VIRTUAL MEMORY SUPPORT FOR BRANCH-AND-BOUND ALGORITHMS

C. F. Yu and Benjamin W. Wah
School of Electrical Engineering
Purdue University
W. Lafayette, IN 47907

ABSTRACT

In this paper, we study the virtual memory design for branch-and-bound algorithms under best-first search. The use of a conventional locality based virtual memory system is inadequate because branch-and-bound algorithms do not exhibit strong locality. It is not apparent how the amount of locality exhibited may be enhanced. The alternative is to adapt the virtual memory system to the algorithm. Simulations indicate that for branch-and-bound algorithms a well designed virtual memory system may achieve a seven- to nine-fold improvement in paging traffic over a general purpose virtual memory system with the LRU replacement algorithm.

KEYWORDS AND PHRASES: Best-first search, B^+-tree, branch-and-bound, page size, paging traffic, replacement algorithm, virtual memory.

1. INTRODUCTION

Many of the deterministic problems in computer science, operations research, and other application areas are NP-complete [GAR79]. The computation time for all known optimal algorithms for this class of problems increases exponentially with the problem size. The most general technique for solving a wide variety of these problems is the branch-and-bound algorithm.

A branch-and-bound algorithm [LAW66] is an efficient algorithm for solving problems that can be put into the form of a constrained optimization.

$$\begin{aligned} &\text{Minimize} && C_0(x) \\ &\text{subject to} && g_1(x) \geq 0 \\ & && g_2(x) \geq 0 \\ & && \bullet \\ & && \bullet \\ & && \bullet \\ & && g_m(x) \geq 0 \\ &\text{and} && x \in X \end{aligned}$$

in which X represents the domain of optimization defined by the m constraints, normally an euclidean n-space, and x denotes a vector $(x_1, x_2, \ldots, x_n)$. (Problems that are NP-complete can be put into this form. There exists problems that are not NP-complete, but are put into this form as well.) A solution vector that lies in x is called a feasible solution, and a feasible solution for which $C_0(x)$ is minimal is called an optimal solution.

Research supported partially by National Science Foundation Grant ECS81-05968 and by CIDMAC, a research unit of Purdue University, sponsored by Purdue, Cincinnati Milicron Corporation, Control Data Corporation, Cummins Engine Company, Ransburg Corporation, and TRW.

The branch-and-bound algorithm is an organized and intelligently structured search of the space of all feasible solutions. It has been extensively studied in areas such as artificial intelligence and operations research [HOW78, NIL80]. Problems in scheduling [LEN76], knapsack [ING77], traveling salesman [GAR73], facility allocation [EFR66], integer programming [GAR72], and many others can be solved optimally or sub-optimally by branch-and-bound algorithms. Dominance relations similar to those used in dynamic programming have been used to prune search tree nodes [MOR76].

Besides requiring exponential time complexity, branch-and-bound algorithms may require exponential memory space for storing intermediate results. Conventional virtual memory system has been found to be inadequate for supporting efficient execution of branch-and-bound algorithms. In this paper, we have studied the virtual memory design for these algorithms. The relationship between branch-and-bound algorithms and virtual memory is discussed in Section 2. The features of a virtual memory that can be tailored to support these algorithms include the data structure, replacement policy, page size, and secondary storage data organization. These are investigated in Sections 3 to 5. Simulation results concerning the performance of branch-and-bound algorithms for integer programming are shown in Section 6. Although it seems that this study is directed towards application-dependent virtual memory systems, our results help designers to discover the essential features of a virtual memory for supporting specialized applications. Our long term objective is to design a virtual memory system with a set of alternative features that can be selected dynamically to support a variety of different applications.

2. BRANCH-AND-BOUND ALGORITHMS AND VIRTUAL MEMORY

2.1 Branch-and-bound Algorithms

In branch-and-bound algorithms, the space of all feasible solutions is repeatedly partitioned into smaller and smaller subsets, and both the lower and upper bounds are calculated for solutions within each subset. After each partitioning, subsets with lower bounds (in the case of minimization) that exceed either the value of a known feasible solution or the least upper bound of all subsets are excluded from further consideration. The partitioning process continues until a feasible solution is found such that the value is no greater than the lower bound of any subset. Each partition is also known as a subproblem.

There are two essential features of a branch-and-bound algorithm: the branching and bounding rules. The

branching algorithm examines the set of active subproblems and, based on some predefined criteria, selects one for expansion. If the set of active subproblems is maintained in a first-in first-out (FIFO) list, the algorithm is called a breadth-first search. If the set is maintained in a last-in, first-out list, the algorithm is called a depth-first search. Lastly, if the subproblem selected for expansion is one with the minimum lower bound, the search algorithm is called a best-first search.

Once the subproblem has been selected for partitioning, some undetermined parameters in the subproblem must be selected so that alternatives for these parameters can be defined and multiple subproblems created. For example, in the traveling salesman problem, the undetermined alternatives are the set of untraversed edges. In expanding a subproblem, an untraversed edge (i,j) is selected, and two alternatives can be created: (1) the edge is traversed and the salesman goes directly from city i to city j, and (2) the edge is not traversed. The parameter chosen to be expanded is usually done in an ad hoc manner.

After new subproblems are created, the bounding algorithm is applied to evaluate the upper and lower bounds of a subproblem. Generally, only the lower bound is evaluated, and the upper bound is updated when feasible solutions are found. The bounding algorithm is highly dependent on the problem.

As an example to illustrate the use of branch-and-bound algorithms, the evaluation of an integer programming problem [MIT70] is shown here. Integer programming problems may be expressed as,

$$\text{Minimize } CX$$
$$\text{subject to } AX \geq B$$
$$X^T = (x_1, x_2, \ldots, x_n)$$
$$x_i\text{: non-negative integer, } i=1,2,\ldots n.$$

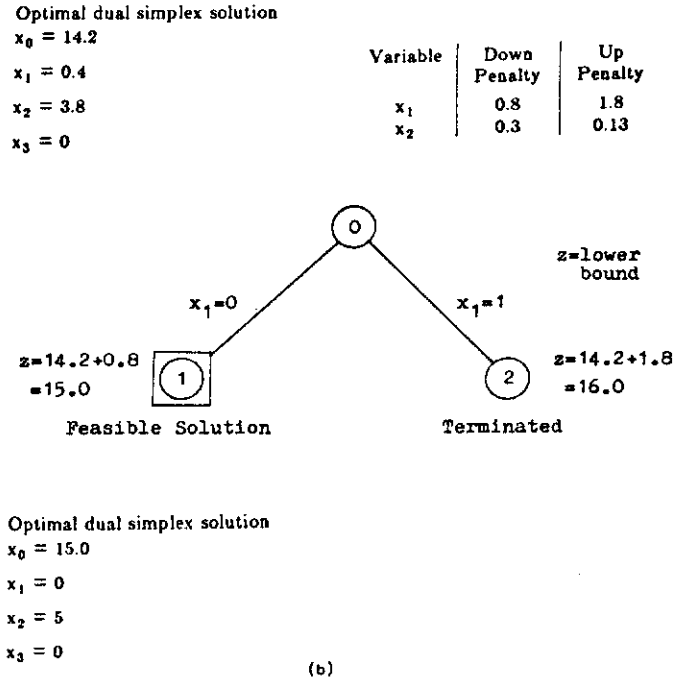
These problems differ from ordinary linear programming problems in that the variables are restricted to non-negative integer values.

One approach to the problem is the following. Apply the dual simplex method to a subproblem and solve it as a linear program. If the optimal solution is integral, a feasible solution has been generated; otherwise, create two new subproblems as follows. Choose a variable that has a non-integer value (say $x_i = 4.4$) and restrict that variable to the next lower integral value for one problem ($x_i \leq 4$) and to the next higher integral value ($x_i \geq 5$) for the other. The variable chosen is the one with the greatest up or down penalty. The up penalty for a variable x_i having a value of a_i is the estimate of the amount by which the solution to the current subproblem would increase if the integral constraint $x_i \geq \lceil a_i \rceil$ was introduced. The down penalty is similar, except that it is associated with the constraint $x_i \leq \lfloor a_i \rfloor$. The lower bound of a new subproblem is the sum of the optimal simplex solution and the associated penalty. This process is repeated on the new subproblems.

Figure 1(b) shows the branch-and-bound tree for the problem in Figure 1(a). The dual simplex method gives an optimal solution of 14.2 for the original problem. Since the variables are not integral, a feasible solution cannot been generated. Up and down penalties are calculated for the variables. x_1 has the greatest penalty (U = 1.8). Two new subproblems are then created, one with $x_1=0$, and the other with $x_1 \geq 1$. The lower bounds are calculated as in Figure 1(b). The dual simplex method is

$$\min x_0 = 7x_1 + 3x_2 + 4x_3$$
$$x_1 + 2x_2 + 3x_3 \geq 8$$
$$3x_1 + 2x_2 + x_3 \geq 5$$
$$x_1, x_2, x_3 \geq 0, \text{ integer}$$

(a)



(b)

Figure 1 (a) An example of an integer programming problem and (b) the corresponding branch-and-bound solution.

then applied to the subproblem with the smallest lower bound again, and a feasible solution is generated with all variables having integral values. This constitutes an optimal solution since the lower bound of the remaining subproblem is greater.

The total number of nodes expanded under best-first search is minimum in the sense that any branching operation performed under this policy must also be performed under other search criteria, provided that all the bounds are unique [LAW66]. The evaluation time will be minimized using best-first search. However, best-first search is space-consuming because all active subproblems must be stored as intermediate data in the computer. It has been shown [YU83] that the memory space required under best-first search grows exponentially as the size of the input. On the other hand, both depth-first and breadth-first searches are space-saving because the memory space for storing the state of the parent subproblem that lead to the next subproblem is very small. Since time is a more critical factor in evaluating large optimization problems, best-first search is a better policy to use if the memory requirement can be satisfied. This implies the need for an efficient virtual memory system to manage the large addressing space.

2.2 Virtual Memory Systems

An enormous amount of literature has been published concerning virtual memory systems [DEN70,

DOR76]. The set of addresses that a task can reference is called the (virtual) address space V. The set of physical locations in main memory allocated to the task is the memory space M. The address and memory space are usually divided into contiguous blocks called pages. During execution, reference to addresses in the object code will be to virtual addresses. The virtual memory system automatically maps the address space into memory space.

Central to the successful operation of paging systems is the principle of locality. This principle states that, during execution, a process will favor a subset of V, i.e., only a subset of its pages have to be resident in main memory to allow seldom interrupted execution intervals. There are two components to this locality of reference: temporal locality which is the tendency for a program to reference in the near future those pages referenced in the recent past; and spatial locality which is the tendency for a program to reference neighboring pages. Loops, constants, temporary variables, and working stacks are constructs which lead to temporal locality. Sequential portions of code and traversals of arrays give rise to spatial locality.

These have led to investigations into the behavior of programs in order to determine methods for enhancing the locality of programs. Some rules to be followed have been proposed [MOR66]. Studies have been performed on automatic restructuring of programs in order to tailor the programs to the locality-based virtual memory environment [BAE76, HAT71, FER74, FER75, FER76]. However, few studies have been made on the effects of data structures on virtual memory performance. Some examples of studies on algorithms utilizing special data structures include matrix manipulation [MCK69, FIS79], sorting [BRO70], and database operations [SMI78]. Nearly all the virtual memory systems available today are geared towards the locality of programs. At this time, a general methodology for designing virtual memory systems for data structure oriented algorithms is lacking.

3. VIRTUAL MEMORY DESIGN FOR BRANCH-AND-BOUND ALGORITHMS UNDER BEST-FIRST SEARCH

The general assumptions made under this study are:

(a) During partitioning, each subproblem generates a fixed number (denoted by s) of new subproblems

(b) The difference between the lower bound of a subproblem and the lower bound of its parent subproblem is exponentially distributed. Simulations have verified this assumption for integer programming problems [WAH82].

3.1 Data Structures

The data structures selected for the implementation of branch-and-bound algorithms depend on the operations performed on the subproblem list, which include,

(a) insert(x) — insert a subproblem with lower bound x into the subproblem list;

(b) delete–smallest(y) — delete the subproblem with the smallest lower bound y from the subproblem list;

(c) delete–not–less–than(z) — delete all subproblems with lower bounds not less than z from the subproblem list.

Each iteration of the branch-and-bound algorithm commences with a delete-smallest operation. Let v be the lower bound of this subproblem. If a feasible solution w is generated, then a delete-not-less-than(w) operation must be carried out. Otherwise s new subproblems with lower bounds $x_1, x_2, ..., x_s$ are generated and inserted into the subproblem list. Under best-first search, subproblems are ordered by increasing lower bounds in an ordered list. Since x_i, $i = 1, ..., s$, are random variables, it is impossible to predict where the subproblems will be inserted.

Table 1 Time complexities of key operations in branch-and-bound algorithms for various data structures.

	insert	delete smallest	delete not less than
sequential list	O(m)	O(m)	O(log m)
linked list	O(m)	O(1)	O(m)
B-tree	O(log m)	O(log m)	O(m log m)
heap	O(log m)	O(log m)	O(m log m)

The time complexities of the key operations for various data structures are shown in Table 1. It can be seen that no single data structure is optimal for all the three operations. A suitable choice will depend upon the relative frequencies of the operations. Recalling our assumption that each expansion produces s son subproblems, insertions should be approximately s times as frequent as deletions. Simulations show that the delete-not-less-than operation is very infrequent. The B-tree[1] [COM79] and the heap[2] are the best data structures for implementing branch-and-bound algorithms.

The amount of locality exhibited by each implementation is shown in Table 2. We see that the algorithm exhibits locality for either insertion or deletion, but not both. This implies that branch-and-bound algorithms will be unable to run efficiently in a conventional locality based virtual memory environment. Modification of the algorithm to enhance the amount of locality exhibited does not appear to be a feasible approach.

Table 2 Amount of locality exhibited by the key operations

	insert	delete smallest	delete not less than
B-tree	weak	strong	strong
heap	strong	weak	weak

3.2 Organization of the Primary and Secondary Memeory

We propose that the list of subproblems be split into two disjoint sub-lists, a primary list which resides in the main memory and a secondary list which resides in the secondary storage. Newly generated subproblems are inserted into the primary list. When the primary list grows to its maximum size, subproblems are removed from the primary list and inserted into the secondary list. In the mean time, computation can still be carried out. This use of a primary list in this manner is equivalent to batching the insertions into the virtual space, which

1. A B-tree of order m is a search tree which is either empty of of height greater than or equal to one and satisfies the following properities:
   (i) The root node has at least two children;
   (ii) Each node contains at most 2m keys and 2m+1 pointers;
   (iii) Each node contains at least m keys and m+1 pointers.
2. A heap is a complete binary tree with the property that the value of each node is at least as small as the value of its children nodes.

reduces the cost of insertions.

The organization chosen for the primary list has to allow efficient insertion and retrieval of the subproblem with the smallest lower bound. A suitable choice is the heap. The subproblem selected for partitioning is the smaller of the smallest subproblems in the primary and secondary lists. Thus it is necessary to keep a number of the smallest subproblems of the secondary list in a portion of the main memory, called the p-buffer.

The secondary list is organized as a B^+-tree [COM79]. Each leaf of the B^+-tree corresponds to a page of the virtual memory system where the subproblems are stored. (The terms page and leaf will be used interchangeably.) The upper levels are organized as a B-tree and consists only of indexes to enable rapid location of subproblems. Since the internal nodes of a B^+-tree are very much smaller in size than the leaf nodes, it is possible to keep the internal nodes in main memory. This reduces the number of secondary accesses required by each page access to O(1).

To further reduce the cost of each page access, the B^+-tree was modified by removing the restriction that subproblems in a page have to be stored sequentially by non-decreasing lower bounds. The block size is a parameter that can be set by the operating system, subject to constraints like sector size and overall disk capacity. The block size was chosen to be the smallest number of sectors equal to or greater than the size of a subproblem and a block was allowed to hold only one subproblem. Each page consists of h_s successive blocks. A subproblem is allowed to reside in any block as long as it is in the correct page.

Each page is assigned a unique number between 0 and $\ell_s - 1$, where ℓ_s is the number of pages in the B^+-tree. A set of ℓ_s status vectors are maintained in main memory. Status vector ω is assigned to page ω. Each status vector contains h_s bits. Bit ς of status vector ω will be set whenever block ς of page ω contains a subproblem, and cleared otherwise.

To insert j subproblems into page ω, the operating system uses status vector ω to select the j empty blocks into which the j subproblems are to be written. As soon as the write head is positioned over each of the chosen blocks, the corresponding subproblem is written onto the disk, and the status vector is updated. Two approaches to the selection of blocks have been considered:

(a) *Sequential allocation:* The j vacant blocks with the smallest block numbers are selected. This scheme ensures that the subproblems are kept in successive blocks.

(b) *First available allocation:* Assuming that the operating system knows the current position of the write heads after they have been moved to the cylinder containing the page, the status vector is scanned for the first j vacant blocks that will come under the write heads. Under this scheme, subproblems will occupy clusters (due to the batching of insertions) scattered throughout the page.

Due to our relaxed conditions on the secondary storage organization, subproblems are not stored in a sorted order in a page. Since the loading of the p-buffer requires the selection of the k subproblems with the smallest lower bounds in a page, this may be done by building a selection tree of lower bounds of size h_s in main memory. An alternative is to modify the best-first search process of the branch-and-bound algorithm. The k subproblems residing in blocks with the smallest block numbers are loaded. Now, instead of partitioning the subproblem with the smallest lower bound, a subproblem with a lower bound that is at least the q-th smallest is partitioned, where q is the number of subproblems remaining in the page. The effect of this, at worst, is to increase the number of iterations by q.

When overflow occurs, before the overflowing page can be split, the median lower bound has to be determined. All subproblems with lower bounds less than the median lower bound remain in this page while those with lower bounds greater than the median lower bound are moved to a new page. Similarly, when underflow occurs, before the actual redistribution of subproblems between two pages can be carried out, it is necessary to find the median lower bound which will divide the subproblems evenly between the pages.

In the following sections, the design of the virtual memory operating system will be considered by the appropriate choice of the replacement algorithm and page size. The expected number of page accesses is derived in the next section. This is used to select an optimal replacement algorithm.

4. THE EXPECTED NUMBER OF PAGE ACCESSES

The two principal causes of page accesses are the transfer of subproblems from the main memory to secondary storage during replacement, and the transfer of subproblems from the secondary storage to p-buffer. In this section, we derive an expression for a quantity η which is the expected number of page accesses normalized by the expected number of iterations. By minimizing this expression, a suitable replacement algorithm may be determined.

4.1 The Expected Number of Page Accesses due to Replacement

Let the maximum size of the primary list be n_p subproblems; that is, the replacement algorithm is called when the primary list grows to n_p subproblems. Suppose then ϕ subproblems are removed from the primary list and inserted into the pages of the secondary list. Also, let the number of subproblems in the secondary list just before the invocation of the replacement algorithm be N_s subproblems.

Let $C_{p,i}$, $i=1,\ldots,n_p$ be the lower bounds of subproblems in the primary list such that $C_{p,1} \leq C_{p,2} \cdots \leq C_{p,n_p}$. Similarly, let $C_{s,i}$, $i=1,\ldots,N_s$ be the lower bounds of subproblems in the secondary list with $C_{s,1} \leq C_{s,2} \cdots \leq C_{s,N_s}$. We can then define,

$$X_i = C_{p,i} - C_{p,1} \qquad i = 1, \cdots, n_p \tag{1}$$

$$Y_j = C_{s,j} - C_{s,1} \qquad j = 1, \cdots, N_s \tag{2}$$

It is assumed that,

(i) $C_{p,1} = C_{s,1}$ (3)

(ii) X_i, $i=1,\ldots,n_p$ are independent, identically distributed random variables having an exponential density function,

$$f_X(x) = \lambda_p e^{-\lambda_p x} \tag{4}$$

(iii) Y_j, $j=1,\ldots,N_s$ are independent, identically distributed random variables with exponential density function,

$$f_Y(y) = \lambda_s e^{-\lambda_s y} \tag{5}$$

Assumptions (ii) and (iii) have been justified experimentally for integer programming problems [YU83].

One way of reducing the cost of insertions during replacement is to batch the insertions. The lower bound of a subproblem determines the page into which the subproblem is to be inserted. The subproblems that are

most likely to be inserted into the same page as the subproblem with a lower bound of $C_{p,i}$ are subproblems with lower bounds of $C_{p,i-1}$ and $C_{p,i+1}$. This suggests that these subproblems should be removed together. Suppose the ϕ subproblems selected for removal are the subproblems with lower bounds $C_{p,\theta+1}, C_{p,\theta+2}, \cdots, C_{p,\theta+\phi}$, where $X_{\theta+1}$ has the value v and $X_{\theta+\phi}$ has the value w. Let M_s be the number of subproblems in the secondary list with $v < Y_i \leq w$. The probability of $v < Y_i \leq w$ for any subproblem in the secondary list is given by,

$$\Pr\{v < Y_i \leq w\} = \int_{x=v}^{w} \lambda_s e^{-\lambda_s x}\, dx$$

The probability that $M_s = m$ constitutes a binomial distribution.

$$\Pr\{M_s=m\} = \binom{N_s}{m} \left(\Pr\{v<Y_i\leq w\}\right)^m \left(1-\Pr\{v<Y_i\leq w\}\right)^{N_s-m}$$

The expected value of M_s given that $X_{\theta+1} = v$, $X_{\theta+\phi} = w$, and $N_s = n$ is,

$$E[M_s|\, X_\theta=v, X_{\theta+\phi}=w, N_s=n] = n \Pr\{v<Y_i\leq w\} \quad (6)$$

Since N_s and $P\{v < Y_i \leq w\}$ are independent, Eq. (6) may be evaluated as,

$$E[M_s] = E[N_s] \int_{v=0}^{\infty}\int_{w=0}^{\infty} \Pr\{v<Y_i\leq w\}\, f_V(v)\, f_W(w)\, dv dw$$

$$= E[N_s] \left(\frac{n_p!}{(n_p-\theta)!} \frac{\Gamma(n_p-\theta+1+\varsigma)}{\Gamma(n_p+1+\varsigma)} - \frac{n_p!}{(n_p-\theta-\phi)!} \frac{\Gamma(n_p-\theta-\phi+1+\varsigma)}{\Gamma(n_p+1+\varsigma)} \right) \quad (7)$$

where

$$\varsigma = \frac{\lambda_s}{\lambda_p}$$

Let the M_s subproblems reside in R pages, and let $Q_1, Q_2, \cdots, Q_R$ be the number of subproblems in these R pages. Assume that they are independent and identically distributed with uniform distribution between $1/2\, h_s + 1$ and h_s, where h_s is the size of each page in subproblems. Let Q be a random variable representing the number of subproblems in a page. Since R is a stopping time, renewal theory and Wald's equation [ROS70] gives,

$$E[M_s] = E[R]\, E[Q]$$

and

$$E[R] = \frac{4}{3h_s+2} E[N_s] \left(\frac{n_p!}{(n_p-\theta)!} \frac{\Gamma(n_p-\theta+1+\varsigma)}{\Gamma(n_p+1+\varsigma)} - \frac{n_p!}{(n_p-\theta-\phi)!} \frac{\Gamma(n_p-\theta-\phi+1+\varsigma)}{\Gamma(n_p+1+\varsigma)} \right) \quad (8)$$

The derivation of the distribution function of R is extremely difficult. For mathematical tractability, the expected value of R will be used instead. Of the E[R] pages, let subproblems be actually inserted into G pages. Assuming that the ϕ insertions are uniformly distributed among the E[R] pages, the number of subproblems inserted into any page, K, satisfies the binomial distribution,

$$\Pr\{K=j\} = \binom{\phi}{j} \left(\frac{1}{E[R]}\right)^j \left(1-\frac{1}{E[R]}\right)^{\phi-j}$$

The expected number of pages into which subproblems are inserted is,

$$E[G] = E[R]\, \Pr\{K \geq 1\} = E[R] \left[1 - \left(1 - \frac{1}{E[R]} \right)^{\phi} \right] \quad (9)$$

Out of the G pages, H pages have to be split due to overflow. Suppose a page has q subproblems in it. Then the page has to be split only if more than $(h_s - q)$ subproblems are inserted into the page. Thus,

$$\Pr\{\text{split page}|\, Q=q\} = \Pr\{K \geq h_s - q + 1\}$$

The probability that a page has to be split, f_r, is given by,

$$f_r = \sum_{q=\delta}^{h_s} \Pr\{Q=q\} \Pr\{\text{split page}|\, Q=q\} \quad (10)$$

$$= \sum_{q=\delta}^{h_s} \frac{2}{h_s} (1 - \Pr\{K \leq h_s - q\})$$

$$= \frac{2}{h_s}(h_s+1-\delta) - \frac{2}{h_s} \sum_{j=0}^{h_s-\delta} \left\{ \left(h_s+1-\delta-j\right) \times \binom{\phi}{j} \left(\frac{1}{E[R]}\right)^j \left(1-\frac{1}{E[R]}\right)^{\phi-j} \right\} \quad (11)$$

where

$$\delta = \max(1/2h_s + 1\ ,\ h_s - \phi + 1)$$

Since any of the G pages is either split or not split, this is a binomial experiment with f_r as the probability of success and $1 - f_r$ the probability of failure. The expected number of pages split is

$$E[H] = f_r\, E[G]$$

Splitting of a page requires two page accesses. Thus the expected number of page accesses incurred during a replacement, $E[T_s]$, is given by,

$$E[T_s] = (E[G] + E[H]) + 2E[H] = (2 + f_r) E[G]$$

$$= \left\{ 1 + \frac{6}{h_s}(h_s+1-\delta) - \frac{6}{h_s} \sum_{j=0}^{h_s-\delta} \left[(h_s+1-\delta-j) \times \binom{\phi}{j} \left(\frac{1}{E[R]}\right)^j \left(1-\frac{1}{E[R]}\right)^{\phi-j} \right] \right\} \times E[R] \left[1 - \left(1 - \frac{1}{E[R]} \right)^{\phi} \right] \quad (12)$$

4.2 Expected Number of Iterations between Consecutive Replacements

The main memory contains the primary list and the p-buffer which holds part of the page of the B^+-tree containing subproblems with the minimum lower bounds. At present, the space set aside for the primary list and p-buffer are fixed. The case where the space allocated is variable dynamically will be considered in the future.

When a subproblem is chosen for partitioning, let f_s be the probability that the subproblem is selected from

the p-buffer. Also, let the number of subproblems partitioned between two consecutive replacements be D. Of these D subproblems, let D_s subproblems be from the p-buffer and D_p subproblems be from the heap (primary list). Thus,

$$D = D_s + D_p$$

and

$$\Pr\{D_s = k \mid D\} = \binom{D}{k} (f_s)^k (1 - f_s)^{D-k}$$

The expected number of subproblems from the p-buffer partitioned between two consecutive replacements is,

$$E[D_s] = E[E[D_s \mid D]]$$
$$= f_s E[D] \quad (13)$$

Similarly, the expected number of subproblems from the heap partitioned between two consecutive replacements is,

$$E[D_p] = (1 - f_s) E[D] \quad (14)$$

Each partitioning of a subproblem generates s new subproblems. These newly generated subproblems are inserted into the heap. The size of the heap will increase by s whenever a subproblem from the p-buffer is partitioned. When a subproblem from the heap is partitioned, the increase in the size of the heap is only s − 1 since the subproblem being partitioned has been deleted from the heap. Equating the increase in the size of the primary list between two replacements to the fixed number of subproblems removed during a replacement, we have,

$$\phi = s E[D_s] + (s-1)E[D_p]$$
$$= s f_s E[D] + (s-1)(1-f_s) E[D]$$
$$= (s - 1 + f_s) E[D]$$

and

$$E[D] = \frac{\phi}{s - 1 + f_s} \quad (15)$$

4.3 The Expected Number of Page Accesses Due to Loading of the p-buffer

Let the D_s subproblems reside in F pages of the secondary memory, and $Q_1', Q_2', \ldots, Q_F'$ subproblems be the number of subproblems in each of these F pages. These Q_i's have the same distribution as Q. Let F be a stopping time. The use of renewal theory and Wald's equation gives,

$$E[D_s] = E[F] E[Q]$$

and,

$$E[F] = \frac{4}{3h_s + 2} E[D_s]$$
$$= \frac{4}{3h_s + 2} \frac{f_s \phi}{s - 1 + f_s}$$

Let the expected number of page accesses due to the loading of p-buffer between two successive replacements be $E[T_l]$. Then,

$$E[T_l] = E[F]$$
$$= \frac{4}{3h_s + 2} \frac{f_s \phi}{s - 1 + f_s} \quad (16)$$

4.4 An Expression for η

From its definition, η may be written as,

$$\eta = \frac{E[T_s] + E[T_l]}{E[D]} \quad (17)$$

Substituting Eq.'s (12), (15) and (16) into Eq. (17) yields,

$$\eta = \frac{s - 1 + f_s}{\phi} \left\{ \frac{4}{3h_s + 2} \frac{f_s \phi}{s - 1 + f_s} + \left[1 + \frac{6}{h_s} (h_s + 1 - \delta) - \frac{6}{h_s} \sum_{j=0}^{h_s - \delta} \left(h_s - \delta + 1 - j \right) \binom{\phi}{j} \left(\frac{1}{E[R]} \right)^j \left(1 - \frac{1}{E[R]} \right)^{\phi - j} \right] \times E[R] \left[1 - \left(1 - \frac{1}{E[R]} \right)^{\phi} \right] \right\} \quad (18)$$

4.5 Comparison of η with Simulation Results

To verify Eq. (18), simulations were performed using integer programming problems. For these problems, each partitioning generates two new subproblems (s = 2). Using the values of f_s, λ_p, and λ_s obtained from simulations, the corresponding values of η were computed and compared against η_{actual} (Table 3). The results show that Eq. (18) is a reasonably good measure of the number of page accesses for integer programming problems.

Table 3 Comparison of η with η_{actual} for two 20 variables, 20 constraints integer programming problems (n_p = 100, h_s = 50).

θ	ϕ	Problem 1		Problem 2	
		η	η_{actual}	η	η_{actual}
0	50	0.80	0.80	0.47	0.46
10	50	0.65	0.73	0.37	0.41
20	50	0.56	0.68	0.31	0.39
30	50	0.51	0.64	0.29	0.36
40	50	0.50	0.61	0.28	0.33
50	50	0.54	0.59	0.28	0.31
80	20	0.65	0.73	0.34	0.41
70	30	0.60	0.68	0.30	0.36
60	40	0.56	0.63	0.29	0.35
50	50	0.54	0.60	0.28	0.31
40	60	0.53	0.57	0.28	0.31
30	70	0.53	0.54	0.29	0.30
20	80	0.54	0.54	0.30	0.31
10	90	0.55	0.55	0.31	0.31

5. REPLACEMENT RULE AND PAGE SIZE

As discussed in the last section, the proposed replacement rule $S(\theta,\phi)$ selects subproblems with lower bounds $C_{p,\theta+1}, C_{p,\theta+2}, \cdots, C_{p,\theta+\phi}$ for removal. Two special cases of this rule may be identified: $S(0,\phi)$ or *Front Replacement* where the ϕ subproblems with the smallest lower bounds are replaced; and $S(n_p-\phi,\phi)$ or *Back Replacement* where the ϕ subproblems with the largest lower bounds are replaced.

Thus the problem of selecting the proper replacement rule reduces to the selection of the proper values of:

(i) θ which is the index of the first subproblem to be removed from main memory;

(ii) ϕ which is the number of subproblems to be removed from main memory during a replacement.

The page access measure, η, allows the value of θ and ϕ to be selected. These values have to be chosen so that they will minimize η as given by Eq. (18) for a given n_p, h_s, and $E[N_s]$. However, this is difficult as f_s, λ_p and λ_s are complex functions of both θ and ϕ. We will optimize θ and ϕ iteratively here. Let θ^* be the value of θ which minimizes Eq. (18) for a given n_p, h_s, $E[N_s]$ and ϕ. Simulations based on integer programming problems show that the parameters f_s, λ_p and λ_s vary as θ varies [YU83]. Quadratic approximation of these variations was used, whereby,

$$f_s = \alpha_{1,1} + \alpha_{1,2}\,\theta + \alpha_{1,3}\,\theta^2$$
$$\lambda_p = \alpha_{2,1} + \alpha_{2,2}\,\theta + \alpha_{2,3}\,\theta^2 \qquad (19)$$
$$\lambda_s = \alpha_{3,1} + \alpha_{3,2}\,\theta + \alpha_{3,3}\,\theta^2$$

The optimization of θ was then performed using non-linear programming. The results are consistent and show that Back Replacement should be used. The details of this are not shown here [YU83]. The use of Back Replacement is also suggested by statistics on the number of page accesses collected during the simulations for integer programming problems (Figure 2).

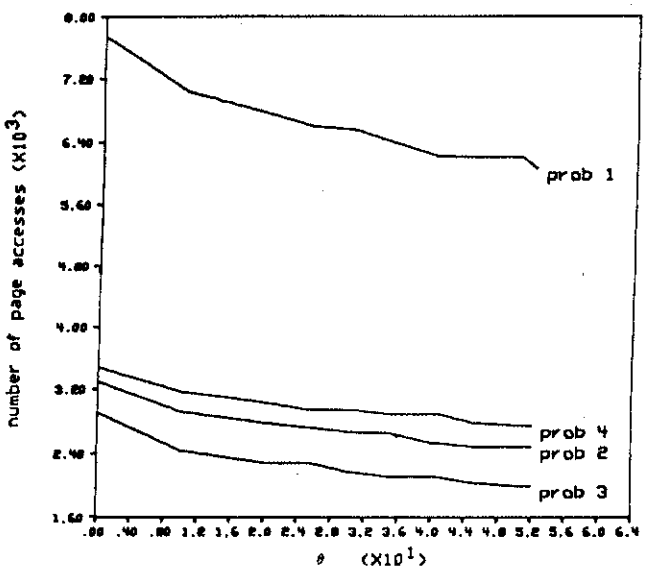


Figure 2 Variation of the number of page accesses with θ for four 20 variables, 20 constraints integer programming problems (n_p = 132, h_s = 33, ϕ = 80)

Assuming that Back Replacement is used, ϕ may be chosen. Let ϕ^* be the value of ϕ which minimizes Eq. (18). Once again, complications arise from the variations of f_s, λ_p and λ_s with ϕ. Quadratic approximation was again used.

$$f_s = \beta_{1,1} + \beta_{1,2}\,\phi + \beta_{1,3}\,\phi^2$$
$$\lambda_p = \beta_{2,1} + \beta_{2,2}\,\phi + \beta_{2,3}\,\phi^2 \qquad (20)$$
$$\lambda_s = \beta_{3,1} + \beta_{3,2}\,\phi + \beta_{3,3}\,\phi^2$$

In the actual simulation results, some problems exhibited spikes in the values of parameters f_s, λ_s, and λ_p [YU83]. Quadratic approximation was unable to describe these sharp variations.

Non-linear programming was again used to solve for ϕ^*. It is observed that the values of ϕ^* should be between $0.7\,n_p$ and $0.9\,n_p$ [YU83]. Statistics on page accesses collected by simulations on integer programming problems also suggest such a result (Figure 3). It is noticed that the number of page accesses is minimum at a ϕ which is problem-dependent. This value is difficult to predict. A conservative approach is to select ϕ in the range of $0.7n_p$ to $0.9n_p$. Here the number of page accesses is consistently close to the minimum and differs from it by at most 10%.

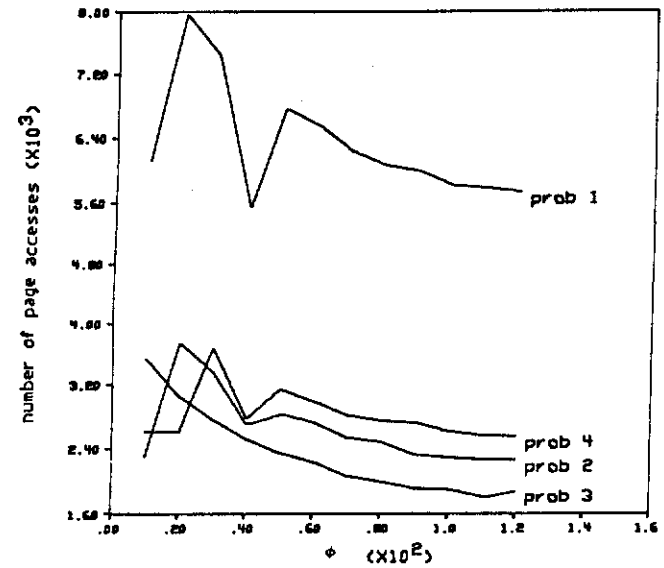


Figure 3 Variation of the number of page accesses with ϕ for four 20 variables, 20 constraints integer programming problems (n_p = 132, h_s = 33, $\theta = n_p - \phi$)

Given the replacement algorithm, the optimal page size can be determined. The cost of accessing a page on a moving head disk is given by the seek, rotational, and transfer time. By simulations on integer programming problems, it is found that a page size, h_s, in the range of $0.65\,n_p$ and $0.85\,n_p$ should be used (Figures 4-5).

6. PERFORMANCE EVALUATION

Figures 6 and 7 compare the paging costs incurred in the modified virtual memory system against those incurred in a general purpose virtual memory system with LRU replacement algorithm for two 20 variables, 20 constraints integer programming problems. Other simulation results are reported elsewhere [YU83]. It can be seen that the modified virtual memory system has a better performance than the conventional virtual memory system and that it shows a greater improvement with increasing disk density.

For an integer programming problem, let P_m be the paging cost on the modified virtual memory system, and P_l the paging on a LRU virtual memory system. The improvement factor R_c is defined as,

$$R_c = \frac{P_l}{P_m}$$

For a head per track device, R_c increases from between 1.3 to 1.4 at 16 kbytes/track to between 3.7 and 4.0 at 64 kbytes/track. For a moving head disk, the increase in R_c is from between 3.2 and 3.9 at 16 kbytes/track to between 7.7 and 9.1 at 64 kbytes/track.

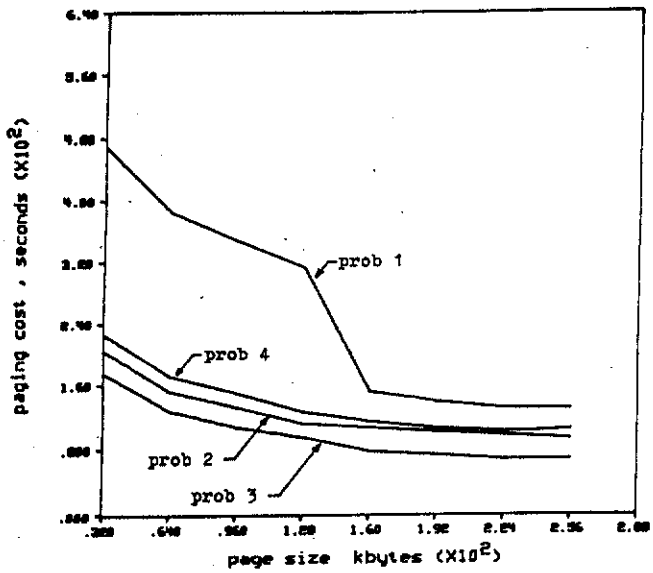


Figure 4 Variation of the total paging cost with page size on a moving head disk (seek time = 28 ms., rotational delay = 16.67 ms.) with a disk density of 16 kbytes/track for four 20 variables, 20 constraints integer programming problems ($n_p = 132$, $\phi = 120$, $\theta = n_p - \phi$)

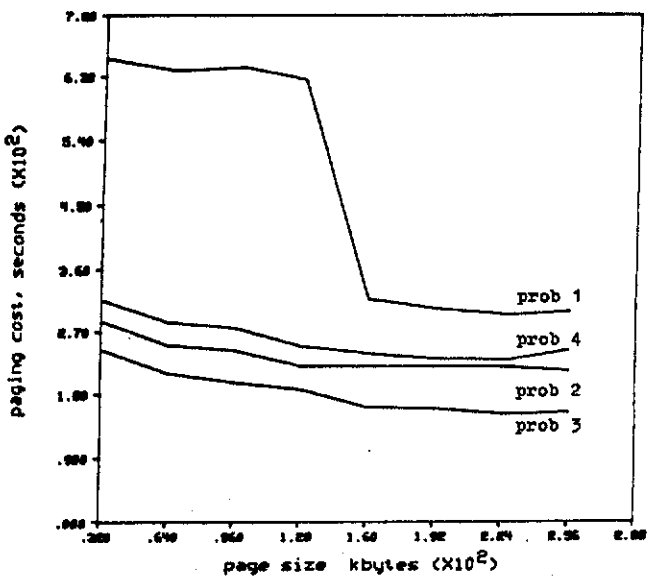


Figure 5 Variation of the total paging cost with page size on a moving head disk (seek time = 28 ms., rotational delay = 16.67 ms.) with a disk density of 64 kbytes/track for four 20 variables, 20 constraints integer programming problems ($n_p = 132$, $\phi = 120$, $\theta = n_p - \phi$)

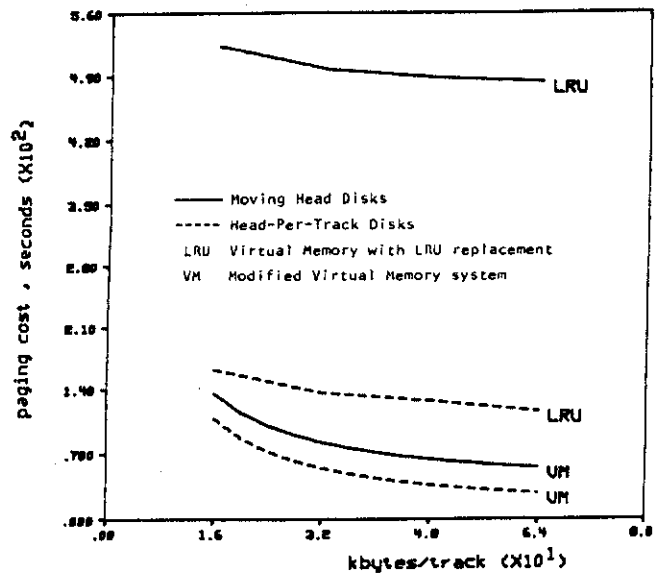


Figure 6 Variation of the paging costs for a 20 variables, 20 constraints integer programming problem (problem 1) with disk density ($n_p = 132$, $h_s = 100$, $\phi = 120$, $\theta = n_p - \phi$)

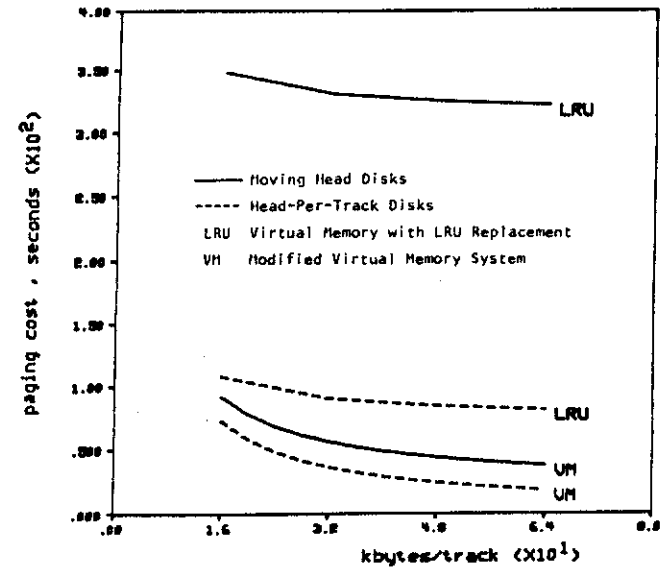


Figure 7 Variation of the paging costs for a 20 variables, 20 constraints integer programming problem (problem 2) with disk density ($n_p = 132$, $h_s = 100$, $\phi = 120$, $\theta = n_p - \phi$)

7. CONCLUSION

In this paper, we have proposed a virtual memory operating system that is based upon the access characteristics of branch-and-bound algorithms. Conventional virtual memory systems are founded on the locality of accesses of programs; and is unsuitable for applications that require large specialized data structures. Although the users may reconfigure the data structures to the requirements of the operating system, this is not always possible as we have discovered in the evaluation of branch and bound algorithms. Other parameters that include the replacement algorithm, page size, and fetch policy may have to be modified. From our analysis and simulations, they indicate that a nine-fold reduction in paging traffic can be achieved in the modified system. The amount of improvement observed increases as disk density increases.

The trend is true in the future design of virtual memory operating systems. When it is impossible to tailor the applications to the characteristics of the virtual memory, it may be necessary to do the reverse. Future virtual memory systems should provide a set of alternatives that can be dynamically switched based on the characteristics of the applications.

REFERENCES

[BAE76] J. L. Baer and G. R. Sager, "Dynamic Improvement of Locality in Virtual Memory Systems," *IEEE Trans. on Software Engineering*, Vol. SE-2, No. 1, pp. 54-62, March 1976.

[BRO70] B. S. Brown, F. G. Gustavson and E. S. Mankin, "Sorting in a Paging Environment," *CACM*, Vol. 13, No. 8, pp. 438-494, August 1970.

[COM79] D. Comer, "The Ubiquitous B-Tree," *Computing Surveys*, Vol. 11, No. 2, pp. 121-137, June 1979.

[DEN70] P. J. Denning, "Virtual Memory," *Computing Surveys*, Vol. 2, No. 3, pp. 153-187, Sept. 1970.

[DOR76] R. W. Doran, "Virtual Memory," *Computer*, pp. 27-37, Oct. 1976.

[EFR66] M. A. Efroymson and T. C. Ray, "A Branch and Bound Algorithm for Plant Location," *Operations Research*, Vol. 14, pp. 361-368, 1966.

[FER74] D. Ferrari, "Improving Locality by Critical Working Sets," *CACM*, Vol. 17, No. 11, pp. 614-620, Nov. 1974.

[FER75] D. Ferrari, "Tailoring Programs to Models of Program Behavior," *IBM Journal of Research and Development*, Vol. 19, No. 3, pp. 244-251, May 1975.

[FER76] D. Ferrari, "The Improvement of Program Behavior," *Computer*, pp. 39-47, Nov. 1976.

[FIS79] P. C. Fisher and R. L. Probert, "Storage Reorganization Techniques for Matrix Computation in a Paging Environment," *CACM*, Vol. 22, No. 7, pp. 405-415, July 1979.

[GAR72] R. S. Garfinkel and G. L. Nemhauser, *Integer Programming*, John Wiley and Sons, Inc., New York, 1972.

[GAR73] R. Garfinkel, "On Partitioning the Feasible Set in a Branch and Bound Algorithm for the Asymmetric Travelling Salesman Problem," *Operations Research*, Vol. 21, No. 1, pp. 340-342, 1973.

[GAR79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman and Company, San Francisco, 1979.

[HAT71] D. J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory," *IBM Systems Journal*, Vol. 10, No. 3, pp. 168-192, 1971.

[HOW78] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Maryland, 1978.

[ING77] G. Ingargiola and J. Korsh, "A General Algorithm for One Dimensional Knapsack Problems," *Operations Research*, Vol. 25, No. 5, pp. 752-759, 1977.

[LAW66] Lawler, E. L. and Wood, D. W., "Branch and Bound Methods: A Survey," *Operations Research*, Vol. 14, pp. 699-719, 1966.

[LEN76] J. Lenstra, "Sequencing by Enumerative Methods," *Math. Centre. Tract 69*, Mathematisch Centrum, Amsterdam, 1976.

[MCK69] A. C. McKellar and E. G. Coffman, Jr., "Organizing Matrices and Matrix Operations for Paged Memory Systems," *CACM*, Vol. 12, No. 3, pp. 153-164, 1969.

[MIT70] L. Mitten, "Branch and Bound Methods: General Formulation and Properties," *Operations Research*, Vol. 18, pp. 24-34, 1970.

[MOR66] J. E. Morrison, "User Program Performance in Virtual Storage Systems," *IBM Systems Journal*, Vol. 5, No. 2, pp. 216-237, 1966.

[MOR76] T. L. Morin and R. E. Marsten, "Branch-and-Bound Strategies for Dynamic Programming," *Operations Research*, Vol. 24, No. 4, July-August 1876.

[NIL80] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Palo Alto, 1980.

[ROS70] S. M. Ross, *Applied Probability Models with Optimization Applications*, Holden-Day, San Francisco, 1970.

[SMI78] A. J. Smith, "Sequentially and Prefetching in Database Systems," *ACM Trans. on Database Systems*, Vol. 3, No. 3, pp. 223-247, Sept. 1978.

[WAH82] B. W. Wah and C. F. Yu, "Probabilistic Modelling of Branch-and-Bound Algorithms," *Proc. of COMPSAC*, pp. 647-623, 1979.

[YU83] C. F. Yu, "Virtual Memory Support for Branch and Bound Algorithms," *MSEE Thesis*, Purdue University, 1983.