This study is based on a virtual-storage concept that provides for automatic memory allocation.

Several algorithms for the replacement of current information in memory are evaluated.

Discussed is the simulation of a number of typical program runs using differing replacement algorithms with varying memory size and block size. The results are compared with each other and with a theoretical optimum.

A study of replacement algorithms for a virtual-storage computer

by L. A. Belady

One of the basic limitations of a digital computer is the size of its available memory.[1] In most cases, it is neither feasible nor economical for a user to insist that every problem program fit into memory. The number of words of information in a program often exceeds the number of *cells* (i.e., word locations) in memory. The only way to solve this problem is to assign more than one program word to a cell. Since a cell can hold only one word at a time, extra words assigned to the cell must be held in external storage. Conventionally, overlay techniques are employed to exchange memory words and external-storage words whenever needed; this, of course, places an additional planning and coding burden on the programmer. For several reasons, it would be advantageous to rid the programmer of this function by providing him with a "virtual" memory larger than his program. An approach that permits him to use a sufficiently large address range can accomplish this objective, assuming that means are provided for automatic execution of the memory-overlay functions.

Among the first and most promising of the large-address approaches is the one described by Kilburn, et al.[2] A similar virtual-addressing scheme was assumed as a starting point for the studies reported in this paper. Within this framework, the relative merits of various specific algorithms are compared. Before

discussing these algorithms in detail, we review the basic features of virtual addressing and operation.

Virtual-storage computer

A virtual-storage computer (VSC) can decode addresses that are longer than those of its memory. The longer address is treated by the VSC as a *virtual address* that must be transformed to the actual, shorter memory address. As in the conventional approach, overflow words are stored externally; in VSC operation, however, the virtual addressing of a word in external storage triggers a procedure that automatically brings the addressed word into memory. For the sake of brevity, the activity involved in moving a word from external storage to memory is termed a *pull*. Similarly, the activity required to move a word from memory to external storage is called a *push*. Generally, a pull must be preceded by a push. This *replacement* operation includes the updating of a *mapping table* which keeps track of the virtual addresses and corresponding actual memory addresses of all words currently in memory.

VSC variables

Conceptually, the simplest VSC operation would limit pulls and pushes to single words. However, since external storage devices are relatively slow and the time delay involved for a single word is frequently very close to that of a *block* of words, it is advantageous to move entire blocks rather than individual words. Experience also shows that, once a word is used, the probability is high that adjacent words in the same block will soon be needed; this is particularly true for the problem programs in machines sequenced by instruction counters. Moreover, the size of the mapping table can be considerably reduced by dealing with large blocks. On the other hand, the larger a block, the higher the proportion of its words that will not be needed soon. In summary, *block size* is clearly an important VSC design parameter.

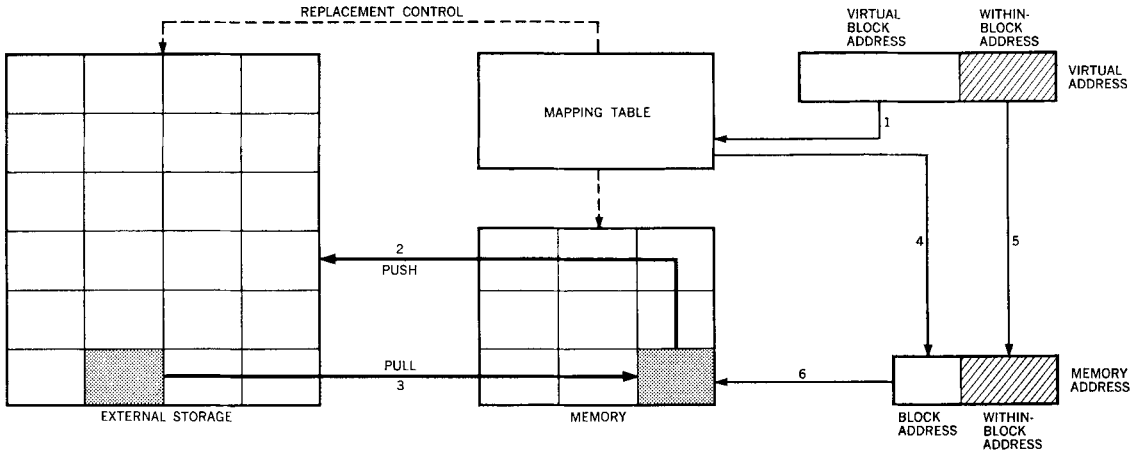
Another parameter is *memory size*. A small memory is inexpensive but entails a high frequency of VSC activity. The maximum memory size that need be considered is, of course, the one which can be directly addressed by the problem program's virtual addresses, without address mapping.

An important design factor is the *replacement algorithm*, which determines which block to push whenever memory space is needed. Of course, it is preferable to first replace the block that has the least chance of soon being referenced. However, as shown later, there is an inherent uncertainty about the occurrence of block usage. Three classes of many conceivable replacement algorithms are discussed in this paper.

virtual addressing process

The virtual addressing process referred to in this paper is illustrated in Figure 1. The CPU references a word by issuing a virtual address, which is divided into two parts. The low-order part, which provides the within-block address of the desired word, is temporarily ignored. The remainder of the virtual address, the virtual block address, is used as an argument for search in

Figure 1 Virtual addressing



the mapping table (Step 1). This table lists the virtual addresses of all blocks presently in memory and links them, row by row, with the associated memory addresses. If the virtual address which serves as an argument is not in the table, a memory block designated by the replacement algorithm is pushed (Step 2), the requested block is pulled (Step 3), and the table is properly updated. If the argument is found in the list at the end of Step 1, Steps 2 and 3 are skipped. The actual memory address of the requested block is extracted from the appropriate row in the table (Step 4). Then, the temporarily held low-order part of the original virtual address and the extracted block address are concatenated (Step 5), forming the complete actual memory address with which the desired information becomes accessible (Step 6).

Heuristic replacement algorithms

For a particular problem program, the performance behavior of a virtual storage computer is determined by memory and block sizes and by the nature of the replacement algorithm. Block size and memory size, being obvious parameters, require no further discussion. Replacement algorithms, however, deserve analysis because they are based on a variety of assumptions and design considerations.

As previously mentioned, the replacement algorithm determines which block to push whenever space is needed in memory. To minimize the number of replacements, we attempt to first replace those blocks that have the lowest probability of being used again. Conversely, we try to retain those blocks that have a good likelihood of being used again in the near future. To study the replacement behavior, it is sufficient to examine the problem program's *sequence of address references*.

When the need for replacement arises, usually no information is available about subsequent references. Since it is not possible

to delay the replacement decision until sufficient information for an optimal solution is attainable, we must rely on the previous distribution of references or assume randomness.

Theoretically, the best replacement pushes a "dead" block, i.e., a block no longer needed by the program run. The worst replacement occurs when a block is referenced immediately after being pushed.

Although a wide variety of possible replacement algorithms exists, they can be grouped into three main classes:

- *Class 1*—It is assumed that all blocks are equally likely to be referenced at any time. The replacement algorithm is not based on information about memory usage.
- *Class 2*—Blocks are classified by the history of their most recent use in memory. The replacement algorithm uses corresponding information.
- *Class 3*—Blocks are classified by the history of their absence and presence in memory. Information is recorded about all blocks of the entire program.

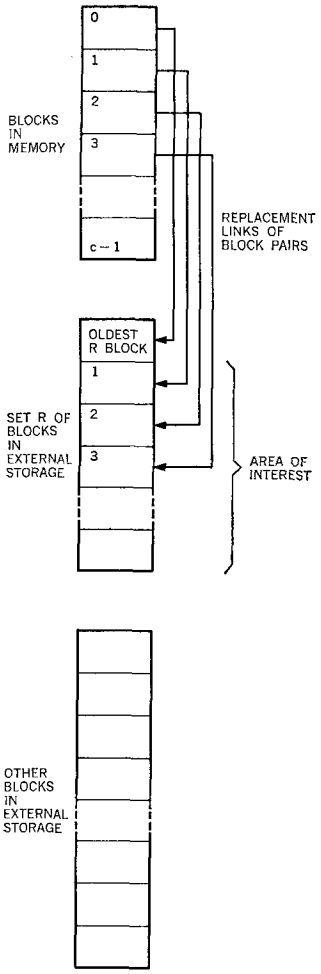
probabilistic model

As a useful benchmark for comparison purposes, we first develop a probabilistic replacement model. For this purpose, we make the primitive assumption that references occur at random, i.e., evenly distributed over the range of all program blocks. Under this assumption, historical information is irrelevant, and the use of any specific replacement rule does not ensure any relative advantage. Therefore, we might as well choose a simple, random replacement scheme in building the probabilistic model. This scheme (let us call it RAND) chooses the block to be pushed at replacement time at random over the range of all blocks in memory.

To find the efficiency of RAND, it suffices to determine the probability of a wrong decision when using RAND. Let s be the number of blocks in the problem program. Then the probability of hitting a particular block at any address reference time is $1/s$. Let c be the number of blocks in memory. Then the probability of referencing a block in memory is c/s, and the probability of a replacement is $(s - c)/s$. A reference to a block already in memory can be considered a *repetition* because at least one previous reference must have occurred (when the block was pulled). From the above expressions, we can deduce that the ratio of repetitions to replacements is $c/(s - c)$.

For the set of all problem program blocks, there is—at any given time—a set R of blocks that were pushed to make room for a new set of blocks in memory. After the initial loading period in a run, each block in memory is associated with a block in R. However, not all blocks in R are necessarily in external storage; a block does not lose its membership in R when referenced and pulled again. Furthermore, a block may be pushed more than once into R by distinct blocks in memory. Of course, according to the above definition, a given block loses membership in R

Figure 2 Block pairs of a probabilistic model



as soon as its *associate* (the block in memory that pushed the given block) is itself pushed out of memory.

Equipped with these definitions, we now examine the possibility of poor replacement decisions. Obviously, a reference to a block in memory does not reveal a previous bad choice of replacement, since the required block was left in memory where it is now needed. Also, reference to a non-R block in external storage does not indicate a previous poor decision, since the block has not recently been pushed. However, a previous poor replacement decision may be revealed by a reference to a recently replaced block (a block in R). Thus, in studying misguided replacement decisions, we can limit the investigation to references to blocks in R.

As a first approximation for our calculations, we assume that there are c distinct blocks in R, i.e., the complete set R is in external storage. Then we can pair the blocks in R with their associates in memory. We order these pairs in the same order in which they are being formed—the first replacement producing the oldest pair, the next replacement the next younger pair, etc. This is illustrated in Figure 2.

A reference to the oldest block in R does not reveal a previous poor replacement choice, because the block's associate is—by definition—the oldest block in memory; hence, none of the other blocks now in memory could have been considered as an alternate choice.

A reference to any younger block in R indicates a previous poor choice if at least one of the blocks now in memory has not been referenced since the replacement under consideration. (Actually, it is sufficient to check merely those blocks in memory that are older than the appropriate associate, because all younger blocks were pulled into memory later than the associate and referenced at that time.) If there is such a non-referenced block in memory, it would have been better to replace that block rather than the block in R under consideration, thus avoiding one pull operation.

We can conclude that there are $c - 1$ blocks in R to which a reference can reveal a possible previous bad choice. This area of interest to us is shown in Figure 2. For the ith such element, the probability that there has been at least one better candidate is

$$1 - \left[1 - \left(1 - \frac{1}{c}\right)^{k_i}\right]^i$$

where

$$k_i = (c - i)\frac{c}{s - c}$$

and $1 - 1/c$ is the probability that a particular block in memory has not been referenced by a repetition; k_i is the number of repetitions since the replacement of the ith block (the oldest being the 0th block). For the ith block, there were i possible better block candidates; hence the exponent i. Since there are

$s - c$ blocks to which a reference causes a replacement, the probability that—at replacement time—a previous bad choice shows up is

$$\frac{(c-1) - \sum_{i=1}^{c-1} \left[1 - \left(1 - \frac{1}{c}\right)^{ki}\right]^{i}}{s - c}$$

still assuming that R consists of c distinct blocks.

Actually, the number of distinct blocks in R is reduced if one of its blocks is pulled again, even though that block does not lose its membership in R. Such a block may become a duplicate block of R if it is pushed again. Assuming $i \geq 1$, the probability that a block in R also appears in memory or has at least one higher-order duplicate in R is $1/(s - c)$ for one single replacement. Hence $1 - 1/(s - c)$ is the probability that the block in R is not pulled again. For the ith block in R $(1 \leq i < c - 1)$, the probability of again being pulled during its most recent presence in R is

$$\left(1 - \frac{1}{s - c}\right)^{(c-1)-i}$$

The $(c - 1)$th block of R certainly exists and has at most only lower-order duplicates. With this, we can refine our formula for the probability of choosing a wrong block for replacement to

$$w = \frac{\sum_{i=1}^{c-2} \left\{1 - \left[1 - \left(1 - \frac{1}{c}\right)^{[c/(s-c)](c-i)}\right]^{i}\right\}\left(1 - \frac{1}{s - c}\right)^{(c-1)-i} + 1 - \left[1 - \left(1 - \frac{1}{c}\right)^{c/(s-c)}\right]^{c-1}}{s - c}.$$

Conversely, the probability of being right is $(1 - w)$, which is the efficiency of any replacement algorithm processing a random sequence of VSC references. For a length l of the random string, the optimal replacement scheme MIN discussed later in this paper would—over a long period—generate

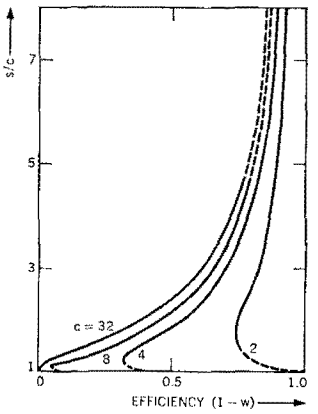
$$l \frac{s - c}{s} (1 - w)$$

replacements, where $(s - c)/s$ is the probability of referencing a block not in memory. For given s and c $(s > c)$, MIN could optimize the random reference string to have a replacement for only every jth reference, where

$$j = \frac{s - c}{s} (1 - w)$$

Figure 3 is a family of curves of w values, each curve computed for fixed c and steps of s/c. In spite of our primitive assumption that the references are at random over s, it is still interesting to see the tendencies. By increasing s/c (larger programs relative to memory capacity), w decreases and the efficiency increases. In other words, RAND does a reasonably good job if the problem

Figure 3 Efficiency values of a probabilistic model



program must be "squeezed" into a relatively much smaller memory. By increasing c and keeping s/c constant, the efficiency goes down.

class 1 algorithms

The above probabilistic model assures us that RAND gives reasonably consistent results: with large s/c, the high relative frequency of replacements is somewhat compensated by a better efficiency; with small s/c, this frequency is low anyway. This is the justification for RAND, which is the most representative member in Class 1 of the replacement algorithms because it does not use any special information—it has a static rule. Such static rules can be justified by the random-reference assumption (in which case they are as good or bad as any other rule); however, it is hard to imagine their usefulness while considering true program behavior as described later. Another Class-1 algorithm, called FIFO (first in, first out), has been investigated. FIFO always replaces the block having spent the longest time in memory. The strongest argument for FIFO is the fact that it is easier to step a cyclic counter than to generate a random number. As to its logical justification, the notion of locality should first be introduced.

In our primitive model, references are uniformly distributed over the range of all problem program blocks. This assumption is certainly not true for a large collection of possible programs; the number of references to memory between two successive replacements is actually high. Suppose now that, for a given program section, we let f denote the number of repetitions between two successive replacements. Now increase c to $c + \Delta$; then f also increases, say by δ. Calling c and $c + \Delta$ *localities* of the program, both starting at the same point, f and $f + \delta$ express the respective lifetimes of these localities. Now if

$$\frac{f + \delta}{f} \approx \frac{c + \Delta}{c}$$

we can assume that the primitive random assumption holds. However, if

$$\frac{f + \delta}{f} \gg \frac{c + \Delta}{c}$$

two possibilities exist: either the proportion of references to the additional Δ blocks was relatively high, or the additional δ repetitions were distributed roughly over all $c + \Delta$ blocks. Thus, in the case of inequality, we cannot assume with confidence that the references are uniformly distributed.

The idea behind FIFO is this switching from one locality to another, at which time a certain block is abandoned and another block is picked up for a new interlude of c block replacements. It is hoped that the probability of the oldest block in memory being the abandoned block exceeds $1/c$.

In Class-2 replacement algorithms, one hopes to improve re-

placement decisions by anticipating future references on the basis of previous references. We try to improve the techniques of FIFO, which selects the blocks according to their age in memory, but does not provide information about the distribution of references.

class 2 algorithms

The idea is to dynamically order the blocks in memory according to the sequence of references to them. When a replacement becomes necessary, we replace the block to which reference has not been made for the longest time. We hope that the fact that this block has not been needed during the recent past indicates that it will not be referenced in the near future. This is a significant refinement relative to FIFO, since now frequency of use rather than stay in memory is the decisive factor.

The dynamic reordering of all blocks in memory may be a costly procedure; moreover, we are not particularly interested in the entire order. Of interest, rather, is a split of blocks into two subsets: the one to which recent references have occurred and the other to which no recent reference has been made. There is a relatively easy way to facilitate the split: any time a reference is made to a block, a status bit (which we call the "P bit") is set to 1 for the particular block. For pushes, unmarked blocks (for which P is 0) are preferred. However, it may happen that the set of unmarked blocks vanishes; at this instant, all P bits are reset to 0, except the just-marked block, and the procedure starts afresh. Variation in the relative size of each subset resembles a sawtooth function of time. Because on the average there is more than one element in the unmarked set, an additional subrule is needed to pick a specific block from the set. Subrule variations can easily lead to a wide variety of replacement algorithms, all based on the idea of marking references.

Class-2 algorithms can be justified by the following reasoning. All blocks encountered in a program can be divided, very roughly, into two main groups. The first group is characterized by high-frequency usage; most of its blocks contain program loops. The second group contains blocks used with relatively low frequency (initializing programs, low-frequency data blocks, sometimes I/O or other service routines). Using a Class-2 replacement algorithm, one may hope that mostly blocks of the second group will be pushed; RAND or FIFO usually do not justify such hope.

It is convenient to introduce another status bit, the A bit, which is used to mark whether the content of a block has changed during the block's most recent stay in memory. In contrast to P bits, A bits are not resettable by the replacement algorithm. We refer to the P and A status bits in this (P,A) order; e.g., (1,0) indicates that a block has been referenced one or more times, but that the block content has not been changed.

For simulation purposes, the following Class-2 algorithms were chosen as reasonably distinct and sufficiently representative of parameter influences (block and memory sizes).

S-3: Blocks in memory are classified in subsets according to (P,A) values (0,0), (0,1), (1,0), and (1,1); this order is significant.

At replacement time, P bits are reset only if all of them are found to be 1. A block is chosen at random from the lowest order (leftmost) non-empty (P,A) subset.

AR-1: Same as S-3, but the reset occurs immediately and automatically whenever the last P bit is set to 1. This implies that the (0,0) and (0,1) subsets are never empty at the same time.

T: Same as AR-1, but instead of choosing a block at random from the relevant subset, a sequential search is started from the last-replaced block.

ML: Same as AR-1, but ignoring the A bit.

LT: Chooses the block to which no reference has been made for the longest time. No status bits are used; instead, all blocks are dynamically reordered.

class 3 algorithms

Class-3 replacement algorithms represent an extension of the Class-2 algorithms. Dynamic information is kept about blocks in external storage as well as in memory. Here, for a typical algorithm, we may refer to the one used on the Ferranti Atlas.[2] The ATLAS algorithm features a buffer block—an empty memory block that permits a pull without waiting for the corresponding push. Because the buffer leaves only $c - 1$ usable blocks in memory, the buffer may—for very low c—have a significant deteriorating influence on replacement efficiency. Only the published version of the ATLAS algorithm was simulated, and no other Class-3 algorithms were postulated.

general remarks

The algorithms described herein are meant to typify rather than exhaust the set of all algorithms that try to anticipate future references by observing summary information on past references. Some proposals, suggesting elaborate age-measuring schemes using digital counters or electrical analogs, are not treated here.

Optimal replacement algorithm

All the replacement algorithms discussed thus far attempt to minimize the number of block replacements. However, none of these algorithms can reach the actual optimum because at push time nothing is known about the subsequent block references. For an optimal replacement algorithm, which must be based on such information, the necessary complete sequence of block references can be supplied by a pre-run of the program to be used. Although this is impractical for most applications, an optimal replacement algorithm is of value for system study purposes. Such an algorithm, let us call it MIN, is described after an introduction to the underlying principles.

principles

The optimal solution can be found by storing the program's entire sequence of references and then working backward to reconstruct a minimum-replacement sequence. This is a two-pass job; moreover, an excessive number of tapes is necessary to store the sequence for a long program. Fortunately, the amount of information to be stored can be reduced. First of all, it suffices

to store reference information by block rather than by word. Furthermore, as long as the memory is not entirely filled, a pulled block can be assigned to any free location, no replacement decision is necessary, and information need not be recorded. However, once the memory becomes full, information must be collected. When another block in external storage is then referenced and must be pulled, a *decision delay* starts because no block is an obvious push candidate. However, if a block in memory is now referenced (previously defined as a *repetition*), this block should be kept in memory and is therefore temporarily disqualified as a push candidate, thus reducing the number of candidates. When $c - 1$ blocks in memory have been disqualified, uncertainty decreases to zero because only one block remains as a push candidate. Thus, we can now make the delayed push decision; we know which block should have been pushed to make room for the new pull. Together with the new pull, the $c - 1$ disqualified blocks form a *complete set* of c blocks that define a new memory state.

The above case assumes that the delayed push decision can be made before a second pull is necessary. In general, however, not enough repetitions occur between two pulls to make the decision that early. Then, the decision must be further delayed while we continue to investigate the sequence of program references as explained below. Usually, many push decisions are being delayed at the same time because each new pull requires another push. The maximum decision delay ends with the program run; usually, however, the delay terminates much earlier. If not enough repetitions have occurred by that time, the program picks blocks for replacement in a simple manner—either repeatedly from a relatively small number of blocks, or by straight-line sequencing.

MIN algorithm

The MIN algorithm, applicable to the general case of many delayed push decisions, is based on the elimination of complete sets as push candidates. Whenever a complete set exists between consecutive pulls, a new memory state has been defined and all remaining blocks are pushed. This is now explained in detail.

In processing the string of block references for a given program, pulls are numbered consecutively by MIN. Let p denote the integer that identifies the most recent pull and hold p in the *current register*. Whenever a block is referenced, its associated *index*—stored in a table and updated by MIN—is set to p. For example, if Blocks A through E are referenced and pulled, their respective index values are 1 through 5 (assuming an empty memory to start with); if Blocks B and C are now referenced again, their index values are both changed to 5, not to 6 and 7. Thus, at this point of reference, blocks 2, 3, and 5 have the same index value. This is shown in Column 5 of Table 1.

Assuming $c = 3$ for our example, we now have a complete set for a new memory state. For explanatory purposes only, the indices of pulls and repetitions (including blocks saved for later repetition) are encircled in Table 1. Going from left to right in the table, a complete set is defined whenever c encircled index

Table 1 Index settings for MIN example

		Current-register settings									
		0	1	2	3	4	5	6	7	8	9
Block	A	0	①	1	1	1*	1*	1*	⑦	7	7
	B	0	0	②	②	②	⑤	⑤	⑦	7	7
	C	0	0	0	③	③	⑤	5*	5*	5*	5*
	D	0	0	0	0	④	4*	4*	4*	4*	⑨
	E	0	0	0	0	0	⑤	5	5*	5*	5*
	F	0	0	0	0	0	0	⑥	⑥	⑥	⑨
	G	0	0	0	0	0	0	0	0	⑧	8

Assumption: c = 3
⟶ complete sets
• decision points
* pushed blocks

values appear in a column.

We say that we have reached a *decision point* when sufficient information has become available for one or more previously delayed push decisions (so that a group of blocks can be pushed). At that point, a *complete register* is increased to the highest index value of the latest complete set. In our example, we have now reached such a decision point and therefore step the complete register from 1 to 5. Any block having a lower index than that of the defined complete set is pushed and can reappear in memory only by pull. The pushes occur in ascending order with respect to the index values of the blocks. In our example, Blocks 1 and 4 should have been pushed at current-register settings 4 and 5, respectively. Pushed blocks are indicated in Table 1 by asterisks. Since pushed blocks are not in memory, it is now apparent that the table agrees with the actual memory state in that actually no more than c blocks are in memory at any one time.

In the general case, a complete set frequently exists before the decision point has been reached. However, this is not apparent at that time, and the set cannot be defined until we have arrived at the decision point. This is illustrated in Columns 6 through 9 of Table 1. Here, we do not reach the decision point before the reference to Block F in column 9, although a complete set is then found in Column 7. In this case, the complete register is stepped from 5 to 7. If several complete sets are defined at any one decision point, only the latest set is of interest. This happened at the first decision point of our example.

MIN process

The actual MIN process (for $c > 1$) is now described. Initially, all block index values and the current register are reset, and the complete counter is set to 1. Note that the actual process keeps only the current index value for each block; thus, for our

example, the table would actually show only the latest of the columns in Table 1.

Step 1. Pick the next block reference and check the index of the corresponding block in the table. If the block index is lower than the complete register, increase the current register by one, set the block's index equal to the current register and go back to the beginning of Step 1. If the index is equal to the current register, go back to the beginning of Step 1. If the index is lower than the current register but not lower than the complete register, set the index equal to the current register, set the temporary register equal to the current register, reset the counter to zero and go to Step 2.

Step 2. Add to the counter the number of blocks having an index equal to the temporary register. If the content of the counter is equal to c, or if the temporary register is equal to the complete register, set the complete register equal to the temporary register and go back to Step 1. If the content of the counter is less than c, decrease the content of the counter by 1, decrease the temporary register by 1 and go back to the beginning of Step 2.

After the last reference, the current register shows the minimum number of pulls required, k. For $s > c$, the number of pushes is $k - c$, discounting c inputs for the initial load of the memory. For $s \leq c$, no pushes occur.

special version

It is possible for MIN to mark a block in memory as "active" whenever its information content has been changed In this application a memory block selected for replacement is pushed only if active; otherwise it is simply overwritten.

In contrast to the previously described regular MIN, which minimizes only the number of pulls, a special version of MIN minimizes the *sum* of all pulls and pushes.[3] Since in some rare cases, the elimination of one or more pushes by overwriting may later necessitate an extra pull, the number of actual pulls may slightly exceed that of the regular MIN.

Both versions of the MIN algorithms are also capable of defining the sequence of blocks to be pushed. This would be done during the processing of the program's reference sequence. Whenever the complete register is increased, all blocks having lower index values than this register are collected in ascending order and form the sequence of pushes. In other words, MIN is now a compression box with the reference sequence as input and a replacement sequence as output.

MIN can be used for general system studies and specifically for an investigation of the heuristic replacement algorithms presented earlier. We define replacement *efficiency* as the ratio of MIN pulls to the pulls generated by a particular algorithm. Therefore, the efficiency of any algorithm is non-zero and never exceeds 1—the efficiency of MIN.

System simulation

It is obvious that the behavior (running time, etc.) of a VSC machine depends upon the problem programs using it. Conversely, a program run is influenced by the VSC. Although VSC machines and programs specifically written for them are presently scarce, system behavior can be studied by simulation.

Given an IBM 7094 and a specific problem program written for it, we assume for simulation purposes that the memory is too small for the program's requirements. This means that whenever the program needs information that is by assumption not in memory but in "external" storage, VSC activity is necessary. Each simulated block replacement is counted as part of the sequence pattern we are looking for. The counting is continued until all memory references have been considered. The final count gives the number of block replacements necessary to run the specific program in the hypothetical machine. Since usage of different replacement algorithms stimulates different sequence counts, the number of replacements is influenced by the algorithm used. By varying both design parameters (block size and memory size) as well as the replacement algorithm, a three-dimensional response surface can be constructed.

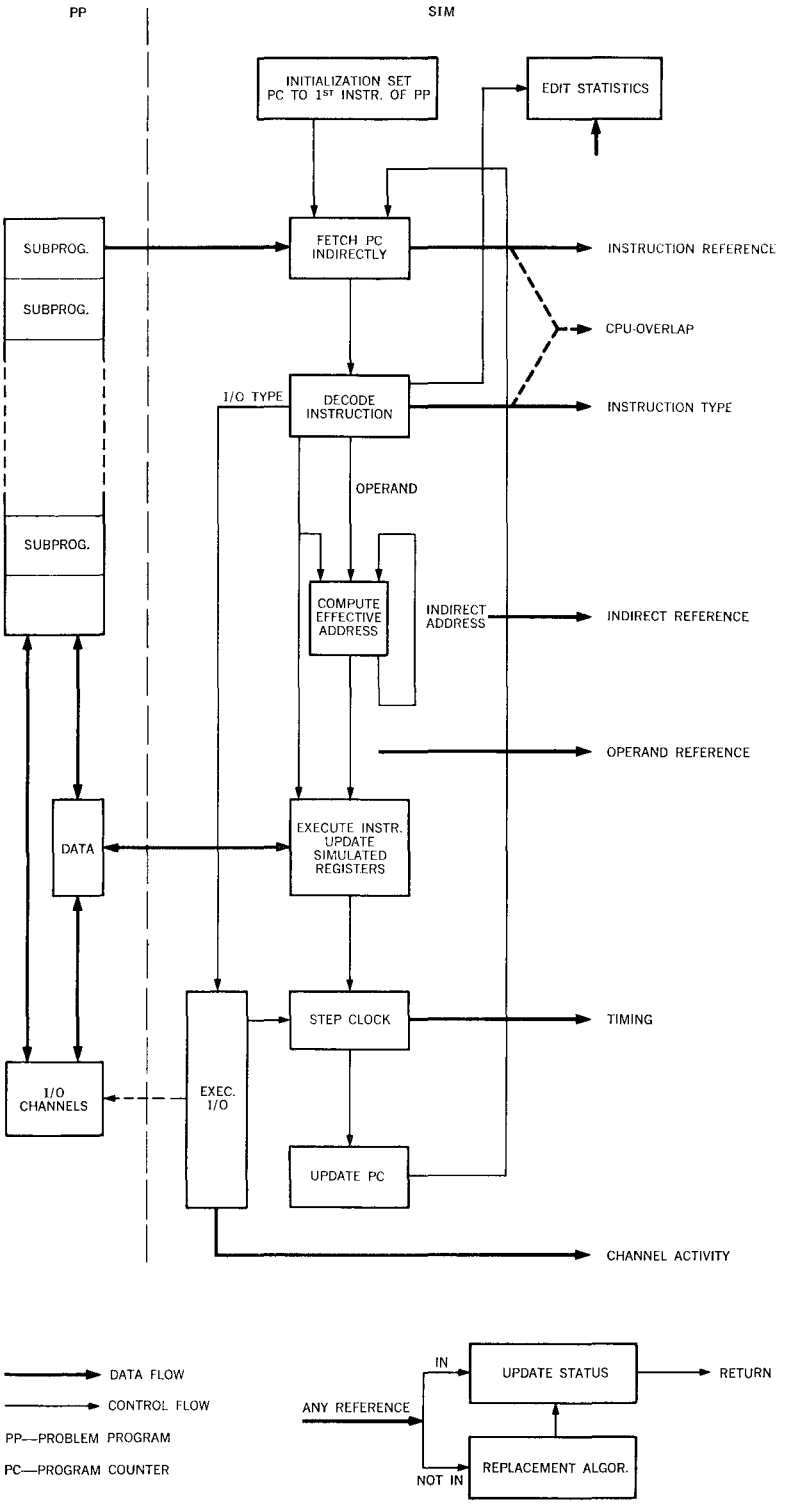
The tool for this simulation work is a family of programs called SIM (Statistical Interpretive Monitor).[4] SIM is capable of executing 7090/94 object programs by handling the latter as data and reproducing a program's sequence of references. This sequence, in turn, is an input to a replacement algorithm that determines the block replacement sequence.

The simulation procedure starts with decoding of the first executable instruction of the object program to be tested. A CONVERT instruction is used to transfer control to the proper section of SIM which—depending upon the word being tested—changes the contents of the simulated registers, computes the virtual address, or executes a load/store. A program counter plays the role of an instruction counter for the problem program being tested. A simulated branch, for example, changes the program counter. By defining some artificial constraints on the sequence of instruction and operand references, it is easy to keep track of different events, because SIM has control and can enter statistics-gathering routines at predefined points. One SIM program examines at each reference the location of the word referenced—whether already in memory or still in external storage. For non-trivial results, the simulated memory should of course be smaller than that of the machine being used. Figure 4 shows a block diagram of SIM.

Simulation results

The following is a summary and evaluation of SIM run outputs. Sample programs for simulation were picked at random; some of them were hand-coded, others written in FORTRAN. Almost all

Figure 4 Non-detailed block diagram of SIM



of them are sufficiently long to justify confidence in the significance and applicability of their respective results. A brief description of the sample programs is listed in the Appendix. As in any simulation, the number of samples represents a compromise between statistical confidence and study costs; the results are reasonably uniform and can therefore be considered satisfactory.

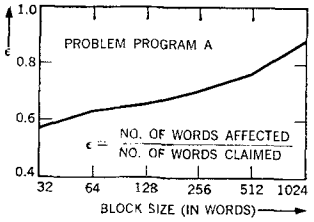
The simulation results contain a large amount of useful information. More evaluation and plotting is possible, and future behavior studies could be based on the results presented here. The following figures and tables represent a high degree of data compression; they show only the most important results and are used as a basis for the following evaluations.

memory space saving

Depending upon the associated data set, some program paths and buffer areas are bypassed and never used in a particular run; the program could produce the same results by loading only the information actually used. Provided that the change in instruction sequencing and data referencing is taken care of, the program could run in a smaller memory than the one originally claimed. This is true for vsc which pulls blocks only if and when needed. In addition to the words actually used, these blocks include, of course, unused words that happen to be in the blocks. Since these words—whether actually used or not—are affected by pulls and use up memory space, we call them *affected* words. In contrast, *unaffected* words are in blocks not pulled during a particular program run. In similar manner, we speak about unaffected information, program areas, and data areas.

Since vsc brings only affected words into memory, this means a saving of memory space in comparison to conventional computers, which must fit the entire program into memory. The tendency observed in the vsc results of sample program A, shown in Figure 5, has been found typical for any problem program. It shows that the number of words affected during a particular program run decreases with decreasing block size. The use of medium block sizes (e.g., 256 words) generally results in a 20 to 30 percent storage saving.

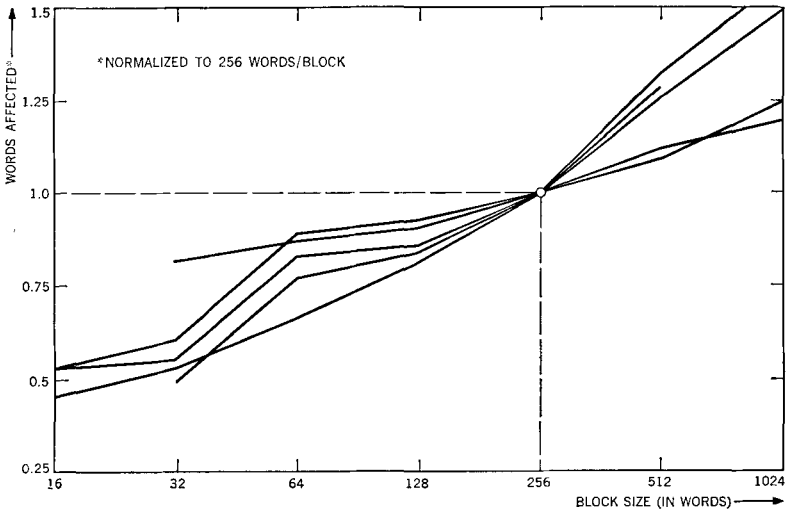
Figure 5 Typical VSC memory space saving



Although the main objective of vsc is convenient addressing for oversized programs, the saving of memory space is an important by-product. This by-product is best illustrated by a program that exceeds memory space and therefore can be run on a conventional computer only with the help of programmer-initiated techniques. On a vsc, automatic block exchange is expected during the run between memory and external storage. However, once initially loaded, the program may not need any replacement if the number of words affected does not exceed memory space. Neglecting vsc mapping time, the program behaves in this case as if on a conventional machine. Block linkages are supplied by the mapping device and the blocks in memory may be in any order. This can ease multiprogramming since the space allocated to a given program need not be contiguous.

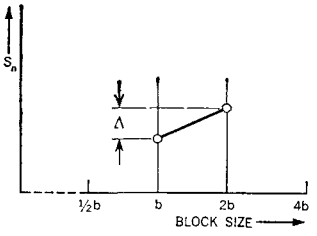
Figure 6 shows the memory space requirements of several

Figure 6 VSC memory space saving for five unrelated programs



unrelated programs normalized at 256-word block size. The tendencies are reasonably consistent. On the average, halving the block size leads to about ten to fifteen percent space saving. Some programs behave even more abruptly, at least for certain block sizes, because program or buffer areas are unaffected (as defined earlier) in the order of the particular block size considered; in comparison, when doubling the block size, large unaffected areas become affected without being used. In general, in the arbitrary locality of Figure 7, let Δ represent the derivative $ds_n/d(\log_2 b)$ of the function $s_n = f(\log_2 b)$, where b is the block size, s_n is the total number of affected words in the program, and Δ is the difference between consecutive s_n values. If, at a given value of b, Δ is negligible, the unaffected areas scattered along the program are—in general—smaller than b; thus, for an increase in block size to $2b$, only small unaffected areas are absorbed by already affected blocks (and also become affected without being used). For a large Δ, the unaffected areas scattered along the program are generally at least as large as b. Thus, for an increase in block size to $2b$, large unaffected areas become affected without being used.

Figure 7 Interpretation of space saving

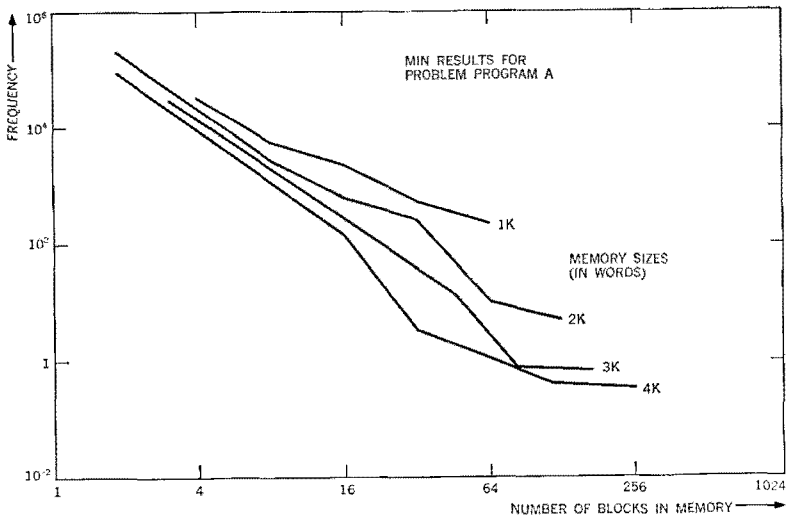


To avoid ambiguity, programs were reloaded in such a manner that the lowest address word of the program is the first word of a block. Other conventions are also conceivable; for example, each subprogram, routine, or independent data area could start in a new block.

Figure 8 shows the effect of block structure on a typical problem program. Each curve representing a given memory size as parameter, the ordinate gives the frequency of completely changing the contents of the memory, and the abscissa reflects the fineness of block structure (a value 64 means, for example, that the memory

replacement frequency

Figure 8 Frequency of full memory load



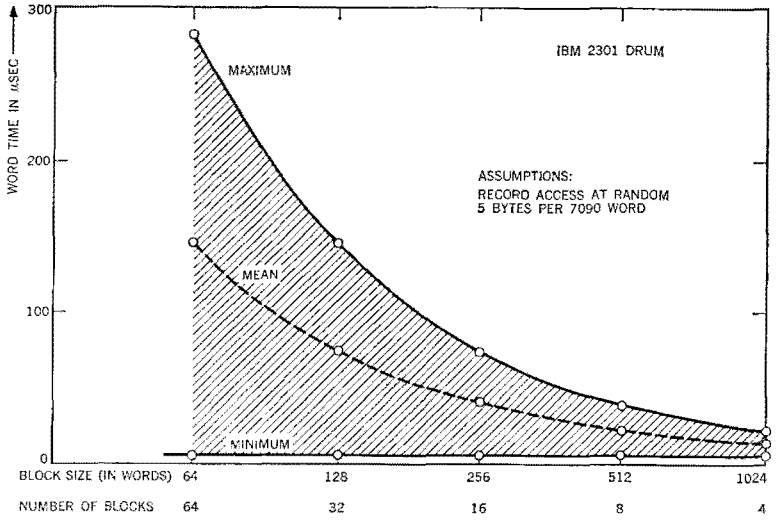
contains 64 blocks of equal size). The frequency is always proportional to the number of replacements in the problem program. Considering the logarithmic scale, it is striking how abruptly the number of replacements is reduced by decreasing the block size. The curves of different memory sizes differ little; moreover, their slopes hardly differ at all. A linear approximation based on half a dozen unrelated programs suggests that the number of block replacements is proportional to a/c^k where a varies considerably with the individual program and is also a function of the memory size, and k (although a function of the program) is mostly in the interval $1.5 < k < 2.5$.

The considerable improvement accomplished by finer block structuring (relatively fewer replacements when more, but smaller, blocks are used) can only partially be due to the smaller affected program area itself; this is evident from comparing Figure 8 with Figures 5 and 6. The predominant factor is that more useful information can be stored in memory, thus is directly accessible, resulting in a new, extended locality of the program. Nevertheless, vsc system efficiency is not necessarily optimized with minimum block size because the number of entries in the mapping table may eventually exceed technological feasibilities, and the effective data rate of external storage depends upon block size. The latency effect of a drum is evident from the numerical example of Figure 9, which gives data transmission rates for different block sizes on a drum. If disks are used, seek time must also be considered, and very small block sizes become prohibitive.

other replacement statistics

The work with the MIN algorithm suggests some statistical measures that may possess significance for future program-behavior studies. Recalling the previous description of MIN, let us define three of the derived statistics: delay in replacement decision

Figure 9 Latency effect of a drum



(Δ_1), average number of replacements between successive complete sets as defined earlier (Δ_2), and cumulative frequency of decision points.

The size of Δ_2 is affected by the distribution of repetitive references to memory blocks. If the repetitions between replacements extend over the entire memory, Δ_2 consists of only one block. Another interpretation of Δ_2 views it as a measure of uncertainty; on the average, Δ_2 is the number of replacement candidate blocks to which no references have been made between successive replacements. Hence $c - \Delta_2$ is the local need of the program before it switches to another locality by causing a new pull.

Figure 10 shows typical Δ_2 values for a single problem program. The tendency is for Δ_2 to decrease as block size increases, because larger areas are affected by repetition references and the degree of uncertainty is reduced. To a degree, increases in memory size have a reverse effect. Starting from a small memory, Δ_2 first increases; but a further increase in memory size changes this tendency. A possible explanation for this is that the program has blocks with high repetition frequency, and it changes to another locality by using some transitional blocks. With an increase in memory size, the increased number of transitional blocks generally leads to a higher degree of uncertainty. However, beyond a certain limit, the memory becomes large enough to reduce the vsc traffic to such a rate that time between replacements becomes very long and repetitive references cover a large area, hence reducing the degree of uncertainty.

Figure 11 shows values of $f = (\Delta_1 + \Delta_2)/(s - c)$ for the same problem program. Clearly, $f \leq 1$ since the limiting case exists when all $s - c$ blocks in external storage have been collected

Figure 10 Typical Δ_2 values for MIN

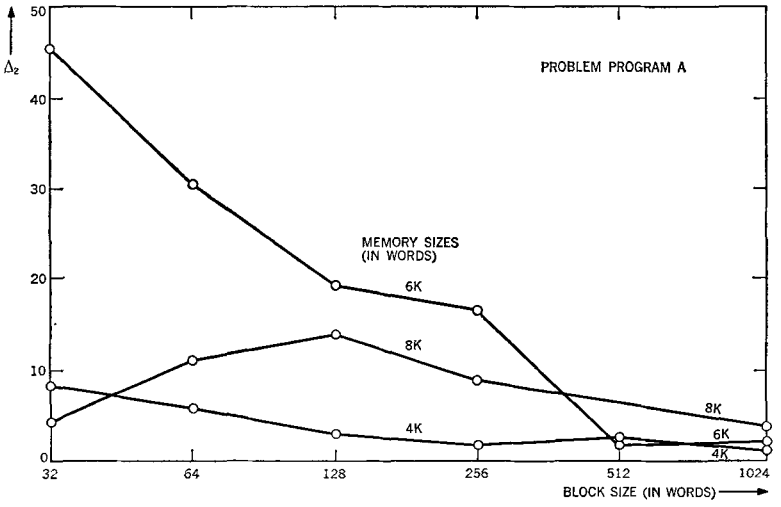
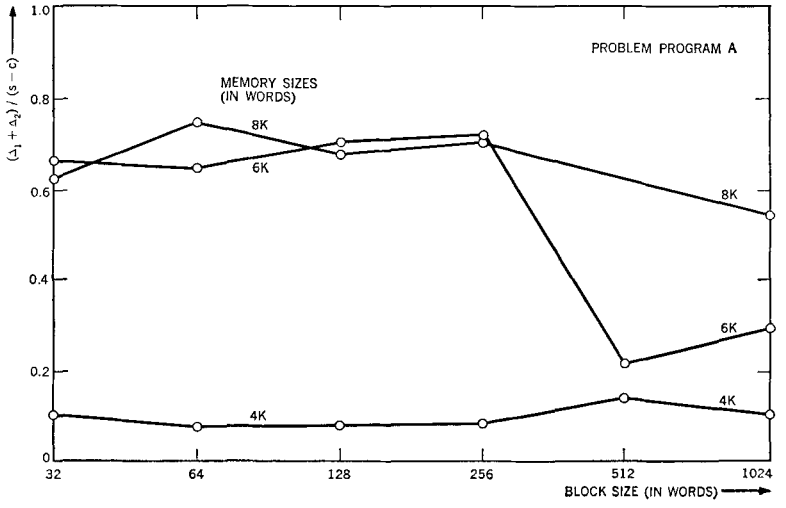


Figure 11 Normalized Δ values for MIN

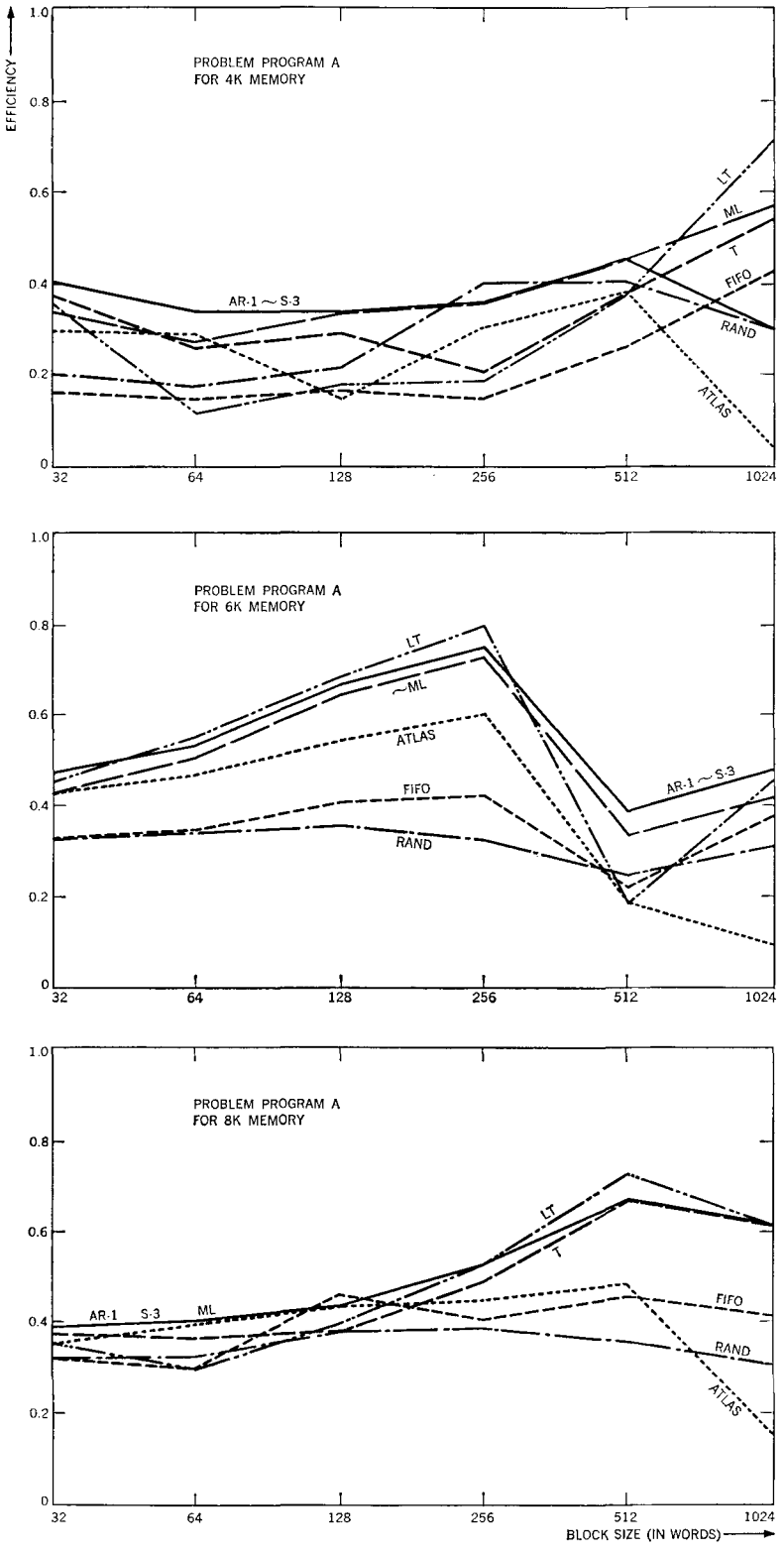


as needed but still no decision is possible. After this point, only repetitive references can be used to narrow the degree of uncertainty, and f is the percentage of non-memory blocks about which information should be kept. Again, due to the short replacement interval, the amount of information to be kept about blocks is somewhat less for small memories, but does not change very much with block size.

efficiency

Figure 12 displays efficiencies of all simulated algorithms for a typical problem program for three different memory sizes, the efficiency of an algorithm being related to MIN results.

Figure 12 Efficiency of algorithms for three memory sizes



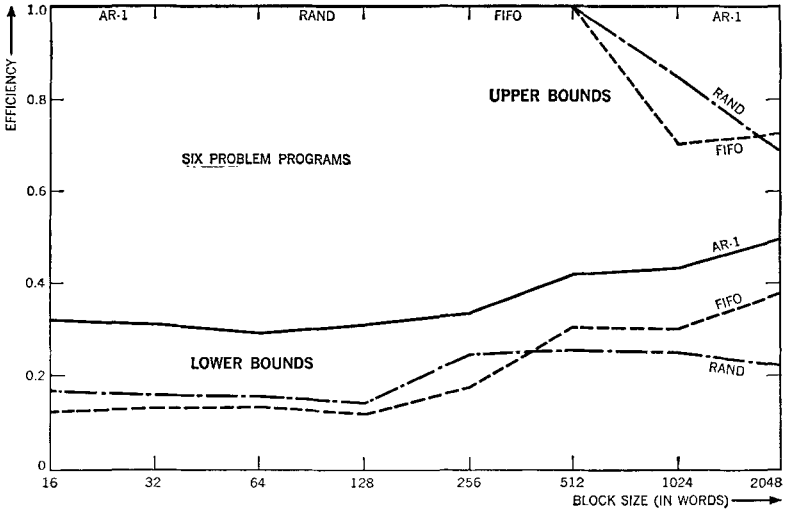
First examining the two Class-1 algorithms FIFO and RAND, the results show that RAND is the most uniform and does not vary much with the two independent parameters (block size and memory size). RAND efficiencies are about 30 percent, and it is interesting to observe that this efficiency corresponds to those s/c values in Figure 3 that are less than 2. This gives some evidence that references are not randomly distributed over all possible blocks but rather over some localities. For this particular example of Figure 12, s/c is actually slightly above 3 in the 4K memory case.

Among the Class-2 algorithms, AR-1 and S-3 exhibit the best overall efficiency and consistency. The difference between these algorithms becomes significant for a memory with a few large blocks, in which case AR-1 is consistently superior to S-3. In this situation, the P bits are usually set for S-3, and only the A bits are informative. With large blocks, the A bits as well are too often set, and the algorithm degenerates to RAND.

The other Class-2 algorithms generally perform better than those of Class 1, but not as well as AR-1. The only non-status-bit algorithm (LT) is the least consistent; although LT is the best for certain parameter values, its overall efficiency is unpredictable. Its efficiency is good for very few memory blocks, where the degree of uncertainty is already low, and the block to which no reference occurred for the longest time is frequently the best replacement candidate.

ATLAS, the only Class-3 algorithm tested, typically does slightly better than RAND or FIFO. For very large blocks, however, its performance is poor because the memory always contains an unused reserve block.

Figure 13 Efficiency bounds of algorithms for a wide range of memory sizes



After testing the various replacement algorithms, RAND, FIFO, and AR-1 were selected as having efficiencies that could justify implementation (subject to other technical constraints). The first two have the advantage that updating of memory status involves only block presence or absence. FIFO is especially simple in that even a (pseudo) random-number generator is unnecessary. As pointed out earlier, AR-1 needs such a generator because it picks at random from the subset defined by status bits.

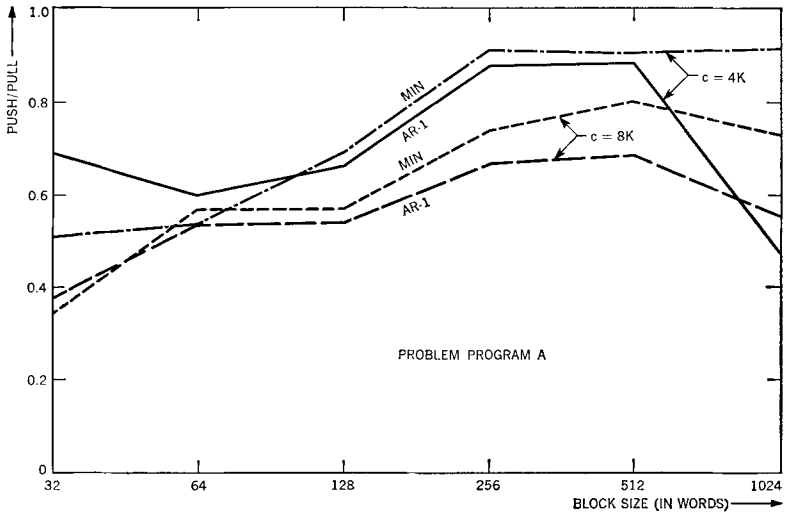
Figure 13 exhibits efficiency bounds of the selected algorithms for a wide range of memory sizes. The results were obtained using six problem programs. The superiority of AR-1 is clear; its efficiency is at least 30 percent in all cases. FIFO and RAND are close to each other; FIFO is preferable because of its simpler implementation; RAND is interesting because of its simple mathematical formulation.

Computation of the efficiencies refers to the number of pulls only, and—as mentioned before—each pull may or may not be coupled with a push. If A bits are used, it is useful to recall the special version of MIN that minimizes the sum of pulls and pushes and to define an adjusted efficiency as the ratio of this sum to that generated by another algorithm using A bits.

push/pull ratio

Figure 14 shows the average ratio of pushes to pulls for AR-1 and the regular MIN (using A bits) on the same problem program. The relative number of pushes increases for larger blocks, because a change in a single word causes the entire block to become active, and the probability of at least one change per block is higher for larger blocks. Note that the push/pull ratio of AR-1 is higher than that of MIN for small blocks, and lower for large blocks. The latter case can be explained by AR-1's property to

Figure 14 Typical VSC push/pull ratios



prefer a nonactive block for replacement. With only a few blocks in memory, AR-1's saving in pushes can be significant. This is also the explanation for the interesting fact that sometimes the combined push and pull efficiency of AR-1 for very large blocks exceeded unity because MIN minimizes only the pulls. Reruns with the special version of MIN showed that the minimum of the sum is such that the adjusted efficiency actually does not exceed unity. The fact that AR-1 produces relatively more pushes than MIN for very small block sizes suggests that the use of A bits is not only irrelevant in this case but might even be detrimental to the efficiency.

Summary comment

This paper groups replacement algorithms into three major classes according to their use of relevant information, and develops an optimal replacement algorithm for system study purposes. Simulation of typical programs on a postulated virtual-storage computer reveals the behavior of several algorithms for varying block and memory sizes.

In most cases, the simulated algorithms generated only two to three times as many replacements as the theoretical minimum. The simulated system was found to be sensitive to changes in block size and memory size. From the viewpoint of replacement efficiency (which disregards the operational effects of external-storage access times), small block sizes appear to be preferable. In spite of the obvious advantages of a large memory, the algorithms achieved reasonable efficiency even for large program-to-memory ratios.

The results of the study suggest that a good algorithm is one that strikes a balance between the simplicity of randomness and the complexity inherent in cumulative information. In some cases, too much reliance on cumulative information actually resulted in lower efficiency.

The virtual-storage concept appears to be of special relevance to time-sharing/multiprogramming environments. Here, it enables the user to deal with a larger and private virtual storage of his own, even though the computer's memory is shared with other users. Because of the capability of loading small program parts (blocks), the memory can be densely packed with currently needed information of unrelated programs.

ACKNOWLEDGMENT

The author wishes to express his thanks to R. A. Nelson who initiated and guided the study described in this paper, and to Dr. D. Sayre for his helpful suggestions in preparation of the manuscript.

CITED REFERENCE AND FOOTNOTES

1. The term *memory*, as used in this paper, refers only to the direct-access main storage of a computer.
2. T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One level storage system," *IRE Transactions on Electronic Computers* **EC-11**, No. 2, 223–235 (1962).
3. An algorithm applicable to this problem is presented by L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd in "Index register allocation," *ACM Journal* **13**, No. 1, 43–61 (January 1966).
4. SIM, an experimental program package, was developed by R. A. Nelson of the IBM Research Division.

Appendix—List of sample programs

	Description	*Number of executed instructions*	*Program size (without* COMMON, *system, and library routines)*	*Remarks*
A	Integer Programming (of the class of linear programming)	11 000 000	3.2K	FORTRAN
B	SIMC (a version of SIM, interpretively executing sample program A)	325 000	6.6K	FAP
C	Pre-processor (for FORTRAN language programs)	59 000	12.5K	FAP
D	Logical manipulations on bits (study for logical connectives)	8 500 000	18.0K	FAP
E	Arithmetic translator (an experimental compiler module)	1 900 000	16.5K	FAP
F	Numerical integration	20 000 000	1.8K	FORTRAN
G	WISP (list processor)	≈15 000 000	≈28.0K	
H	SNOBOL	≈1 300 000	≈26.0K	