total ordering (list) over all k distinct blocks that occur at least once in the trace up to reference x_{t-1}. (Only blocks already referenced can occur in a buffer of any size.) This ordering is called the *stack* S_t and is shown in Fig. 3.

According to this description then *the block contents of a buffer of size D at any time t are simply the top D elements of the stack*. We define the *stack distance* $s_t(E)$ of a block E as its position relative to the top of S_t. If E is the top element, then $s_t(E) = 1$. If E is not present in S_t, we write symbolically $s_t(E) = \infty$. This happens when E has not yet been referenced.

Condition 2) states that precisely one block $y_t(D)$ is replaced in a full buffer that does not contain x_t; no block is replaced otherwise. This amounts to "replacement on demand" or demand paging.

Stack S_{t+1} is obtained from S_t according to the diagram in Fig. 4. Block x_t becomes the top element of S_{t+1} and no change in the stack is made below position $s_t(x_t)$, where x_t is found in S_t. The diamonds represent decisions about the occupancy of intermediate positions in S_{t+1}. Each decision $Q(D)$ determines the occupant of the Dth position of S_{t+1} by choosing between $y_t(D-1)$ and the block at the Dth position of S_t.

The stack distance $s_t(x_t)$ of the current reference indicates *the minimum size that the buffer must be to contain* x_t at time t. If $D_1 \geq s_t(x_t)$, the reference results in a hit. The success function can be obtained by developing the cumulative distribution of the values of $s_t(x_t)$, $t = 1, \cdots, L$. Thus, stack processing of a reference string consists of finding the stack distance for the current reference x_t, building the new stack S_{t+1}, finding the distance of x_{t+1}, etc. The initial stack is assumed to be empty.

Seen from this perspective the replacement algorithm appears as the source of decisions $Q(D)$. A convenient way of representing all such decisions is by introducing *priority lists*, R_t, $t = 1, \cdots, L-1$. List R_t is similar in structure to a stack, in that both are total orderings over a set of blocks. List R_t should contain (at least) those

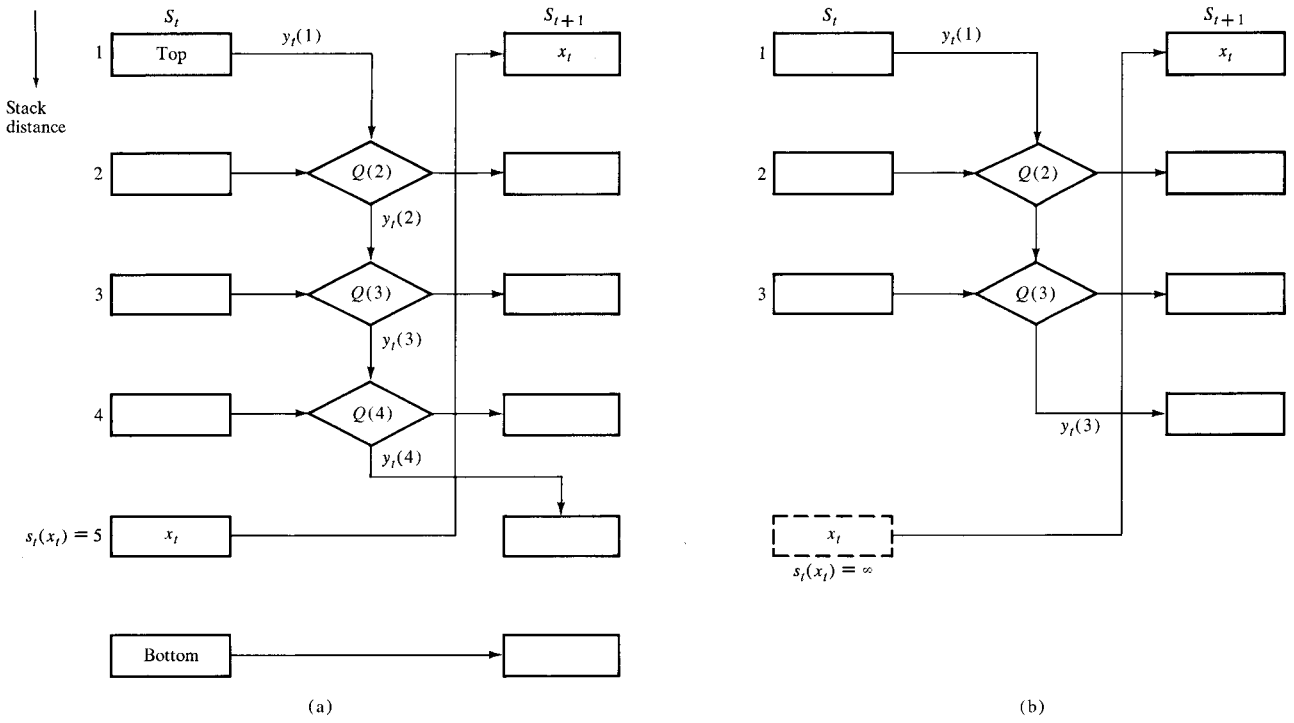

Figure 4 Stack updating scheme with (a) $x_t \in S_t$ and (b) $x_t \notin S_t$.

blocks that enter into decisions at time t. The position of a block E in R_t is denoted $r_t(\mathrm{E})$. If two blocks E and F, such that $r_t(\mathrm{E}) < r_t(\mathrm{F})$, enter in a decision $Q(D)$ in the course of constructing S_{t+1} from S_t, then $y_t(D) = \mathrm{F}$. Restated for a particular buffer of size D, this means that $y_t(D)$ is the block in the lowest position in R_t among the D blocks present in the buffer. Thus, knowledge of R_t's (or having a way to derive them) amounts essentially to defining a stack algorithm.

Example 1 Figure 5 shows the processing of a reference string by three representative stack algorithms, LRU, LFU, and OPT, and the resulting success functions. Successive times are indicated in the top row, and the symbolic names of the pages referenced at each time are given in the next row. The contents of stack S and of priority list R for each reference appear in the rows below, followed by the stack distances for each reference. First references ($s_t(x_t) = \infty$) are not used in the calculation of the numerical values of $p_1(C_1)$. In LRU the priority list R_t is the order in which blocks occur when scanning X backwards from x_t (which is at the top of R_t). LRU is the only stack algorithm in which $R_t \equiv S_{t+1}$, which makes it important both theoretically and practically. The construction of S_{t+1} from S_t is particularly simple. It amounts to pushing down by one position each element is S_t between the top and x_t. Many actual replacement schemes use LRU or its variants.

With LFU all priority lists are constant in time. Blocks are ordered according to their frequency of occurrence in X, with the most frequently referenced block on top. Thus, prior information about X is necessary. A modified form of LFU uses frequency counts from the past part of X only. Then, of course, R_t's might change in the course of processing.

OPT is an algorithm of theoretical interest requiring knowledge of the future values in X. Priority list R_t is always the order in which blocks occur on scanning X forward from x_t (which is at the top of R_t). OPT yields the highest possible value of hit ratio for any given buffer size and can be used as a yardstick for evaluating the performance of other algorithms. Recent developments [4] have shown a seemingly paradoxical result: It is possible to find the success function for OPT *without* information about the future of X. (But it is not possible to make the actual correct replacement decisions).

An attempt was made in this section to review earlier results informally, omitting some details not directly relevant to joint stack processing; a rigorous treatment of these topics is provided in [1].

As a final remark, stack algorithms and stack processing can be seen heuristically as parts of a dynamic op-

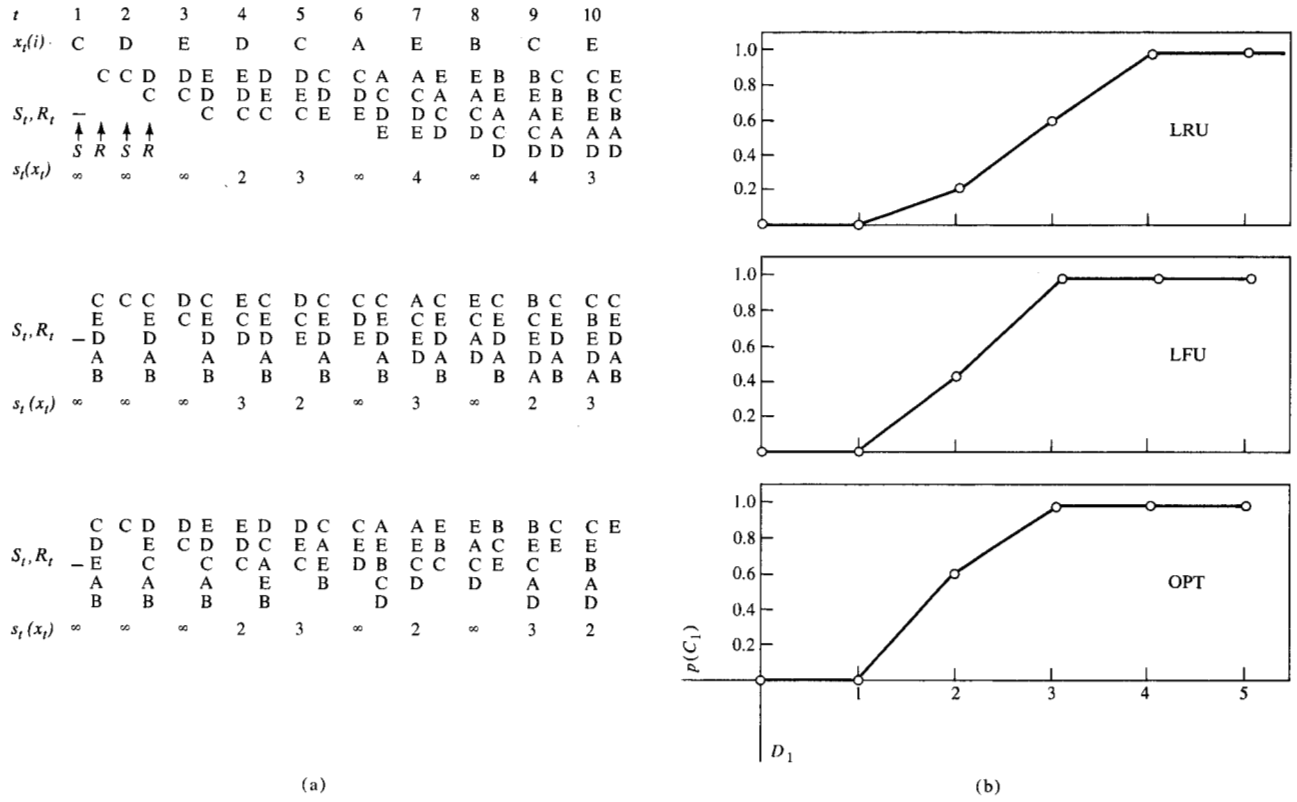

Figure 5 Stack processing of a reference string (a) by three representatives algorithms and (b) the resulting success functions.

timization problem. Stack S_t represents the state of a system, and R_t can be thought of as the system's expectations about its environment. (Actually, R_t is its current assumption about the immediate future of the reference string). The goal of the system is to generate its next state S_{t+1}, matching the environment as well as possible. The constraints in doing so are conditions 1) and 2), which limit the possible amount of change in S_t. By modifying 2) to allow for more than one block to be replaced at each reference, we effectively arrive at what is known as "nondemand" or "anticipatory" replacement algorithms.

Staging hierarchies

A memory hierarchy may be thought of as the association of hardware components (storage devices, data transfer paths) and a set of rules that control the dynamics of data movement within the hierarchy. The class of *staging hierarchies* is roughly delimited by the following attributes:

- A staging hierarchy has two or more levels of memory, denoted $M_1, M_2, \cdots, M_H$, with capacities $C_1 \leq C_2 \leq \cdots \leq C_H$(bytes).
- Information is moved between adjacent levels M_i and M_{i+1} in blocks (pages) of size B_i (bytes); $B_1 \leq B_2 \leq \cdots \leq B_{H-1}$.
- Each block of size B_i is composed of an integral number of blocks of size B_{i-1}, called its *descendants*. Thus, each block of size B_1 has a unique *parent* block of size B_2, which in turn has a unique parent of size B_3, etc.
- A staging hierarchy is always accessed at the top (highest) level M_1.
- Movement of information within the hierarchy follows the *staging rule*: Whenever a block F′ is moved upward into level M_i, its parent block F is moved into M_{i+1} (if not already there), the parent of F into M_{i+2}, etc. In the case of *demand staging*, blocks are always moved upward until M_1 is reached, but this happens only on demand by a current reference. Thus, demand staging is always accomplished by a sequence of block transfers (one across each interface), starting at the highest level where the required information exists and ending at level M_1. Downward movement of blocks is the result of space limitations at one or more levels above M_H, and it occurs only when space is needed for an upcoming block. Replacement algo-

rithms determine the block to be replaced (pushed out) to a lower level, where it is returned to its parent block. Hence the logical sequence of events on each demand for staging is *first* to create space (if necessary) by moving blocks downward and *then* to stage upward. In an actual implementation, this sequence can be reversed by providing buffer space for one block at each level. Demand staging is contrasted with anticipatory staging, in which blocks may move upwards prior to actual reference to their contents. In this paper we deal with demand staging only.

- Initially all information resides at the lowest level of the hierarchy. Later, several copies of each byte can exist in the system, but only one at each level.
- The two-level hierarchy described previously is a staging hierarchy in which $H = 2$.

Figure 6 shows a four-level hierarchy and the process of staging a block from M_4 to M_1.

These general features of staging hierarchies appear to be reasonable in the light of the following facts about storage technology and common data accessing patterns. The decreasing capacities and block sizes usually used toward the top level of memory hierarchies are motivated by typically higher cost and lower access time of devices at the upper levels. The concept of staging is effective due to the locality of reference that is characteristic of most data processing environments: data requests to blocks or their groups tend to be repetitious; hence it "pays" (from the standpoint of average access time to the hierarchy) to stage a referenced block into a higher—and faster—level.

Hierarchy management

The definition of staging hierarchies in the preceding section is clearly insufficient for a precise description of the movement of blocks between levels. What is needed is the specification of replacement algorithms $A_1, \cdots, A_{H-1}$. By replacement algorithm A_i we mean the rule that determines the block to be replaced in level M_i when space is required for a new block. Generally, A_i may depend on all A_j and C_j, $i > j$.

The collection of replacement algorithms in a given staging hierarchy is called the *hierarchy management*. Within the scope of demand staging, we investigate systems of replacement algorithms that facilitate the determination of hit ratios $p_1, \cdots, p_H$. Each hit ratio p_i is the fraction of all references to the hierarchy for which information has to be staged from level M_i.

As shown previously, the hit ratios for a two-level hierarchy depend (for fixed X, B_1, and A_1) on C_1; $p_1 = p(C_1)$. This function was termed the success function. In the case of multilevel hierarchies, p_i in general depends on C_i, $C_{i-1}, \cdots, C_1$ (X, all B_i and A_i fixed). Thus, no

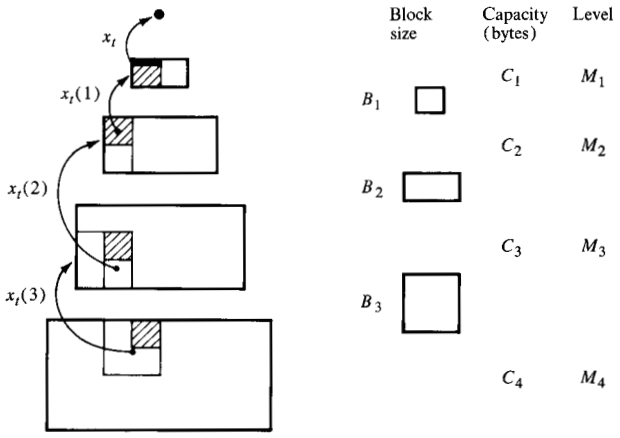

Figure 6 Four-level staging hierarchy.

direct analogy with stack processing exists, and the only apparent way to determine the hit ratios is through direct simulations, one for each combination of capacities. This approach turns out to be uneconomical and sometimes unmanageable, even for two-level hierarchies, and clearly gets quickly out of bounds with the growing dimensionality of the problem.

Our aim is to find a framework for using success functions (i.e., results of two-level simulations or stack processing) for ascertaining hit ratios in H-level staging hierarchies.

The following properties of hierarchy management (which may or may not hold in a given staging hierarchy) are of interest:

Property 1 All replacement algorithms A_i (i.e., the replacement decisions taken by A_i's) are *independent* of C_j, $i > j$. An equivalent way of expressing this is that the block contents of all levels M_i at any time and for any reference string are independent of the capacities of higher levels. In [2] this is called the "two-level property"; each level M_i can be seen as the upper level of a hypothetical two-level hierarchy, the lower level of which consists of all levels below M_i. Levels above M_i are transparent to M_i, since C_j can be assumed to be zero.

Property 2 The presence of a block in M_i implies the presence of its parent block in M_{i+1}, $i = 1, \cdots, H-1$. This is called the "nesting property" in [2]. It means that no block may be removed from M_{i+1} before all its descendants have been returned from M_i. (Note that this property implies a dependence of A_i on A_j, $i > j$.)

Suppose now that success functions $p(C_1), \cdots, p(C_{H-1})$ are formed by processing block traces $X(1), \cdots, X(H-1)$ with replacement algorithms $A_1, \cdots, A_{H-1}$. If Property 1 holds, then $p(C_i)$ can be interpreted as the fraction of all references to a staging hierarchy such that

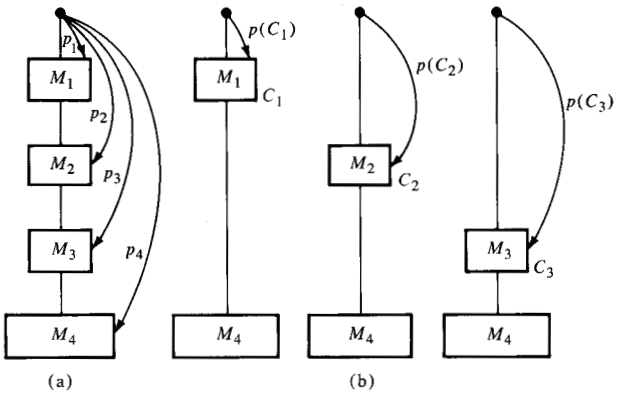

Figure 7 Hierarchy properties include (a) four-level hierarchy and the required hit ratios and (b) success functions and hypothetical two-level hierarchies.

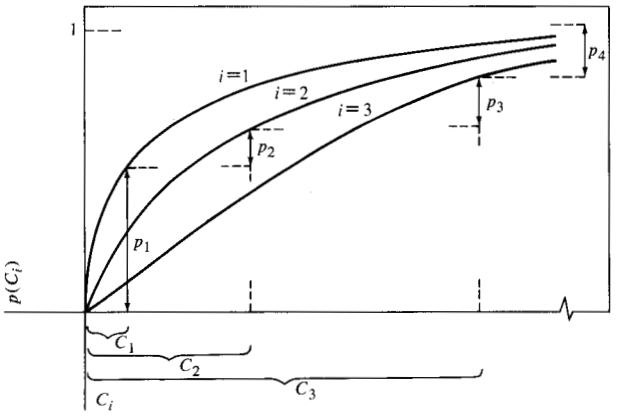

Figure 8 Relationship between hit ratios and success functions.

a copy of the referenced byte is found at level M_i. Clearly, the $p(C_i)$ are not equivalent to the hit ratios p_i, since x_t may be present also in some higher level and, thus, staging from M_i may not result.

However, if Property 2 holds too, then references that are found in M_i are also found in all levels below M_i. This provides for a simple way to calculate p_i's, summarized in the following theorem originally proved in [2].

Theorem 1 In a staging hierarchy with block sizes B_i and level capacities C_i, and with hierarchy management such that Properties 1 and 2 hold at all times, the hit ratios are determined by

$$p_i = p(C_i) - p(C_{i-1}),\ i = 1, \cdots, H, \qquad (1)$$

where $p(C_0) = 0$ and $p(C_H) = 1$. Figures 7 and 8 illustrate the above result and its informal proof.

The significance of Theorem 1 is that it may be used for calculating p_i for a range of capacities at all levels from a single set of success functions.

Another result shown in [2] is that if the capacities satisfy the constraint

$$D_1 \leq D_2 \leq \cdots \leq D_{H-1},$$

where $D_i = C_i/B_i$ are integers, and if all replacement algorithms are LRU, then Properties 1 and 2 hold and hence Eq. (1) is valid. In the next section this result is generalized for algorithms other than LRU.

Joint stack algorithms

Hierarchy management has thus far been treated as a collection of independent replacement algorithms. It was shown that if all algorithms are *defined* to be LRU, then the hit ratios can be obtained from Eq. (1).

Now we take a different approach, in that the replacement algorithms are viewed as *dependent* upon the algorithm A_1 at the highest level. The nature of this dependence should be such that Properties 1 and 2 hold at all times. Thus, A_2 appears to be driven by A_1, A_3 driven by A_2, etc., rather than each algorithm operating autonomously. This view of hierarchy management is quite natural because the dependence of A_i on A_{i-1} is inherent in Property 2.

The result to be shown in this section is that A_1 may be an *arbitrarily chosen* stack algorithm. However, if Properties 1 and 2 are to hold, then the remaining algorithms $A_2, \cdots, A_{H-1}$ are uniquely determined by A_1. In the next section an extension of stack processing is outlined for determining the success functions $p(C_2)$ simultaneously for $i = 1, \cdots, H-1$.

We assume that all replacement algorithms in a demand staging hierarchy are stack algorithms. If each A_i is applied to its corresponding block trace $X(i)$, $i = 1, \cdots, H-1$, a sequence of stacks results for each A_i. We denote $S_t(i)$ as the stack of A_i immediately *before* the reference to block $x_t(i)$. It follows from Theorem 1 that, as long as Properties 1 and 2 hold, the upper D_i entries in $S_t(i)$ represent the block contents of level M_i of a staging hierarchy. As explained earlier, a priority list $R_t(i)$ is used by each A_i to obtain $S_{t+1}(i)$. It is known that if A_i is LRU, then $R_t(i) = S_{t+1}(i)$.

Definition 1. The *parent stack* $P_t(i)$ of $S_t(i-1)$ is a stack that contains exactly those blocks (of size B_i) whose descendants occur in $S_t(i-1)$. The stack distances from the top of $P_t(i)$ of any two blocks F and G are ordered in the same way as are the distances of the *highest* (closest to the top) descendants of F and G in $S_t(i-1)$.

Lemma 1 If F′ is a descendant block of F, $p_t(\mathrm{F})$ the stack distance of F in $P_t(i)$, and $s_t(\mathrm{F}')$ the distance of F′ in $S_t(i-1)$, then

$$p_t(\mathrm{F}) \leq s_t(\mathrm{F}').$$

Proof The proof is evident from Definition 1.

Example 2 Suppose that:
a_1, a_2, a_3 are descendants of block A,
b_1, b_2, b_3 are descendants of block B,
c_1, c_2, c_3 are descendants of block C.
If $S_t(i-1) = b_2$, then $P_t(i) =$ B

b_1	A
a_3	C
b_3	
c_1	
a_2	
a_1	

Definition 2 A block that is present in level M_i of a staging hierarchy is said to be *free* if none of its descendants is present in M_{i-1}. All blocks in M_1 are free. If a block is not free, it is *bound.*

Lemma 2 Property 2 of hierarchy management holds if only free blocks are selected for replacement by all replacement algorithms A_i, $i = 1, \cdots, H-1$, at all times.

Proof It follows from the staging rule that immediately after a block F′ is staged up to M_1, M_2 contains its parent block F, M_3 the parent of F, etc., down to M_H. If subsequently only free blocks are removed from each level, no hole in the sequence of parent blocks down the hierarchy can ever occur.

Lemma 3 (*a*) If the capacity of level M_{i-1} is D_{i-1} blocks (of size B_{i-1}), then any block $F \in P_t(i)$ such that

$$p_t(F) > D_{i-1}$$

is free. (*b*) If $F \in P_t(i)$ is free and $G \in P_t(i)$ is bound, then $P_t(F) > P_t(G)$.

Proof (a) Level M_{i-1} contains the top D_{i-1} blocks of $S_t(i-1)$. According to Lemma 1 no parent of these blocks can be lower than the D_{i-1}th position in $P_t(i)$. (*b*) The proof is evident from the proof of (*a*) and from Definition 1.

Definition 3 Stack algorithms $A_1, \cdots, A_{H-1}$ constitute a system of *joint algorithms* if

$$R_t(i) = P_{t+1}(i), \quad i = 2, \cdots, H-1, \quad t = 1, \cdots, L.$$

Algorithm A_1 is an arbitrarily chosen stack algorithm, and it is called the *principal algorithm.*

Thus, A_1 can use arbitrarily ordered priority lists for updating its stack. The priority lists for subsequent levels are the parent stacks derived from the immediately higher level. We observe that $R_t(i)$ always contains exactly the same blocks as $S_t(i)$.

The stacks in a system of joint algorithms are updated as follows: Starting with all stacks given at time t, $S_{t+1}(1)$ is obtained on each new reference first, then $P_{t+1}(2)$, which in turn is used as the priority list to obtain $S_{t+1}(2)$, etc. This sequence is illustrated in Figure 9.

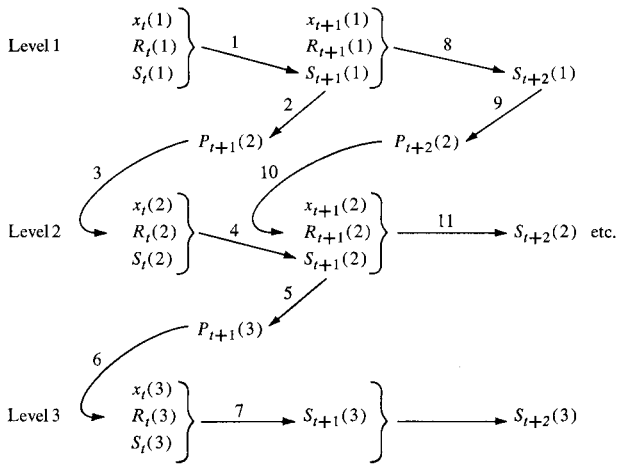

Figure 9 Updating stacks in a system of joint algorithms.

It can be seen that the replacement process starts at the highest level so that a block freed at a level may be considered for replacement out of that level at the same time t.

Lemma 4 Consider level M_i of a staging hierarchy with joint replacement algorithms. The capacity of M_i is D_i blocks. Suppose that M_i is full and the current reference $x_t(i) \notin M_i$; therefore A_i must make a replacement decision. Denote $y \in M_i$ the block selected for replacement and $r_t(y)$ the position (distance) of y in the priority list $R_t(i)$. Then

$$r_t(y) > D_i.$$

Proof We recall from Definition 3 that $R_t(i) = P_{t+1}(i)$. Thus $x_t(i) \notin M_i$ is the top element of $R_t(i)$, $r_t(x_t(i)) = 1$. Observe now that block y, selected by A_i for removal from among D_i blocks, must always be the one with distance

$$r_t(y) = \max_k \, (r_t(F_k)), \quad k : F_k \in M_i.$$

This follows from the way $R_t(i)$ is used for replacement decisions.

Therefore, there must be at least $D_i - 1$ distinct blocks $F_k \in M_i$ such that

$$2 \leq r_t(F_k) < r_t(y).$$

This directly implies

$$r_t(y) > D_i.$$

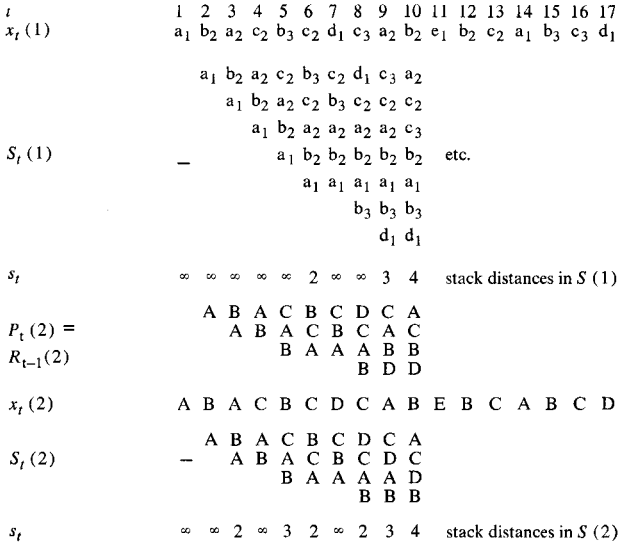

Figure 10 Joint stack processing.

Now we are ready to formulate and prove the following theorem.

Theorem 2 Assume an H-level staging hierarchy, G, $H \geq 2$, with block sizes $B_1, \cdots, B_{H-1}$ and capacities $C_1, \cdots, C_{H-1}$ such that

$$D_1 \leq D_2 \leq \cdots \leq D_{H-1}.$$

If the hierarchy management consists of a system of joint algorithms $A_1, \cdots, A_{H-1}$, then properties 1 and 2 of hierarchy management hold.

Proof of Property 1 It is observed from Definition 3 that the stack updating (i.e., replacement) decisions made at time t by A_i are uniquely determined by $x_{t+1}(i)$ and $S_{t+1}(i-1)$, $i = 2, \cdots, H-1$. But $S_{t+1}(i-1)$ is a *stack* and hence independent of C_{i-1}. By the same argument the decisions made by A_{i-1} are independent of C_{i-2}, etc. As a result, $S_{t+1}(i)$ and A_i are independent of C_{i-1}, C_{i-2}, $\cdots$, C_1 as required by Property 1.

Proof of Property 2 From Lemma 4 and Definition 3 we have

$$P_{t+1}(y) = r_t(y) > D_i.$$

Substituting $D_{i-1} \leq D_i$ yields

$$P_{t+1}(y) > D_{i-1}.$$

According to Lemma 3, then, block y (selected by A_i for replacement from M_i) is free. Finally, from Lemma 2 and from observing that the above reasoning is valid for all A_i, $i = 2, \cdots, H-1$, and for all t, $t = 1, \cdots, L$, we conclude that Property 2 holds.

Corollary 1 The hit ratios for a hierarchy G with joint stack algorithms are

$$p_i = p(C_i) - p(C_{i-1}), \quad i = 1, \cdots, H, \tag{1}$$

where $p(C_0) = 0$ and $p(C_H) = 1$.
This follows directly from Theorems 1 and 2.

Joint stack processing

The success functions $p(C_i)$ used in Eq. (1) can be obtained by means of a procedure that is an extension of the standard stack processing technique. This extended procedure is called *joint stack processing*, and it uses the dependence of $S_t(i)$ on $S_{t+1}(i-1)$ as described in Definition 3 and Fig. 9. Joint stack processing with $A_1 =$ OPT, $H = 3$, and $B_2 = 3B_1$ is shown in Fig. 10.

Given the principal algorithm A_1 and block sizes $B_1, \cdots, B_{H-1}$, all $H-1$ stacks are maintained and updated after each reference. Stack distance statistics are kept for each stack individually exactly as in standard stack processing. Thus joint stack processing is a one-pass procedure. It appears to be similar to a method described in [5] for $A_1 =$ LRU.

The following two sections describe an application of joint stack processing to the design of storage hierarchies.

Implementation of hierarchy management

It is important to distinguish the notion of joint stack processing from the process of actual management of a given staging hierarchy. The former is a procedure for determining $p(C_i)$, $i = 1, \cdots, H-1$, for the entire range of C_i, while the latter is the real-time processing of a string of memory requests by a hierarchy with fixed capacities C_i.

As indicated in the preceding section, joint stack processing requires updating of each stack at the time of each reference. In the real-time environment it appears to be necessary to maintain $H-2$ stacks. However, these stacks serve not for gathering distance statistics but to make actual replacement decisions: Stack $S_{t+1}(i)$ is used to generate the priority list R for level M_{i+1}. The block selected for replacement (if any) in M_{i+1} is the one in the lowest position in $R_t(i+1)$ among blocks currently present at M_{i+1}.

With these thoughts in mind, we can visualize two ways of implementing hierarchy management, as shown in Fig. 11.

In Fig. 11(a) the hierarchy management is centralized in a single module (hardware, software, or a combination), which receives all requests from the reference string, contains all stacks, and controls data transfers between adjacent levels. In Fig. 11(b) hierarchy management is distributed among individual control modules, each associated with a level in the hierarchy, e.g., they could be part of the device control units. Using this

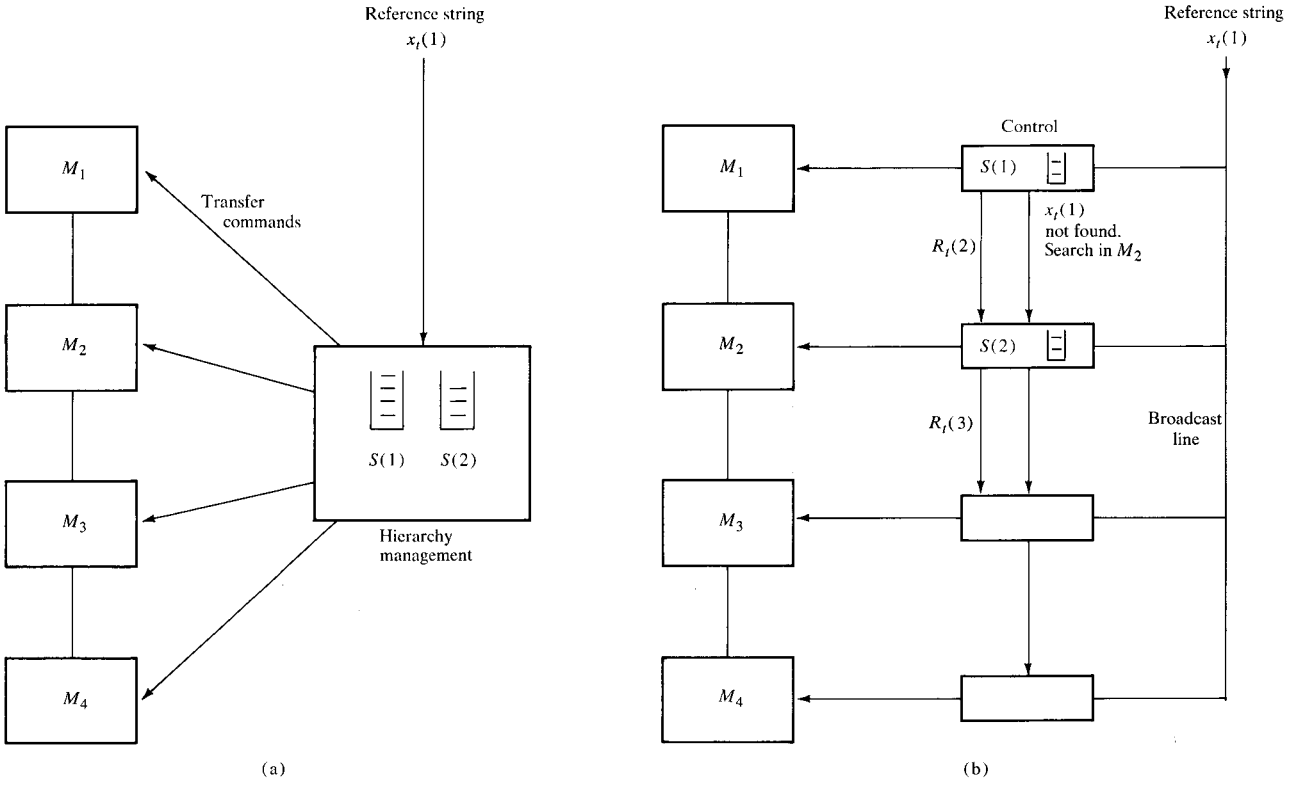

Figure 11 Hierarchy management with (a) centralized control and with (b) distributed control with broadcasting.

modular approach, it is conceivable to easily construct or change hierarchies, with little or no impact on the remaining part of the system. In order to work properly, however, these control modules must be interconnected, and to update the stacks they must receive information at the time of each reference. Reference x_t is received via a *broadcast line* and the priority lists by connection from the adjacent level. This implies a serious disadvantage of the otherwise attractive idea of distributed hierarchy management: a lower level (e.g., archive management) must be capable of high speed, since it has to process all references from a string, including those that cause no actual data transfer in or out of that level.

In the next section we show that, if the principal algorithm is LRU, distributed hierarchy management is possible *without broadcasting* references and priority lists to all levels. This means that stacks $S_t(i)$ do not have to be updated on each reference. In fact, only a small part of each stack has to be maintained to make the correct LRU decisions.

Distributed hierarchy management with LRU

Follow first in detail the general (not necessarily LRU) interaction between the controls of two levels M_{i-1} and M_i to determine the block to be replaced (deleted) from M_i. We assume that reference x_t has to be staged from below M_i and that M_i is full at t. The interaction evolves in time as follows:

1. Determine stack $S_{t+1}(i-1)$.
2. Determine the parent stack $P_{t+1}(i)$ and the priority list $R_t(i)$ for level M_i. By definition $R_t(i) = P_{t+1}(i)$.
3. Determine block y to be replaced from M_i as $r_t(y) = \max_k (r_t(\mathrm{F}_k))$, k: $\mathrm{F}_k \in M_i$.

   In words, y is the block having the lowest priority among those present in M_i. From the proof of Theorem 2 we know that y is always free.
4. Construct stack $S_{t+1}(i)$.

Note that step 4 is needed only to create $R_t(i+1)$, not for making the replacement decision in M_i!

Now assume that the principal algorithm A_1 is LRU. Then all algorithms [2] will be LRU, and all stacks will become LRU stacks (with block size B_i at each level). It will be shown that in this case y can be found by a procedure much simpler than steps 1–4. First we notice that except for $x_t(i)$, which is at the top of $R_t(i)$, the LRU stack $S_t(i)$ and the priority list $R_t(i)$ induce identical orderings over all blocks of size B_i. Therefore, y can also be determined from $S_t(i)$:

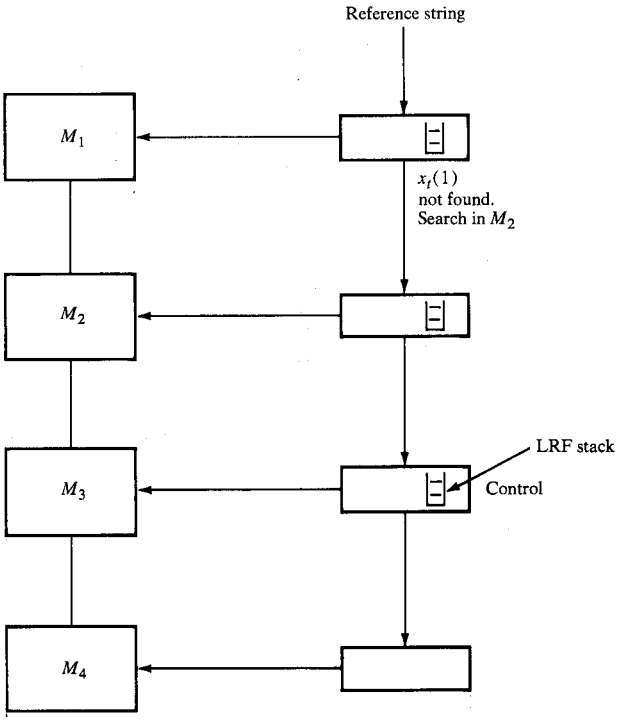

Figure 12 Staging hierarchy constructed of autonomous modules without broadcasting.

$$s_t(y) = \max_k (s_t(F_k)), k: (F_k \in M_i, F_k \text{ is free.})$$

In words, y is the *lowest free block* in $S_t(i)$ among all blocks present in M_i.

The following Lemma makes it possible to determine this lowest free block from the *time order* in which blocks became free (i.e., the time their last descendants were released from M_{i-1}).

Lemma 5 Let $F \in M_i$ and $G \in M_i$ be two free blocks in the LRU stack $S_t(i)$, let n_F and n_G be the respective times when they were last referenced, and let m_F and m_G be the respective times when they last became free; n_F, n_G, m_F, $m_G < t$. If $A_1 = A_i = \text{LRU}$ and $m_F < m_G$, then $n_F < n_G$.

Proof Consider the priority list $R_{t'}(i)$ for $m_F \leq t' < m_G$. From Lemma 3 and Definition 3 we have

$$r_{t'-1}(F) > r_{t'-1}(G).$$

But for LRU $R_{t-1}(i) = S_t(i)$ for all t; therefore

$$s_{t'}(F) > s_{t'}(G).$$

Since $n_F < m_F \leq t' < t$ (else F would not be free at time t) and observing that the order of any two blocks in an LRU stack cannot change unless one of them is referenced, it follows that

$$s_t(F) > s_t(G).$$

This in combination with the definition of LRU stacks finally implies

$$n_F < n_G.$$

Loosely speaking, Lemma 5 asserts that the time order in which blocks in M_i become free is the same as their order in the LRU stack. Thus, keeping track of this time order alleviates the need to maintain full stacks. A partial LRU stack containing only the free blocks in M_i can be easily maintained in the control module of every level. The block to be replaced is always the "longest time free," or "least recently freed," LRF.

We can summarize the action of each control module:

1. When a block becomes free, put it on the top of a LRF stack.
2. When a block becomes bound, remove it from the stack.
3. When replacement is required, select the bottom element of the stack, and remove it from the stack.
4. A directory of the current contents of the level must exist in the control module for search procedures, but this is a quite separate issue. For replacement decisions, only steps 1-3 have to be done.

The potential of this scheme is that replacement decisions are dependent only on information *local* to a memory level, i.e., on previous block transfers in and out of that level. No broadcasting of references is required, and interaction between controllers of adjacent levels is limited to times when actual data transfers take place between these levels. Still, the sequence of replacements thus generated is exactly the same as in the system with broadcasting. Figure 12 shows this type of configuration.

In conclusion, localized hierarchy management is an architectural feature of its own, not limited to the context of LRU. Any other local algorithm may be used. If the replacement decisions are for free blocks only, then Property 2 still holds, Property 1, possibly not. Evaluation based on joint stack processing then becomes a more or less good approximation to the true values of hit ratios.

Summary of notation

A_i	Replacement algorithm operating at ith level of hierarchy
B_i	Block size (bytes) at ith level
C_i	Capacity (bytes)
D_i, D	Capacity (blocks)
M_i	ith level of hierarchy, set of blocks contained in that level
$M_t(D)$	Set of blocks in buffer of size D, before reference to x_t
p_i	Hit ratio at ith level
$P(C_i)$	Success function

$P_t(i)$	Parent stack (before reference to x_t, block size B_i)
$p_t(\mathrm{E})$	Distance of block E from the top of $P_t(i)$
$R_t(i)$	Priority list
$r_t(\mathrm{E})$	Distance of block E from the top of $R_t(i)$
$S_t(i)$	Stack (before reference to x_t, block size B_i)
$s_t(E)$	Distance of block E from the top of $S_t(i)$
$Q(D)$	Decision at level D of a stack
T_i	Access time to ith level
$x_t(i), x_t$	Block (size B_i) referenced at time t
$X(i), X$	Reference string (block size B_i)
$y_t(D), y$	Block replaced from buffer (size D) at time t

Acknowledgment

The author is grateful for the constructive remarks made by one of the referees concerning Lemma 5.

References

1. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal* **9,** 718 (1970).
2. D. R. Slutz and I. L. Traiger, *Determination of Hit Ratios for a Class of Staging Hierarchies*, IBM Research Report RJ 1044, San Jose, California, May 1972.
3. G. S. Shedler, *A Cyclic Queue Model of a Paging Machine*, IBM Research Report RC 2814, Yorktown Heights, New York, March 25, 1970.
4. L. A. Belady and F. P. Palermo, "On-line Measurement of Paging Behavior by the Multivalued MIN Algorithm," *IBM J. Res. Develop.* **18,** 2 (1974).
5. I. L. Traiger and D. R. Slutz, *One-pass Technique for the Evaluation of Memory Hierarchies*, Research Report RJ 892, July 1971.

Received December 10, 1973; revised March 1, 1974

The author is located at IBM Canada, 5 Place Ville Marie, Montreal, Canada.