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

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

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

Evaluation techniques for storage hierarchies

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

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

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

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

hierarchy concepts

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

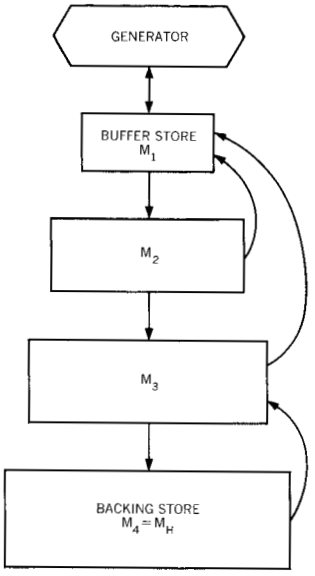
A hierarchy management facility is included to control the movement of pages and to effect the (generally dynamic) association between the logical address space and the physical address space of the hierarchy. When the CPU references a logical address, the hierarchy management facility first determines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

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

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

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

Figure 1 Linear storage hierarchy



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

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

objectives of the paper

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

The system model

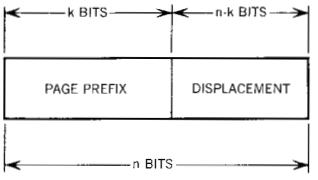
basic model concepts

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

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

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

Figure 2 Logical address

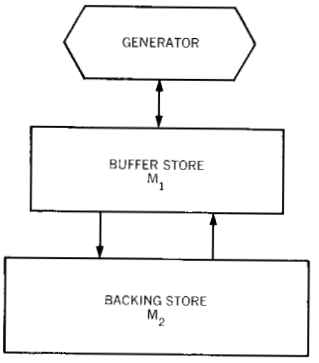


The capacity of the backing store is assumed to be at least 2^k page frames, and all logical pages initially reside in the backing store. At any time, each logical page resides in exactly one page frame of the hierarchy. A *mapping function* is associated with each hierarchical level, and specifies for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

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

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

Figure 3 Two-level hierarchy



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

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

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

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

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

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

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

$$B_0(C) = \phi$$

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

$$\gamma_t = |\Gamma_t|$$

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

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

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

Least recently used replacement

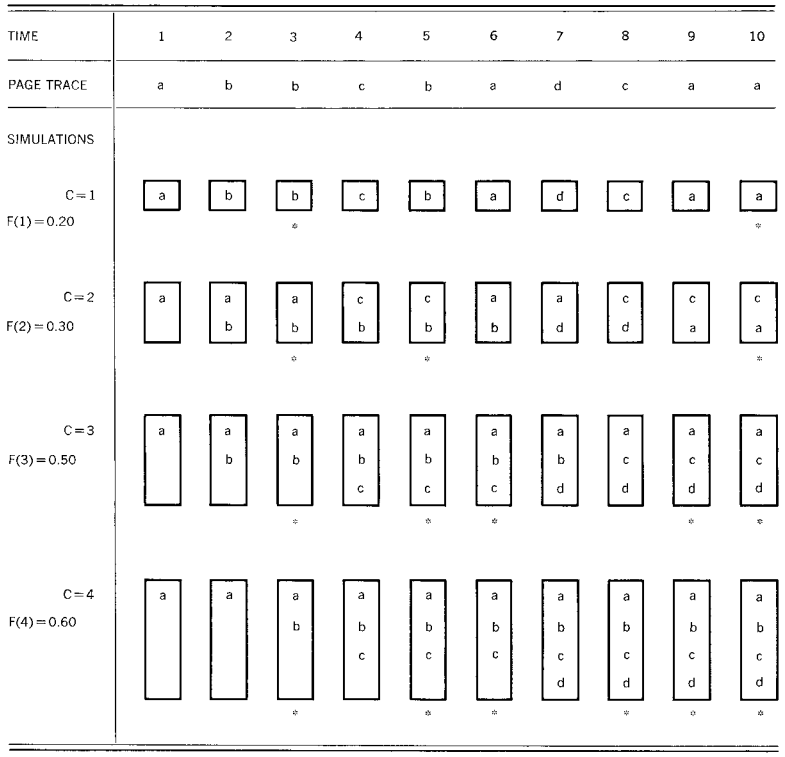
We now consider a particular replacement algorithm called "least recently used" (LRU), and show that the entire success function can be obtained by stack processing in a single pass of the address trace. Briefly, the single-pass technique requires the maintaining of a list of pages, called an LRU stack,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

success function

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

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

Figure 4 Determining success function by buffer simulation



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

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

$$B_t(1) \subset B_t(2) \subset \cdots \subset B_t(\gamma_t) = B_t(\gamma_t + 1) = \cdots \tag{1}$$

where

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

and

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

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

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

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

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

and

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

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

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

Hence

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

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

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

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

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

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

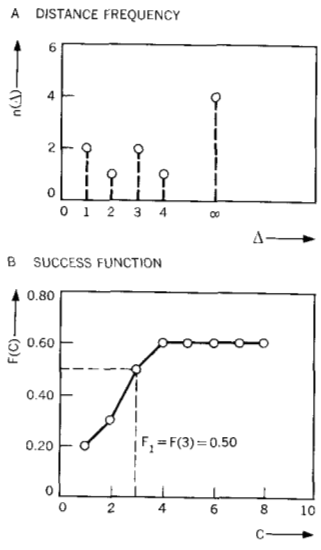
We call C_t the *critical capacity* since, from the inclusion property given in Equation 1, $x_t \in B_{t-1}(C)$ if and only if $C \geq C_t$. If x_t has not been previously referenced, we set $C_t = \infty$ because x_t is not contained in a buffer of any finite capacity.

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

Figure 5 Sequence of LRU stacks

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

Figure 6 Obtaining success function from distance frequencies



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

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

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

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

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

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

and the success function is given by the expression

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

In practice, the set $\{n(\Delta)\}$ can be determined from a set of distance counters, as shown in Figure 5. All counters are set initially to zero, and the counter for each distance Δ is incremented whenever

that distance occurs. For k-bit page numbers, we need at most $2^k + 1$ counters, corresponding to $1 \leq \Delta \leq 2^k$ and $\Delta = \infty$. At the conclusion of a page trace, the final values of the distance counters are the values $\{n(\Delta)\}$, and $F(C)$ is obtained from Equations 4 and 5.

numerical example

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

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

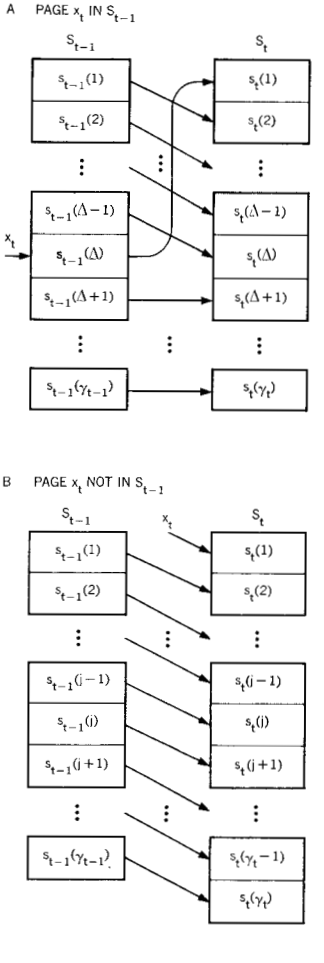
Note that $F(C)$ is always a monotonic, non-decreasing function of C for LRU replacement, since $F(C)$ is obtained by cumulative summation as shown in Equation 4. Also, $F(C)$ never exceeds $(L - \gamma_L)/L$ for any capacity, because all pages initially reside in the backing store.

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

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

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

Figure 7 Constructing LRU stacks



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

Stack algorithms

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

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

stack generation

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

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

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

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

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

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

and

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

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

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

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

for

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

stack algorithm identification

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

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

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

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

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

and

$$y_t(C+1) = \min [B_{t-1}(C+1)] \tag{8}$$

$$= \min [B_{t-1}(C), s_{t-1}(C+1)] \tag{9}$$

$$= \min \{\min [B_{t-1}(C)], s_{t-1}(C+1)\} \tag{10}$$

$$= \min [y_t(C), s_{t-1}(C+1)] \tag{11}$$

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

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

stack updating

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

$$s_t(1) = x_t \tag{12}$$

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

$$s_t(\Delta_t) = y_t(\Delta_t - 1) \tag{14}$$

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

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

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

$$s_t(1) = x_t \tag{16}$$

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

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

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

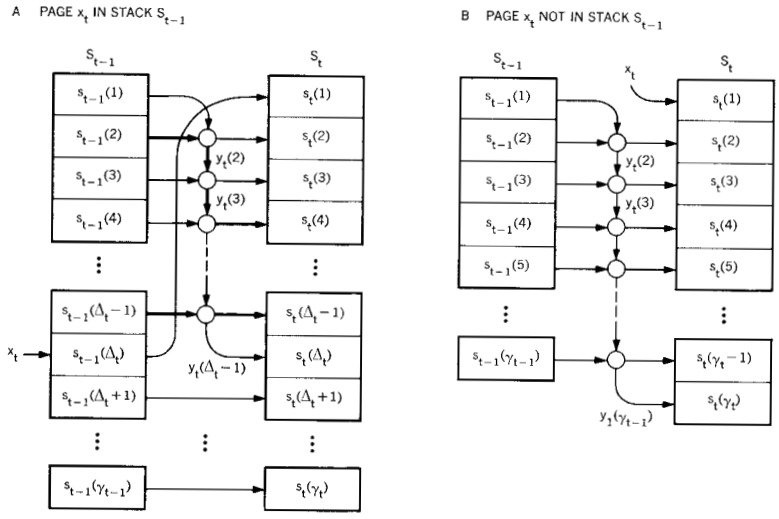
$$\gamma_t = \gamma_{t-1} + 1$$

is now full.

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

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

Figure 8 Stack updating



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

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

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

and Equations 13 and 17 then reduce to

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

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

examples of stack algorithms

Let us now examine several examples of stack algorithms. In general any replacement algorithm that bases its decisions on some page usage quantity, whether measured or predicted, naturally induces a priority list and is, therefore, a stack algorithm. One example, of

course, is LRU, and another example previously mentioned is least frequently used (LFU) replacement.

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

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

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

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

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

Other stack algorithms may base their decisions on information from the programmer or compiler, or on properties of the computer system. For example, the programmer or compiler may supply to the system[14] special "program directives" that indicate which pages

should be given high priorities in the immediate future. Another case is where the operating system assigns priorities to program pages in a multiprogrammed system, based perhaps on the position of the program in a task queue. If all the pages in the address space can be ordered in a priority list P_t for each time t, the resulting replacement algorithm is a stack algorithm.

first-in/ first-out

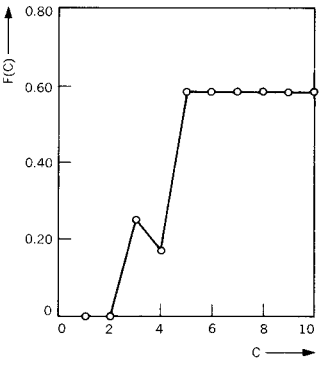
In the examples given, we see that priority lists can arise in a variety of ways. We now consider a replacement algorithm called "first-in/first-out" (FIFO) that is not a stack algorithm. Under FIFO, the page that has remained in the buffer for the longest (continuous) time up to time t is removed.

A peculiarity of FIFO is illustrated by the following page trace

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

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

Figure 9 Success function for FIFO replacement



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

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

An optimum replacement algorithm

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

an optimum replacement algorithm. Using certain properties of LRU and OPT, the entire success function for OPT can be determined in two passes of a page trace.

OPT

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

Figure 10 Example of OPT replacement

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

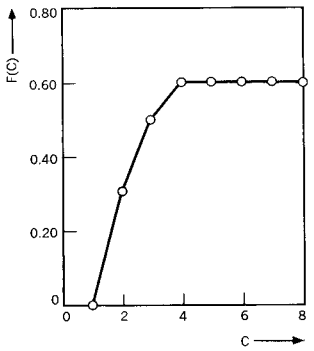
A formal proof that OPT is an optimal replacement algorithm is given in the Appendix. We note here that OPT is not realizable in an actual computer system because it requires knowledge of future page references. However, OPT does serve as a useful benchmark for any replacement algorithm, including stack-type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

stack processing example

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

Figure 11 Stack processing and success function for OPT replacement

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



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

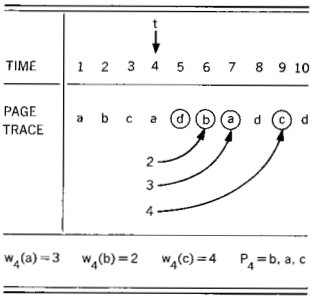
forward distance

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

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

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

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



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

maximum success function

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

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

Figure 13 Two-pass technique for LRU and OPT replacement

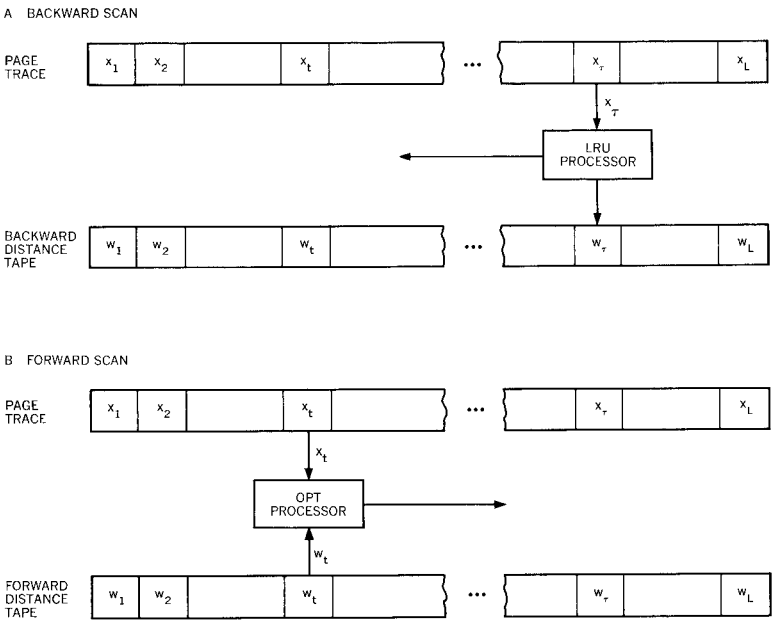
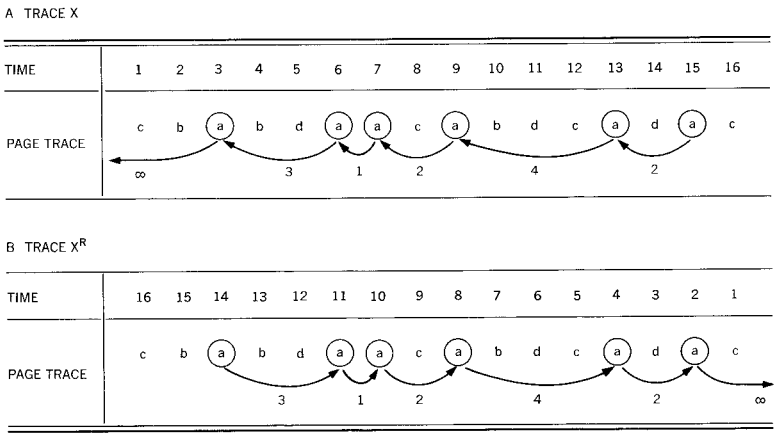


Figure 14 Sequence of LRU distances for page a



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

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

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

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

Random replacement

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

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

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

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

One difficulty in implementing RR is the generation of the random priority list P_t. Fortunately, it is possible to update the stack without actually constructing the entire priority list. Assuming that $\Delta_t > j$,

let $q_j(t)$ denote the probability that page $s_{t-1}(j)$ has priority over page $y_t(j-1)$ at time t. If $s_{t-1}(j)$ does not have priority over $y_t(j-1)$, we know that $s_{t-1}(j) = \min[B_{t-1}(j)]$. Since this occurs with probability $1/j$, we obtain

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

or

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

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

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

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

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

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

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

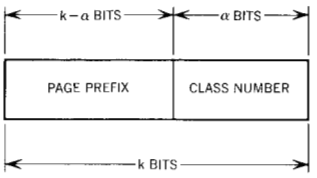
to reflect the probabilistic properties of the desired replacement algorithm for every capacity C.

Congruence mapping

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

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

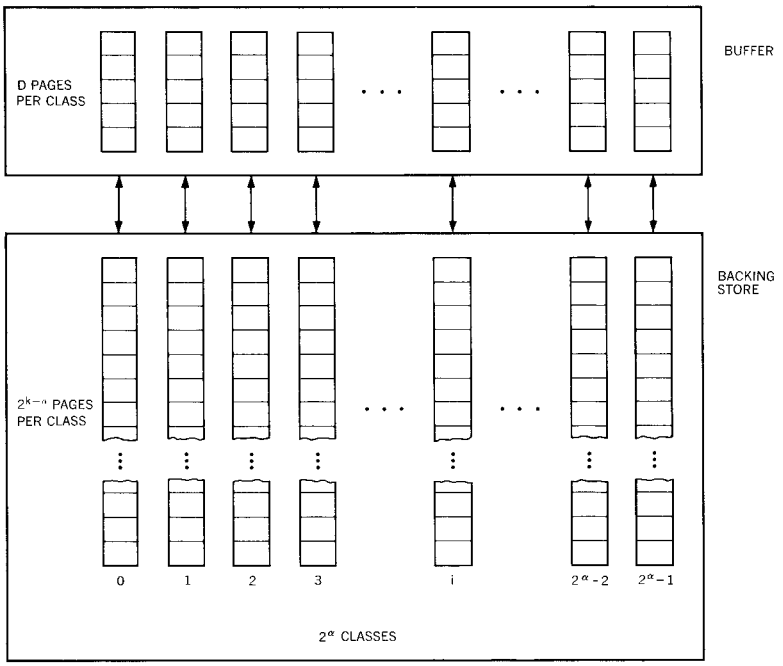
Figure 15 Page number



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

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

Figure 16 Two-level hierarchy with congruence mapping



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

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

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

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

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

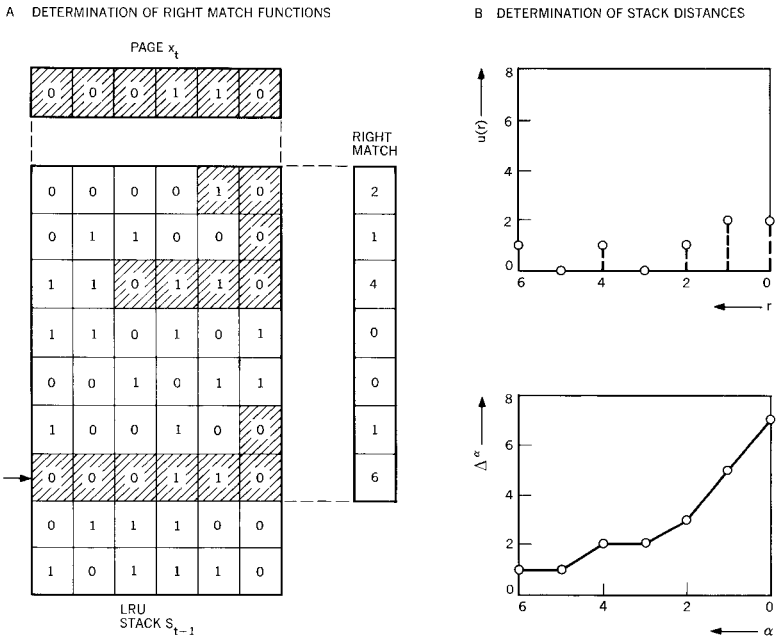
A simple procedure for determining Δ_t^α for all α is to scan down the stack, and maintain a set of right match frequency counters $\{\mu(r)\}$ for $0 \leq r \leq k$. Counter $\mu(r)$ is incremented whenever RM(x, y) is equal to r. If page x has been previously referenced, we eventually find RM$(x, y)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

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

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

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

Figure 17 Right match function for LRU replacement



Multilevel hierarchies

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

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

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

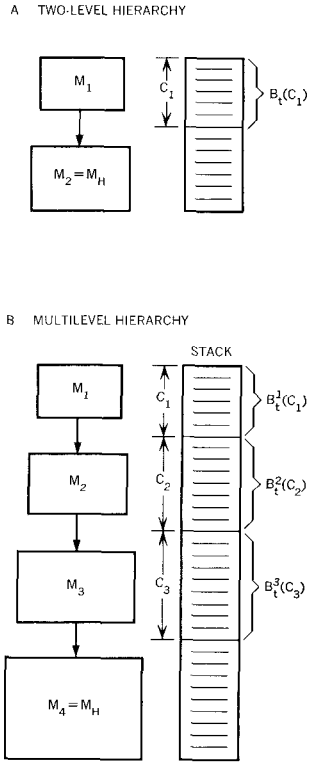
When page x_t is accessed from some level M_i (for $2 \leq i \leq H-1$), a page is replaced from each of the levels $M_1, M_2, \cdots, M_{i-1}$. The page replaced from level M_{i-1} is guaranteed to find space at level M_i, since a page frame was vacated by x_t. When page x_t is accessed from the backing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

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

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

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

Figure 18 Relationship between stack and hierarchy levels



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

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

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

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

with the result that

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

and

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

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

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

and

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

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

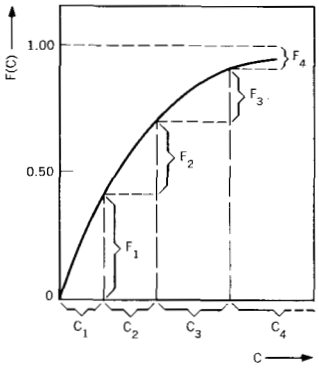
The significance of this result is that a stack distance Δ, where $C_1 + \cdots + C_{g-1} < \Delta \leq C_1 + \cdots + C_g$, corresponds to an access to hierarchy level M_g, and the relative number of such Δ's is simply the access frequency F_g to that level. Thus

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

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

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

Figure 19 Obtaining access frequencies from success function



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

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

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

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

$$F_i^\alpha = \begin{cases} F^\alpha(\sigma_i) - F^\alpha(\sigma_{i-1}) & \text{for} \quad 1 \leq i \leq H - 1 \\ 1 - \sum_{i=1}^{H-1} F_i^\alpha & \text{for} \quad i = H \end{cases} \tag{27}$$

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

Possible extensions

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

- Pre-loading program pages into the buffer for starting execution
- Loading a working set[19] of pages into the buffer when resuming program execution
- Returning all pages to the backing store upon program interruption
- Maintaining copies of pages in several levels of the storage hierarchy
- Bringing pages to the local store only for fetch operations
- Returning pages to the backing store for references such as stores from an I/O channel
- Moving unequal size pages or segments between levels

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

The evaluation of fetch-store hierarchies requires that the generator tag each reference as either a fetch or a store. For fetches, the priority list and the stack are updated, and a fetch distance Δ^f is recorded. For stores, neither the priority list nor the stack is up-

dated, but a store distance Δ^s is recorded. The distributions $\{n^f(\Delta^f)\}$ and $\{n^s(\Delta^s)\}$ can then be used to determine the fetch and store access frequencies to each level of the hierarchy. It should be clear that this technique also works if congruence mapping is included. We can also consider a modified fetch-store design where the page usage statistics are updated for a store operation even though no page motion occurs. This change is incorporated by updating the priority list for both fetches and stores. Thus, for modified fetch-stores, the net change in our model is that the stack is not updated for store operations.

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

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

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

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

Concluding remarks

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

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

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

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

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

ACKNOWLEDGMENT

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

Appendix

Two results mentioned in the paper concerning the OPT replacement algorithm are proved here. To do this, it is first shown that given any trace and replacement algorithm (not necessarily using demand

paging) another replacement algorithm exists that uses demand paging and causes the same or a fewer total number of pages to be loaded into the buffer. This result is used to show that OPT is an optimal replacement algorithm and, in fact, that OPT causes the minimum total number of pages to be loaded into the buffer. Finally, it is shown that the success function under OPT for any trace is identical to the success function under OPT for the reverse of the trace.

Definition

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

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

Definition

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

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

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

Definition

Let X be a trace and B_0 (where $|B_0| \leq C$) an initial state. A sequence of states $B = B_0, B_1, \cdots, B_L$ is a *valid sequence* if $x_t \in B_t$,

for $1 \leq t \leq L$. A policy pair P and Q is a *valid pair* for X and B_0 if application of the pair results in a valid sequence.

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

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

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

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

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

Theorem 1

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

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

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

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

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

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

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

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

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

optimum replacement algorithm

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

Definition

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

Definition

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

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

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

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

Lemma 1

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

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

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

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

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

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

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

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

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

$$B_{i_a}' = T_{i_a} + \{a\}$$

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

which can also be written as follows:

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

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

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

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

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

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

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

Rewriting the buffer states at time ℓ as

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

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

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

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

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

which again can be treated as in Case 3A.

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

Theorem 2

OPT is an optimal replacement algorithm

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

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

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

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

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

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

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

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

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

Theorem 3

OPT minimizes page loading

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

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

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

Lemma 2

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

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

Lemma 3

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

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

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

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

Theorem 4

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

forward/ backward OPT

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

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

Consider now the state sequence ${}^rB_L, {}^rB_L, {}^rB_{L-1}, \cdots, {}^rB_2, {}^rB_1$. Since $x_L \in {}^rB_1, x_{L-1} \in {}^rB_2, \cdots, x_2 \in {}^rB_{L-1}, x_1 \in {}^rB_L$, we see that this sequence is a valid (not necessarily demand) sequence for the trace X. Let us denote the corresponding valid policy pair as P' and Q'. We observe first that, since ${}^rE_C = B_C$, we have ${}^rB_L = B_C = B_0$. Thus P' and Q' (as well as P^O) are valid policies for X and B_0. Next we observe that ${}^rB_L = {}^rB_{L-1} + \{{}^rq^O_L\} - \{{}^rp^O_L\}$ can be written as ${}^rB_{L-1} = {}^rB_L + \{{}^rp^O_L\} - \{{}^rq^O_L\}$. But we also have ${}^rB_{L-1} = {}^rB_L + \{q'_2\} - \{p'_2\}$, which yields $q'_2 = {}^rp^O_L$ and $p'_2 = {}^rq^O_L$, since ${}^rp^O_L \cap {}^rq^O_L = \phi$. Similarly, since ${}^rB_{L-1} = {}^rB_{L-2} + \{{}^rq^O_{L-1}\} - \{{}^rp^O_{L-1}\}$, we have $q'_3 = {}^rp^O_{L-1}$ and $p'_3 = {}^rq^O_{L-1}$. Continuing in this manner we can show that

$$\left.\begin{aligned} q'_t &= {}^rp^O_{L+2-t} \\ p'_t &= {}^rq^O_{L+2-t} \end{aligned}\right\} \quad \text{for} \quad 2 \leq t \leq L \tag{A5}$$

Now, since $x_L \in {}^rB_0$ (recall that ${}^rB_0 = {}^rB_C$), it follows that ${}^rp^O_1 = {}^rq^O_1 = \phi$. Similarly, since $x_1 \in B_0$ (recall that $B_0 = B_C$), it follows that $p'_1 = q'_1 = \phi$. We can then trivially assume that $p'_1 = {}^rq^O_1$ and $q'_1 = {}^rp^O_1$. The significance of this is that, using Equation A5, we have established a one-to-one correspondence between P' and ${}^rQ^O$, and between Q' and ${}^rP^O$. In particular, $|\phi|_{P'} = |\phi|_{{}^rQ^O}$ and $|\phi|_{Q'} = |\phi|_{{}^rP^O}$. We now observe that $|\phi|_{{}^rQ^O} = |\phi|_{{}^rP^O}$, since $|{}^rB_0| = |{}^rB_1| = \cdots = |{}^rB_L| = C$. In other words, ${}^rp^O_t = \phi$ if and only if

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

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

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

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

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

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

CITED REFERENCES

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

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