A SET OF STRATEGY-INDEPENDENT RESTRUCTURING ALGORITHMS

by

M. Kobayashi

Memorandum No. ERL-M77/5

21 January 1977

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A SET OF STRATEGY-INDEPENDENT RESTRUCTURING ALGORITHMS*

Makoto Kobayashi

Computer Science Division
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California
Berkeley, California 94720
U.S.A.

Summary

This paper proposes a set of new program restructuring algorithms which can be used to reorganize programs so as to increase their performance under various memory management strategies in virtual-memory computer systems. The new algorithms are based on a recently proposed program behavior model called the bounded locality intervals model, which allows us to give a precise definition of the localities of a program. The paging activities of a program restructured with the new algorithms under working-set and global LRU-like memory management strategies are simulated to evaluate the new algorithms. Some of them are shown to be almost as good as the strategy-oriented algorithms, which have been reported to be very successful.

Keywords: virtual memory, program restructuring, performance improvement, working set, page survival index, bounded locality intervals

*The research reported here has been supported by the National Science Foundation under grant DCR74-18375.

Introduction

The overall system performance of a virtual memory computer system strongly depends on the performance of the memory subsystem, which consists of physical memory devices and the system programs in charge of the management of the information stored on the devices.[1] Since the concept of virtual memory has emerged, memory management policies, particularly page replacement strategies, have been paid much attention and have been extensively studied, both to reduce software overhead in implementing virtual memory systems and to achieve reasonable performance over a wide spectrum of programs.

Another relatively new approach to performance improvement is to rearrange programs so that they can be efficiently executed in virtual memory systems. This is especially true for complicated and large programs, e.g., system programs, compilers and large simulation programs, etc., and it is often the case that these programs actually behave poorly. This second approach is called program restructuring.

If we knew, or were able to estimate, the performance improvement that would be obtained by restructuring and the frequency of execution, we could decide whether a given program should be restructured or not. Unfortunately, the improvements that restructuring will afford are generally very difficult to predict reliably. System programs and compilers, as well as other programs in certain installations, however, have such a high frequency of execution that restructuring is likely to be advantageous even if the performance improvement for each execution is small, since the cost of restructuring will be paid off quite soon.

The algorithms used to restructure programs can be divided into two categories. Restructuring algorithms in the first category take advantage

of the knowledge of memory management strategies, whereas those in the second category are independent of memory management strategies. The restructuring algorithms in the first category are expected to be superior to those in the second category due to the fact that they use additional information. Several experiments have confirmed this intuitive result.[2,3] However, programs should be exclusively restructured for each virtual memory system in which they are executed. Thus, in certain cases, for example when little is known about memory management strategies, or when a program is to be executed on a number of different systems, it would be desirable to use restructuring algorithms in the second category, provided that their efficiency were comparable to that of the algorithms in the first category.

This paper presents and evaluates a set of new restructuring algorithms of the type we have just described. Though being strategy-independent, these algorithms produce restructured programs which in general perform substantially better than those produced by the previously proposed algorithms in the second category under a variety of memory management strategies.

In the next section, two classes of widely used memory management strategies will be discussed. They correspond to the environments in which the performance of the restructured programs will be evaluated. In the following section, several previously proposed restructuring algorithms and a set of new restructuring algorithms will be introduced, following a description of a general restructuring procedure. Then, the experimental results produced by the new restructuring algorithms in the two different environments will be given.

Memory Management Strategies

In multiprogrammed virtual memory systems, an executable program does not usually fit entirely into main memory. Therefore, a copy of the program is stored in fast secondary memory before execution. Segments of a program containing instructions or data to be referenced are transferred automatically from secondary memory to main memory or vice versa according to a memory management strategy which is implemented in a module of the operating system. The memory management strategy determines when and which segment of a program is to be transferred to what part of the main memory, and when and which segment resident in the main memory is to be transferred back to the secondary memory to make room for an incoming segment. In the sequel we assume that the segments of a program are of equal size and the main memory is also divided into frames of the same size as the program segments. The fixed-sized segments are called pages. We also assume that the pages needed are transferred into main memory on demand, as in most virtual memory systems. The pages to be transferred back to secondary memory are determined by the so-called page replacement algorithm, which is a component of the memory management strategy. There are two groups of widely used page replacement algorithms: global LRU-like page replacement algorithms and working-set-like strategies. Under a global LRU-like page replacement algorithm, a page which has not been referenced for a relatively long period of time is likely to be replaced. This algorithm is simple to implement, for example, by maintaining a table of reference bits, each of which corresponds to a page frame in main memory, and a pointer to an entry of the table. When a page resident in the main memory is referenced, the reference bit associated with the page frame is turned on. The pointer circles around the reference bit table so that, when it is necessary to choose a page to be replaced, the

pointer proceeds turning off the reference bits until it encounters a reference bit off. The page associated with that reference bit is the one to be replaced. Algorithms of this type are implemented, for example, in MULTICS[4] and OS/VS1.[5]

The paging activities of the programs executed concurrently under a global LRU-like page replacement strategy interfere with each other. For instance, programs which require many pages in a short time interval tend to steal page frames from other programs resident in the main memory. The determination of paging activities in this situation is too complicated to be performed by analytical methods. Therefore, simulation is to be used. However, the simulation of paging activities under a global LRU-like strategy would require that several programs were fed into a virtual memory simulator simultaneously, and it is very difficult to choose and procure enough programs whose combinations constitute reasonable and representative job mixes in a virtual memory system. Fortunately, a simulation model of paging activities under global LRU-like strategies in time-sharing environments has been proposed and successfully verified with the CP-67 system.[6] This model is based on the following observations. A program loses its pages resident in the main memory only while it is in the interrupted state, that is, when its execution is suspended. A program interruption (simply called an interruption in the sequel) occurs when the program issues a page request or an I/O request, or when its time quantum expires. When the system paging activity is low and the demand for page frames is small, the unreferenced pages of a program can survive (that is, remain in main memory) a relatively larger number of interruptions. On the other hand, when the system paging activity is high, the unreferenced pages will be lost from a program after relatively few interruptions. The effect of the overall system activity on

a single program is thus characterized by a single parameter called the page survival index (PSI), which is defined as the number of interruptions that an unreferenced page resident in the main memory can survive, that is, stay in the main memory. The set of pages of a program which reside in the main memory at time t under the PSI model with a particular value ψ of PSI is called the resident set at a given virtual time t with $PSI = \psi$, and is denoted by $R(t,\psi)$. The resident set size is defined as the number of pages in the resident set.

The working set of a program at a given virtual time t, $W(t,\tau)$, is dynamically defined as the set of pages referenced by the program during the time interval $(t-\tau,t]$, where τ is a parameter called window size. The working set size is defined as the number of pages belonging to the working set. The working-set memory management strategy ensures that the working set of a program at any given time t, $W(t,\tau)$, is maintained in the main memory. The working set is, in most cases, not updated continuously at each memory reference, but only at sampling times, so as to reduce the software overhead. Working-set-like memory management strategies are implemented, for instance, in MANIAC II,[7] VMOS,[8] and OS/VS2 Release 2.[9]

The PSI and working-set memory management strategies have been used in our experiments as the environments in which the execution of programs was simulated and their paging activities were measured.

Throughput and response time for time-sharing systems or turnaround time for batch systems are ideal performance indices for the evaluation of virtual memory computer systems performance. The evaluation of these system-wide indices requires a relatively complex analytical[10] or simulation[11] model. However, for the sake of simplicity, we decided to use intermediate performance indices like the number of page faults generated by a program

and its memory demand, which will be defined below. The number of page faults (NPFs) is the total number of pages not found in the main memory when needed by the CPU during a program's execution. If the number of page faults of a program is reduced, the execution of the program finishes in a shorter time and less main memory is unnecessarily wasted during page transfers. The memory demand is defined as the average resident set size in a global LRU-like environment or the average working set size in a working set environment over the total virtual time used for the program's execution. If programs occupy less main memory space on the average, then the degree of multiprogramming can be increased and therefore the throughput should be expected to increase.

Program Restructuring

A program to be restructured is divided into relocatable segments (called blocks in the sequel) such that the relative addresses of instructions or data within a block are not modified but the ordering of blocks can be changed in the virtual address space. COMMONs in FORTRAN and BLOCKs in ALGOL can be blocks. Then the block reference string, which is a series of block identifiers ordered by the time of reference, is obtained. There are two methods to obtain block reference strings. The first method consists of instrumenting a program before its execution so that the instrumented program generates and records the block reference string on a file during its execution. The second method is to execute, under control of a software interpreter or instruction tracer, an uninstrumented program interpretively instruction by instruction, and to record on a file the addresses referenced. The block reference string is obtained from the address reference string by mapping addresses into blocks. A restructuring graph

of a program is obtained by applying a restructuring algorithm to the block reference string which was recorded on a file. Alternatively, the graph may be generated directly from the program during its execution. A restructuring graph is non-directed. The nodes of the graph represent blocks of the program and the edge labels represent the strength of connectivity between the corresponding nodes. By applying a clustering algorithm to the restructuring graph, nodes, i.e. blocks, are grouped together so that the total strength of connectivity among blocks which belong to different groups is minimal. Finally, the blocks of the program are reordered in the virtual address space so as to obtain the groups suggested by the clustering algorithm.

The restructuring algorithms may be divided into two categories: strategy-oriented algorithms and strategy-independent algorithms. A strategy-oriented algorithm assumes a particular target memory management strategy and takes advantage of the knowledge of that strategy. The goal is to obtain a restructured program which can be executed efficiently in a virtual memory system controlled just by the target memory management strategy. Some successful strategy-oriented restructuring algorithms are summarized in the sequel.

Before proceeding to describe these restructuring algorithms, we will introduce some notations and assumptions. We assume that references are made at the discrete time instants $1,2,3,\ldots,t,\ldots$ to the blocks $b(1),b(2),b(3),\ldots,b(t),\ldots$, respectively. In this section, blocks are the units of memory management. Therefore, the working set and the resident set are defined in terms of blocks rather than of pages. The working set at time t with window size τ is denoted by $W_b(t,\tau)$, and the resident set with the PSI value ψ is denoted by $R_b(t,\psi)$. A critical reference

is defined as a reference to a block which is not resident in the main memory, which causes a block fault.

The Critical Working Set (CWS) restructuring algorithm[2] is intended to decrease the number of page faults generated by a program which is executed in a virtual memory system with the working-set memory management strategy. CWS increments by one, whenever a block fault occurs, the labels of the edges (b_i,b_f), where b_f is the block the reference to which causes a block fault at time t, and b_i is any block which belongs to $W_b(t,\tau)$. When the program is executed under the working set strategy after it has been restructured by using the CWS algorithm, it is expected to generate fewer page faults than the unrestructured original program, because a number of former critical references will not be critical anymore.

The Minimum Working Set (MWS) restructuring algorithm[12] has also been suggested for the working set strategy in order to decrease the working set size. The MWS algorithm increments by one the labels of all the edges (b_i,b_j), where both blocks b_i and b_j are members of $W_b(t,\tau)$ at sampling time t.

The Critical PSI (CPSI) restructuring algorithm, which is proposed and described in detail in a companion paper,[13] intends to decrease the number of page faults generated by programs which are executed in global LRU-like environments. The basic concepts behind the CPSI algorithm are the same as those behind the CWS algorithm. That is, if a block b_f referenced at time $t+1$ is not a member of $R_b(t,\psi)$, then the CPSI algorithm increments the label of all the edges (b_i,b_f), where b_i is a member of $R(t,\psi)$.

The Minimum PSI (MPSI) restructuring algorithm, also proposed and described in the companion paper,[13] tries to minimize the resident-set size by incrementing by one the labels of all the edges (b_i,b_j), where b_i and b_j are members of $R_b(t,\psi)$.

The strategy-independent restructuring algorithms do not require any specific information about the memory management strategies under which programs to be restructured are executed. The Nearness Method (NM) is a classic strategy-independent restructuring algorithm.[14] The NM just increments the edge labels of block pairs referenced consecutively. That is, NM increments the label of the edge (b(i),b(i+1)) for i = 1,2,3,...,t,... .

Programs almost always exhibit the so-called locality of reference in the virtual address space and in virtual time.[15] This means that a proper subset of a program is accessed for relatively long periods of virtual time and the items in the subset are located relatively close to each other. Virtual memory computer systems take advantage of this property of programs. Memory management strategies, for instance global LRU-like strategies and working-set-like strategies, assume that programs exhibit some locality of reference and can only achieve reasonable levels of performance if this assumption is satisfied. Thus a restructuring algorithm which enhances a program's locality will improve the efficiency of program execution much more than the NM, which makes little use of the locality.

Bounded Locality Intervals (BLIs) have been recently proposed by Madison and Batson[16] to model dynamic program behavior from the locality's point of view. Some of the nice features of Madison and Batson's model are that it contains no parameters, so that localities are determined naturally, and that it defines hierarchies of localities. A locality in this context is a proper subset of a program's blocks defined by the BLI model. A set of k distinct blocks of a program is said to be "formed" when it consists of the k most recently referenced blocks of the program, and is said to be "terminated" when a block not belonging to the set is referenced for the first time after the set is formed. An activity set at time t is defined

as any set of blocks all members of which have been re-referenced since the set was formed. Activity sets can be conveniently identified by the Extended Least Recently Used (LRU) stack defined by Madison and Batson.[16] The Extended LRU stack at time t consists of the following three vectors:

$$L(t) = (L_1(t), L_2(t), \ldots, L_n(t)) ;$$
$$\alpha(t) = (\alpha_1(t), \alpha_2(t), \ldots, \alpha_n(t)) ;$$
$$T(t) = (T_1(t), T_2(t), \ldots, T_n(t)) ;$$

where n is the number of blocks in the program, $L(t)$ is the normal LRU stack and $L_i(t)$ is the i-th most recently referenced block at time t, $\alpha_i(t)$ is the time at which the block in the i-th position of $L(t)$ was last referenced, and $T_i(t)$ is the most recent time at which a reference was made to a stack position greater than i. Therefore, $T_i(t)$ is the formation time of the set $S_i = \{L_1(t), L_2(t), \ldots, L_i(t)\}$. If at time $t+1$ a reference is made to the block $L_i(t)$, then the Extended LRU stack is updated as follows:

$$L_1(t+1) = L_i(t) ; \quad \alpha_1(t+1) = t+1 ; \quad T_i(t+1) = T_i(t) ;$$

for $1 \leq j \leq i-1$

$$L_{j+1}(t+1) = L_j(t) ; \quad \alpha_{j+1}(t+1) = \alpha_j(t) ; \quad T_j(t+1) = t+1 ;$$

for $i+1 \leq k \leq n$

$$L_k(t+1) = L_k(t) ; \quad \alpha_k(t+1) = \alpha_k(t) ; \quad T_k(t+1) = T_k(t) .$$

An <u>activity set</u> at time t, $A_i(t)$, is defined as any set $S_i(t)$ for which $\alpha_i(t) > T_i(t)$.

The lifetime of an activity set is defined as the interval between the time the activity set is established and the termination time. In other words, all members of an activity set have been re-referenced after it was formed. A Bounded Locality Interval (BLI) is defined as the pair consisting of an activity set and its lifetime. Suppose that two activity sets at time t, $A_i(t)$ and $A_j(t)$, exist with $i < j$. Then, $A_i(t)$ is a subset of $A_j(t)$ and the lifetime of $A_i(t)$ is shorter than that of $A_j(t)$, because the set of the i most recently referenced blocks is a subset of the set of the j most recently referenced blocks. Thus, BLIs at a particular time constitute a hierarchy in terms of both set membership and lifetime. The top level (level 1) of the hierarchy corresponds to the largest BLI and the level of a BLI is defined as its distance from the top of the hierarchy minus one. Thus, the higher the level of a BLI, the smaller the activity set size and the shorter its lifetime.

A set of new strategy-independent restructuring algorithms can be derived from the concept of BLI. If a set of blocks belong to an activity set of high level, they should be put close to each other in the virtual address space so as to decrease the working set size or the resident set size of the program. In order to reflect this situation, the Activity Set algorithm-1 (AS1) is defined as a restructuring algorithm which increments the label of the edge between blocks b_i and b_j by an amount equal to the level of the smallest activity set to which blocks b_i and b_j belong. Practically, the labels are incremented by one when an activity set is established because of the inclusion property of BLIs. Suppose that blocks b_i and b_j belong to an activity set of level 3. This implies that these blocks are the members of the three activity sets, say, $A_\ell(t)$, $A_m(t)$, and $A_n(t)$, with $\ell > m > n$, where $A_\ell(t)$ is the activity set of level 1

and $A_n(t)$ is the smallest activity set which includes b_i and b_j. Since $A_\ell(t)$, $A_m(t)$, and $A_n(t)$ are established in this chronological order, incrementing by one the label of the edge between b_i and b_j each time one of the three activity sets is established, is equivalent to incrementing the label by three, which is exactly what the AS1 algorithm does.

A simpler algorithm can be used if the hierarchy has only a few levels or if the average size of the activity sets of level 1 is small enough to fit in the main memory. The Activity Set algorithm - 2 (AS2) keeps track of the activity sets of level 1 only and may be described as the algorithm which increments by one all the labels of the edges between the blocks which belong to an activity set of level 1.

Establishment of an activity set does not necessarily mean that all the members of the set will be referenced repeatedly in the future. Some of them can stay in the set unreferenced. In other words, not all the members of an activity set are always active. Therefore, a stricter termination condition of activity sets is required in order to remove inactive members. One possible solution to the termination condition problem is to introduce a parameter α which plays a role similar to the one of the window in the working set strategy. This allows us to define a set of blocks called a strict activity set. A strict activity set at time t is defined as any activity set all the blocks of which have been rereferenced after the time $t-\alpha$. Note that strict activity sets also have a hierarchical structure. By using the notion of strict activity sets, two restructuring algorithms analogous to AS1 and AS2 may be introduced. The Strict Activity Set algorithm - 1 (SAS1) is defined as the algorithm which increments the label of the edge between a pair of blocks b_i and b_j by the level of the strict activity set to which blocks b_i and b_j belong. The Strict Activity

Set algorithm - 2 (SAS2) is defined as the algorithm which increments by one the label of the edge between a pair of blocks only when these blocks belong to a strict activity set of level 1.

The performance of the restructured programs obviously depends also on the clustering algorithm employed in the restructuring procedure.[12] However, since investigating the clustering algorithms was not the objective of this research, we used the same simple clustering algorithm throughout the experiments. The clustering algorithm will be described briefly in the next section.

Experimental Results

Trace driven simulators were developed for the working set environment and the PSI environment. Some experiments were performed to evaluate the performances of programs restructured with the new algorithms under the two memory management strategies using five block reference strings which had been obtained from an actual program by Ferrari and Lau.[17] The reference strings (called S1 through S5 in the sequel) were generated from an instrumented PASCAL compiler compiling five different source programs. The PASCAL compiler is 17,836 60-bit words large and has 139 instruction blocks (procedures). The mean block size is 129 words, the maximum 669 words and the minimum 18 words. The reference strings contain only instruction references. However, it is worthwhile to mention that the instruction and the data portions of a program can be restructured independently, resulting in the same orderings and performance improvements as if the restructuring were based on the complete reference string, provided that the program runs in a working set environment and instructions and data are stored in different pages. The input data to the PASCAL compiler, used to obtain

the five strings, were: S1) a part of the PASCAL compiler itself; S2) a program of about 500 predominantly arithmetic statements; S3) the same as in S2) but with numerous errors inserted; S4) a global flow analysis program of about 400 source statements; and S5) a tree traversal program of about 65 statements.

There are many factors which influence the performance indices of the restructured programs: the program to be restructured; the restructuring algorithm; the input data used in the restructuring procedure; the memory management strategy; and the input data to the program. Since a multivariate analysis performed by Ferrari and Lau[17] shows that a very large portion of the variation about the grand mean of the number of page faults in a working set environment is to be attributed to the restructuring algorithms, and since it is obvious that the memory management strategy is another major factor, only these two major factors were varied during the experiment. The other factors were fixed, that is, the same input data was used for restructuring and for executing the program, the page size was set equal to 1024 words, the window size was chosen to be 50 ms both for the working set strategy and for the CWS and the MWS algorithms, and a PSI value of 13 was selected both for the PSI environment and for the CPSI and the MPSI algorithms. In the PSI environment, interruptions were caused by page faults and quantum time expirations, and the quantum time was fixed at 400 ms.

The clustering algorithm used throughout the experiment is a hierarchical clustering algorithm suggested and used in previous experiments by Ferrari. The algorithm proceeds as follows: the two vertices in the restructuring graph whose connecting edge has the highest label are clustered into a single node if the sum of their sizes is not greater than the page size; the label of the edge connecting any other node to the new node is computed as

the sum of the labels of the two edges connecting the node to these two nodes clustered into a new node; and the process is repeated until no further clustering is possible.

As an example of the impact of restructuring, the paging characteristics, which are curves relating the number of pages to the average resident set size for various values of PSI, of the original and the restructured programs are presented in Figure 1.

Performance indices of the original and the restructured programs under a working set environment and under a PSI environment are presented in Table 1 and in Table 2, respectively. The performance indices of the programs restructured by the strategy-oriented algorithms, the CWS and the MWS algorithms in Table 1 and the CPSI and the MPSI algorithms in Table 2 are shown for comparison.

In the working set environment (see Table 1), restructuring algorithms can be divided into the following four classes according to the numbers of page faults (NPF). The algorithm in class 1 is better than those in class 2, which are better than those in class 3, and so on.

1) CWS
2) AS1 and AS2
3) MWSS
4) SAS1, SAS2, and NM

Table 1 shows that the CWS algorithm is superior to the AS1 and the AS2 algorithms. However, the differences among the NPFs are not significant enough to reject the hypothesis that there is no difference between the CWS and the AS1 algorithms. This result has been obtained by performing a multivariate analysis of variance (MANOVA) at the 1% level. The same hypothesis between the CWS and the AS1 algorithms is rejected at the same level. The

hypothesis that there is no difference between the average working set sizes (AWSS) of the algorithms is not rejected by MANOVA at the 5% level. The CWS, AS1, AS2, and MWS algorithms decreased AWSS by about 13%, and the SAS1, SAS2, and NM algorithms decreased AWSS by about 5%.

In the PSI environment (see Table 2), restructuring algorithms can also be categorized into the following four classes in order of increasing NPFs and decreasing performance.

1) CPSI and MPSI
2) AS1 and AS2
3) NM
4) SAS1 and SAS2

A statistical difference between the NPFs of the CPSI and the AS2 algorithms, but not between those of the CPSI and the AS1 algorithms, is found by MANOVA at the 5% level. The CPSI, MPSI, AS1 and AS2 algorithms decrease the average resident set size (ARSS) by about 10%, whereas the SAS1, SAS2, and NM algorithms have very little impact on it.

The performance of the program restructured with the AS1 or AS2 algorithm is satisfactorily close to the performance obtained by restructuring it with a strategy-oriented algorithm. However, the SAS1 and SAS2 algorithms are not as effective as the AS1 and AS2 algorithms. In order to investigate the causes of their poor performance, another set of experiments were performed with the string S2 and the SAS2 algorithm for various values of α in the PSI environment. The results are presented in Table 3. Note that the SAS2 algorithm with $\alpha = \infty$ is actually the AS2 algorithm. Table 3 shows that the poor performance of the SAS2 algorithm is probably to be partially attributed to the relatively long average lifetime of the BLIs of level one.

Conclusion

A set of new strategy-independent restructuring algorithms has been presented and their effectiveness in a working set environment and in a global LRU-like (PSI) environment has been evaluated with trace-driven simulators and block reference strings obtained from a PASCAL compiler. The new algorithms have been derived from the concept of bounded locality intervals, which allows us to give a precise definition of the localities of a program. Even though the strategy-oriented algorithms are the most effective, the new AS1 and AS2 algorithms have been shown to be almost as good as the strategy-oriented algorithms in terms of the number of page faults and the average memory demand. The other new algorithms, SAS1 and SAS2, perform often as poorly as the classic strategy-independent algorithm NM.

The conclusions of the experiments are of course limited to the program and the input data we have used. However, the evidence we have gathered makes the AS1 and AS2 algorithms likely to be effective in most cases for programs running under various memory management strategies. The AS1 and AS2 algorithms have been derived by focusing on the locality of reference. It is reasonable to expect that even better strategy-independent algorithms will be obtained by considering also locality transitions.

Acknowledgments

The author wishes to thank D. Ferrari for his suggestion to study restructuring problems, for his clustering program and for the working set simulator. The author is also grateful to E. Lau and D. Ching for the reference strings used in the experiments reported in this paper. Finally, I would like to express my thanks to Ruth Suzuki for her elaborate typings of the manuscript.

References

1. P.J. Denning, "Virtual Memory", Computing Surveys, v.2, n.3, September 1970, pp. 153-189.

2. D. Ferrari, "Improving Locality by Critical Working Sets," Comm. ACM, v.17, n.11, November 1974, pp. 614-620.

3. D. Ferrari, "Tailoring Programs to Models of Program Behavior," IBM J. Res. Dev., May 1975, pp. 244-251.

4. E.I. Organick, The Multics System, MIT Press, Cambridge, Mass., 1972.

5. T.F. Wheeler, Jr., "IBM OS/VS1 - An Evolutionary Growth System," NCC 1973, AFIPS Conference Proceedings, v.42, pp. 395-400.

6. Y. Bard, "Characterization of Program Paging in a Time-Sharing Environment," IBM J. Res. Dev., September 1973, pp. 387-393.

7. J.B. Morris, "Demand Paging Through Utilization of Working Sets on the MANIAC II," Comm. ACM, v.15, n.10, October 1972, pp. 867-872.

8. M.H. Fogel, "The VMOS Paging Algorithm: A Practical Implementation of the Working Set Model," ACM Operating Systems Review, v.8, n.1, January 1974, pp. 8-17.

9. M.A. Auslander, J.F. Jaffe, A.L. Scherr and J.P. Birch, "Functional Structure of IBM Virtual Storage Operating Systems," IBM Syst. J., n.4, 1974, pp. 368-411.

10. A. Sekino, "Performance Evaluation of Multiprogrammed Time-Shared Computer Systems," MIT Project MAC, MAC TR-103, September 1972.

11. N.R. Nielsen, "The Simulation of Time Sharing Systems," Comm. ACM, v.10, n.7, July 1967, pp. 397-412.

12. T. Masuda, H. Shiota, K. Noguchi and T. Ohki, "Optimization of Program Organization by Cluster Analysis," Information Processing 74, North-Holland Publishing Co., Amsterdam, 1974, pp. 261-265.

13. D. Ferrari and M. Kobayashi, "Program Restructuring Algorithms for Global LRU Environments," in preparation.

14. D.J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory," IBM Syst. J., n.3, 1971, pp. 168-192.

15. J.R. Spirn and P.J. Denning, "Experiments with Program Locality," FJCC 1972, AFIPS Conference Proceedings, v.41, part I, pp. 611-621.

16. A.W. Madison and A.P. Batson, "Characteristics of Program Localities," Comm. ACM, v.19, n.5, May 1976, pp. 285-294.

17. D. Ferrari and E. Lau, "An Experiment in Program Restructuring for Performance Enhancement," Proceedings of the 2nd International Conference on Software Engineering 1976, pp. 203-207.

TABLE 1

Number of Page Faults in the Working Set Environment

String	Original	AS1	AS2	SAS1	SAS2	NM	CWS	MWS
S1	5868	4144	4353	5805	5255	5938	3830	4524
S2	1963	976	1028	1569	1762	1387	778	1004
S3	1843	983	851	1324	1489	1783	832	1032
S4	1316	909	821	1399	1289	1258	658	840
S5	170	80	92	97	126	122	93	111
Mean	2232	1418	1429	2039	1984	2098	1238	1502

Average Working Set Size

String	Original	AS1	AS2	SAS1	SAS2	NM	CWS	MWS
S1	5.36	4.77	4.72	5.29	5.92	6.07	4.41	4.88
S2	8.16	6.09	6.31	7.93	7.99	7.28	6.33	6.97
S3	8.03	6.82	6.93	7.26	6.96	7.82	7.12	6.58
S4	8.18	8.03	7.78	7.69	9.38	7.62	7.68	7.61
S5	8.49	8.14	7.53	8.38	7.57	7.18	7.67	7.19
Mean	7.64	6.77	6.65	7.31	7.56	7.19	6.64	6.65

TABLE 2

Number of Page Faults in the PSI Environment

String	Original	AS1	AS2	SAS1	SAS2	NM	CPSI	MPSI
S1	603	192	266	445	375	367	192	191
S2	397	89	121	215	226	144	72	87
S3	347	143	120	218	228	161	59	76
S4	250	77	70	307	196	188	51	75
S5	54	24	25	36	31	26	15	19
Mean	330	105	120	244	211	177	78	90

Average Resident Set Size

String	Original	AS1	AS2	SAS1	SAS2	NM	CPSI	MPSI
S1	12.00	9.99	10.42	11.57	11.56	11.99	10.75	10.30
S2	13.30	11.31	11.46	12.77	12.78	12.16	11.39	11.52
S3	13.33	11.63	11.41	12.78	12.81	12.43	11.63	11.28
S4	13.21	12.30	12.26	13.31	13.14	12.99	11.22	11.90
S5	12.49	12.48	12.10	13.00	13.12	13.16	11.32	11.13
Mean	12.87	11.54	11.53	12.69	12.68	12.55	11.26	11.23

TABLE 3

The Effect of α of SAS2 on the Performance Indices of the Program for the String S2 in the PSI Environment

	Original	α = 25 ms	α = 50 ms	α = 100 ms	α = 200 ms	$\alpha = \infty$
NPF	397	279	226	148	204	121
ARSS	13.30	13.11	12.78	11.96	13.03	11.46

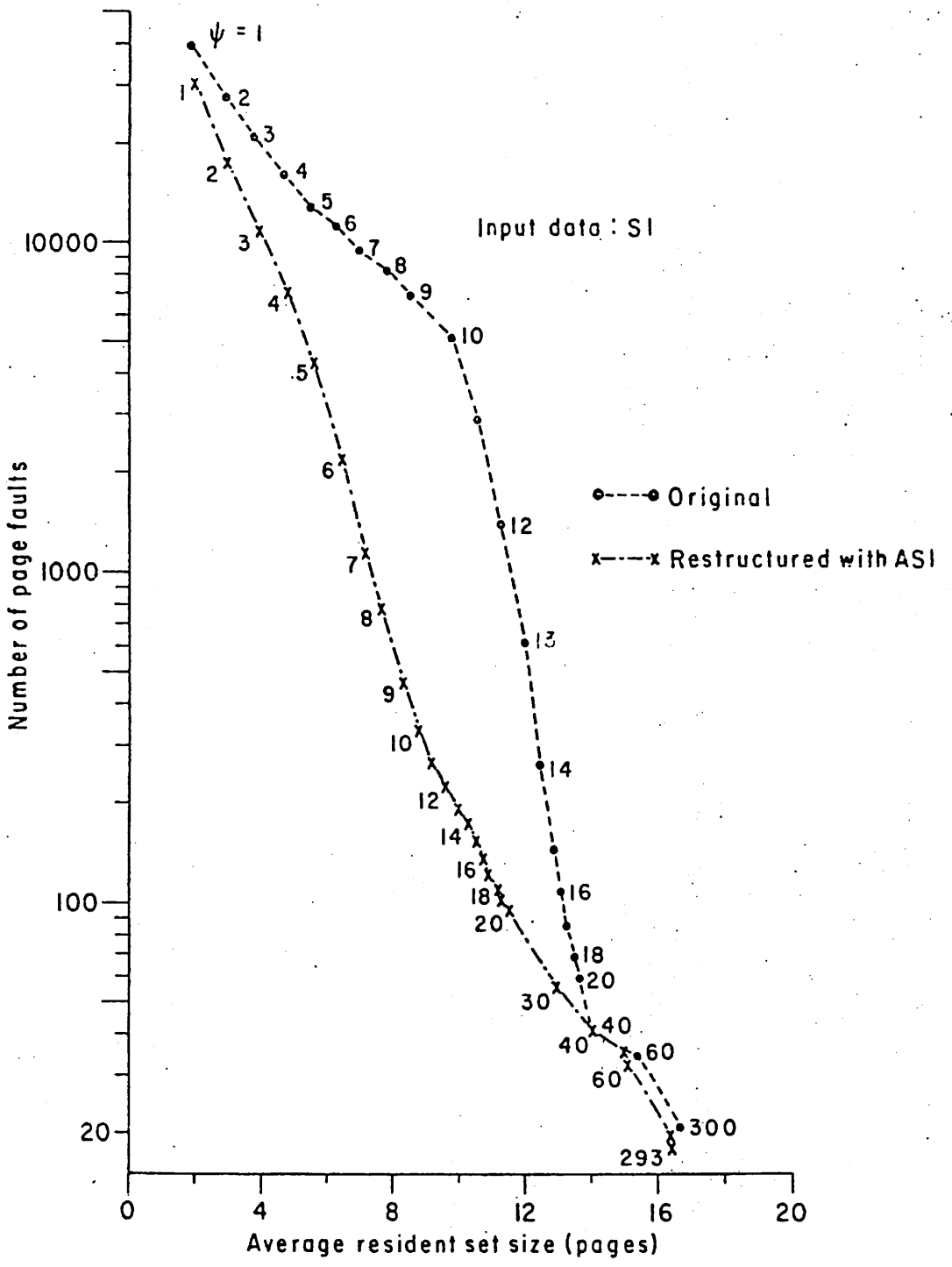

Fig. 1 Paging characteristics of the program