Domenico Ferrari

Tailoring Programs to Models of Program Behavior

Abstract: This paper considers the premise that, in addition to trying to solve the virtual-memory-system performance problem by devising a storage management strategy suitable for the broad spectrum of behavior exhibited by programs, efforts also be made to tailor the behavior of each program to the model underlying the storage management strategy under which the program will have to run. It is observed that a viable approach to program tailoring is offered by restructuring techniques. The application of dynamic off-line techniques to the tailoring problem is discussed, and an algorithm which may be used to fit program behavior to the working set model is described in detail as an example. The performance of this algorithm in dealing with two real-program traces is experimentally evaluated under a variety of conditions and found to be always satisfactory.

Introduction

The problem of achieving reasonable levels of performance in virtual memory systems has received and is still receiving a considerable amount of attention from system designers and performance evaluators. One of the major concerns is with devising methods for managing a storage hierarchy that is responsive to changes in the system's workload. The conventional approach to the storage management problem consists of selecting one among a number of methods based on some properties which have been empirically observed to hold (to a greater or lesser extent) for a number of programs. However, that fraction of the programs in a system's workload which does not appreciably exhibit those properties is often non-negligible. Because the system has no control over the referencing patterns of the programs it executes, the presence in the workload of programs whose behavior does not satisfy the assumptions is bound to degrade the performance of the system.

This paper takes, in some sense, the opposite viewpoint: Instead of accepting behavioral differences among programs as a necessary evil and concentrating our efforts only on improving storage management, trying to make it able to deal efficiently with a broad spectrum of behavior (which is probably a hopeless task), we should try to produce programs that are better suited to the storage management policy under which they will have to run. The purpose of this paper is to show that this result (which would be achievable by proper program design methodologies if they had been found) can be obtained by restructuring programs after they have been implemented.

Storage management strategies and models of program behavior

In virtual memory systems, especially if they are multiprogrammed, the programs to be executed usually do not fit entirely into physical main memory. Thus, programs must be "folded," i.e., divided into (generally non-disjoint) parts to be successively loaded into memory during execution. The folding of programs in a virtual memory system is done automatically by the operating system following a set of rules known as the system's *storage management strategy* [1].

The task of a storage management strategy is to decide which parts of the programs that are ready to run, at any instant, are to be in memory at that instant. The relationships between this function and the one of the scheduling policy (which has to decide the order in which programs will be executed) are very strong and intricate. For the sake of simplicity, the following discussion will assume that the number and identities of the programs to be (partially) loaded into main memory are selected by some other operating system component (for example, by the scheduler or by the general resource manager). Thus, the memory manager will only be concerned with partitioning the available main memory space among the given programs and deciding what portion of each program is to be in main memory at any given time.

In a paging implementation of virtual memory, the address spaces of programs are divided into fixed-size portions called *virtual page frames*, consisting of contiguous virtual addresses. Similarly, the physical main

memory is divided into *physical page frames* having the same sizes as their virtual counterparts. The information stored in a virtual page frame, and occasionally also in a physical page frame, is called a *page*. The page is the unit of memory allocation and of information transfer between main memory and auxiliary storage. Thus, in a paged system, the storage management strategy determines how many physical page frames are to be allocated to each program and which ones of its pages are to be stored into those frames at any given time. In other words, the storage management strategy has to solve the two interrelated problems of *size* and *page identity* for the loaded portion of each program. The objective, in solving these problems, is to maximize performance. Various performance indices are used today in computer system evaluation, the most popular being those expressing the productivity (e.g., the throughput rate) or the responsiveness (e.g., the turnaround or response time) of a system. In order to optimize one of these system-wide indices (or a function of several of them) by making decisions concerned with each individual program separately, we should be able to determine the contribution of each program to our performance index and the impact of our decisions on it. Because this is beyond the current state of our knowledge, at least for most practical systems, indices more easily expressible in terms of the behavior of individual programs are used in storage management studies. One of the most popular indices of this type is the *page fault rate* or, equivalently, the number of page faults (i.e., of pages not found by the central processor in main memory when needed) generated by a program or by an entire workload during execution. Another important index is the program's *memory utilization*: Evidently, in a space-shortage situation, when allocation is dynamic, the strategy must not waste space by keeping in main memory those pages that are no longer needed or that will be needed only in the far future.

Thus, the goal of a good storage management strategy is to predict which pages a program will need in the near future and load those pages in main memory so that the central processor will find them there at the proper time; in other words, to improve the performance index of the system to such a degree that it will approach the one it would have if the whole program could fit into main memory. If the size and page identity decisions alluded to above are made at the virtual-time instants t_1, $t_2, \cdots$, the set of pages referenced by a program during the interval (t_i, t_{i+1}) is said to be the *working information* of that program during that interval. A storage management strategy that is able to determine at t_i the working information of a program during interval (t_i, t_{i+1}) and make sure that all partially loaded programs always have just their working information in main memory will reduce to zero the total number of page faults (of course, the new pages required by the programs would have to be preloaded) and the total number of wasted physical page frames.

As known, the exact prediction of a program's working information is usually an impossible task, and storage management strategies therefore try only to estimate it. They generally do so by making assumptions about the behavior of the program in the near future. These assumptions can be viewed as defining a *model of program behavior*. In other words, a strategy is designed to work optimally with programs whose behavior is accurately represented by the model underlying the strategy, i.e., programs which satisfy the behavioral assumptions mentioned above.

For example, working-set strategies [1, 2] estimate a program's working information by computing its *working set*, defined as the set of pages referenced by the program during the interval $(t_i - T, t_i)$, where T is the *working-set parameter* or *window size*. Thus, the assumption is made that the pages to be referenced during the interval (t_i, t_{i+1}) are all and only those referenced in the backward window $(t_i - T, t_i)$. The model of program behavior underlying working-set strategies is defined by this assumption and is based on the empirical observations summarized by Denning in the principle of locality [1].

Other examples can be drawn from the class of local-replacement algorithms. For instance, the model of program behavior underlying strategies that make use of the *Least Recently Used* (LRU) replacement algorithm for each individual program assumes that the probability that a page will be referenced in the near future decreases as the time from the most recent reference to it increases [3]. Note that local-replacement algorithms help in solving the page identity problem only if the size of a program's working information has been determined or estimated in another way. In fixed-allocation strategies, for example, the size problem is solved very simply: Allocate to each program a main memory partition whose size has been selected once for all and independently of program behavior. Thus, in this case, the working information is estimated as consisting of the p most recently referenced pages, if p is the size of the partition given to the program and LRU is the replacement algorithm.

Similarly, the *First In, First Out* (FIFO) algorithm defines a model in which the reference probability of a page decreases as its "age" in main memory increases [3]. It should be noted that in a system with fixed allocation of main memory, the local-replacement strategies make decisions (i.e., estimate the identities of the pages in the program's working information) every time the program generates a page fault, that is, every time they fail to achieve their ideal goal of eliminating page faults.

Tailoring programs to models by restructuring

Restructuring a program means rearranging it in its virtual address space to improve its performance [4]. Usually, the objective of restructuring is to increase the locality of a program by making spatially contiguous those parts which are likely to be referenced in temporal proximity (in the terminology of Baer and Sager [5], to increase spatial locality).

Several restructuring techniques have been proposed in the recent past. A class of techniques that looks very promising and is attracting considerable interest is the one of *dynamic off-line restructuring methods* [4, 6–9]. The term "dynamic" is used in order to indicate that these methods base their restructuring recommendations on data, collected during execution of the program to be restructured, which describe its dynamic behavior; "off-line" means that the program is not rearranged automatically every time it runs, but once for all before being used in a production environment. A typical dynamic off-line procedure consists of the following phases [6]:

Phase 1 The program (instructions and data) is partitioned into *blocks*, i.e., sets of contiguous information items having an average size substantially smaller than the page size.

Phase 2 The program is instrumented and executed so that its *block reference string* during this execution (or equivalent information about its dynamic behavior) is recorded.

Phase 3 A *restructuring graph* of the program is derived from the information gathered in Phase 2 by applying a *restructuring algorithm*; a restructuring graph is a non-directed graph whose nodes represent the blocks of the program and whose edges have labels quantifying the desirability of grouping into the same page the two blocks which define that edge.

Phase 4 A *clustering algorithm* is applied to the restructuring graph in order to group nodes together so that the sum of the labels of edges connecting nodes belonging to different groups is minimal, and the sizes of the groups do not exceed the page size; in other words, the most desirable arrangement of blocks into pages is sought.

Phase 5 The blocks of the program are reordered in the virtual address space as suggested by the results of Phase 4, and the restructured program is used in its production environment. (Of course, this procedure does not usually pay off if the program is to be run only once or very few times.)

Restructuring techniques can be utilized to tailor the behavior of a program to a given program-behavior model. For instance, a dynamic off-line procedure can evaluate from the block reference string the working information (expressed in terms of program blocks) and rearrange the program so that its dynamic behavior will fit the one assumed by the model better than that of the non-restructured program. More specifically, because models are used by storage management strategies to estimate the working information, the restructuring algorithm will have as its objective that of making the estimation process more successful by allocating blocks to pages so as to decrease the difference between estimated and actual working information. As an example, consider the case of two blocks one of which, say block h, is at a certain instant part of the estimated working information but not of the actual working information, whereas the other, say block k, is at the same instant in the actual but not in the estimated working information. It is evident that if blocks h and k are grouped together in the same page, that page will become part of the estimated working information at that instant and, since it is also in the actual working information, this will contribute to the accuracy of the program's near-term behavior prediction.

Given a model of program behavior, the general approach outlined above suggests one or more restructuring algorithms to be used in Phase 3 of the dynamic off-line restructuring procedure in order to make a program's behavior closer to the given model and therefore more efficient under a storage management strategy based on that model. These restructuring algorithms will be called *tailoring algorithms* in the sequel. The derivation of tailoring algorithms in the case of program-behavior models underlying working-set strategies is described in the following section.

The most serious objection which may be raised against the proposal presented in this section is that, since program reference patterns depend on the input data, the tuning of a program's behavior to a model for a given set of values of the input data (which is what dynamic off-line restructuring techniques do) does not guarantee that the program will also behave consistently with the model for different sets of input data. The data presented in the experimental results section below cannot be used to respond to this question, but a new set of experiments intended to study the sensitivity of the performance of tailoring algorithms to input data variations is now being planned. Because all results of similar investigations performed on other types of restructuring algorithms have indicated that these algorithms were able to improve locality over a wide range of values of the input data [4, 6], we expect our experiments to be reasonably successful.

Some tailoring algorithms for the working set model

As mentioned in the storage management section, the working set model of program behavior identifies the working information of a program during virtual-time interval (t_i, t_{i+1}) with its working set at time t_i. Thus, storage management strategies based on the working set model estimate both the identities of the pages and the size of the working information by equating them to those of the program's working set. The working-set parameter T can be chosen in many different ways and need not be constant; however, in this paper, we shall limit our considerations to the simplest strategy in this class, the one which has a constant working-set parameter T and estimates the working information of a program periodically with a period equal to T (i.e., we assume $t_{i+1} - t_i = T$ for all i). This strategy is the one called "naive" by Bryant [10].

It should be noted that T is assumed to be constant for each program but may vary from program to program. Also, the strategy for which tailoring algorithms will be constructed has been selected for simplicity of illustration; the same conceptual approach can be applied to other strategies (based on the same or different models) as long as their working-information estimations and memory-space allocations do not depend on the total workload but only on the behavior and the properties of each individual program. This condition, which is made necessary by the fact that off-line restructuring procedures are applied to individual programs without having any knowledge of the other programs they will be running with, is satisfied by working-set strategies, which do not allow a program to execute unless its working set is entirely in main memory, thereby preventing other programs from influencing its space allotment with their memory demands.

The general principle stated in the previous section and the example given there to illustrate it suggest some very simple tailoring algorithms for the strategy described above. Let us consider two consecutive virutal-time intervals (t_{i-1}, t_i) and (t_i, t_{i+1}), both of duration T; the estimated working information during (t_i, t_{i+1}) is equal to $W(t_i, T)$, the working set at time t_i, whereas the actual working information during the same interval is $W(t_{i+1}, T)$, the working set at time t_{i+1}. The set

$$M(t_i, t_{i+1}) = W(t_{i+1}, T) - W(t_i, T) \qquad (1)$$

is called the *set of missing pages*, those which are referenced during (t_i, t_{i+1}) but are not included in the estimated working information. Each missing page causes a page fault and is assumed not to replace any page already in main memory (because none of these pages has yet dropped out of the estimated working information). Thus, each missing page will be added to the ones in memory when it is brought in and will increase, until the next measurement of the working set, the size of the program (if there is no room for its growth, the program will be suspended). Similarly, the set

$$E(t_i, t_{i+1}) = W(t_i, T) - W(t_{i+1}, T) \qquad (2)$$

contains the *excess pages*, those which are kept in memory (because they are in the estimated working information) but are not referenced during (t_i, t_{i+1}) and therefore waste memory space.

Working-information prediction would be perfect if both M and E were zero, i.e., if

$$W(t_{i+1}, T) = W(t_i, T). \qquad (3)$$

To approach this goal let us consider the sets of blocks corresponding to the ones defined so far as sets of pages: the working information, the working set W, and the sets defined in Eqs. (1) and (2); the last three will be denoted by W_b, M_b and E_b when referred to as sets of blocks. The grouping of two blocks h and k into the same page may be beneficial in the following cases:

a. $h \in W_b(t_i, T)$ and $k \in M_b(t_i, t_{i+1})$, since the page containing h and k will not be a missing page, and the references to information items in k will not cause any page faults;
b. $h \in E_b(t_i, t_{i+1})$ and $k \in W_b(t_{i+1}, T)$, since the page containing h and k will not be an excess page, and the references to information items in h will not cause any useless page to be kept in memory during (t_i, t_{i+1}).

Several different tailoring algorithms may be derived from these observations. One which is directly suggested by our discussion is the algorithm that, for each pair of consecutive intervals, computes the sets M_b and E_b and increases by 1 the labels of all the edges (p, q) with $p \in W_b(t_i, T)$ and $q \in M_b(t_i, t_{i+1})$ and also those of all the edges (r, s) with $r \in E_b(t_i, t_{i+1})$ and $s \in W_b(t_{i+1}, T)$. This algorithm, which takes care of both cases a and b above, is called the **AB** algorithm.

Simpler to implement than the **AB** algorithm is the **A** algorithm, which only computes M_b and increments the labels of the edges defined in case a, and the **B** algorithm, which only computes E_b and takes care of the edges defined in case b. The **A** algorithm is particularly simple since the appropriate labels can be incremented immediately for each new block reference encountered in scanning the string, without ever having to postpone actions, such as when E_b is to be computed.

A comparison of the performances of these three tailoring algorithms would be interesting to make and will be the subject of a future study. The purpose of the previous section and of this section in the present paper is to demonstrate the feasibility of applying restructuring techniques to the program-tailoring problem. We pro-

Table 1 Sample block reference string and corresponding page reference strings.

	2	1	2	1	8	1	2	7	8	7	8	3	4	5	6	7	2	7	5	1	2	1	8	1	8	7	8	3	4	5	6	7	2	7	5	6
NT	a	a	a	a	d	a	a	d	d	d	d	b	b	c	c	d	a	d	c	a	a	a	d	a	d	d	d	b	b	c	c	d	a	d	c	c
A	b	a	b	a	d	a	b	b	d	b	d	c	c	d	a	b	b	b	d	a	b	a	d	a	d	b	d	c	c	d	a	b	b	b	d	a
CWS	b	a	b	a	a	a	b	c	a	c	a	c	b	d	d	c	b	c	d	a	b	a	a	a	a	c	a	c	b	d	d	c	b	c	d	d

ceed with the demonstration by choosing the **A** algorithm as the tailoring algorithm whose performance is to be evaluated. This evaluation is performed by comparing tailored and non-tailored programs in order to see how much the tailored program helps the storage management strategy estimate the program's working information.

The **A** algorithm is also compared with the Critical Working Set (**CWS**) algorithm [6], which has been proposed and analyzed for restructuring purposes. It could be viewed as a "continuous" **A** algorithm, because it works in exactly the same way but with mobile intervals which follow the current reference. More precisely, the interval boundary t_i always coincides with the time at which the reference immediately preceding the current reference was issued. The strategy assumed by the **CWS** algorithm estimates the working information and makes an allocation decision at every reference. This assumption, based on the definition of working set, is obviously unrealistic and can only be approximately satisfied by most practical working-set strategies.

The performance indices according to which the comparison is made reflect the two objectives of a tailoring algorithm, namely, the reduction of the number of missing pages (i.e., page faults) and the reduction of the number of excess pages. Thus, we have chosen as our performance indices the sums of the cardinalities of the sets $M(t_i, t_{i+1})$ and $E(t_i, t_{i+1})$ over all the intervals contained in the reference string.

An example which illustrates the application of the **A** algorithm to a very simple string and its comparison with the No Tailoring (**NT**) algorithm and with the **CWS** algorithm is reported in Tables 1-4. The program that has generated the block reference string in Table 1 is assumed to contain 8 blocks numbered 1 through 8, each having the size of half a page, and reference times are assumed to be equally spaced, so that an interval of constant duration will always contain the same number of references. The labels of the edges in the restructuring graph produced by the **A** algorithm for an interval T spanning 4 references are shown in Table 2, and an optimal clustering of the graph is reported in column A of Table 3. The other columns in this table represent the grouping corresponding to the non-tailored program (**NT**) and an optimal clustering of the restructuring graph produced by the **CWS** algorithm for a window size of 4 references. The page reference strings obtained by replacing each block number by the page name given in Table 3 are shown in Table 1, and the performance indices of the three page reference strings, assuming that they are processed by the storage management strategy described earlier in this section with $T = 4$ references, are displayed in Table 4. Note the better performance of the **A** algorithm also with respect to index $\Sigma|E(t_i, t_{i+1})|$, which this algorithm does not try to minimize.

Experimental results

Some experiments on block reference strings of real programs have been performed to evaluate the tailoring abilities of the **A** algorithm. The two strings, used in the experiments, here called STR1 and STR2, were respectively produced by a compiler compiling a program and by the compiled program executing. STR1 consists of slightly more than one million references, STR2 of about 2 700 000 references. The performance indices introduced in the previous section and the mean working-set size have been measured in a simulated working-set-strategy environment on the page reference strings obtained from STR1 and STR2 by replacing block numbers by the names of the pages they were assigned to by the **NT**, **A** and **CWS** algorithms. The compiler that generated STR1 was partitioned into 46 blocks; the program that generated STR2, into 90 blocks. Two different page sizes, 512 and 1024 words, were considered in the experiments, and the fact that blocks will generally be positioned across page boundaries when the restructured program is compacted was taken into account in the simulation. Also, three different interval durations (3000, 5000 and 7000 references) were considered.

The results are reported in Tables 5 – 10. Algorithm **A** appears to be successful not only in always drastically reducing the total number of page faults with respect to the one produced by the non-tailored program, but also in almost always outperforming, sometimes substantially, the **CWS** algorithm. These conclusions are almost valid also for the total number of excess pages, with the only exception being string STR2 for $T = 7000$ references, in which case the **CWS** algorithm performs better than the **A** algorithm. The **CWS** algorithm does not have the minimization of excess pages among its objectives; however, it always reduces considerably the number of excess pages with respect to the one produced by the

nontailored program. Interpreting the measured mean working-set sizes is harder, since there seems to be no connection between them and our performance indices (which are rather related to the variance of working-set size). For STR1, for example, the **CWS** algorithm consistently produces a smaller mean working-set size than the **A** algorithm, both being substantially smaller than those of the non-tailored program. The situation is more complex in the case of STR2, in which **CWS** always produces the largest mean working-set size, and the nontailored program for two values of T and a 512-word page size has a smaller mean working-set size than the program tailored by the **A** algorithm.

When one considers the way the **A** algorithm operates on the block reference string, one may wonder whether its success, as displayed in Tables 5 – 10, is mostly due to the fact that the interval boundaries (i.e., the times t_i at which the storage management strategy estimates the working information and makes its decisions) during our simulations were the same as those considered by the **A** algorithm in the restructuring-graph construction phase. In order to investigate this problem, all of our simulations were repeated, each repetition incrementing the starting time by 1000 references, as many times as needed to explore the entire interval. For example, each simulated run with an interval of 7000 references was repeated 7 times beginning at references 1, 1001, 2001, ···, 6001, respectively. All the results obtained were similar to the sample shown in Table 11; i.e., very small differences were found to exist between the various runs, and often the **A** algorithm turned out to work slightly better for interval boundaries quite different from those assumed when the graph was constructed. The same insensitivity to interval-boundary variations was also always found with the page reference strings produced by the **NT** and **CWS** clusterings.

Conclusions

The concept of program tailoring has been proposed to help solve the virtual-memory-system performance problem. Its implications have been illustrated, and the feasibility of its implementation by dynamic off-line restructuring techniques has been demonstrated by discussing a very simple tailoring algorithm (the **A** algorithm, which applies to systems running under a pure working-set storage management strategy) and its performance as determined from experimentation with two real program traces.

Several problems are raised by the introduction of the program-tailoring concept (e.g., the possibility of devising some program design rules that would produce programs better fitted to a given behavioral model); by the idea of using restructuring techniques to tailor programs (e.g., the study of the conditions that a strategy

Table 2 Matrix representing the restructuring graph constructed by the **A** algorithm with $T = 4$ references.

	2	3	4	5	6	7	8
1	0	2	1	1	1	3	2
2		2	2	2	2	4	2
3			2	2	2	1	0
4				0	0	2	2
5					0	2	3
6						2	2
7							2

Table 3 Groupings of blocks recommended by the algorithms.

	Algorithm		
NT	**A**	**CWS**	*Page name*
1,2	1,6	1,8	a
3,4	2,7	2,4	b
5,6	3,4	3,7	c
7,8	5,8	5,6	d

Table 4 Performance comparison of the algorithms for the sample string in Table 1.

Algorithm	$\Sigma\|M(t_i, t_{i+1})\|$ *(total number of page faults)*	$\Sigma\|E(t_i, t_{i+1})\|$
NT	7	5
A	5	4
CWS	7	6

Table 5 Experimental results for STR1: total number of missing pages $\Sigma|M|$.

Interval or window size (references)	*Page size (words)*					
	512			*1024*		
	NT	**A**	**CWS**	**NT**	**A**	**CWS**
3000	212	66	98	109	9	24
5000	88	22	27	44	6	8
7000	67	16	19	32	8	12

Table 6 Experimental results for STR1: total number of excess pages $\Sigma|E|$.

Interval or window size (references)	*Page size (words)*					
	512			1024		
	NT	**A**	**CWS**	**NT**	**A**	**CWS**
3000	203	56	89	103	9	24
5000	77	19	21	36	6	8
7000	55	14	15	24	7	12

Table 7 Experimental results for STR1: mean working set size (pages).

Interval (window size) [references]	*Page size (words)* 512			1024		
	NT	**A**	**CWS**	**NT**	**A**	**CWS**
3000	10.34	9.14	8.78	7.03	5.24	4.99
5000	11.01	9.73	9.51	7.40	5.54	5.20
7000	11.22	10.13	9.62	7.48	5.88	5.80

Table 8 Experimental results for STR2: total number of missing pages $\Sigma|M|$.

Interval window size references	*Page size (words)* 512			1024		
	NT	**A**	**CWS**	**NT**	**A**	**CWS**
3000	380	37	256	146	7	9
5000	141	37	42	91	7	7
7000	91	16	18	61	7	7

Table 9 Experimental results for STR2: total number of excess pages $\Sigma|E|$.

Interval window size (references)	*Page size (words)* 512			1024		
	NT	**A**	**CWS**	**NT**	**A**	**CWS**
3000	373	39	253	149	8	10
5000	143	40	44	93	8	9
7000	94	19	15	63	10	9

Table 10 Experimental results for STR2: mean working set size (pages).

Interval window size (references)	*Page size (words)* 512			1024		
	NT	**A**	**CWS**	**NT**	**A**	**CWS**
3000	6.11	7.15	7.65	4.74	4.03	5.02
5000	7.84	7.18	8.07	4.86	4.03	5.05
7000	8.01	8.10	9.02	4.97	4.08	5.02

Table 11 Performance indices of the A algorithm for different interval boundaries (STR; $T = 7000$ references).

Performance index	*Interval beginning at reference*						
	1	1001	2001	3001	4001	5001	6001
Page size: 512 words							
$\Sigma\|M\|$	16	16	14	16	15	16	16
$\Sigma\|E\|$	14	14	12	14	13	14	14
Mean working set size	10.13	10.12	10.12	10.11	10.12	10.12	10.12
Page size: 1024 words							
$\Sigma\|M\|$	8	6	6	6	7	7	7
$\Sigma\|E\|$	7	5	5	6	6	6	6
Mean working set size	5.88	5.89	5.88	5.88	5.87	5.87	5.87

has to satisfy in order to allow a tailoring algorithm to be constructed; a condition has been stated in the section that describes the tailoring algorithms, but no extensive investigation of its sufficiency has been made); and by the algorithms proposed in that section for working-set strategies (e.g., a comparison of the performance of the **A, B** and **AB** algorithms and a study of the sensitivity of the performance of programs tailored by these algorithms to changes in the duration of the interval). Among the many other research topics which will have to be investigated for their importance, we shall only mention here the study of tailoring algorithms for fixed-allocation local-replacement strategies and for global-replacement strategies (if applicable), and the analysis of the sensitivity of tailoring-algorithm performance to variations in a program's input data.

Acknowledgment

This research was supported in part by the Joint Services Electronics Program Contract F44620-71-C-0087.

References

1. P. J. Denning, "Virtual Memory," *Computing Surveys* **2,** 153 (1970).
2. P. J. Denning, "The Working Set Model for Program Behavior," *Comm. ACM* **11,** 323 (1968).
3. L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Syst. J.* **5,** 78 (1966).
4. D. J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory," *IBM Syst. J.* **10,** 168 (1971).
5. J. L. Baer and G. R. Sager, "On the Dynamic Definition of Locality in Virtual Memory Systems," *Technical Report CS74-01*, Colorado State University, Fort Collins, Colorado (1974).
6. D. Ferrari, "Improving Locality by Critical Working Sets," *Comm. ACM* **17,** 614 (1974).
7. D. Ferrari, "Improving Program Locality by Strategy-oriented Restructuring," *Information Processing 74*, North-Holland Publishing Co., Amsterdam, 1974, p. 266.
8. 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, p. 261.
9. K. D. Ryder, "Optimizing Program Placement in Virtual Systems," *IBM Syst. J.* **13,** 292 (1974).
10. P. Bryant, "Predicting Working Set Sizes," *IBM J. Res. Develop.* **19,** 221 (1975), this issue.

Received December 13, 1974; revised December 31, 1974

Domenico Ferrari is located at the Computer Science Division, Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory, University of California, Berkeley, California 94720.