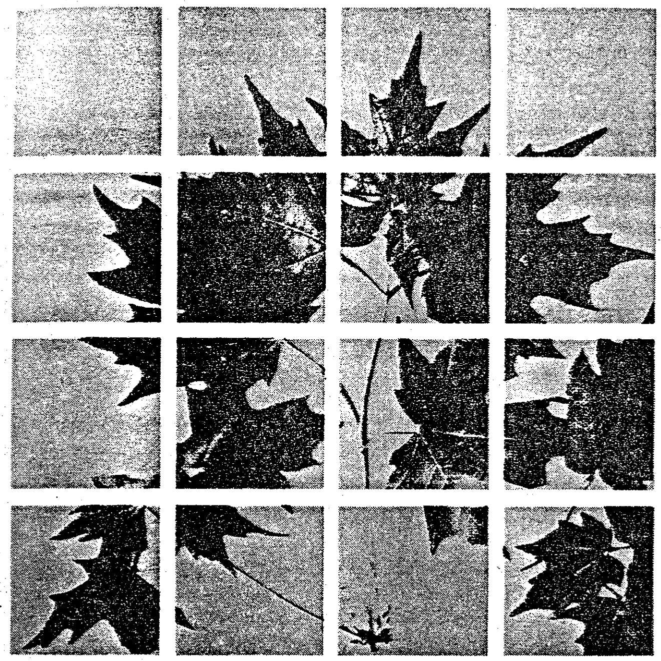


The Improvement of Program Behavior

Domenico Ferrari
University of California, Berkeley

People seldom improve when
they have no other model
but themselves to copy after.
—Oliver Goldsmith

Introduction

The practical objective of program behavior studies is to enhance program and system performance. On the one hand, the knowledge resulting from these studies may be useful in designing new programs and new virtual memory systems that are capable of levels of performance higher than those currently achievable. On the other hand, such knowledge may often be employed to increase the performance of existing programs and systems.

The observation that the performance of a virtual memory system is very sensitive to the referencing behavior of the programs running on it was made very early in the history of such systems. Interest in algorithms which would automatically manage a memory hierarchy, taking program behavior into explicit account, arose quite early also (see, for example, Kilburn et al.[1] and Belady[2]). Program locality[3] was recognized not only as essential to the achievement of acceptable levels of performance by a virtual memory system* but also as a property which directly influences the performance of both the program and the virtual memory system on which the program runs. A "more local" program needs relatively fewer system resources than a "less local" one. This is especially true for those resources which tend to be the most critical and the most scarce in a virtual memory environment: primary memory space and paging channel bandwidth. As a result, a "more local" program will be relatively faster and cheaper to execute, and will positively contribute to the system's overall productivity as expressed, for instance, by the throughput rate.

*A perfectly nonlocal program, which would reference the addresses of its virtual space at random, would make the access time of a memory hierarchy very close to the one of the secondary memory, thereby degrading the system's performance by several orders of magnitude.

The interest in program behavior was originally caused by the desire to devise memory management policies which would best match the predominant referencing habits exhibited by programs. The above remarks naturally suggest that the outcomes of program behavior investigations should also be used to determine methods for enhancing the locality of programs or, more generally, for making programs better suited to the virtual memory environment they have to live in. This approach is philosophically opposite to that of adapting the system (through its memory policy) to the programs, since it may be viewed as attempting to adapt the programs to the system. However, the two approaches are not mutually exclusive. In fact, one could argue that a successful virtual memory system can only result from the coupling of a "program behavior oriented" memory policy with a workload composed of reasonably "policy oriented" programs. Since locality is an essential property in current implementations of virtual memory concepts, and since most memory policies base their decisions on the assumption of local behavior, in the above statement the phrase "program behavior oriented" may often be replaced by "locality based," and "policy oriented" by "local."

When a program is to be designed, the programmer can—and ought to—organize it so that its dynamic behavior is as local as possible. Some rules to be followed have been proposed,[4] and some studies on programs implementing specific algorithms, such as those for matrix manipulation[5] or for sorting,[6] have been performed. However, it is generally difficult or impossible for the human mind to comprehend fully the dynamics of even a medium-size, relatively simple program. Programmers cannot therefore be expected to succeed very often in their attempts to adapt the characteristics of programs to those of systems. It is here that fully or partially mechanizable methods for program referencing-behavior improvement can really help programmers and installation managers. Even programs written by an experienced, locality-minded programmer can often be improved to a surprisingly great extent by using automatic or semi-automatic techniques. The purpose of this paper is to describe how some of these methods, the ones called *program restructuring techniques*, can be used effectively to improve a program's dynamic behavior.

Program restructuring

What is program restructuring? Why can it improve program behavior? A simple example will help us answer these two questions. Consider, say, a Fortran program consisting of one main routine MAIN and three subroutines SUB1, SUB2, and SUB3. Let us assume for simplicity that the sizes of these four *blocks*, measured in machine-language instruction words, are approximately equal. All the variables and arrays the program works on are stored in four COMMON areas, called COM1, COM2, COM3, and COM4, whose sizes are about the same as those of the instruction blocks. To simplify our problem further, we assume that the size of a page in the paged virtual memory system on which the program has to run is twice the size of an instruction or data block. Thus, if we do not consider the library routines which the linker usually appends to a program, the virtual space occupied by the program will consist of four pages, each of which will contain two blocks. No restriction is imposed here on the nature of the two blocks which share the same page, so that it is possible to have an instruction block and a data block together.

The layout of the program in its virtual space, generally decided by the linker, is influenced by the order in which the programmer inputs the instruction blocks and declares the data blocks. Restructuring means modifying a program's layout in virtual memory. In our example, this modification only entails relinking the modules of the program after having changed the order in which they are presented to the linker. If the blocks had not been chosen so as to be relocatable with respect to each other, the program could still be restructured, but some changes to the source code and the recompilation of some modules would probably be required.

To show that the behavior of a program is sensitive to its layout in virtual memory, let us analyze the performance of our program with two different layouts. Table 1, row a, displays a short *block reference string* generated by our program under certain input data. Each element in the string is a reference issued by the program, but the virtual address referenced has been replaced by the number of the block to which that address belongs. The correspondence between block numbers and block contents is given in Figure 1. Note that the block reference string is a characterization of program behavior (more precisely, of the behavior of a program under a certain set of input data) independent of the program's layout. It is just the characterization at the symbolic level that is needed in the context of our problem.

The two layouts we shall consider here are shown in Figure 1. Different layouts generally produce different *page reference strings* from the same block reference string. An example may be seen in Table 1, rows b and c, which show the two page reference strings our program would generate with the two layouts in Figure 1. If the memory policy used by the system which executes our program were a fixed-partitioning, demand-fetching policy with LRU (least recently used) replacement, the two page reference strings in Table 1, rows b and c, would generate the numbers of page faults reported in Figure 2 as functions of the amount of primary memory space allotted to the program. The two curves in Figure 2 illustrate the main conclusion of this section: that the layout has a non-negligible impact on the performance of a program. This conclusion is intuitively plausible since whether a memory reference causes a page fault or not depends on whether the referenced block (or part of block) is in primary memory or not, and the memory contents at any given time differ for different layouts.

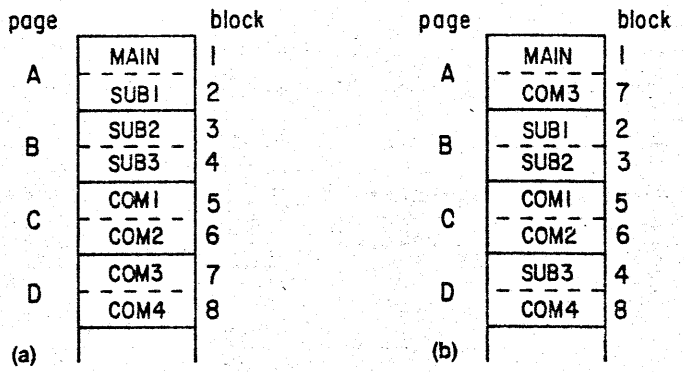


Figure 1. Layouts of a program in virtual memory: (a) the original arrangement; (b) a restructured version of the program.

Table 1. A block reference string and some page reference strings corresponding to it.

(a)	171713562662564884834835353562622561171352626248 83
(b)	ADADABCCACCACCBDDBDBBDBCBCBCCACAACCAADABCACACABDDB
(c)	AAAAABCCBCCBCCDDDDDBDDBCBCBCCBCBBCCAAAABCBCBCBDDDB
(d)	AAAAABBCCCCCBCDDDDDBDDBBBBBBCCCCCBCAAAABBCCCCCDDDB

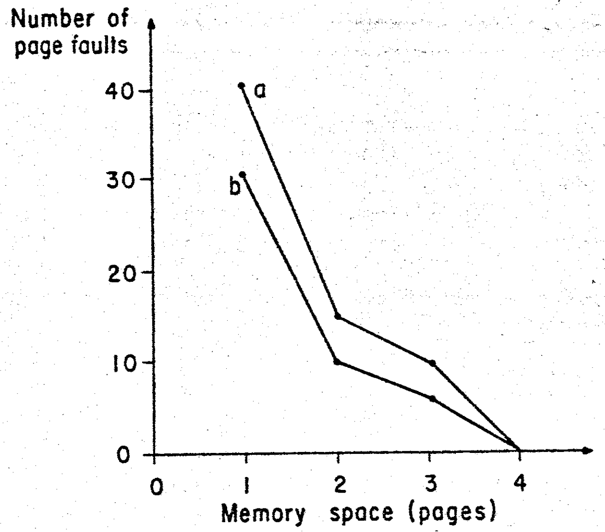


Figure 2. Total number of page faults as functions of the allotted space. Curve a shows those generated under local LRU replacement strings in Table 1, row b; curve b, row d.

The influence of the layout was first experimentally measured by Comeau.[7] Tsao and Margolin,[8] in their multifactor paging experiment on a Fortran compiler, found the following breakdown of contributions to the total variation of the logarithm of the number of page faults:

memory space allotted to the compiler	53.7%
length of the program to be compiled	25.2%
ordering of the subroutines	12.5%
replacement algorithm	2.6%
interaction between memory space and subroutine ordering	4.0%
other interactions among factors	2.0%

Restructuring procedures

We have seen that the layout of the program has an appreciable impact on the program's paging performance and that modifying the layout is always feasible—often quite easy. What procedures should we follow to determine a layout that is likely to produce as substantial a performance improvement as possible?

Ideally, we would like to obtain an optimum layout, one which would minimize the number of page faults, or the memory space demand (as measured by, for instance, the mean working set size of the program), or some other performance index. Finding such an optimum is a highly combinatorial problem for which no efficient, practically applicable solution exists in the general case. Often, however relatively inexpensive procedures are available which produce layouts that are nearly optimal from any practical standpoint. These layouts are usually characterized by performances which can be improved further only by very small amounts and at costs which are often prohibitive with respect to the corresponding improvements. Frequently, these small additional improvements are not worth seeking because of the influence that input data have on the performance of most programs. Since a successful restructuring is based on the dynamic behavior of the program, which is always somewhat sensitive to the input data, an excessive "tuning" of the layout to certain data may be useless—even counterproductive—when the program is executed with different inputs (which is the normal case throughout the program's lifetime).

The above remarks have been made assuming that the layout of a program in its virtual space remains fixed during execution. If we allowed the layout to change dynamically, these remarks would no longer apply. Such an approach to restructuring would conceptually require an additional, "metavirtual" address space, and a dynamic allocation and address mapping mechanism between this space and the virtual memory of a program. The relationships between prefetching memory policies and this type of restructuring have been recognized by Baer and Sager,[9] who proposed a scheme for dynamic layout modification. Effective, practically implementable dynamic procedures seem to be still beyond the state of the art, however.

The other approaches, which modify the program's layout "off-line" but keep it fixed during execution, can be classified into two categories. On one side, we have those procedures which base their decisions on purely static information about the program, that is, on its text. Since such information is available to compilers and linkers, these procedures can be applied at compiling time (for intramodule restructuring) and at linking time (for intermodule restructuring). The early interest in these approaches[10] seems to have decreased considerably in recent years, probably because of the relatively small, sometimes even negative, improvements they produce and the much more impressive performance, at acceptable costs, of the approaches in the second category. These approaches utilize information about the dynamic behavior of the program, gathered during one, or possibly a few, of its runs. Since restructuring is convenient only for those programs which are to be executed many times (actually, the more times they are run, the better, as we shall see), the requirement that the program be executed once for information gathering purposes is an acceptable one.

Most of the procedures in this category can be conceptually described as consisting of the following steps.

Step 1. The program to be restructured is divided into blocks. From the example we discussed in the previous section, it should be evident that blocks are to be substantially smaller than pages. The smaller the blocks, the better the new layout is likely to be. On the other hand, the complexity and cost of the procedure grow rather sharply with the number of blocks. Thus, a compromise solution will have to be reached. As already mentioned, layout modifications will be easier if blocks are relocatable with respect to each other at the source level.

Step 2. The program is executed and the dynamic behavior information gathered is used to construct a nondirected graph, called the *restructuring graph.* The nodes of such a graph represent blocks, and the numerical labels of the edges represent the "desirability" that the nodes (blocks) they connect be laid out together within the same page. The algorithm which is used to compute these labels from dynamic behavior information is called the *restructuring algorithm.* The various approaches which follow the procedure we are describing differ mostly in the restructuring algorithms they use to define and compute the desirability of block pairings.

Step 3. A *clustering algorithm* is applied to the restructuring graph in order to obtain a new layout for the program. If, as in the example of the previous section, groups of blocks fit exactly into pages so that no block resides (or is allowed to reside) across page boundaries, the outcome of this step may consist of the suggested partitioning of the set of blocks into pages, and the relative positions of the pages in virtual memory are immaterial. In all other cases, these positions are important and should be derived during this step along with the composition of each page. Alternatively, the clustering algorithm may produce directly the layout of the blocks in virtual memory. The problem is, loosely speaking, one of determining a linear arrangement (or a partitioning) of blocks which maximizes the vicinity of those pairs having the highest labels. More precise statements of the problem, which can be described in purely graph-theoretical terms, can of course be given. For instance, when the blocks are to be partitioned into pages, a partition which maximizes the sum of all intracluster labels, while satisfying the constraint that the sum of the sizes of all nodes in a cluster be not greater than the page size, is usually sought. Reasonably efficient and good (although not optimal) algorithms for such problems exist. A description of some of those used in restructuring experiments can be found in several papers.[11-13] For example, a very simple clustering algorithm for the block partitioning problem described above entails iteratively sorting the list of edge labels in decreasing order, clustering together the admissible pair of blocks with the highest label, replacing the clustered pair with a single node whose size is the sum of the sizes, replacing the edges connecting a node to the two nodes clustered together with a single edge whose label is the sum of the labels, and updating the list of edge labels accordingly.[13]

Step 4. The program is restructured according to the layout suggested by the previous step.

The rest of this paper will be primarily concerned with the restructuring algorithms to be used in Step 2. Even though there may be differences in the techniques employed to perform the other steps, the restructuring algorithm is the main distinguishing feature of the approaches in the category we are examining. It should also be noted that Step 3, as described above, is independent of Step 2, in the sense that the same clustering algorithm may be used in conjunction with a large number of restructuring algorithms, and the same restructuring algorithms may be followed by a number of different clustering algorithms.

An example of a restructuring algorithm which follows the procedure just described is the *nearness method* proposed by Hatfield and Gerald.[11] The nearness method sets the value of the label of edge ij equal to the number of times block i is referenced immediately after j, or j immediately after i, during the program's execution in Step 2. From an implementation viewpoint, the graph can be very easily built by incrementing the label of edge ij in the block reference string every time a reference to block j follows one to block i, or vice versa. Thus, the desirability of positioning two blocks adjacent to one another in virtual memory is made proportional to the number of times the referencing pattern of the program jumps directly from one to the other or vice versa.

The restructuring graph derived from the block reference string in Table 1, row a, is shown in Figure 3. The same figure also presents a partitioning of the graph suggested by the clustering algorithm described in Step 3 above. The corresponding page reference string is shown in Table 1, row d.

A restructuring algorithm which may be viewed as an extension of the nearness method is the one proposed by Masuda et al.[12] The algorithm is based on an extended definition of nearness: two blocks i and j are near not only when they are consecutively referenced, but also when references to them follow each other at a short distance in

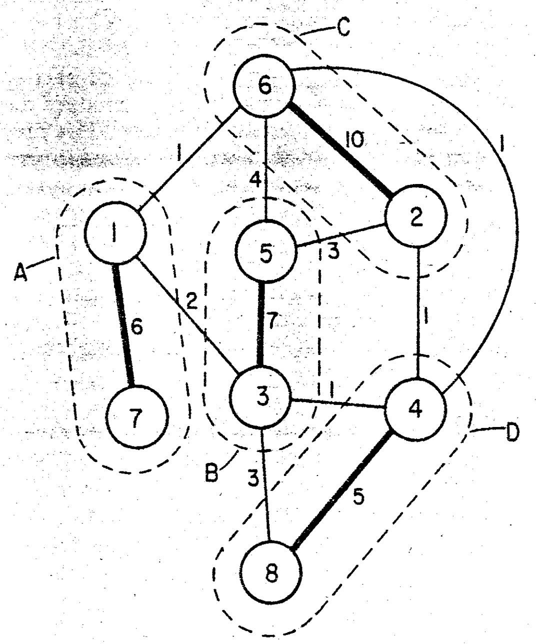


Figure 3. The restructuring graph generated by the nearness algorithm for the string in Table 1, row a. The dashed lines are boundaries of clusters, which contain the blocks to be stored within the same page. The heavy edges are the intracluster links.

time. A *window* T is defined, consisting of a time interval or a certain number of references. At each new reference (say, to block i) issued at virtual time t by the program, the labels of all the edges connecting i to the blocks which have been referenced in the recent past; that is, during the virtual-time interval $(t\text{-}T, t)$, are incremented by 1.

Generally this algorithm, to be discussed again below, works better than the nearness method because of its broader field of observation. Clusters of blocks which are never consecutively referenced but often referenced soon after each other are unlikely to be suggested by the nearness algorithm, even though they would be as convenient as, and sometimes more convenient than, those resulting from the application of this algorithm. With only a minor increase in the cost of the restructuring procedure (processing the block reference string with a backward window containing more than one reference is slightly more expensive than remembering only one, as the nearness method does), the extended algorithm can generally achieve a significant performance improvement. This is not the case of our extremely simple example in Table 1, row a. The reader may in fact verify that, both with a window of three references and with one of four, the partitioning suggested by this algorithm coincides with the one in Figure 3, which was obtained by applying the nearness algorithm.

Program tailoring algorithms

The restructuring algorithms described in the previous section are implicitly based on observations concerning the relationships between program behavior and program performance in a virtual memory environment. For instance, the philosophy underlying the nearness algorithm is that performance can be improved by increasing the number of consecutive references to the same page. This would always be true if only one page of each program were allowed to be in primary memory at any given time. The algorithm proposed by Masuda et al.[12] recognizes that a program generally has more space in memory, and that consecutive references to pages within the current locality are not less desirable than those to the same page. The better results that this algorithm usually obtains are to be attributed to the greater amount of information it provides about equally desirable, alternative block combinations.

A different philosophy for restructuring algorithms has been proposed.[14,15] This philosophy, called *program tailoring* or *strategy-oriented restructuring*, consists of taking explicitly into account, when designing the restructuring algorithm, the memory management strategy under which the programs to be restructured will have to run.

Program tailoring is applicable if the strategy is unique and known; it is much easier to apply if the stretegy is such that the behavior of a program under it does not depend on the other programs with which the program shares the primary memory. Whereas the former assumption is a realistic one in most practical cases, the latter is satisfied only in some systems. However, the class of strategies which satisfy this assumption includes all the fixed-partitioning policies with local replacement algorithms and the working set policies. Many of the strategies used in present day virtual memory systems do not belong to, but can be considered approximations of policies in, these classes.

How can one take the memory policy into account in restructuring a program? One way of describing how to proceed is to say that one has to identify the "ideal behavior" (or, in other terms, the model of program behavior) which the policy assumes, and devise an algorithm which tries to tailor the behavior of the program to be restructured to the ideal one. Although all, or almost all, memory policies use the notion of locality to predict the behavior of a program in the near future, each policy makes a different use of it. In other words, each policy assumes a different model of ideal program behavior, which may be thought of as the behavior of a program for which all the predictions made by the policy are correct. Thus, program tailoring consists of exploiting our knowledge of the prediction mechanisms of a memory policy to make a program's behavior as predictable as possible by that policy. And it clearly identifies a model which, as Oliver Goldsmith said, is highly desirable in any improvement effort.

A few examples will hopefully clarify the principles expressed above. Assume that the system on which the program is to run manages its memory according to a fixed-partitioning, local LRU replacement strategy. This means that each program receives a fixed and constant amount of space, in which pages are replaced, when necessary, following the LRU order. If a program is given m page frames in primary memory, they will contain, whenever the partition is full, the m most recently referenced pages of the program, which this policy considers the most likely to be referenced in the near future. In restructuring the program, we can try to approach this goal by removing as many references to new pages as possible. We know that at any given time the primary memory will certainly contain the m most recently referenced blocks (if a block can lie across page boundaries, some blocks may be in memory only partially). The identity of those blocks which are certainly going to be in memory at any given virtual-time instant of a run may be easily obtained from the block reference to string corresponding to that run. If the next reference is to one of these blocks, that reference will *never* generate a page fault. If, on the other hand, it is a *critical reference*—that is, a reference to a different block—then clustering this block with any of the blocks in memory will also prevent a page fault from occurring. A simple restructuring algorithm (the *critical LRU,* or CLRU, algorithm) therefore consists of increasing by one the labels of the edges connecting the block which causes the critical reference to all of the blocks occupying the first m positions of the LRU block stack. Noncritical references, which involve blocks whose distance from the top of the block stack is not greater than m, are ignored.

The same principles can be applied to all local replacement policies, since it is always possible with such policies to derive from a block reference string the sets of blocks which are certainly going to be in memory at any given time. We thus have, for instance, the CFIFO, the CLFU, the CMRU, and even the CMIN restructuring algorithms, whose objectives are to tailor programs to the first-in-first-out, least-frequently-used, most-recently-used, and MIN[2] replacement policies, respectively. All of these restructuring algorithms belong to the class of *critical algorithms*, a term which alludes to the distinction they make between critical and noncritical references.

Critical algorithms are not confined to the domain of fixed partitioning strategies. They can be applied whenever the portion of a program contained in memory at any instant can be derived or estimated from the behavior of a single program and the index to be minimized is the page fault rate. An example is the *critical working set* (CWS) algorithm,[13] whose objective is to tailor programs to the pure working set policy. This policy gives each program a time-variant amount of space coinciding at any instant t with the size of the program's working set at t. In a multiprogramming environment, if the space available in primary memory is not sufficient to hold the working set of a program, the program is not run. Thus, the information contained in a block reference string and the knowledge of the window size T allow us to identify the blocks which will certainly be in memory at each virtual-time instant. These blocks are the members of the block working set $W_b(t, T)$, that is, those blocks which are referenced between $t - T$ and t. The CWS algorithm increments by 1 all the labels of the edges connecting a critically referenced block to all the members of W_b at the time the critical reference is issued. The layout in Figure 1b is the one suggested by the CWS algorithm for the string in Table 1, row a, with $T = 3$ references.

Restructuring algorithms may be designed taking into account many policy implementation details. An example we shall mention is the one of the *critical sample working set* (CSWS) algorithm, which was called the A algorithm in a previous paper.[15] For convenience of implementation, the pure working set policy may be approximated by measuring the working set periodically instead of at every reference. To simplify the discussion, let the virtual-time sampling period coincide with the window size T. At measurement time, those pages which have been referenced at least once during the last window are considered members of the working set and kept in memory. Those which have not been referenced drop out of the working set, and the frames they occupied are merged into the free list. Any page fault causes the space allocated to the faulting program to grow by one frame. The CSWS algorithm samples a block reference string at the same frequency as the system, measures the block working set W_b and, during the subsequent window, detects all critical references. For each one of them, it increments by 1 the labels of the edges linking the critically referenced block to the members of W_b.

When the memory policy used by the system is of the global type, it is still possible to devise critical algorithms or, more generally, tailoring algorithms if an approximate way is known of determining the portion of a program which is going to be present in primary memory at any instant of virtual time. Several systems use variations of the global LRU algorithm as their memory policy.[16] Bard[17] has proposed an approach to the estimation of a single program's behavior in such an environment. The approach is based on a very simple characterization of the rest of the load on the system in terms of only one parameter, the *page survival index* (PSI). In spite of its crudeness, this characterization has produced good results and could be used as the basis of a tailoring algorithm if the range of values the PSI is expected to take during the executions of the program is known. Another tailoring algorithm for the global LRU policy has been proposed[14] but not tested.

When the main objective of restructuring is not the minimization of the page fault rate, other tailoring algorithms, not belonging to the class of critical algorithms, can be devised. Suppose that, in a pure working set environment, our foremost concern is to reduce as much as we can the mean working set size of a program (this may be the case if, for instance, the installation's charging algorithm penalizes large working sets more than high fault rates). Then, we can apply the algorithm described by Masuda et al.,[12] which has been outlined at the end of the previous section. There is, however, a conceptually important difference between Masuda's application of this algorithm[12] and the one proposed here. In Masuda's proposal, the window size had to be selected according to convenience and reasonableness, and programs would be restructured for any policy. Here, the window size coincides with that used by the working set policy. Thus, while the former is not a tailoring algorithm, the latter, called the *minimum working set* (MWS) algorithm, is.

For the string in Table 1, row a, the MWS algorithm with $T = 3$ references suggests the same layout as the nearness algorithm (see Figure 3). It is easy to verify in Table 1, row d, that this layout would produce 13 page faults and a mean working set size of 1.85 pages under a pure working set policy with $T = 3$ references. The CWS layout (see Figure 1b and Table 1, row c) would on the other hand produce produce only 10 page faults, but the mean working set size would be 2.10 pages. Under the same policy and the same input, the nonrestructured program (see Figure 1a and Table 1, row b) would generate 13 page faults and have a mean working set size of 2.44 pages.

Discussion of the results

A number of experiments have been performed to evaluate the restructuring procedure described above and several restructuring algorithms. Some results have been published, others have not. We shall now attempt to summarize what we consider the most important results available to us in four areas: performance improvement, data dependence, portability, and cost.

Performance improvement. A variety of programs have been restructured (in most cases, restructuring has only been simulated). Most of them were of the systems programming type (compilers, assemblers, editors, operating system modules), but also some simulators and other application programs have been experimented with, as shown in Table 2. Programs purposely designed for virtual memory systems[13] as well as programs written for other types of systems[15] were included. Their authors ranged from the nonprofessional, occasional computer user to the experienced, locality-minded systems programmer. In almost all cases and with almost all restructuring algorithms, significant performance improvements were obtained (see Table 2). Page fault rates were reduced in certain cases by one order of magnitude or even more. Mean working set sizes were almost halved in some experiments. Their coefficient of variation (not reported in Table 2) was also reduced, or not increased, by such algorithms as CWS and MWS. In general, the magnitude of the improvement depends on the quality of the layout of the original programs. A near-optimal program (an endangered species in the realm of nonrestructured programs, a flourishing one among restructured programs) cannot be appreciably improved by restructuring.

An interesting question is the one of how close to the optimum is a layout suggested by a restructuring algorithm. The answer depends, of course, on the algorithm, on the program, and on the memory policy. Not surprisingly, tailoring algorithms usually outperform others by non-negligible amounts. This can be clearly seen in the comparisons of the nearness and CWS algorithms reported in Table 2, row c, and in those of CWS and CSWS in a sampled working set environment (Table 2, row d). The reader will notice that in some of the latter comparisons CWS is more effective than CSWS in reducing the mean working set size. However, the objective of both algorithms is to minimize the number of page faults, not the working set size (which in some cases turned out to be bigger than in the nonrestructured program).

The restructuring procedure dealt with in this paper is by its nature suboptimal, since it reduces all the information contained in the string to that contained in the restructuring graph, where only relationships between pairs of blocks can be represented. Only if no page contains more than two blocks may the information in the graph be sufficient to determine an optimum layout. In fact it can be shown that, under this assumption, the CWS algorithm is optimum, in the sense that the layout it would recommend if an optimum clustering algorithm were employed would produce the minimum page fault rate.

In the general case, optimum layouts are unknown and can only be determined by a prohibitively expensive, exhaustive search. There are, however, two arguments suggesting that the tailoring algorithms of the type described in the previous section tend to produce near-optimum layouts. First, some attempts at improving the performance of tailored programs further by more

Table 2. Results of program restructuring experiments.

REFERENCE		RESTRUCTURED PROGRAM	RESTRUCTURING ALGORITHM	MEMORY POLICY	PAGE FAULT RATE REDUCTION FACTOR	MEAN WORKING-SET SIZE REDUCTION FACTOR
(a)	11	AED compiler	Nearness	LRU, FIFO, Random	2-4	1.1-1.25
(b)	12	Fortran compiler	(MWS)	—	—	1.5-1.8
(c)	13	Interactive editor	Nearness	Pure working set	1.56	—
			CWS	Pure working set	1.86	—
		File system	Nearness	Pure working set	2.32	—
			CWS	Pure working set	3.60	—
(d)	15	Fortran compiler	CWS	Sampled working set	2.1-5.5	1.15-1.42
			CSWS	Sampled working set	3.2-12	1.08-1.34
		Application program	CWS	Sampled working set	1.4-16.2	0.79-0.99
			CSWS	Sampled working set	3.8-20.8	0.85-1.21
(e)	18	Pascal compiler	CSWS	Sampled working set	1.2-2.4	1.18-1.22
(f)	—	Fortran compiler	CLRU	LRU	1.1-1.9	—
			CFIFO	FIFO	1.1-2.0	—
(g)	—	Simulator	CWS	Pure working set	1.7-3.2	1.06-1.42
			MWS	Pure working set	1.1-1.75	1.22-1.54
(h)	—	Fortran compiler	MWS	Pure working set	1.16-1.32	1.3-1.65
		Application program	MWS	Pure working set	1-3	1.13-1.80

sophisticated algorithms have resulted in negligible differences obtained at high cost. Second, while both the page fault rate and the mean working set size of a program are usually decreased by restructuring, there seems to be a tradeoff between these two indices for tailored programs. Any attempt at decreasing the page fault rate of a tailored program is likely to increase its mean working set size and vice versa. For instance, the CWS algorithm normally suggests layouts having lower fault rates and larger working set sizes than those suggested by MWS (see Table 2, row g).

The performance improvement obtainable by restructuring depends also on the relative sizes of blocks and pages. In general, the smaller the blocks with respect to the pages, the better the improvement. The larger page sizes have in fact been found to increase the effectiveness of restructuring.[11,13,15]

Data dependence. During Step 2 of the restructuring procedure the program to be restructured is run so that information on its dynamic behavior can be gathered. This information is necessarily dependent on the input data used in that run. The program is then restructured according to this information, but will be executed under a variety of different sets of input data. How sensitive to the various inputs is the improvement due to the restructuring procedure?

Most of the programs for which restructuring is convenient are not very data-dependent, and for them the effectiveness of restructuring algorithms should not be expected to be data-dependent either. That this had been true of their programs was reported by Hatfield and Gerald.[11] A statistically designed experiment on a Pascal compiler[18] has confirmed these conclusions quantitatively. Some of the results of the latter study are summarized in Table 3. Also, the CWS algorithm was shown to keep, under a variety of inputs, its superiority over the nearness method in the cases of an editor and a file-system module,[13] which are probably more sensitive than compilers to their input data. To the writer's knowledge, experiments with more data-sensitive programs—for example, sort-merge packages—have not been performed yet.

Table 3. Results of an experiment on input-data dependence.[a]

	PAGE-FAULT RATE [faults/second]	
LAYOUT[b]	UNDER INPUT USED TO DETERMINE LAYOUT	UNDER OTHER INPUTS
0	34.05-37.54	
1	23.59	19.60-24.29
2	14.67	17.50-28.71
3	15.54	15.54-27.85
4	17.93	19.00-24.93
5	14.25	21.66-23.50

[a]From D. Ferrari and E. Lau, "An Experiment in Program Restructuring for Performance Enhancement," *Proc. 2nd Int. Conf. Software Engineering*, San Francisco, California, October 1976.

[b]Layout 0 is the original layout of the Pascal compile experimented with. Layouts 1-5 were obtained by restructuring the compiler using five block reference strings generated by it under five different inputs.

Portability. That a program whose behavior has been tailored to a memory policy should perform better than its nontailored versions under that policy is not surprising. But what happens when the policy changes, for instance when the program is transported to another system? Does not tailoring exceed in "tuning" a program to a policy, so that a variation in the policy may cause serious performance degradations?

Some previously unpublished experiments performed by the author have shown that this is not the case, at least for the programs and tailoring algorithms considered (see Table 4). The approaches to restructuring taken by the tailoring algorithms described in this paper are more sophisticated than others but still quite primitive. Most of their efforts, we might say, are spent improving the locality of the program and produce fully portable gains. Thus, the results resemble those we would obtain if only the extra improvements due to the tailoring could be taken away by a change of policy, and the performance could be degraded, at worst, to the level of that produced by restructuring algorithms of the nontailoring type.

Cost. Is restructuring economically convenient? An analysis of the costs of the restructuring procedure we have described shows them to concentrate mostly in the areas of block selection, program-behavior data collection, restructuring-graph construction, and clustering.

Data collection is perhaps the most difficult operation in the procedure. Generally, today's systems are not, and cannot easily be, instrumented to permit the gathering of referencing-behavior information. Instrumenting the code portions of programs for this purpose is relatively easy, but tracing data references is usually much more difficult and expensive. Interpretive execution of the program to be restructured requires an instrumented interpreter and large amounts of computer time. The problem can certainly be solved, as has been done within several research environments and in the design of the restructuring tools now available on the market. However, the designers of the next generation of systems could substantially help the collection of trace data (if they wanted to) with relatively little effort.

The partitioning of the program into blocks is usually done by the programmer but could be automatically performed by the linker in most practical cases.

The restructuring algorithm has a cost in terms of computer time which varies roughly linearly with the number of references to be examined, that is, with the duration of the program. A typical figure for the cost of the MWS algorithm, which is one of the most expensive among the algorithms described in this paper, is about 4 seconds of CDC 6400 CPU time for a string of 1,000,000 references.

The cost of the clustering algorithm described earlier has been found to increase slightly less than quadratically with the number of blocks in the program; a typical clustering time for a program partitioned into 45 blocks is about 1 second of CDC 6400 CPU time and becomes slightly more than 3 seconds for 90 blocks.

If we weigh these costs against the expected improvements and the savings they will bring about, we will easily convince ourselves that, for programs to be executed a large number of times, restructuring is very likely to be convenient even from an economical standpoint. Since a program is restructured only once (or a few times, when its users suspect that the program's usage patterns have significantly changed), there is a number of executions at which the cost of restructuring is compensated for by the savings obtained. Beyond that

Table 4. Some results of a portability experiment.

MEMORY POLICY	NUMBER OF PAGE FAULTS				
	NO RESTRUCTURING	NEARNESS	CLRU (8 page frames)	CFIFO (8 page frames)	CWS (T = 4000 refs)
Fixed partitioning					
LRU replacement					
7 page frames	1610	1028	801	904	1345
8 page frames	1034	835	699	786	795
9 page frames	785	696	596	666	552
FIFO replacement					
7 page frames	2481	1657	1502	1377	2159
8 page frames	1803	1335	1296	1114	1251
9 page frames	1394	1116	1081	965	985
Variable partitioning					
Pure working set					
3000 references	1670	1551	1434	1400	1250
4000 references	1121	1143	1066	1078	790
5000 references	980	1103	1037	1007	741

number of executions, all the advantages due to restructuring are free. This threshold depends on many factors and cannot be easily determined *a priori*. However, our feeling is that the thresholds for all the programs we have experimented with would be quite low.

It has already been mentioned that tailoring algorithms are more expensive, being more sophisticated, than other restructuring algorithms such as the nearness method. Is their additional cost worth their better performance? A general answer cannot be given. We observe, however, that a tailoring algorithm's cost is higher in the construction of the restructuring graph and possibly, but by a very minor amount, in the collection of the data (storing nearness data may be less expensive). Since the cost of restructuring-graph construction has been found to be quite reasonable even for the MWS algorithm, and is only a fraction of the total cost anyway, there does not seem to be any general reason to give up the substantially greater improvements produced by tailoring algorithms. Because of what has been said earlier about the near optimality of these algorithms, using the same arguments to advocate the introduction of more sophisticated algorithms does not seem wise, and some of our experiments have confirmed the validity of this feeling.

Conclusion

Restructuring has proved to be a viable, effective, and economical way of improving the performance of programs in virtual memory systems. In particular, the procedure outlined in this paper and those restructuring algorithms which try to tailor the behavior of a program to the ideal behavioral model assumed by the memory policy have been quite successful in a number of experiments, and constitute the basis of some commercial restructuring tools which have recently appeared on the market.

Several avenues of research remain to be explored. One of the most important, in our opinion, is the study of procedures and algorithms which can be completely automated. The goal of such research could be the implementation of "intelligent" memory hierarchies which would restructure programs and data bases without requiring any human intervention.

Another topic to be investigated is the relationship between prefetching policies and restructuring, which is particularly interesting when data base systems are to be restructured. These systems exhibit a peculiar referencing behavior, for which prefetching policies seem to be sometimes more convenient than demand-fetching ones. The development of restructuring algorithms for systems managed by prefetching policies and the incorporation of prefetching mechanisms into the intelligent memory hierarchies mentioned above could contribute to the solution of the severe performance problems which several data base systems encounter.

Even when (or if) software design methods are so sophisticated as to allow a programmer to write programs exhibiting a preassigned referencing behavior, restructuring procedures are likely to be useful as performance debugging and testing tools. ■

Acknowledgments

The support of the National Science Foundation (grant DCR74-18375) is gratefully acknowledged. The author is also indebted for many of the above ideas and results to the students who have participated in the PROGRES project—Miguel Borges, Chung Chan, David Ching, Connie Chung, Makoto Kobayashi, Edwin Lau, Mark Liu, Jehan Pâris, and Michael Pompa—and to Ruth Suzuki for her typings of the manuscript.

Domenico Ferrari is the guest editor of this issue of *Computer*. His biography appears on p. 8.

References

1. T. Kilburn, D.B.G. Edwards, M. J. Lanigan, and F. H. Sumner, "One-Level Storage System," *IRE Trans. on Electronic Computers*, Vol. EC-11 (4), April 1962, pp. 223-235.

2. L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Systems Journal*, Vol. 5 (2), 1966, pp. 78-101.

3. P. J. Denning, "The Working Set Model for Program Behavior," *CACM*, Vol. 11 (5), May 1968, pp. 323-333.

4. J. E. Morrison, "User Program Performance in Virtual Storage Systems," *IBM Systems Journal*, Vol. 12 (3), 1973, pp. 216-237.

5. A. C. McKellar and E. G. Coffman, Jr., "Organizing Matrices and Matrix Operations for Paged Memory Systems," *CACM*, Vol. 12 (3), March 1969, pp. 153-164.

6. B. S. Brown, F. G. Gustavson, and E. S. Mankin, "Sorting in a Paging Environment," *CACM*, Vol. 13 (8), August 1970, pp. 438-494.

7. L. W. Comeau, "A Study of the Effect of User Program Optimization in a Paging System," *Proc. ACM Symp. on Operating Systems Principles*, Gatlinburg, Tenn., 1967.

8. R. F. Tsao and B. H. Margolin, "A Multi-factor Paging Experiment: II. Statistical Methodology," in W. Freiberger, ed., *Statistical Computer Performance Evaluation*, Academic Press, New York, 1972, pp. 135-158.

9. J. L. Baer and G. R. Sager, "Dynamic Improvement of Locality in Virtual Memory Systems," *IEEE Trans. on Software Engineering*, Vol. SE-2 (1), March 1976, pp. 54-62.

10. C. V. Ramamoorthy, "The Analytic Design of a Dynamic Look Ahead and Program Segmenting System for Multiprogrammed Computers," *Proc. ACM National Conference*, New York, 1966, pp. 229-239.

11. D. J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory," *IBM Systems Journal*, Vol. 10 (3), 1971, pp. 168-192.

12. T. Masuda, H. Shiota, K. Noguchi, and T. Ohki, "Optimization of Program Organization by Cluster Analysis," *Information Processing 74* (Proc. IFIP Congress 74), North-Holland, Amsterdam, 1974, pp. 261-265.

13. D. Ferrari, "Improving Locality by Critical Working Sets," *CACM*, Vol. 17 (11), November 1974, pp. 614-620.

14. D. Ferrari, "Improving Program Locality by Strategy-Oriented Restructuring," *Information Processing 74* (Proc. IFIP Congress 74), North-Holland, Amsterdam, 1974, pp. 266-270.

15. D. Ferrari, "Tailoring Programs to Models of Program Behavior," *IBM Journal of Research and Development*, Vol. 19 (3), May 1975, pp. 244-251.

16. N. A. Oliver, "Experimental Data on Page Replacement Algorithm," *AFIPS Conf. Proc.*, Vol. 43 (NCC 1974), pp. 179-184.

17. Y. Bard, "Characterization of Program Paging in a Time-Sharing Environment," *IBM Journal of Research and Development*, Vol. 17 (5), September 1973, pp. 387-393.

18. D. Ferrari and E. Lau, "An Experiment in Program Restructuring for Performance Enhancement," *Proc. 2nd Int. Conf. on Software Engineering*, San Francisco, Calif., October 1976.

SOFTWARE RELIABILITY ENGINEERS

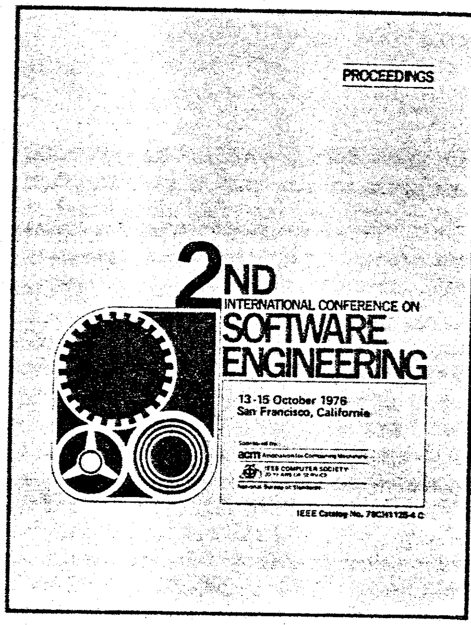
Intel Corporation, the world's foremost manufacturer of microcomputers, has openings for software engineers who want to advance the art and science of making software reliable.

If software engineering means more to you than eliminating GO TO's—if the joy of writing structured code no longer satisfies your concern for quality—then consider software product engineering at Intel. Successful applicants will analyze and approve software specifications, review the internal design of software products, manage testing projects, consult with development engineers to propose quality assurance strategies, and develop tools for the evaluation and guarantee of software reliability in all its aspects.

Candidates must have the technical understanding to implement software of superior quality and the experience to evaluate the economic consequences of reliability. Familiarity with recent software reliability theory is desired, together with an MS or PhD in Computer Science.

For immediate and confidential consideration, please send your resume, including salary history, to Intel Employment, 3065 Bowers Ave., Santa Clara, CA 95051. An Equal Opportunity Employer M/F.

intel


PROCEEDINGS
2ND INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING
13-15 October 1976
San Francisco, California

Proceedings of the 2nd International Conference on

SOFTWARE ENGINEERING

October 13-15, 1976 (700 pages)

Topics covered in this proceedings include software requirements and specifications, program synthesis, operating systems, education, performance evaluation, networks, design and development, programming languages, modeling, testing, tools, and case studies.

Non-members — $20 Members — $15