UC 32
LBL-12370

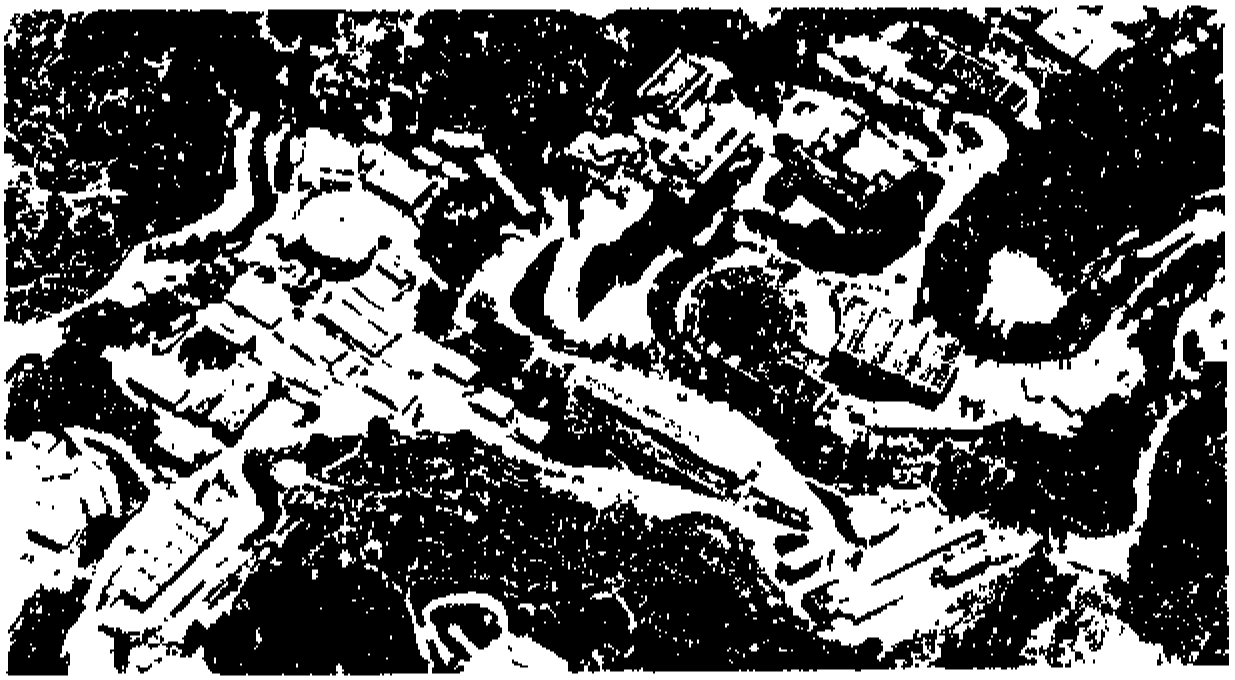
Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Physics, Computer Science & Mathematics Division

EFFICIENT METHODS FOR CALCULATING THE SUCCESS FUNCTION OF FIXED SPACE REPLACEMENT POLICIES

Frank Olken
(M.S. thesis)

May 1981



Prepared for the U.S. Department of Energy under Contract W-7405-ENG-48

LBL 12370

Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies

Frank Olken

Electrical Engineering and Computer Science Dept.
University of California, Berkeley

and

Computer Science and Mathematics Dept.
Lawrence Berkeley Lab,
University of California
Berkeley, California 94720

May 22, 1981

This research was supported by the Applied Mathematical Sciences Research Program of Office of Energy Research, U.S. Dept. of Energy under contract W-7405-ENG-48.

Abstract

In this paper we are concerned with efficient methods for calculating the success function of replacement policies used to manage very large fixed size caches. Such problems arise in studying the caching of files on disk.

We review earlier work by Coffman and Randell, and Mattson et al. We characterize a class of replacement policies for which it is possible to evaluate the success function for a single cache size in time O(n*log(s)), where n is the number of memory references in the trace and s is the size of cache. We then construct an algorithm to evaluate the success function for the Least Recently Used replacement policy in time O(n*log(s)), for cache sizes smaller than s. This algorithm runs in bounded memory, O(s). We also show how to modify Bennett and Kruskal's algorithm to run in bounded space. The two algorithms have the same asymptotic running times (within a constant factor). Measured running times for the classic LRU algorithm, Bennett and Kruskal's algorithm, and our new algorithm are compared.

We consider the impact of variable size segments (files, rather than fixed size pages), and deletions on algorithms for calculating success functions.

Table of Contents

1. Introduction

1.1. File System Storage Hierarchy Management

We wish to study policies which automatically manage the migration of data within storage hierarchies used to implement file systems. We shall use the term "virtual memory system" to denote a hierarchy of storage devices and a set of policies to manage the movement of information between the levels of the storage hierarchy [1]. The effect of such a virtual memory system is to present the illusion of a single level, uniformly addressable memory. The user need not concern himself with the actual location of the information in the storage hierarchy.

The topmost level of the storage hierarchy is comprised of fast, expensive storage devices, such as semiconductor random access memories (RAM). Lower levels are built from progressively slower, cheaper storage devices: magnetic disks and tape drives.

Classical virtual memory studies have been concerned with the caching of executing programs.

We are interested in studying two types of caches used to implement file systems. One such cache keeps fixed size blocks (sometimes called pages, typically 512 bytes to 4K bytes in size), which normally reside on disk, in a buffer pool in RAM. We will call this a disk cache. Some database management systems such as IMS use fairly large ones [Benn75]. Another type of cache moves entire files between disk (the cache) and tape (the secondary store). We will call this a file cache [Smith79, Stritt77]. On those occasions where we are indifferent to the nature of the cache, we shall use the generic term segment to refer to page, block, or file.

File caches are distinguished by the variable size unit of information moved between different levels of the storage hierarchy. Deletions of blocks from disk caches and files from file caches are commonplace, whereas deletions are nonexistent in most program caches.

It has been observed that programs do not reference memory randomly. Instead programs exhibit locality of reference, i.e., they tend to reference memory locations which are "near" those referenced recently. If one keeps recently referenced information in the higher levels of the storage hierarchy, then it is possible to build a storage system whose average access time is close to that of the topmost (most expensive) level and yet whose average unit cost is close to that of the bottommost (cheapest) level.

Below are definitions of some standard terms used in this area [Coff73]. Consider a two level hierarchy in which fixed size blocks (pages) are transferred between levels. If a page is referenced which is not in the top level (the cache) and the cache is full then a "replacement policy" decides which page already in the cache should be evicted ("pushed") to make room for the incoming ("pulled") page. If the page referenced was in the cache we say that a "hit" occurred. If the page was not in the cache we call the event a "miss". The hit ratio for a specific size cache, managed by a specific policy for a particular

[1] Others have used this term to refer specifically to the storage system used to hold executing programs. We use it here in a generic sense.

sequence of memory references is simply the ratio of hits to the total number of memory references. The miss ratio is the ratio of the number of misses to the total number of memory references. The success function is simply the hit ratio as a function of cache size.

We would like to investigate how well various policies for managing the file system caches work and how big a cache is needed to obtain a specified hit ratio. To answer these questions we would like to have efficient methods of calculating the success functions of replacement policies. The traces of memory references (address traces) which we will use to evaluate various replacement policies are typically several hundred thousand references long[2]. Evidence to date suggest that the traces in question will exhibit poor locality, i.e., that very large caches are necessary to obtain high hit ratios[3]. Earlier work studying caches for holding executing programs typically dealt with cache sizes of a few dozen pages. Since the running time of the classic methods of computing success functions depends on the locality of reference in the address traces the efficiency of the calculations was not as pressing a problem in the earlier work.

For the problems we wish to address, both the running time and storage requirements of the evaluation algorithms are important.

In this paper we begin by reviewing a taxonomy of replacement policies, characterized according to the computational complexity of the algorithms used to evaluate their success functions. We then report a new algorithm for evaluating the success function of the Least Recently Used (LRU) policy. The proper treatment of deletions of files from a cache, and of references (updates) which alter the size of files in a cache is explained. Finally we present performance measurements of various algorithms for calculating LRU success functions.

1.2. Machine Model for Complexity Results

In the next sections, we will examine the running time and storage requirements of several algorithms both theoretically and empirically.

The theoretical results on computational complexity of the evaluation algorithms will be presented in terms of a machine with a fixed size random access memory. We shall assume that the usual computer instructions can be performed on single word operands in constant time, that the algorithms execute on a word machine, and that the words are of sufficient size to hold the variables we are interested in: the segment name and the reference time. Memory requirements are stated in words. These are conventional assumptions in complexity theory.

[2]These traces are obtained by processing System Management Facility traces of file system activities on IBM mainframes. In this paper we use only synthetic traces to assess the efficinncy of various methods of calculating success functions.

[3][Benn75] is concerned with evaluating caches containing several hundred pages and [Stritt77] and [Smith79] are concerned with file caches containing thousands of files.

2. Stack Policies

In 1970 Mattson, et al. [Matt70] characterized a class of replacement policies, called "stack policies", for which it is possible to evaluate the success function for all cache sizes in one pass across the address trace. All stack policies also have monotone nondecreasing hit ratios as a function of increasing cache size [Matt70].

A stack policy specifies, each time a page is referenced, a total ordering on all pages which have been referenced up to the present. This ordering is called a priority list[4]. For a given cache size, the stack policy will always replace (push out of the cache) the page with the lowest priority. Mattson et al. showed that this is equivalent to the requirement that such policies satisfy an inclusion property. The inclusion property holds if, for all address traces, a given size cache always includes all of the pages in smaller caches at the same point in the address trace[5].

As a consequence it is possible to represent the contents of all size caches (at a specific point in time) by a single stack equal to the size of the largest cache. The top n cells of the stack contain the names of the pages in a cache of size n at a time t in the address trace. The location of a page in the stack (measured from the top of the stack) is the minimum memory capacity (MMC) of a cache which would contain the page referenced. Because of the inclusion property all larger caches would also contain this page. The hit ratio for a cache of size S is the fraction of references whose MMC is less than or equal to S. Hence to evaluate the success function for all cache sizes one calculates the MMC for each reference, tabulates the the frequencies of MMC's, normalizes by the total number of references to give a probability distribution, and then integrates to yield the cumulative distribution function of the MMC's (i.e., the success function).

To calculate the MMC's one must update the stack at each memory reference to reflect the new cache contents. For example, consider the stack shown in figure 1 and suppose that the next page referenced is page 3 located fifth from the top in the stack. In this example, small numbers designate high priorities. Assume that the priority number for each page has been generated by some policy (not LRU) and that it does not change except when the page is referenced.

[4]For example one might order the pages according to the time since each was last referenced, with least recently used pages having the lowest priority.

[5]The first-in-first-out (FIFO) replacement policy does not satisfy the inclusion property and hence is not a stack policy.

Stack	Stack before reference		Page	Stack after reference	
Position	Page	Priority	Pushed	Page	Priority
1)	6	4	6	3	0
2)	4	1	6	4	1
3)	5	6	5	[illegible]	4
4)	7	3	5	[illegible]	3
5)	3	9	none	5	6
6)	8	2	none	8	2
7)	9	5	none	9	5

Figure 1: Stack before and after reference to page 3, at stack position 5. Note that 0 = highest priority. For s[illegible]plicity we have assumed that the priorities of nonreferenced pages remain unchanged.

Clearly the first element of the stack, page 6, must be pushed (from the cache of size 1) to make room for the newly referenced page. The newly referenced page 3 will be placed in stack position 1. The page pushed from the cache of size two will either be the page pushed from the cache of size one, page 6, or the page which was in the cache of size two but not size one (i.e., page 4 at stack position 2). Page 6 has priority 4, while page 4 has priority 1. Since lower numbers designate higher priorities in this example, page 4 has higher priority. Hence the lower priority of these two pages, page 6, is pushed out of the cache of size 2, while the higher priority page, page 4, remains (at stack location 2). Similarly the page pushed from the cache of size 3 will either be the page pushed from the cache of size 2 (page 6), or the page (page 5 at position 3 in the stack) which was in the cache of size three but not the cache of size two. Again the lower priority page (page 5) will be pushed, while the higher priority page (page 6) is kept in the cache of size three (at stack location 3). The fourth stack position holds the page which would be contained in cache of size four, but not size three. Here one must either push page 5 (priority 6) which was pushed out the cache of size three, or page 7 (priority 3) at stack location 4 which was the page missing from a cache of size three but contained in a cache of size four. Page 5, having the lower priority, is pushed from the cache of size four, while page 7 rem[illegible]t stack location 4. The fifth stack location contains the page [illegible] is contained in a stack of size five, but not a cache of size 4. [illegible] is page 3, the page being referenced. It is now placed at the top of the stack. The page pushed from the cache of size four, page 5, is placed in stack location 5. A hit is recorded for MMC 5.

Updating the stack for a memory reference thus amounts to a single pass of a bubble sort down the stack until one reaches the currently referenced page [Coff73]. The stack may be implemented either as an array or as a linked list. Assuming that one can determine the priority of a page in constant time this gives a running time of O(n*d), where n is the number of memory references in the trace and d is average distance from the top of the stack of the pages referenced.

2.1. Characterization of Stack Policies

In the remainder of the paper we characterize various classes of stack policies in terms of the computational complexity of algorithms for computing their hit ratios for various size caches. Section 3 concerns policies which can be processed in space proportional to the largest cache size considered. Section 4 concerns policies which are amenable to multi-pass hit ratio evaluation algorithms for problems too large to fit in memory. Section 5 is about policies for which the hit ratio can be efficiently calculated for a single cache size. Section 6 concerns a very small class of policies (containing only the Least Recently Used policy) for which one can efficiently calculate the hit ratio for all size caches.

3. Bounded Cache Sizes

Suppose that, given an trace of segment references, one only wants to evaluate the success function for a stack policy for cache sizes less than some constant C. One would hope that this could be done in space O(C).

If the policy does not use any information concerning the history of the segment prior to its most recent reference, there is no difficulty. Pages pushed from the cache of size C are simply discarded. Pages not found in the stack are recorded as misses.

But suppose that the policy does use the prior history of the segment to compute its priority, e.g. the Least Frequently Used (LFU) policy. Then pushing segments out of the stack of size C renders their history inaccessible when they are next referenced. This problem can be dealt with by preprocessing the trace in the manner described below.

Suppose that the replacement policy calculates the priority of a segment from only a small fixed set of statistics which summarize the previous history of the segment, and that, as new references occur, these statistics can be updated in constant time without recourse to any other information concerning the history of the segment. For example, the LFU policy would record the time since creation of the segment and the total number of references to the segment. All practical replacement policies can be so charactcterized.

Then one can preprocess the trace as follows. Split the trace by segment name into sufficient subtraces that the summary statistics for each subset of segments can be kept in main memory, retaining the chronological ordering within each subtrace. If the segment names are not contiguous the splitting can be done by calculating a hash function on the segment name. If the segment references do not already include timestamps, append timestamps to each segment reference.

Each subtrace is then processed separately as follows. Read through the subtrace updating a table of summary history statistics for each segment. Whenever a segment is referenced write out the segment reference along with the summary statistics calculated at the previous reference to the segment.

Finally merge all the subtraces of timestamped references (including the appended summary statistics) back together in chronological order. We can then process the trace using a stack algorithm for a bounded cache size, employing the preprocessed summary statistics whenever we reference a segment we have pushed out of the stack.

The computational complexity of the preprocessing is $O(n*\log(S/R))$, where n is the length of the trace, S is the number of distinct segments referenced (maximum stack size), R is the number of segment histories which will fit into main memory[6].

A similar technique can be used to evaluate the OPT policy. This policy has been shown to be optimal if we count all segment faults equally and ignore the cost of pushing dirty (modified) segments out the cache [Bela65],[Matt70]. It consists of pushing the segment which will be referenced furthest in the future. The technique described above can be applied here by reading the trace in reverse order. This can be done readily if the trace is stored on disk. Otherwise one might have to sort the trace twice.

4. Multi-pass Stack Algorithms

4.1. The Extension Problem

Now suppose that we want to construct a multi-pass evaluation algorithm which will evaluate the success function for cache sizes too large to fit the stack in memory. For those segments not found in the stack on the first pass, we pass the segment being pushed from the stack (with its history), and the segment being pulled to a second pass. In the second pass we simply continue with our bubble sort, adding the size of the stack in the first pass to the MMC's we calculate.

This solution to the construction of multi-pass evaluation algorithms was given by Coffman and Randell [Coff71]. They called it the "extension problem". He was concerned with finding the hit ratio for cache sizes greater than C, given information only on the occasions of faults (misses) from the cache of size C. This information (a "reduced trace" of the faulted segments) is generally much shorter than the original address trace. Furthermore, in a real system, a segment fault causes a trap to the operating system. Thus, acquiring a reduced trace is much easier than collecting a full trace.

Because the reduced trace is generally much smaller than the original trace, the combined space-time product of a two pass evaluation algorithm may be much better than a one pass algorithm[7]. (The stack for the second pass need only be kept around for a time proportional to the length of the reduced trace.)

Smith [Smith77] has advocated the use of reduced traces to obtain approximate results for paging policies where the policy under investigation in the second pass is not the same as in the first (reducing)

---

[6] S/R is the number of subtraces we split the original trace into, so that the tables required to preprocess each subtrace will fit into memory. If we are constrained (by memory requirements for file buffers) to split files at most k ways on each pass, then $\log_k(S/R)$ passes may be required to completely split the trace, i.e., the height of a merge tree whose fan-out is k and which has S/R leaves. Hence the splitting would require time $O(n*\log(S/R))$. The preprocessing for all of the subtraces can be done in time $O(n)$. Merging the subtraces back together will require time $O(n*\log(S/R))$.

[7]The space-time product of a computation is the integral of instantaneous memory usage over the course of the computation.

pass.

4.2. Separable Priorities

In order for multi-pass algorithms to be practical one must bound the amount of information which must be passed to the second pass at each segment fault from the first pass cache. For most policies, such as LRU or LFU, this is not a problem. The segment priorities in the second pass can be calculated from a fixed set of history statistics passed along with the segment when it was pushed by the first pass.

But suppose that the priority of the segment does not depend solely on its own history, but also that of other segments ("precursor segments"). Then one might have to pass a new priority list out with every segment pushed from the first pass. This could grow to enormous size.

In order to obtain practical multi-pass algorithms we will therefore restrict consideration to policies which have <u>separable priorities</u>, i.e., priorities which are solely the function of a (small) fixed set of statistics summarizing the previous history of the segment.

5. Time Invariant Relative (TIR) Priority Policies

We will now consider a class of stack policies for which one can efficiently calculate the hit ratio for a single large cache size. This is the same problem as managing a fixed size cache. Hence these policies are practical candidates for managing very large fixed size caches. Also, by constructing multi-pass evaluation algorithms for these policies one can evaluate the success function at a small number of points efficiently. For example one might be constrained to purchase memory for the cache in fixed size increments (each disk drive might hold several hundred megabytes). Alternatively, one might use a single size evaluation algorithm as the first pass in solving the extension problem.

To calculate the hit ratio for a specific size cache one simply simulates the management of the cache. For each memory reference, one must determine whether it is in the cache. If it is, one merely updates its history statistics (which are used in calculating its priority). If the segment is not in the cache then one must find the lowest priority segment in the cache and push it out.

Now suppose that the relative priorities are "time invariant", i.e., that the relative priorities of two segments do not change unless one is referenced[8]. Any policy in which the priority of a segment is a function solely of the segment's history up to the last reference plus a constant (identical for all segments) times the time since last reference to the segment is a "time invariant" policy, e.g., LRU. Thus at each reference the only change in the priority list is the position of the currently referenced segment. Hence one can represent the priority list as a heap[9]. In a heap one can find the minimum priority segment and delete it in time O(log C), where C is the number of segments in the cache. Thus the total computation time would be O(n*log(C)).

[8]Time invariant priority functions have been used in the context of processor scheduling by Ruschitzka and Fabry [Rusch77].

[9]If the relative priorities were not time invariant one would have to rebuild the heap at each reference, instead of just deleting and reinserting as single element.

This notion can be extended (as described below) to form a larger class of replacement policies at a modest increase in the computational complexity.

One could divide the segments into k classes, with time invariant relative priorities within each class. The priority of a segment could thus be a function of the history up to the last reference plus one of k constants times the time since last reference, e.g. a generalized LRU with different aging rates for different types of segments. To calculate a hit ratio one now must calculate the minimum of the k minima of each heap. Thus the running time will be bounded by $O(n*k)+O(n*\log(C))$.[10]

Another possibility would be to construct a composite piecewise time invariant relative priority function, by switching between several time invariant relative priority functions a finite number (k) of times as the time since last reference to a segment grows. This would allow the priority of a segment to be an arbitrary function of the segment history up to the last reference plus a piecewise linear function of the time since last reference. A hit ratio calculation for such a policy has running time bounded by $O(n*k) + O(n*k*\log(C))$, but the average running time will depend on the distribution of inter-reference intervals.[11]

Among the class of functions which generate time invariant relative (TIR) priorities are priority functions of the form $p(i,t)=a*t+b(i)$, where $p(i,t)$ is the priority of segment i at time t, a is constant for all segment and $b(i)$ is an arbitrary number unique to each segment and invariant between successive references to the file (e.g., it could be a function of the segment's history up to the time of last reference). In particular if one let t_{ref} = time since last reference to the segment, then $p(i,t-t_{ref})=a*(t-t_{ref})+b(i)$ generates time invariant relative priorities. Thus any linear function of the time since last reference will generate TIR priorities. Futhermore any monotone transformation of the linear priority function will generate TIR priorities. In particular

$$p(i,t-t_{ref}) = e^{(a*(t-t_{ref})+b(i)+c)} = g(i)*e^{(a*(t-t_{ref})+c)}$$

[10] The linear term n*k may be insignificant, in terms of the average running time, because one need only compute the minimum of the k minima of each heap when a segment fault occurs. For large cache sizes this should not occur very often.

[11] The distribution of inter-reference intervals determines the number of times the segment priority will have to be recalculated between successive references. Note that the priority of the segment must be recalculated, and the segment deleted from one heap and inserted in another each time one switches priority functions. In the example of a piecewise linear priority function of time since last reference one "switches" TIR priority functions whenever the time since last reference grows beyond the domain of a single linear segment of the piecewise linear priority function, i.e., at each of the kinks in the priority function. This accounts for the $O(n*k*\log(C))$ term in the bound on the running time. However, the minimum of the k minima of each heap need only be calculated when a segment fault occurs. This accounts for the $O(n*k)$ term in the bound on the running time.

will generate TIR priorities. Then for k-class or piecewise TIR priority policies one can choose different constants g(i),a,c for each class.

For example, suppose that our replacement policy ranks segments according to the hazard rate of each segment's inter-reference time distribution, i.e., according to the probability that the segment will be referenced in the next instant. Then one could employ multiple classes and piecewise TIR priority functions to attempt to fit the empirically estimated hazard rates for various types of segments[12]. In particular, the exponential priority function, $p(i,t)=g(i)*e^{(a*(t-t_{ref}))}$, represents a very strict notion of a proportional risk model, wherein g(i) represents the relative risk. Similar sorts of models have been widely used in biomedical research and there are techniques for fitting these models from censored data[13].

6. Least Recently Used Policy

6.1. Time Invariant Stack Positions

Now suppose that the relative ordering of two pages in the stack is time invariant (i.e., does not change except when one of the pages is referenced). For what class of stack policies is this the case? It is true for only one policy, Least Recently Used. To see this, observe that for LRU (and only for LRU [Coff73]), the priority ordering and the stack ordering are identical. Consider any policy for which this is not true, then the possibility arises that two pages at stack positions i and i+1 would be out of priority order. Thus any reference to a page further down in the stack would result in a reordering of the stack, but this violates time invariance of relative stack position.

6.2. Tree Representations

If the ordering of pages in the stack is time invariant, then one can effectively represent the stack as a binary tree[14]. Each node of the tree represents a page and the tree is ordered according to the last reference time for each page. The ordering used is "inorder" as defined by [Wirth76], i.e., a chronological ordering of the nodes in ascending time of last reference corresponds to visiting the nodes in the order of left, root, right.

When a page is referenced one finds it in the tree,[15] determines its position in the stack (as described below), deletes it from the tree,

---

[12]For example, one might distinguish segments by whether or not they were directory segments or file segments, whether the last reference was a read or a write, etc.

[13]Censored data on inter-reference times arises in file migration studies, because one can only observe the file system for a finite period of time. Thus there will be many inter-reference intervals which must be truncated when one ceases recording references.

[14]If the ordering of the pages on the stack was not time invariant then one would have to reconstruct the tree after each reference instead of simply deleting and reinserting the referenced page.

[15]One can find it in constant time by constructing a hash table on the page ID's with pointers into the tree.

and then reinserts it in the tree at the top of stack, i.e., as the rightmost child.

Since the top of stack position is always at one side of the tree, one would generate a very unbalanced tree unless some effort is made to rebalance the tree occasionally. We implemented a generalized AVL tree ([Fost73] and [Knuth73]), but other types of balanced trees would also work. AVL trees guarantee that the height of the tree (the longest distance of any node to the root) is O(log(C)), where C is the number of nodes in the tree (equal to the number of pages in the largest cache size considered). Furthermore any node can be inserted or deleted and the tree balance maintained in time O(log(C)) in the worst case. One can calculate the position of a page in the stack, delete it, and reinsert it in the tree, all in time O(log(C)).

To facilitate the determination of the position of a page in the stack, one stores in each node the number of pages in the subtree rooted at that node[16]. The tree is ordered on the time of last reference, so that all pages in the righthand subtree of a page have been referenced more recently than it. Then the set of pages which have been referenced more recently than page p are those which lie in the righthand subtree of p, contain p in their lefthand subtree, or lie in the righthand subtree of a page which contains p in its lefthand subtree. The position of a page p in the stack can thus be found by traversing a path from the page in question to the root of the tree[17]. Whenever one reaches a node (page) which includes p in its lefthand subtree (i.e., has been referenced more recently than p) we add the size of that node's righthand subtree, plus 1 (for the node itself). Finally, one adds the size of p's righthand subtree. This sum is the position of page p in the stack. Thus the stack position can be calculated in time O(log(C)) if the tree height is so bounded[18]. Furthermore, it is clear that recalculating the subtree sizes does not increase the complexity of insertions and deletions of an AVL tree except by a constant factor.

6.3. Bennett and Kruskal's Algorithm

In 1975 Bennett and Kruskal [Benn75] described an algorithm for calculating LRU hit ratios. This algorithm was built around the observation that the LRU minimum memory capacities (MMC's)[19] are simply the number of distinct pages referenced since the last reference to the page currently referenced.

They construct a bit vector as long as the address trace. Initially all the bits are set to 0. As each memory reference occurs at time t, bit b[t] is set to 1, and the bit, $b[t_{prev}]$, corresponding to the time

---

16 The algorithm for calculating the position of a page in the linear ordering represented by a the tree is taken from [Knuth73].

17 Recall that we obtained a pointer to the node containing the last reference to page p in constant time (on average) via a hash table.

18 If one wished to dispense with the father pointers one could search the tree from the root to the page in question, using the page's last reference time (obtained via the hash table) as a key.

19 i.e., the LRU hit depths (see section 2).

of last reference to the currently referenced page is cleared[20]. Thus at any point t in processing the trace, all the bits are 0 except those corresponding to the most recent references of each page. We can calculate the MMC of a reference by counting the number of 1 bits in the interval between the current time and the last reference to the page.

Bennett and Kruskal contrive to do this counting in time O(log(inter-reference time)). They do so by constructing a fixed structure m-ary tree atop the bit vector[21]. The leaves of the tree are the elements of the bit vector. Each internal node contains the sum of the counts stored in its children, i.e., the count of number of 1's in the leaves of the subtree (cf. the partial sum tree in Figure 2).

One can calculate the position of a page in the stack by first looking up the most recent previous reference time (t_{prev}) of the page (in the table one maintains) and then traversing a path from the leaf $b[t_{prev}]$ to the closest common ancestor in the tree of $b[t_{prev}]$ and $b[t]$[22]. At each node which contains $b[t_{prev}]$ in its lefthand subtree, we add the size of the righthand subtree. Updating the counts can be done by traversing the same path. One need not ascend higher than the closest common ancestor, at height O(log(inter-reference time)), because at that point the increase in subtree caused by setting b[t], and the decrease occasioned by clearing the bit corresponding to the previous reference cancel. Note that this does not occur on initial references to a page, when one must go all the way to the root of the tree. An example is shown in Figures 2,3 and 4.

[20] A table is maintained containing the most recent reference time for each page (t_{prev}).

[21] That is, a tree with an arbitrary m way fan out at each node.

[22] Assume one clock tick per reference.

Stack:

Page	Time last referenced	Depth in stack
6	11	1
8	10	2
1	9	3
7	8	4
4	7	5
5	2	6
2	1	7

Partial sum tree

```
                              7
              +++++++++++++++++++++++++++++++++
              3                               4
      +++++++++++++++++               +++++++++++++++++
      2               1               4               0
  +++++++++       +++++++++       +++++++++       +++++++++
  1       1       0       1       2       2       0       0
+++++   +++++   +++++   +++++   +++++   +++++   +++++   +++++
0   1   1   0   0   0   0   1   1   1   1   1   0   0   0   0
    ^
    |
    ----last reference to page 2
```

Figure 2: Stack and partial sum tree at time 11 prior to referencing page 2 at time 12. Location of last reference to page 2 is found via a hash table. The leftmost reference occurred at time zero.

Stack:

Page	Time last referenced	Depth in stack
6	11	1
8	10	2
1	9	3
7	8	4
4	7	5
5	2	6

Partial sum tree

```
                                  6
                 +++++++++++++++++++++++++++++++++
                 2                                4
         +++++++++++++++++                +++++++++++++++++
         1               1                4               0
     +++++++++       +++++++++       +++++++++       +++++++++
     0       1       0       1       2       2       0       0
   +++++   +++++   +++++   +++++   +++++   +++++   +++++   +++++
   0   0   1   0   0   0   0   1   1   1   1   1   0   0   0   0
```

Figure 3: Stack and partial sum tree after page 2 is pulled at time 12. A hit is recorded with MHC = 7. The leftmost reference occurred at time zero.

Stack:

Page	Time last referenced	Depth in stack
2	12	1
6	11	2
8	10	3
1	9	4
7	8	5
4	7	6
5	2	7

Partial sum tree

```
                                7
                ++++++++++++++++++++++++++++++++++
                2                               5
       ++++++++++++++++++               ++++++++++++++++++
       1                1               4                1
   +++++++++        +++++++++       +++++++++        +++++++++
   0       1        0       1       2       2        1       0
 +++++   +++++    +++++   +++++   +++++   +++++    +++++   +++++
 0   0   1   0    0   0   0   1   1   1   1   1    1   0   0   0
                                                   ^
                                                   |
                          most recent reference to page 2 ----
```

Figure 4: Stack and partial sum tree after page 2 is pushed onto stack at time 12. Hash table entry for page 2 is updated to point to the new most recent reference. The leftmost reference occurred at time zero.

As described by Bennett and Kruskal the algorithm does not run in bounded memory for a finite size cache and an arbitrarily long trace. Suppose one is only interested in caches smaller than C pages. On each fault from a full cache, clear the bit corresponding to the last reference to the pushed page and adjust the tree accordingly. Every C references, compress the bit string so that it is contiguous and rebuild the tree so that it begins with the earliest t_{prev}[23]. The algorithm now runs in bounded space O(C). To compress the bit string and recalculate the tree requires time O(C), but the event only occurs every C references, hence the complexity introduced by periodic compression is only O(n). The height of the tree is now bounded by O(log(C)). The time for each reference is thus min[O(log(C)), O(log(inter-reference time))]. The total running time is thus bounded by O(n*log(C)), the same result we got for the AVL tree algorithm.

[23] The table which used to contain the most recent reference times for each page (assuming 1 clock tick per reference), which was used to index into Bennett and Kruskal's bit vector, now simply contains pointers into the bit vector (not true reference times).

The similarity in running times is not surprising since both algorithms build similar trees. Bennett and Kruskal's algorithm builds a fixed structure sparse tree, which is periodically recompressed, whereas the AVL algorithm maintains a compact tree which is continuously rebalanced.

6.4. Two-level Linked List Algorithm

In section 2 a linked list algorithm for implementing stack policy hit ratio calculations was discussed, with running time O(n*d), where d is the average stack depth of referenced segments. The linked list algorithm can be be used to evaluate LRU, a stack policy.

The linked list algorithm can be modified to improve its performance by exploiting the time invariant relative stack positions generated by the LRU policy. The modified algorithm is known as Franta's [Fran77] Two-Level linked list algorithm for maintaining simulation event sets (a linear ordering with insertions, deletions). This algorithm uses a second linked list as an index into the primary linked list (hence the name). If the number of nodes in the index is the square root of the number of nodes in the primary list, and the index entries are uniformly spaced over the primary list, then at most sqrt(C) nodes in the index and sqrt(C) nodes in the primary list must be visited to find, rank, insert or delete a node in the primary list (where C = number of primary nodes, i.e., segments in the stack). Hence the running time for processing a trace of n references, with a stack size bounded by C segments is bounded by O(n*sqrt(C)).

This algorithm has been used by Ozalp Babaoglu [Baba80]. It offers intermediate complexity and performance. It could also be used to evaluate the hit ratio for a single cache size of a time invariant relative priority policy (other than LRU).

6.5. Deletions and Variable Segment Sizes

Thus far we have limited the discussion to memory hierarchies in which one moves fixed-size blocks of information (pages) between various levels. Furthermore we have ignored the question of how to treat deletions.

The file system replacement policies we wish to investigate move entire files (segments) of various sizes between levels of the storage hierarchy. External fragmentation of storage (disks) is not an issue since most file systems can store files on noncontiguous disk pages.

All of the LRU algorithms discussed can be extended to accommodate variable size segments. If the size of a given segment never decreases and segments are never deleted then we merely sum the sizes of the segments instead of counting pages.

Suppose one were to delete a segment from the stack. Then the MMC's of all segments further down the stack would decrease, because one calculates the MMC's by summing the sizes of segments higher on the stack. But this is impossible, because under a demand fetch policy a segment never moves back into a smaller size cache unless it is referenced. To be accurate one would have to record the maximum depth in the stack a segment ever attained since the last reference. Similar problems arise if one allows segment sizes to decrease when segments are referenced (e.g., a file is overwritten by a newer smaller version).

Greenberg [Green74] discussed deletions but did not implement a correct algorithm. Instead he estimated the error introduced by his approximate algorithm. Smith [24] suggested a method of dealing with deletions which we have incorporated into these LRU algorithms. Whenever a segment is pulled from the stack (either due to a reference or deletion) one records a gap at the location in the stack where the file was. This gap is the same size as the pulled file. Whenever a file is pushed onto the stack one starts at the top of the stack to squeeze out as much gap space as the size of the file pushed. Finally whenever one wishes to calculate the depth of a segment in the stack one sums the sizes of the segments and gaps above the target segment in the stack.

This modification to the LRU algorithms increases the space requirements of each algorithm by at most a constant factor (<2), since one can collapse adjoining gaps so that we have at most one gap per segment in the stack. Whereas before one only had to keep track of the space occupied by files on the stack, one now must maintain parallel records of the gaps adjoining each file. In the linked list algorithm one simply stores the size of the gap (if any) prefixing each segment as an entry in the corresponding segment descriptor. In both the AVL trees and Bennett and Kruskal partial sum hierarchies each node now contains one entry for the total space occupied by the files in the subtree and one entry for the total gap space within the subtree. The parallel data structures for gap space and occupied space were suggested by the the work reported in [Eggers80].

For each pull or deletion one must create (or increase) one gap. For each push, one squeezes the topmost gaps until the total space squeezed equals the size of the pushed segment. The average number of gaps which must be squeezed for each push thus depends on the the distribution of sizes of pushed segments and the distribution of the sizes of the gaps (generated by pulls or deletions). If all segment are the same size (pages), or if segments never change size nor are deleted then the number of gaps squeezed on a push is one (zero if no gaps exist). If all segment references (pushes, pulls, and deletions) are drawn from stationary distributions of segment sizes (typically the same distribution) then one can construct a bound on the average number of gaps squeezed per push which independent of both the stack size, and the length of the trace.

Hence impact of deletions on the running time of the LRU algorithms hinges on the effort required to find and modify each gap. For the AVL tree and Bennett and Kruskal's algorithms one can incorporate the gaps into the same data structures and treat them similarly to the file sizes. In each the time to access or modify the gaps is bounded by $O(\log S)$, where S is the size (i.e., cardinality) of the stack. For the AVL algorithm this represents no change in running time. But Bennett and Kruskal's algorithm has now slowed down to within a constant factor of the AVL algorithm.

One could proceed in the same fashion for the linked list algorithm[25], storing the size of the preceding gap (if any) with each segment node on the linked list representing the stack. However, the average time to perform a push would then be proportional to the average

[24] Private communication.

[25] See sections 2 and 6.4.

stack depth of the topmost gaps. Potentially this could be much worse behaved than the average stack depth of referenced segments since gaps are generated by deletions as well as references. While most deletions in a file system are temporary files (found near the top of the stack) others are old relics near the bottom of the stack. Furthermore, references to temporary files will mostly be be deleted from compressed traces used for studying long term file migration. Installations (such as Lawrence Livermore Lab) which have adopted file system caches have experienced growing file systems, i.e., users do not bother to delete old files but treat them as archival backups. In such contexts one finds more file creations (pushes) than deletions. Hence one would expect more gaps to be consumed than produced. Such chronic gap deficits mean that deep gaps produced by deleting relics will become topmost gaps. If such gaps are large, they may be squeezed repeatedly before they are entirely consumed. Finding and squeezing such deep gaps may consume inordinate amounts of time.

We have therefore incorporated a second linked list for the gaps, threaded through those segment descriptor nodes which have prefixed gaps. This makes finding the topmost gap (at the head of the gap list) to squeeze out during a push trivial, i.e., a constant time operation. However, now one must arrange to insert all new gaps into this gap list (i.e., for every pull or deletion). While traversing the linked list of segment descriptors constituting the stack to find the position of a pulled segment, one can record the location of the previous gap, if any. Inserting the new gap occasioned by the pull can then be done in constant time. Thus the running time of the linked list algorithm increases (for pushes and pulls) by at most a constant factor (to accommodate the search for the previous gap during a pull).

Deletions could be treated as pulls (searching the segment list from the top). However, determining the depth of a deleted segment in the stack is not necessary, since deletions do not produce cache misses. Thus instead of searching the segment linked list constituting the stack from the top to determine the depth of the deleted segment (as one would for an ordinary pull) one could access the segment node to be deleted via a hash table index. The hash table access can be done in constant time, versus time proportional to stack depth for searching the linked list. If a new gap is created by the deletion (rather than merely enlarging an adjacent gap) then one must insert the new gap into the gap list. The appropriate location for the insertion can be found by noting the last reference time recorded in each segment node inspected while searching the gap list. The gap list will undoubtedly be much smaller than the segment list in realistic file cache studies (it may even be empty).

6.6. LRU Model Reference Generators

The tree algorithms described above can be used to generate synthetic reference strings from an LRU model. Instead of climbing a tree to its root to calculate the position of a page in stack, one searches the tree from the root to find the page at the specified position in the stack. This target stack distance is generated by a pseudo-random number generator with a specified distribution. The search algorithms are simple analogues of the distance calculation algorithms. The trees are maintained in the same manner as for the LRU success function calculations. Thus the running times for reference generation are identical to those for calculating success functions to within a constant factor.

6.7. Performance Measurements

We have implemented and timed the three algorithms discussed above for calculating LRU success functions: the classic linked list algorithm (LL), the AVL tree algorithm (AVL), and the modified Bennett and Kruskal algorithm (MBK).

All of the algorithms were coded by the author in Pascal and run on a DEC VAX 11/780 under the VMS operating system. The only differences between the programs was the code for maintaining the stack data structures. In particular all three programs employ an identical hash table to check for the presence of the segment in the stack (so that misses need not search the entire stack)[26].

All of the codes are capable of handling variable size segments, deletions and multi-pass operation. However, in these timing experiments only single pass operation of the algorithms was measured[27]. Furthermore all segment sizes were set to one.

The timing measurements record the time required to calculate the success function for various synthetic reference strings (traces) generated from an LRU model. Before timing for each traces commenced, a warm start was performed by loading the cache with pages in random order. All three algorithms were timed on identical sets of traces.

Since the theory predicts that the performance of the linked list algorithm will depend solely on the mean stack depth of accessed segments, while the performance of the other two algorithms should be a function primarily of the stack size, we employed synthetic reference strings generated from an LRU model to control these parameters separately.

No deletions were generated. However, the code to implement deletions (i.e., to create and squeeze out gaps) is exercised by the normal push-pull sequences. (Deletions for the linked list code were treated similarly to pulls rather than only searching the gap list.) Furthermore, except for the warm start (omitted from the timing statistics), all of the references generated were to previously referenced pages, i.e., there were no complete misses (references to new pages) generated.

[26]The hash table is also used to provide an index into the trees maintained by the AVL tree and Bennett and Kruskal algorithms. We use linear hashing with chained overflow. The chained overflow was adopted to facilitate deletions from the hash table (required whenever a segment is pushed out from the stack). The access time to find an entry in the hash table depends on its position in the overflow chain. Hence the average access time depends on the average overflow chain length, i.e., the loading density of the hash table. The size of the hash table was chosen for each of the timing runs so as to preserve the same loading density of hash table on all runs, i.e., 1.0 entries/bucket.

[27] Multi-pass operation would produce similar results except that the linked list algorithm would have better performance, since the hash table index avoids the need to search the linked list for missing segments and deletions from the linked list are faster than from the AVL or Bennett and Kruskal trees.

The LRU stack distance distributions chosen were discrete analogues of beta distributions. This family of distributions was chosen, both because it can generate plausibly shaped distributions (i.e., positive, highly skewed toward zero)[28] and because we could easily generate pseudo-random variates. The synthetic stack distances were chosen by generating a standard beta variate x distributed as B(p,q) on the unit interval (0,1)[29]. This variate was then multiplied by the stack size, truncated, and then added to 1 (i.e., scaled to generate integers between 1 and the stack size).

The pseudo-random uniform random number generator used is a Pascal implementation of the standard IMSL multiplicative congruential random number generator, GGUBS.

The parameters p,q of the beta distribution were chosen in conjunction with the stack sizes in an attempt to generate an orthogonal experimental design of stack size versus mean stack depth of accessed segments. Such a design facilitates regression studies. For each stack size (100,200,400,800,1600) strings were generated from the beta distributions B(1,1), B(3,1), B(7,1), B(15,1), B(31,1). This generates the following approximate mean stack depths.

Table of Approximate Mean Stack Depths

	Parameters of Beta Distribution				
	B(1,1)	B(3,1)	B(7,1)	B(15,1)	B(31,1)
Stack size					
100	50	25	12	6	3
200	100	50	25	12	6
400	200	100	50	25	12
800	400	200	100	50	25
1600	800	400	200	100	50

For each design point (stack size, mean stack depth) 4821 references were processed after the warm start. The number of references was chosen to correspond approximately to an integral number of compactions for the modified Bennett and Kruskal algorithm employing a buffer twice the size of the stack. Thus for a stack size of 1600 entries, compaction would occur after 3200 references and 4800 references. Hence the timings correspond to the long run average for very long reference strings with many compactions. The number of references was also a tradeoff between available computer time and minimizing the variance of the results. Each such experiment was repeated a second time.

For each experiment we tabulated the following measurements:

(1) E(t) = the mean processing time per reference (in microseconds).

[28]LRU stack distance distributions have commonly been observed to be highly skewed with most references occurring to recently referenced segments (i.e., small stack distances).

[29] See [John70].

(2) $E(s)$ = the average stack size.

(3) $E(\log_2(s))$ = the average of the log of the stack size.

(4) $E(d)$ = the average stack depth of hits.

(5) $E(\log_2(d))$ = the average of the log of the stack depth of hits.

Regression studies were then conducted to estimate the parameters of models to predict the mean processing time per reference. The results are given below.

(1) For the linked list algorithm, the following model accounted for 99.8% of the variance:

$$E(t) = 2.28 \times E(d) + 356$$

(2) For Bennett and Kruskal's algorithm, the following model accounted for 98.5% of the variance:

$$E(t) = 71.7 \times E(\lceil \log_2(s) \rceil) + 525$$

(3) For the AVL tree algorithm with a balance factor of 1 (i.e., height imbalances of adjacent nodes are constrained to 2 or less) the following model accounted for 83% of the variance:

$$E(t) = 39.0 \times E(\lceil \log_2(s) \rceil) + 429$$

The poorer fit of the model compared to the Bennett and Kruskal or linked list algorithms is presumably due to the fact that the stack size is an imperfect predictor of the depth in the tree at which a file node will be found. Also differently shaped LRU stack depth distributions may affect somewhat the amount of rebalancing required. Neither factor is an issue with the Bennett and Kruskal or linked list algorithms.

(4) For the AVL tree algorithm with a balance factor of 2 (i.e., height imbalances of adjacent nodes are constrained to 2 or less) the following model accounted for 93% of the variance:

$$E(t) = 32.4 \times E(\lceil \log_2(s) \rceil) + 505$$

Again the poorer fit of the model is presumably due to the fact that the stack size is an imperfect predictor of the depth in the tree at which a file node will be found. Also differently shaped LRU stack depth distributions may affect the amount of rebalancing required (to lesser extent than above because of the relaxed balance constraints).

The reader will note that the AVL tree algorithm outperforms the modified Bennett and Kruskal algorithm. However, the difference is sufficiently small that it might be reversed by clever coding of the modified Bennett and Kruskal algorithm. Also interesting is the fact that permitting greater imbalance in the AVL tree produces faster running times for large stacks.

6.8. Summary of LRU Algorithms

Comparison of LRU algorithms

Algorithm	Time	Space	Coding Difficulty	Actual Performance
Linked list	O(n*d)	O(s)	easy	poor
B + K	O(n*log(i))	O(n)	medium	-----
modified B+K	O(n*log(s))	O(s)	medium	good
AVL tree	O(n*log(s))	O(s)	hard	best

Notes: B+K is Bennett and Kruskal's algorithm.
n = length of address trace,
d = average stack depth of references,
s = largest stack size considered (number of entries),
log(i) = average of log of inter-reference time.

6.9. Efficiency in a Virtual Memory Environment

Thus far we have been concerned with the efficiency of computations executing with real memory. In this section we will consider algorithms appropriate to running in a virtual memory environment (or explicitly referencing secondary storage).

We will assume that the operating system will replace the least recently used page in the computation (leaving aside questions of how many pages are allocated to the computation).

None of the algorithms as implemented made any attempt to keep logically contiguous nodes physically contiguous (to enhance locality).

For the Bennett and Kruskal algorithm the tree linearization function could be modified to map a node and its closest descendents onto a contiguous page of memory. Unfortunately the compression phase would still flush the working set from real memory.

In virtual memory one would replace the linked list algorithm with a two-level algorithm, where the top level consisted of an linked list index to pages of nodes. Adjacent half-empty pages would be combined upon detection. Presumably the index would be small enough to stay in real memory.

The analogue of AVL trees on secondary storage is a B-tree [Comer79]. B-trees provide access and updating times which are proportional to the log of the tree size, where the radix is the number of nodes which will fit on a page. Within each page we would probably use an AVL tree. The resulting code, although somewhat complex, should provide excellent performance.

7. Conclusions

In this paper we have given a new algorithm (AVL) for calculating LRU success functions efficiently in bounded space. We have shown how to treat deletions of segments in calculating LRU success functions. We have also shown how to modify Bennett and Kruskal's algorithm for calculating LRU success functions to run in bounded space while preserving its efficiency. Empirically both algorithms perform well.

8. Acknowledgements

The author would like to thank the following people for their comments on the paper: Susan Eggers, Dennis Hall, Juan Porcar, Arie Shoshani, Prof. A.J. Smith, Prof. D. Ferrari, Prof. M. Stonebraker, Ozalp Babaoglu, and Peggy Keenan.

This research was supported by the Applied Mathematical Sciences Research Program of Office of Energy Research, U.S. Department of Energy under contract W-7405-ENG-48.

Bibliography

[Baba80]

Babaoglu, O. "Analysis of a Class of Hybrid Page Replacement Policies" , Publication No. 246 , Institute of Numerical Analysis Pavia, Italy , May 1980

[Bonn75]

Bennett, B.T., and Kruskal, V.J. "LRU Stack Processing" , IBM Journal of Research and Development, vol. 19, no. 4 , July 1975 , pp. 353-357

[Coff71]

Coffman, E.G., and Randell, B. "Performance Predictions for Extended Paged Memories" , Acta Informatica, vol. 1, no. 1 , 1971 , pp. 1-13

[Coff71a]

Coffman, Edward G., and Jones, N.D. "Priority Paging Algorithm and the extension problem", Proceedings of the 11th Switching and Automata Conference , October 1971

[Coff73]

Coffman, Edward G., and Denning, Peter J. Operating Systems Theory , Prentice Hall Englewood Cliffs, New Jersey , 1973, Chapter 6

[Comer79]

Comer, Douglas "The Ubiquitous B-tree" , Computing Surveys, vol. 11, no. 2 , June 1979

[Egg80]

Eggers, Susan and Shoshani, Arie "Efficient Access to Compressed Data" , Proceedings of the Sixth International Conference on Very Large Databases, Montreal, Canada , October, 1980

{Fost73}

Foster, Caxton C. "A Generalization of AVL Trees" , Communications of the ACM, vol. 16, no. 8 , August 1973 , pp. 513-517

[Fran77]

Franta, W. R. and Maly, Kurt "An Efficient Data Structure for the Simulation Event Set" , Communications of the ACM, vol. 20, no. 8 , August 1977 , pp. 596-602

[Green74]

Greenberg, Bernard S. An Experimental Analysis of Program Reference Patterns in the Multics Virtual Memory , MAC TR-127 , MIT, Project MAC Cambridge, Mass. , January 1974

[John70]

Johnson, Norman L. and Kotz, Samuel Continuous Univariate Distributions, Vol. 2 , Wiley , 1970, Chapter 24

[Knuth73]

Knuth, Donald E. The Art of Computer Programming, Vol. 3, Sorting and Searching, , Addison Wesley , 1973 , pp. 451-468

[Matt70]

Mattson, R.L., J. Gecsei, D.R. Slutz, and I.L. Traiger "Evaluation Techniques for Storage Hierarchies" , IBM Journal of Research and Development, vol. 9, no. 2 , 1970 , pp. 78-117

[Rusch77]

Ruschitzka, Manfred, and Fabry, R.S. "A Unifying Approach to Scheduling" , Communications of the ACM, vol. 20, no. 7 , July 1977 , pp. 469-477

[Smith77]

Smith, Alan Jay "Two Methods for the Efficient Analysis of Memory Trace Data" , IEEE Transactions on Software Engineering vol. SE-3, no. 1 , January 1977 , pp. 94-101

[Smith79]

Smith, Alan Jay "Long Term File Migration: Parts I and II" (submitted for publication)

[Stritt77]

Stritter, Edward P. File Migration , doctoral dissertation , SLAC-200 , STAN-CS-77-594 , Stanford Linear Accelerator Center , Stanford University Stanford, Calif. , January 1977

[Traig71]

I.L Traiger, and D.R. Slutz "One Pass Techniques for the Evaluation of Memory Hierarchies" , IBM Research Report RJ 892 , IBM Yorktown Heights , July 1971

[Wirth76]

Wirth, Niklaus Algorithms + Data Structures = Programs , Prentice Hall , 1976