B. T. Bennett
V. J. Kruskal

LRU Stack Processing

Abstract: Stack processing, and in particular stack processing for the least recently used replacement algorithms, may present computational problems when it is applied to a sequence of page references with many different pages. This paper describes a new technique for LRU stack processing that permits efficient processing of these sequences. An analysis of the algorithm and a comparison of its running times with those of the conventional stack processing algorithms are presented. Finally we discuss a multipass implementation, which was found necessary to process trace data from a large data base system.

Introduction

Storage hierarchy evaluation is often accomplished by simulating the hierarchy under loads determined by "representative" address traces. Stack processing as proposed by Mattson, Gecsei, Slutz, and Traiger [1] allows efficient evaluation of multilevel hierarchies for a class of replacement algorithms called stack algorithms. Of these algorithms, least recently used (LRU) is the most extensively simulated.

The original LRU stack processing algorithm proposed by Mattson et al. calculates for one page size a histogram of stack distances, which determines the frequency of access to each level of a multilevel linear hierarchy for any set of level capacities. The method involves the conversion of each address to a page reference and the maintaining of a list of pages called an LRU stack, wherein the pages are in order of the most recent reference. For each reference the stack distance, the position in the stack of the current referenced page, is obtained.

To maintain the LRU stack and to obtain the stack distance for each reference, Mattson et al. proposed a concurrent search and update from the top down. The current page is placed on top of the stack, and each page in the stack is down-shifted by one until the current page is encountered. That position in the stack is recorded as the stack distance. If the current page is not found, i.e., if it has not occurred before, then downshifting proceeds to the bottom of the stack and a stack distance of infinity is recorded.

The number of tests for a match with the current page is equal to the stack distance or, for a new page, to the number of distinct pages encountered so far. Thus, stack-processing a trace that has a large number of distinct pages or a large average stack distance may require excessive computing time. Traiger and Slutz [2] showed that for a little additional overhead this method could also produce stack distances for page sizes that are successive multiples of the basic page size. However, for some data base reference traces and some program address traces the method was found not to be feasible for the page sizes of interest.

In this paper we describe a new algorithm for LRU stack processing; this algorithm is much more efficient for the analysis of trace data for a single page size when the number of pages and the average stack distance are large, but separate page sizes require essentially separate calculations. Second, we present an analysis of the algorithm and a comparison of running times. Finally, we discuss a multipass implementation of the algorithm, which we have found necessary in order to efficiently process trace data from a large data base system.

New LRU stack processing algorithm

The purpose of the algorithm described here is to determine the stack distance for each reference. Rather than regarding the stack distance as the position in an LRU stack, we observe that the distance is equal to one plus the number of distinct pages that have been referenced since the current page was last referenced.

If we denote the page referenced at time t by x_t and the stack distance by d_t, then we have

$$d_t = 1 + C(x_{p+1}, \cdots, x_{t-1}),$$

where $p = \max i < t$ such that $x_i = x_t$, i.e., p is the position of the most recent past reference to page x_t and $C(S)$ is the number of distinct pages in S.

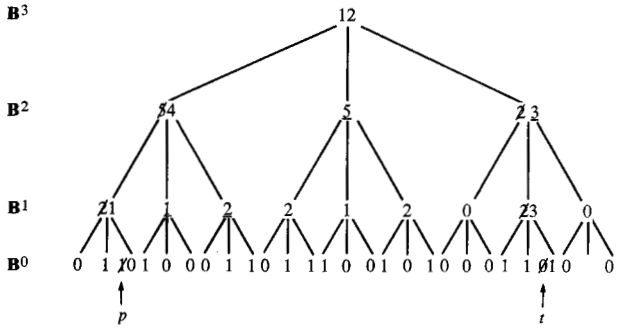

Figure 1 Partial sum hierarchy for $m = 3$ illustrating the elements changed (/) and the elements summed (_) to maintain the hierarchy and to calculate the stack distance for a reference at time $t = 23$ to a page previously referenced at time $p = 2$.

Let

$$\mathbf{B}_t^0(i) = \begin{cases} 1 \text{ if } x_i \neq x_j (j = i+1, \cdots, t), \\ 0 \text{ otherwise,} \end{cases}$$

and for any page x in the virtual address space, let

$$\mathbf{P}_t(x) = \begin{cases} \max i \leq t \text{ such that } x_i = x, \\ -1 \text{ if there is no } x_i = x \ (0 \leq i \leq t). \end{cases}$$

Thus $\mathbf{B}^0$ is an array of 0s and 1s whose positions correspond to positions in the reference string, and a 1 in position i means that page x_i has not occurred since time i. Array $\mathbf{P}$ contains for each referenced page the position of its most recent reference.

Let $p = \mathbf{P}_{t-1}(x_t)$. The stack distance is one plus the count of the 1s in $\mathbf{B}_{t-1}^0$ from $p+1$ to $t-1$,

$$d_t = 1 + \sum_{i=p+1}^{t-1} \mathbf{B}_{t-1}^0(i), \tag{1}$$

because each 1 in $\mathbf{B}^0$ after position p uniquely represents a page that has been referenced since x_t was last referenced. In order to compute the sum efficiently, we maintain in arrays $\mathbf{B}^1, \cdots, \mathbf{B}^L$ a hierarchy of partial sums of $\mathbf{B}^0$. For some chosen interval size m the sum of the first m elements of $\mathbf{B}^s$ is maintained in the first element of $\mathbf{B}^{s+1}$, and the sum of the second m elements of $\mathbf{B}^s$ is maintained in the second element of $\mathbf{B}^{s+1}$, and so on. With zero-based indexing,

$$\mathbf{B}_{t-1}^s(j) = \sum_{i=jm}^{i=(j+1)m-1} \mathbf{B}_{t-1}^{s-1}(i) = \sum_{i=jm^s}^{i=(j+1)m^s-1} \mathbf{B}_{t-1}^0(i)$$

for each j, $0 \leq j < m^{L+1}/m^s$, and for each s, $0 \leq s < L$. In array $\mathbf{B}^L$, where L is defined by $m^L <$ total number of references $\leq m^{L+1}$, the sums of all elements of $\mathbf{B}^0$ are contained in $\leq m$ elements. To accumulate the sum of $\mathbf{B}^0$ from $p+1$ to $t-1$, this interval is considered to be a sequence of intervals whose sums are obtained by summing partial sums from increasing levels of the hierarchy. The elements of $\mathbf{B}^0$ from $p+1$ to $(\lfloor p/m+1 \rfloor)m-1$ must be summed, the elements of $\mathbf{B}^0$ from $(\lfloor p/m+1 \rfloor)m$ to $(\lfloor p/m^2+1 \rfloor)m^2-1$ may be summed by summing the elements of $\mathbf{B}^1$ from $\lfloor p/m+1$ to $(\lfloor p/m^2+1 \rfloor)m-1$, and so on until level $r-1$ is reached, where

$$r = \min j \text{ such that } (\lfloor p/m^j = \lfloor t/m^j). \tag{2}$$

Then the elements of $\mathbf{B}^0$ from $(\lfloor p/m^{r-1}+1 \rfloor)m^{r-1}$ to t are summed using elements of $\mathbf{B}^{r-1}$ to sum up to $(\lfloor t/m^{r-1}+1 \rfloor)m^{r-1}+1$. It is assumed that $\mathbf{B}_{t-1}^0(i) = 0$ for $i \geq t$. Note that

$$(\lfloor p/m^{s+1}+1 \rfloor)m^{s+1} \geq (\lfloor p/m^s+1 \rfloor)m^s \geq p+1$$

for all p, s nonnegative integers and m a positive integer. A negative range of summation indicates that no summation need be made.

The sum (1) may be expressed as

$$1 + \sum_{i=p+1}^{t-1} \mathbf{B}_{t-1}^0(i) = 1 + \sum_{i=p+1}^{(\lfloor p/m+1 \rfloor)m-1} \mathbf{B}_{t-1}^0(i) + \sum_{i=(\lfloor p/m+1 \rfloor)m}^{(\lfloor p/m^2+1 \rfloor)m^2-1} \mathbf{B}_{t-1}^0(i) \cdots + \sum_{i=(\lfloor p/m^{r-2}+1 \rfloor)m^{r-2}}^{(\lfloor p/m^{r-1}+1 \rfloor)m^{r-1}-1} \mathbf{B}_{t-1}^0(i) + \sum_{i=(\lfloor p/m^{r-1}+1 \rfloor)m^{r-1}}^{(\lfloor t/m^{r-1}+1 \rfloor)m^{r-1}-1} \mathbf{B}_{t-1}^0(i). \tag{3}$$

Then, representing each term as a sum of partial sums, we obtain

$$1 + \sum_{i=p+1}^{t-1} \mathbf{B}_{t-1}^0(i) = 1 + \sum_{i=p+1}^{(\lfloor p/m+1 \rfloor)m-1} \mathbf{B}_{t-1}^0(i) + \sum_{i=\lfloor p/m+1}^{(\lfloor p/m^2+1 \rfloor)m-1} \mathbf{B}_{t-1}^1(i) + \cdots + \sum_{i=\lfloor p/m^{r-2}+1}^{(\lfloor p/m^{r-1}+1 \rfloor)m-1} \mathbf{B}_{t-1}^{r-2}(i) + \sum_{i=\lfloor p/m^{r-1}+1}^{\lfloor t/m^{r-1}} \mathbf{B}_{t-1}^{r-1}(i). \tag{4}$$

Processing each reference involves obtaining the value of p, the position of the previous reference, and then calculation of the sum (4) and updating of arrays $\mathbf{B}^0, \cdots, \mathbf{B}^L$, viz.,

$$\mathbf{B}_t^s(\lfloor p/m^s) = \mathbf{B}_{t-1}^s(\lfloor p/m^s) - 1,$$

$$\mathbf{B}_t^s(\lfloor t/m^s) = \mathbf{B}_{t-1}^s(\lfloor t/m^s) + 1,$$

for $s = 0, \cdots, r-1$.

Thus an algorithm for processing reference x_t is as follows:

1. Obtain the value of $p = \mathbf{P}_{t-1}(x_t)$ and set $\mathbf{P}_t(x_t) = t$. If $p = 0$ (new page), go to step 6, else set $s = 0$, $d_t = 0$.
2. Update $\mathbf{B}_t^s(\lfloor p/m^s) = \mathbf{B}_{t-1}^s(\lfloor p/m^s) - 1$ and $\mathbf{B}_t^s(\lfloor t/m^s) = \mathbf{B}_{t-1}^s(\lfloor t/m^s) + 1$; note that when $s = 0$, $\lfloor p/m^s = p$, and $\lfloor t/m^s = t$.
3. Calculate $\lceil p/m^{s+1}$ and $\lceil t/m^{s+1}$. If equal, go to step 5.

4. Set $d_t = d_t + \mathbf{B}_t^s(i)$ for $i = \lfloor p/m^s + 1, \cdots, (\lfloor p/m^{s+1} + 1)m - 1$. Then set $s = s + 1$, and go to step 2.
5. Set $d_t = d_t + \mathbf{B}_t^s(i)$ for $i = \lfloor p/m^s + 1, \cdots, \lfloor t/m^s$. Processing is complete. [The additional 1 in sum (4) has been included by updating $\mathbf{B}_t^s(\lfloor t/m^s)$ prior to this step.]
6. Set $s = 0, d_t = \infty$.
7. Update $\mathbf{B}_t^s(\lfloor t/m^s) = \mathbf{B}_{t-1}^s(\lfloor t/m^s) + 1$.
8. Set $s = s + 1$. If $s \leq L$, then go to step 7, else processing is complete.

Figure 1 illustrates the hierarchy of partial sums, the elements changed (/), and the elements summed (__) for a reference at time $t = 23$ to page previously referenced at time $p = 2$. The bottom level $\mathbf{B}_{t-1}^0$ contains the binary array. The next level contains the sums of three consecutive values in $\mathbf{B}_{t-1}^0$, and so on. Changes made at levels 0, 1, and 2, respectively, are a decrease by 1 for $p = 2$, $\lfloor p/3 = 0$, and $\lfloor p/9 = 0$, and an increase by 1 for $t = 23$, $\lfloor t/3 = 7$, $\lfloor t/9 = 2$. The summations are from $p + 1 = 3$ to $(\lfloor p/3 + 1)3 - 1 = 2$, thus nothing on level 0, from $\lfloor p/3 + 1 = 1$ to $(\lfloor p/9 + 1)3 - 1 = 2$ on level 1, and from $\lfloor p/9 + 1 = 1$ to $\lfloor t/9 = 2$ on level 2.

Analysis

This section demonstrates that the average amount of processing for a reference to a previously referenced page is approximately a linear function of the logarithm of the inter-reference distance n:

$$n = t - p.$$

In performing the algorithm, step 1 is carried out once for each reference, step 2, r times, and step 3, $r+1$ times, where r is determined by Eq. (2).

Step 4 is performed $r - 1$ times; the number of additions to calculate the distance is

$$m - 1 - (\lfloor p/m^s \bmod m) \text{ for each } s = 0, \cdots, r - 2. \quad (5)$$

Step 5 is performed once and requires

$$\lfloor t/m^{r-1} - \lfloor p/m^{r-1} \text{ additions.} \quad (6)$$

Thus, the amount of computation is a function of both p and n. We shall show that averages of r and the number of additions over a range of values of p are approximately linear functions of $\log n$.

Let $\lfloor \log_m n = c$, i.e., $n = n_0 m^c + n_1$, where $1 \leq n_0 < m$, $0 \leq n_1 < m^c$. Equation (2) implies that

$$c + 1 \leq r \leq \log_m t + 1$$

and that for each positive integer q, $1 \leq q \leq (\log_m t) - c$, $r > q + c$ whenever $km^{q+c} - n \leq p < km^{q+c}$ for some positive integer k. Therefore, in the range $1 \leq p \leq m^a$ for integer $a > c$, $r > c + q$ for $nm^{a-(c+q)}$ distinct values of p for each q, $1 \leq q \leq a - c$, and $r > a$ implies $r = a + 1$. Thus, in this range the average value of r is

$$\begin{aligned} & c + 1 + \left(\sum_{q=1}^{a-c} nm^{a-(c+q)}\right) \Big/ m^a \\ &= 1 + c + n/m^{c+1} + n/m^{c+2} \cdots + n/m^a \\ &= \log_m n - \log_m(n/m^c) + 1 \\ &\quad + [n(1 - m^{c-a})]/[m^c(m-1)], \end{aligned} \quad (7)$$

the predominant term of which is $\log_m n$.

The number of additions in step 5 is given by (6), which is one if $r > c + 1$, and if $r = c + 1$, is $\lceil n/m^c$ if there exists integer q, $qm^c - n_1 \leq p < qm^c$, or $\lfloor n/m^c$ otherwise.

If $r = c + 1$, i.e., if there is no k, $km^{c+1} - n \leq p < km^{c+1}$, but there exists q, $qm^c - n_1 \leq p < qm^c$, then $0 < q \bmod m < m - n_0$. Thus, in the range $1 \leq p \leq m^a$, the number of additions in step 5 is one for $nm^{a-(c+1)}$ values of p, is $n_0 + 1$ for $n_1 m^{a-(c+1)}(m - 1 - n_0)$ values of p, and is n_0 for all others. Thus, the average number of additions in step 5 is

$$\begin{aligned} & [n_0(m^a - nm^{a-(c+1)}) + nm^{a-(c+1)} \\ &\qquad + n_1 m^{a-(c+1)}(m - 1 - n_0)]/m^a \\ &= n_0 + (n(1 - n_0) + n_1(m - 1 - n_0))/m^{c+1} \\ &= n_0 + (n_0 - n_0^2)/m + n_1(m - 2n_0)/m^{c+1}. \end{aligned} \quad (8)$$

The number of additions to the distance d in step 4 depends on s, the level in the hierarchy. For $s = 0, \cdots, c - 1$, expression (5) takes each value $0, \cdots, m - 1$ exactly m^{a-1} times for p, $1 \leq p \leq m^a$. For $s = c, \cdots, r - 2$, additions occur only if $r > s + 1$, i.e., for p such that there exists an integer k, $km^{s+1} - n \leq p < km^{s+1}$. Then $\lfloor (km - n/m^s) \leq \lfloor p/m^s < km$; thus $\lfloor p/m^s \bmod m \geq m - \lceil n/m^s$, and $m - 1 - (\lfloor p/m^s \bmod m) \leq \lceil n/m^s - 1$. Thus, for $s = c$ the maximum number of additions is $\lceil n/m^c - 1$, and there are no additions for $s \geq c + 1$.

For $s = c$ there are additions if there exists an integer k such that $p = km^{c+1} - i$, $1 \leq i \leq n$, and the number of additions is

$$\begin{aligned} & m - 1 - (\lfloor (km^{c+1} - i)/m^c \bmod m) \\ &= (\lceil i/m^c \bmod m) - 1 \\ &= \begin{cases} n_0 \text{ if } n_0 m^c + 1 \leq i \leq n_0 m^c + n_1, \\ n_0 - j \text{ if } (n_0 - j)m^c + 1 \leq i \leq (n_0 - j + 1)m^c \\ \quad \text{for } 1 \leq j \leq n_0. \end{cases} \end{aligned}$$

Thus, in the range $1 \leq p \leq m^a$ there are $n_1 m^{a-(c+1)}$ values of p with n_0 additions for $s = c$ and m^{a-1} values of p with $n_0 - j$ additions for each j, $1 \leq j \leq n_0$.

Table 1 Timing results for the three stack distance algorithms for processing 100 000 references to a large data base.

Experiment number	Page size in data base blocks	Number of distinct pages	Average stack distance	Execution time (seconds)		
				Original algorithm	Modified algorithm	New algorithm
1	16384	185	12.53	6.5	7.0	5.7
2	512	1367	60.29	24.6	22.2	6.3
3	64	4709	151.13	85.1	49.4	6.5
4	8	9996	237.56	234.5	72.2	6.8
5	2	14262	328.53	>450	94.4	7.5

The average number of additions in step 4 is

$$\{m(m-1)cm^{a-1}/2 + n_0n_1m^{a-(c+1)} + m^{a-1}[n_0^2 - n_0(n_0+1)/2]\}/m^a$$
$$= c(m-1)/2 + n_0n_1/m^{c+1} + (n_0^2 - n_0)/2m. \quad (9)$$

Combining (8) and (9) we obtain the average total number of additions necessary to calculate the stack distance as

$$c(m-1)/2 + n_0 - (n_0^2 - n_0)/2m + n_1(m-n_0)/m^{c+1}$$
$$= [(m-1)\log_m n]/2 + [(m-1)\log_m(n/m^c)]/2 + n_0 - (n_0^2 - n_0)/2m + n_1(m-n_0)/m^{c+1}. \quad (10)$$

The predominant term is $[(m-1)\log_m n]/2$.

Performance Comparison

In this section the new algorithm described in this paper is compared with the original algorithm proposed by Mattson et al. Included in the comparison is an important modification of the original algorithm, which has been used by Hempy [3] to process data base traces.

In the original algorithm each new page resulted in a costly, unsuccessful search of the complete stack. The modified algorithm avoids the unsuccessful searches by maintaining a directory in which it records for each page whether it has previously been referenced. Most often the set of page names is such that a search technique (see Knuth [4]) is required to map the page names to directory entries. For this comparison we have used a hash table with linear probing and a division hash function. However, consecutive references to the same page are checked for without looking up the hash table.

The array **P** in the new algorithm records the previous reference position for each page in the virtual address space. Unless the virtual address space is small, or the trace has been preprocessed to number the pages from one to the total number of distinct pages, it will be more efficient to use a search technique and a table of observed page names to map the page names to the locations in which the previous reference positions are stored.

For the comparison we used a hash table identical to that used by the modified algorithm and checked for consecutive references to the same page. These references do not require reference to the hash table or to the **B** arrays. For efficiency the binary array $\mathbf{B}^0$ is stored in bit strings of length $m = 8$. Since the sum of the bits is a function of the binary number represented by the string, the values of $\mathbf{B}^1$ and the partial sums of $\mathbf{B}^0$ are obtained by table lookup. For programming convenience the arrays $\mathbf{B}^2, \cdots, \mathbf{B}^L$ are treated as levels in m-ary trees (Knuth [5], p. 401). For the comparison $m = 10$ was chosen. Optimal choice of m depends on the implementation of the algorithm, on the speed of division of the computer, and on the proportion of references to new pages. The number of divisions is roughly $2 + 2\log_m n$, and the number of other operations is roughly proportional to $m\log_m n$ plus a constant for references to previously referenced pages; otherwise the number of divisions and other operations is proportional to $\log_m N$ plus a constant (N is the total number of references). In assembly language shifting may be substituted for division if m is a power of 2.

Timing results are presented in Table 1 for 100 000 references to a large data base. By varying the page size a range of numbers of distinct pages and average stack distances is obtained. The algorithms were coded in PL/I, compiled under the most optimization option of the PL/I Optimizing Compiler, and executed on an IBM System/370 Model 168. Average stack distance was calculated in each program.

These results show that the new algorithm requires much less computing time, particularly when the number of distinct pages and the average stack distance are large. Note, however, that the storage requirements of the faster algorithms are greater than that of the original algorithm. Whereas the original algorithm requires storage only for a stack of page names whose maximum size is the number of distinct pages, the modification requires storage for a hash table of page names. To be efficient the length of the hash table should be significantly larger than the maximum number of distinct pages. For the

comparison the length was chosen as 32 599. The new algorithm has the hash table and array **P** of the same length and so also the arrays $\mathbf{B}^0 \cdots, \mathbf{B}^L$. For traces of data base systems these tables may require so much space that the use of secondary storage must be considered.

Implementation for data base systems

Traces from data base systems may have many references and large numbers of distinct pages. Techniques for resolving the storage problems created by these qualities are discussed in this section.

The binary array $\mathbf{B}^0$ has length $\lceil (N+1)/8$ bytes, where N is the number of references, and partial sum arrays $\mathbf{B}^2, \cdots, \mathbf{B}^L$ have lengths $\lceil (N+1)/8m, \cdots, \lceil (N+1)/8m^{L-1}$, respectively. For large N these arrays may occupy many pages of virtual memory. To ensure a minimum of paging $\mathbf{B}^0$ is divided into, for example, sections of $8m^3$ bits and stored contiguously with the corresponding sections of $\mathbf{B}^2$ and $\mathbf{B}^3$.

The hash table and array **P** contain page names and the positions of the last reference to each page. To be efficient the lengths of the tables must be significantly larger than the maximum number of distinct pages. If this number is too large, then the algorithm should be performed in two passes of the data. In the first pass the hash table and array **P** are used to convert the trace of page names to a trace of positions of previous references. Then, on the second pass, the **B** arrays are used to calculate the stack distances.

If the number of distinct pages is sufficiently large, then the one pass to obtain the previous reference position trace may itself best be replaced by a multipass scheme. If from prior knowledge it is known how to partition the set of page names into approximately equal-size groups, then the following scheme is suggested.

1. Add the reference number to each record and partition the trace into subtraces, one for each page name group.
2. Use a hash table and **P** array to create for each subtrace a previous reference position subtrace.
3. Merge the previous reference position subtraces.

Concluding remarks

We have described a new algorithm for LRU stack processing with a single page size and have shown by timing results that it is much more efficient than previously suggested algorithms for processing a trace with a large average stack distance. Efficient use of virtual storage and ways of utilizing secondary storage to make up for the increased storage required by the new algorithm have been discussed. The amount of computation for each reference has been analyzed for the new algorithm, and the average for a fixed inter-reference distance over a range of previous reference positions is shown to be approximately a linear function of the logarithm of the inter-reference distance.

Acknowledgment

The authors appreciate the useful discussions held with A. C. McKellar, M. A. Auslander, M. C. Easton, and P. G. Capek.

References

1. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Syst. J.* **9**, (1970).
2. I. L. Traiger and D. R. Slutz, "One-Pass Technique for the Evaluation of Memory Hierarchies," *Research Report RJ 892*, IBM Research Laboratory, San Jose, CA, July 28, 1971.
3. Private communication from H. Hempy of the IBM General Products Division Laboratory, Boulder, CO.
4. D. E. Knuth, *Sorting and Searching; The Art of Computer Programming*, Vol. 3, Addison-Wesley Publishing Co., Reading, MA, 1973.
5. D. E. Knuth, *Fundamental Algorithms, The Art of Computer Programming*, Vol. 1, Addison-Wesley Publishing Co., Inc., Reading, MA, 1968.

Received September 6, 1974; revised March 10, 1975

The authors are located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.