View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Purdue E-Pubs

Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1981

An Analytical Study of Strategy-Oriented Restructuring Algorithms

Jehan-François Päris

Domenico Ferrari

Report Number:

81-395

Päris, Jehan-François and Ferrari, Domenico, "An Analytical Study of Strategy-Oriented Restructuring Algorithms" (1981). *Department of Computer Science Technical Reports.* Paper 321.
https://docs.lib.purdue.edu/cstech/321

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

An Analytical Study of Strategy-Oriented Restructuring Algorithms

Jehan-François Pâris

Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907

Domenico Ferrari

Department of Electrical Engineering and Computer Sciences
Computer Sciences Division
and
Electronics Research Laboratory
University of California, Berkeley
Berkeley, CA 94720

CSD-TR-395
PROGRES Report 82.1

ABSTRACT

Considerable experimental evidence has been accumulated showing that the performance of programs in virtual memory environments can be significantly improved by *restructuring* the programs, i.e. by modifying their block-to-page or block-to-segment mapping. This evidence also points out that the so-called *strategy-oriented* algorithms, which base their decisions on the knowledge of the memory management strategy under which the program will run, are more efficient than those algorithms which do not take this strategy into account.

We present here some theoretical arguments to explain why strategy-oriented algorithms perform better than other program restructuring algorithms and determine the conditions under which these algorithms are optimum. In particular, we prove that the algorithms oriented towards the working set or sampled working set policy are optimum when applied to programs having no more than two blocks per page, and that, when this restriction is removed, they minimize an upper bound of the performance index they consider as the figure of merit to be reduced. We also prove that the restructuring algorithms aimed at reducing the page fault frequency of programs to be run under such policies as LRU, Global LRU, and PFF (the Page Fault Frequency policy) minimize a upper bound of the page fault rate, and we extend some of our results to some non-strategy-oriented algorithms. Throughout the paper, the only assumption about program behavior is that it can be accurately modeled as a stationary stochastic process.

Key Words and Phrases: virtual memory, program restructuring, restructuring algorithms, program behavior, page replacement, working set policy, sampled working set policy, LRU policy, PFF policy.

1. INTRODUCTION

Program Restructuring (P.R.) [5] is one of the various techniques aimed at improving the behavior of programs in virtual memory environments. It has the distinguishing feature of being applicable to already written programs, and operates by modifying the order according to which the various blocks of code or data constituting a program are stored in the program's virtual address space. If well conducted, this reordering will result in a new block-to-page—or block-to-segment—mapping, which will improve the degree of locality of the program.

Considerable experimental evidence has been accumulated on the performance of P.R. algorithms and this evidence clearly shows that P.R. can significantly improve the behavior of programs in both *paged* [16, 20, 9, 10, 12, 13, 23] and *segmentation* [22, 23] environments. The observations also point out that the so-called *strategy-oriented* algorithms, which base their decisions on the knowledge on the memory management strategy under which the program will run, are more efficient that those algorithms which do not take this strategy into account.

We present here some theoretical arguments to explain why strategy-oriented algorithms perform better than other program restructuring algorithms and determine the conditions under which these algorithms are optimum. Section 2 of the paper briefly reviews existing strategy-oriented restructuring algorithms. Sections 3 to 5 study their performance under various memory policies, including Working Set, Sampled Working Set, LRU , Global LRU and Page Fault Frequency. Section 6 presents some analytical results on non-strategy-oriented P.R algorithms and Section 7 contains our conclusions.

Although we shall refer in this paper to paging environments, most of our considerations could be applied to segmented systems as well [23].

2. STRATEGY-ORIENTED RESTRUCTURING ALGORITHMS

With very few exceptions (e.g., [2]), all P.R. algorithms share the same organization in four phases [10]:

(i) partitioning of the program to be restructured into *blocks*, the size of which should ideally be less than or equal to one half of the page size,

(ii) construction of a *restructuring matrix* –or restructuring graph—A, the elements of which express the "affinities" between blocks,

(iii) application of a *clustering algorithm* that tries to gather into the same page blocks exhibiting the strongest mutual affinities, and

(iv) relocation of the blocks in the program's virtual address space according to the results of the clustering algorithm.

Among these four phases, phase (ii) is definitely the most critical for both the algorithm's performance and its run-time. The first P.R. algorithms based their restructuring matrix on the analysis of the static structure of programs. Since then, they have been outclassed by the so-called *dynamic algorithms*, which take into account the run-time referencing behavior of programs. Gathering such information normally involves simulating or monitoring one or more executions of the program to be restructured and this step is often the most expensive and time-consuming part of the whole restructuring process.

All of the most efficient dynamic restructuring algorithms known today belong to the class of the so-called *strategy-oriented* algorithms introduced by one of the authors [9-11]. These algorithms construct the restructuring matrix in a manner that

(i) takes into account the *memory management strategy* of the system in which the program will be run, and

(ii) is explicitly based on a *measurable indicator* of the program's performance.

The Critical Working Set Algorithm (CWS) [10] is probably the best known example of these strategy-oriented algorithms. It attempts to minimize the page fault frequency of programs and assumes that the restructured program will be executed on a system using a working set (WS) replacement policy [7,8]. Define a *critical reference* as a reference to a block that is not guaranteed to be present in memory at that time. Under a WS policy, this will be any block that has not been referenced during the last τ time units, where τ is the value of the policy's parameter (the *window size*). If we store a block to which a critical reference is made into the same page as a block that is guaranteed to be present in memory at that time, we avoid the page fault that could have occurred otherwise.

Let $r_1, r_2, \ldots, r_n$ be a reference string collected during one execution of the program we want to restructure. Define $R_b(t)$, the resident set of blocks at time t, as being the set of blocks guaranteed to be present in memory after the t-th reference is processed. In a WS environment, $R_b(t)$ contains all blocks that have been referenced during the time interval $(t-\tau, t)$.

The restructuring matrix $C=(c_{ij})$, which has initially all zero entries, will be constructed in the following way:

```
(a) For all t from 1 to n do
        if $r_t \notin R_b(t-1)$ then
            increment by one all $c_{ij}$'s such that $i \in R_b(t)$ and $j = r_t$
        fi
    od;
(b) For all $i$ and $j<i$ do
        $c_{ij} := c_{ji} := c_{ij} + c_{ji}$
    od.
```

Other critical algorithms have been developed and tested for LRU (CLRU [13]),FIFO (CFIFO [13]), Sampled Working Set (CSWS [11, 12]), Global LRU (CPSI [14]), and Page Fault Frequency environments (CPFF [23]). They can be derived from the CWS algorithm by modifying in an appropriate manner the definition of the resident set of blocks $R_b(t)$.

Unlike critical algorithms, minimal algorithms [13] attempt to minimize the *memory occupancy* of restructured programs. To achieve this goal, they attempt to store within a common page blocks that will often be simultaneously resident in memory. Thus, the algorithm will increment by one, at fixed sampling intervals during a simulated execution of the program, all entries of the restructuring matrix corresponding to a pair (i,j) of blocks which are members of the current resident set of blocks.

Let $r_1, r_2, ..., r_n$ represent again a block reference string collected during a run of the program to be restructured. Assume that the algorithm's sampling interval is equal to K references. Then, the restructuring matrix $M=(m_{ij})$ will have initially all zero entries and will be constructed in the following way:

```
(a) For all $t$ from 1 to $n$ do
        if $t$ mod $K$ = 0 then (* sampling time *)
            increment by one all $m_{ij}$'s such that $i \in R_b(t)$ and $j \in R_b(t)$
        fi
    od;
(b) For all $i$ and all $j<i$ do
        $m_{ij} := m_{ji} := m_{ij} + m_{ji}$
    od.
```

Minimal algorithms have been developed and tested for various memory policies, including Working Set (MWS), Sampled Working Set (MSWS), Global LRU (MPSI), and Page Fault Frequency (MPFF).

The effectiveness of these algorithms obviously depends on the value of the sampling constant K. In order to avoid this problem, we will restrict ourselves for the sequel of this paper to minimal algorithms with full sampling, i.e., with $K=1$.

Strategy-oriented restructuring algorithms of a third kind have been recently introduced by one of the authors [21-23]. They are the so-called *balanced algorithms*, which attempt to minimize the *space-time product* of the programs being restructured.

The space-time product characterizes the behavior of a program in a virtual memory environment by its main memory usage expressed in space-time units, for instance in page-seconds or byte-seconds. Every page fault occurring during the execution of a program will increase the space-time product of the program by a quantity equal to the product $S(t_f)T_w$ of the program's memory occupancy $S(t_f)$ at the time t_f of the fault by the page wait time T_w. Similarly, the cost of increasing the program's memory occupancy by s memory units during a time interval Δt would be equal to $s\Delta t$ space-time units. Any restructuring algorithm attempting to minimize the space-time product of a program will have to reduce the sum of these contributions. It will thus attempt to reduce simultaneously the program's page fault frequency and its mean memory occupancy, leading thus to a more "balanced" improvement of the program's performance.

One of the difficulties encountered in the design of balanced restructuring algorithms lies in the fact that it is practically impossible to estimate at restructuring time the quantities $S(t_f)T_w$. The solution adopted consists of making the page wait time T_w constant and replacing all $S(t_f)$ by a constant factor $\hat{S}$ which is an estimate of the program's mean memory occupancy $\bar{S}$. Because of this simplification, the balanced restructuring matrix $A=(a_{ij})$ for a given program to be run under a given memory policy will always be a linear combination of the corresponding critical and minimal restructuring matrices, and one will have

$$a_{ij} = \hat{S}T_w c_{ij} + KT_m m_{ij},$$

where KT_m is the sampling interval of the minimal algorithm.

Balanced algorithms have been developed and tested for several memory policies, including Working Set (BWS), Sampled Working Set (BSWS), Global LRU (BPSI), and Page Fault Frequency (BPFF). A more complete description of these algorithms may be found in [23].

3. ANALYSIS OF THE CWS, MWS AND BWS ALGORITHMS

The traditional approach to the analytical study of the performance of P.R. algorithms implied the choice of a well defined model of program behavior in virtual memory environments like the Independent Reference

Model (IRM) [6, 1, 4], the Simple LRU Stack Model (SLRUSM) [24, 4], or the first-order Markov model [15]. Rather than restricting ourselves to one of these models, we will only assume that the behavior of the program to be restructured can be accurately described by a stochastic chain having a steady-state solution. From the practitioner's viewpoint, this assumption means that the program exhibits an essentially stable behavior, which should obviously be a prerequisite for any attempt to restructure the program.

A restructuring matrix is *not* a complete representation of all interactions between the various blocks of a program. In particular, it does not provide any information on the possible interactions involving more than two blocks. We will thus first consider the case of programs that contain *at most* two blocks per page and examine later which results can be extended to the more general case of programs having an arbitrary number of blocks per page.

3.1. Programs with No More Than Two Blocks per Page

Let us consider a program consisting of m blocks occupying a total of n pages with the restriction that no page will ever contain more than *two* blocks. We must then necessarily have $m \le 2n$.

For convenience, we would like to have always exactly two blocks per page. If this is not the case, we will add to the m original blocks $2n-m$ fictitious blocks of size 0, which will never be referenced. Since these blocks will never cause a page fault or occupy any memory space, their introduction will not alter the performance of the program. Besides, they will appear in the restructuring matrix as empty rows and empty columns without any influence on the clustering process.

Taking into account these fictitious blocks, one can assume that each page i contains two blocks with indices i_1 and i_2 respectively. The infinite sequence $r_1, \ldots, r_{t-1}, r_t, r_{t+1}, \ldots$ represents an infinite block reference string produced by the program. In a Working Set environment, the mean page fault frequency and the mean memory occupancy can be written in terms of block reference probabilities and of the probability that a given block is in the resident set of blocks $R_b(t)$, if these probabilities do indeed exist. Rather than restricting our analysis to a specific class of stochastic models, we will assume, as mentioned above, that the program's behavior can be described by a stochastic model having a steady-state solution. Under these assumptions, the steady-state probability that page i causes a fault at time t exists and is equal to the probability that either block i_1 or i_2 is referenced at time t given that neither of them is a member of $R_b(t-1)$. Thus,

$$Pr\{i \text{ causes a fault at time } t\} =$$
$$Pr[i_1=r_t \cap i_1 \notin R_b(t-1) \cap i_2 \notin R_b(t-1)]$$
$$+ Pr[i_2=r_t \cap i_2 \notin R_b(t-1) \cap i_1 \notin R_b(t-1)]\},$$

and the page fault rate f is given by

$$f = \sum_{i=1}^{n}\{Pr[i_1=r_t \cap i_1 \notin R_b(t-1) \cap i_2 \notin R_b(t-1)]$$
$$+ Pr[i_2=r_t \cap i_2 \notin R_b(t-1) \cap i_1 \notin R_b(t-1)]\}.$$

Similarly, the probability that page i is in memory at time t exists and is equal to

$$Pr[i_1 \in R_b(t) \cup i_2 \in R_b(t)]$$

The mean memory occupancy $\bar{S}$ of the program is then given by

$$\bar{S} = \sum_{i=1}^{n} Pr[i_1 \in R_b(t) \cup i_2 \in R_b(t)].$$

THEOREM I: The CWS algorithm minimizes the page fault rate of all programs whose behavior can be described by a chain having a steady-state solution and which have at most two blocks per page.

Proof:

Assume without loss of generality that each page contains exactly two blocks. In the CWS algorithm, each element c_{ij} of the restructuring matrix is then proportional to

$$Pr[i=r_t \cap i \notin R_b(t-1) \cap j \in R_b(t-1)]$$
$$+ Pr[j=r_t \cap j \notin R_b(t-1) \cap i \in R_b(t-1)].$$

By clustering two blocks per page with the objective of maximizing the sum of intra-page affinities, we attempt to find

$$\max \sum_{i=1}^{n} c_{i_1,i_2} =$$
$$\max \sum_{i=1}^{n} \{Pr[i_1=r_t \cap i_1 \notin R_b(t-1) \cap i_2 \in R_b(t-1)]$$
$$+ Pr[i_2=r_t \cap i_2 \notin R_b(t-1) \cap i_1 \in R_b(t-1)]\}.$$

This maximum is evaluated on the set of all possible block-to-page mappings, rejecting those where the sum of the sizes of the two blocks would be greater than the page size.

Observing that

$$Pr[i=r_t \cap i \notin R_b(t-1) \cap j \in R_b(t-1)] =$$
$$Pr[i=r_t \cap i \notin R_b(t-1)]$$
$$- Pr[i=r_t \cap i \notin R_b(t-1) \cap j \notin R_b(t-1)],$$

we can thus rewrite our objective function as

$$\max \sum_{i=1}^{n} \{Pr[i_1=r_t \cap i_1 \notin R_b(t-1)]$$

$$+ Pr[i_2 = r_t \cap i_2 \not\in R_b(t-1)]$$

$$- Pr[i_1 = r_t \cap i_1 \not\in R_b(t-1) \cap i_2 \not\in R_b(t-1)]$$

$$- Pr[i_2 = r_t \cap i_2 \not\in R_b(t-1) \cap i_1 \not\in R_b(t-1)]\}.$$

Since all non-negative terms are independent of the block-to-page mapping, the objective can be reformulated as

$$\min \sum_{i=1}^{n} \{Pr[i_1 = r_t \cap i_1 \not\in R_b(t-1) \cap i_2 \not\in R_b(t-1)]$$

$$+ Pr[i_2 = r_t \cap i_2 \not\in R_b(t-1) \cap i_1 \not\in R_b(t-1)]\},$$

which is equivalent to minimizing the program's page fault frequency f.

∎

THEOREM II: The MWS algorithm minimizes the mean memory occupancy of all programs whose behavior can be described by a chain having a steady-state solution and which have at most two blocks per page.

Proof:

Assume without loss of generality that each page contains exactly two blocks. In the MWS algorithm with full sampling (K=1), each element m_{ij} of the restructuring matrix is then proportional to

$$Pr[i \in R_b(t) \cap j \in R_b(t)]$$

By clustering two blocks per page with the objective of maximizing the sum of intra-page affinities, we attempt to find

$$\max \sum_{i=1}^{n} m_{i_1, i_2} = Pr[i_1 \in R_b(t) \cap i_2 \in R_b(t)]$$

Observing that

$$Pr[i \in R_b(t) \cap j \in R_b(t)] =$$

$$Pr[i \in R_b(t)] + Pr[j \in R_b(t)]$$

$$- Pr[i \in R_b(t) \cup j \in R_b(t)]$$

we can thus rewrite our objective function as

$$\max \sum_{i=1}^{n} \{Pr[i_1 \in R_b(t)] + Pr[j \in R_b(t)] - Pr[i \in R_b(t) \cup j \in R_b(t)]\}$$

Since all non-negative terms are independent of the block-to-page mapping, the objective can be reformulated as

$$\min \sum_{i=1}^{n} \{Pr[i_1 \in R_b(t) \cup i_2 \in R_b(t)]\}$$

which is equivalent to minimizing the mean memory occupancy $\bar{S}$.

■

Theorems I and II generalize the results in [18], which prove that CWS and MWS are optimal with respect to programs whose behavior can be described by an independent reference model and which have at most two blocks per page. Theorem I also extends the result obtained by Lau [17], who has proved that CWS is optimal with regard to all programs whose behavior could be described by a first-order Markov model and which have two blocks per page.

THEOREM III: The BWS algorithm minimizes a linear combination of the page fault rate and of the mean memory occupancy of all programs whose behavior can be described by a chain having a steady-state solution and which have at most two blocks per page.

Proof:

Assume without loss of generality that each page contains exactly two blocks. If c_{ij} and m_{ij} represent the generic entries of the CWS and MWS restructuring matrices, each element a_{ij} of the BWS matrix A is

$$a_{ij} = \hat{S} T_w c_{ij} + T_m m_{ij}.$$

By clustering two blocks per page with the objective of maximizing the sum of intra-page affinities, we attempt to find

$$\max \sum_{i=1}^{n} a_{i_1,i_2},$$

which is equivalent to

$$\max \sum_{i=1}^{n} \{\widehat{S}.T_w.c_{i_1,i_2} + T_m.m_{i_1,i_2}\}.$$

Using the results of Theorems I and II, we can rewrite our objective as

$$\max \sum_{i=1}^{n} \{\widehat{S}.T_w.Pr[i_1=r_t \cap i_1 \not\in R_b(t-1)]$$

$$+ \widehat{S}.T_w.Pr[i_2=r_t \cap i_2 \not\in R_b(t-1)]$$

$$- \widehat{S}.T_w.Pr[i_1=r_t \cap i_1 \not\in R_b(t-1) \cap i_2 \not\in R_b(t-1)]$$

$$- \widehat{S}.T_w.Pr[i_2=r_t \cap i_2 \not\in R_b(t-1) \cap i_1 \not\in R_b(t-1)]$$

$$+ T_m.Pr[i_1 \in R_b(t)]$$

$$+ T_m.Pr[i_2 \in R_b(t)]$$

$$- T_m.Pr[i_1 \in R_b(t) \cup i_2 \in R_b(t)]\}$$

Observing again that all positive terms of the summation do not depend on the block-to-page mapping, we can reformulate our objective as

$$\min \sum_{i=1}^{n} \{\widehat{S}.T_w.Pr[i_1=r_t \cap i_1 \not\in R_b(t-1) \cap i_2 \not\in R_b(t-1)]$$

$$+ \widehat{S}.T_w.Pr[i_2=r_t \cap i_2 \not\in R_b(t-1) \cap i_1 \not\in R_b(t-1)]$$

$$+ T_m.Pr[i_1 \in R_b(t) \cup i_2 \in R_b(t)]\},$$

which is equivalent to

$$\min \widehat{S}.T_w.f + T_m \bar{S},$$

where f stands for the program's page fault frequency and $\bar{S}$ for its mean memory occupancy.

3.2. Programs with an Arbitrary Number of Blocks per Page

Since the restructuring graph only takes into account interactions between two blocks; the problem of defining affinities among more than two blocks will always remain without a completely satisfactory solution.

Consider, for instance, the case of a critical restructuring algorithm like CWS. As we said before, the affinity c_{ij} between two blocks i and j is equal to the number of page faults that could be avoided if the two blocks i and j were stored into the same page. Suppose now that we want to compute the affinity c_{ijk} among the three blocks i, j and k. Obviously, c_{ijk} should be equal to the total number of page faults that could be avoided by storing blocks i, j and k into the same page.

It could happen that none of the expected beneficial effects of the restructuring process would overlap, that is, that

- storing blocks i and j into the same page would not avoid any of the page faults that would be avoided if i or j were stored with k, and
- storing blocks j and k into the same page would not avoid any of the page faults that would be avoided if j or k were stored with i.

In this case, the affinity c_{ijk} should be set equal to the sum of all affinities between all pairs of blocks in $\{i, j, k\}$ We would then have

$$c_{ijk} = c_{ij} + c_{jk} + c_{ki}$$

and we would then speak of *additive* affinities.

However, it could also happen that some of the page faults that would be avoided if i were stored with j or k could also be eliminated by storing j and k together. Then

$$c_{ijk} < c_{ij} + c_{jk} + c_{ki}$$

In the general case, we have

$$c_{ijk} \le c_{ij} + c_{jk} + c_{ki},$$

and no means to estimate $c_{ij} + c_{jk} + c_{ki} - c_{ijk}$.

From the practitioner's viewpoint, the simplest solution consists of assuming that affinities will always add up and defining the affinities among s blocks $i_1, i_2, \ldots, i_s$ as being equal to

$$c_{i_1, \ldots, i_s} = \sum_{j=1}^{s-1} \sum_{k=j+1}^{s} c_{i_j i_k}.$$

Similar problems also arise with minimal and balanced algorithms and, there too, the simplest solution will be to assume that affinities are additive.

In all three cases, when the restructuring algorithm assumes that all beneficial effects of the restructuring process always add up, it may be construed as being essentially "overoptimistic". Since the algorithm attempts to maximize an optimistic estimate of the beneficial effects of the new block-to-page mapping, it tends to minimize some *lower* bound of its performance index. Note that all performance indices considered by strategy-oriented algorithms are indices to be minimized. We want to show now that this would also cause the algorithm to minimize a relatively weak *upper* bound of the same performance index. In all cases, we will suppose that the program to be restructured consists of m blocks of sizes $s_1, s_2, \ldots, s_m$. After restructuring, these m blocks will be partitioned into n clusters $K_1, K_2, \ldots, K_n$ such that

$$\sum_{j \in K_i} s_j \le s_p \qquad i = 1, 2, \ldots, n,$$

where s_p is the system's page size, in order to allow each cluster to be stored in a single page.

THEOREM IV: The CWS algorithm with additive affinities minimizes both an upper bound and a lower bound of the page fault rate for all programs whose behavior can be described by a stochastic chain having a steady-state solution.

Proof:

Suppose that we apply the CWS algorithm to a program whose behavior can be described by a stochastic chain having a steady-state solution. The result of the restructuring process will be a partition of the program into n clusters of blocks that will maximize

$$\sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k > j} c_{jk}$$

over the set of all possible block-to-cluster mapping.

This last condition can be rewritten as

$$\max \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k \neq j} Pr[r_t = j \cap j \notin R_b(t-1) \cap k \in R_b(t-1)]. \qquad (1)$$

The page fault frequency of the program after restructuring will then be equal to

$$f = \sum_{i=1}^{n} \sum_{j \in K_i} Pr[r_t = j \cap \bigcap_{k \in K_i} k \notin R_b(t-1)],$$

which can be rewritten as

$$f = \sum_{i=1}^{n} \sum_{j \in K_i} Pr[r_t = j \cap j \notin R_b(t-1)] \qquad (2)$$

$$- \sum_{i=1}^{n} \sum_{j \in K_i} Pr[r_t = j \cap j \notin R_b(t-1) \cap \bigcup_{k \in K_i, k \neq j} k \in R_b(t-1)].$$

The first double sum on the right-hand side of the last equation is equal to the frequency of critical references and does not depend on the block-to-cluster mapping. The second double sum,

$$\sum_{i=1}^{n} \sum_{j \in K_i} Pr[r_t = j \cap j \notin R_b(t-1) \cap \bigcup_{k \in K_i, k \neq j} k \in R_b(t-1)], \tag{3}$$

then represents the sum of the frequencies of all page faults that have been avoided because of the new block-to-cluster (and thence block-to-page) mapping. Maximizing this expression would thus result in minimizing the program page fault frequency.

Upper bounds and lower bounds for (3) are respectively given by

$$\sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k \neq j} Pr[r_t = j \cap j \notin R_b(t-1) \cap k \in R_b(t-1)] \tag{4}$$

and

$$\sum_{i=1}^{n} \sum_{j \in K_i} \frac{1}{\tau - 1} \sum_{k \in K_i, k \neq i} Pr[r_t = j \cap j \notin R_b(t-1) \cap k \in R_b(t-1)], \tag{5}$$

where τ is equal to the maximum number of blocks per cluster.

Since CWS maximizes (1), it also maximizes (4) and (5), which are respectively upper and lower bounds of the beneficial effects of the restructuring process. As a result, it minimizes a lower bound of the page fault frequency given by

$$f_{\min} = \sum_{i=1}^{n} \sum_{j \in K_i} Pr[r_t = j \cap j \notin R_b(t-1)]$$

$$- \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k \neq j} Pr[r_t = j \cap j \notin R_b(t-1) \cap k \in R_b(t-1)],$$

and an upper bound of the same page fault frequency given by

$$f_{\max} = \sum_{i=1}^{n} \sum_{j \in K_i} Pr[r_t = j \cap j \notin R_b(t-1)]$$

$$- \sum_{i=1}^{n} \sum_{j \in K_i} \frac{1}{\tau - 1} \sum_{k \in K_i, k \neq i} Pr[r_t = j \cap j \notin R_b(t-1) \cap k \in R_b(t-1)].$$

■

COROLLARY I: Consider a program whose behavior can be described by a stochastic chain having a steady-state solution. If this program is running under a Working Set policy with a given block-to-page mapping $(K_1, K_2, \ldots, K_n)$, its page fault frequency will be bounded by

$$f_{\min} = \sum_{i=1}^{n} \sum_{j \in K_i} Pr[r_t = j \cap j \notin R_b(t-1)] - \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k > j} c_{jk}$$

and by

$$f_{\max} = \sum_{i=1}^{n} \sum_{j \in K_i} Pr[r_t = j \cap j \notin R_b(t-1)] - \sum_{i=1}^{n} \sum_{j \in K_i} \frac{1}{r-1} \sum_{k \in K_i, k > i} c_{jk},$$

where $C=(c_{ij})$ is the CWS restructuring matrix for that program and for the current window size.

∎

THEOREM V: The MWS algorithm with additive affinities minimizes both an upper bound and a lower bound of the mean memory occupancy for all programs whose behavior can be described by a stochastic chain having a steady-state solution.

Proof:

Suppose that we apply the MWS algorithm with additive affinities to a program whose behavior can be described by a stochastic chain having a steady-state solution. The result of the restructuring process will be a partition of the program into n clusters of blocks that will maximize

$$\sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k > j} m_{jk} = \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k > j} Pr[j \in R_b(t) \cap k \in R_b(t)] \qquad (6)$$

over the set of all possible block-to-cluster mappings.

The program's mean memory occupancy will then be equal to

$$\bar{S} = \sum_{i=1}^{n} Pr[\bigcup_{j \in K_i} j \in R_b(t)],$$

which can be rewritten as

$$\bar{S} = \sum_{i=1}^{n} \sum_{j\in K_i} Pr[j\in R_b(t)] - \sum_{i=1}^{n}\sum_{j\in K_i} Pr[\bigcup_{k\in K_i, k>j} j\in R_b(t)\cap k\in R_b(t)]. \qquad (7)$$

The first double sum on the right-hand side of the last equation does not depend on the block-to-cluster mapping. The second double sum,

$$\sum_{i=1}^{n}\sum_{j\in K_i} Pr[\bigcup_{k\in K_i, k>j} j\in R_b(t)\cap k\in R_b(t)], \qquad (8)$$

represents the average memory space that would be saved if the new block-to-cluster (and thence block-to-page) mapping was adopted. Maximizing this expression would thus result in minimizing the program's mean memory occupancy.

Upper bounds and lower bounds for (8) are respectively given by

$$\sum_{i=1}^{n} \sum_{j\in K_i} \sum_{k\in K_i, k>j} Pr[j\in R_b(t)\cap k\in R_b(t)] \qquad (9)$$

and

$$\sum_{i=1}^{n} \sum_{j\in K_i} \frac{1}{r-1} \sum_{k\in K_i, k>j} Pr[j\in R_b(t)\cap k\in R_b(t)], \qquad (10)$$

where r is equal to the maximum number of blocks per cluster.

Since MWS maximizes (6), it also maximizes (9) and (10), which are respectively upper and lower bounds of the beneficial effects of the restructuring process. As a result, it minimizes a lower bound of the mean memory occupancy given by

$$\bar{S}_{\min} = \sum_{i=1}^{n} \sum_{j\in K_i} Pr[j\in R_b(t)] - \sum_{i=1}^{n} \sum_{j\in K_i} \sum_{k\in K_i, k>j} Pr[j\in R_b(t)\cap k\in R_b(t)],$$

and an upper bound of the same mean memory occupancy given by

$$\bar{S}_{\max} = \sum_{i=1}^{n}\sum_{j\in K_i} Pr[j\in R_b(t)] - \sum_{i=1}^{n} \sum_{j\in K_i} \frac{1}{r-1} \sum_{k\in K_i, k>j} Pr[j\in R_b(t)\cap k\in R_b(t)].$$

■

COROLLARY II: Consider a program whose behavior can be described by a stochastic chain having a steady- state solution. If this program is running under a Working Set policy with a given block-to-page mapping $(K_1, K_2, \ldots, K_n)$, its mean memory occupancy will be bounded by

$$\bar{S}_{\min} = \sum_{i=1}^{n} \sum_{j \in K_i} Pr[j \in R_b(t)] - \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k>j} m_{jk}$$

and

$$\bar{S}_{\max} = \sum_{i=1}^{n} \sum_{j \in K_i} Pr[j \in R_b(t)] - \sum_{i=1}^{n} \sum_{j \in K_i} \frac{1}{r-1} \sum_{k \in K_i, k>j} m_{jk},$$

where $M=(m_{ij})$ is the MWS restructuring matrix for that program and for the current window size.

∎

THEOREM VI: The BWS algorithm with additive affinities minimizes both a lower and an upper bound of the same linear combination of the page fault rate and of the mean memory occupancy of all programs whose behavior can be described by a chain having a steady-state solution.

Proof:

If c_{ij} and m_{ij} represent the generic entries of the CWS and MWS restructuring matrices, each element a_{ij} of the BWS matrix A is

$$a_{ij} = \hat{S}T_w c_{ij} + T_m m_{ij}.$$

Our objective,

$$\max \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k>j} a_{jk},$$

can thus be rewritten as

$$\max \ \{\hat{S}T_w \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k>j} c_{jk} + T_m \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k>j} m_{jk}\},$$

which is equivalent to

$$\max \ \{\hat{S}T_w \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k \neq j} Pr[r_t = j \cap j \notin R_b(t-1) \cap k \in R_b(t-1)]$$

$$+ \; T_m \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k>j} Pr[j \in R_b(t) \cap k \in R_b(t)]\}. \qquad (11)$$

Consider now the expression

$$\hat{S}T_w f \; + \; T_m . \bar{S} \qquad (12)$$

which is a linear combination of the program's page fault frequency f and its mean memory occupancy $\bar{S}$. Using equations (2) and (7), it can be rewritten as

$$\hat{S}T_w \sum_{i=1}^{n} \sum_{j \in K_i} Pr[r_t = j \cap j \notin R_b(t-1)]$$

$$- \; \hat{S}T_w \sum_{i=1}^{n} \sum_{j \in K_i} Pr[r_t = j \cap j \notin R_b(t-1) \cap \bigcap_{k \in K_i, k \neq j} k \in R_b(t-1)]$$

$$+ \; T_m \sum_{i=1}^{n} \sum_{j \in K_i} Pr[j \in R_b(t)]$$

$$- \; T_m \sum_{i=1}^{n} \sum_{j \in K_i} Pr[\bigcup_{k \in K_i, k>j} j \in R_b(t) \cap k \in R_b(t)],$$

where all positive terms do not depend on the block-to-page mapping. Upper bounds and lower bounds for (12) are then given by

$$\hat{S}T_w \sum_{i=1}^{n} \sum_{j \in K_i} Pr[r_t = j \cap j \notin R_b(t-1)]$$

$$- \; \hat{S}T_w \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k != j} Pr[r_t = j \cap j \notin R_b(t-1) \cap k \in R_b(t-1)]$$

$$+ \; T_m \sum_{i=1}^{n} \sum_{j \in K_i} Pr[j \in R_b(t)]$$

$$- \; T_m \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k>j} Pr[j \in R_b(t) \cap k \in R_b(t)], \qquad (13)$$

and

$$\hat{S}T_w \sum_{i=1}^{n} \sum_{j \in K_i} Pr[r_t = j \cap j \notin R_b(t-1)]$$

$$- \; \frac{1}{r-1} \hat{S}T_w \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k != j} Pr[r_t = j \cap j \notin R_b(t-1) \cap k \in R_b(t-1)]$$

$$+ T_m \sum_{i=1}^{n} \sum_{j \in K_i} Pr[j \in R_b(t)]$$

$$- \frac{1}{\tau - 1} T_m \sum_{i=1}^{n} \sum_{j \in K_i} \sum_{k \in K_i, k > j} Pr[j \in R_b(t) \cap k \in R_b(t)], \qquad (14)$$

where τ is equal to the maximum number of blocks per cluster.

Since BWS maximizes (11), it maximizes the sum of all negative terms in (13) and in (14) and thus minimizes an upper bound and a lower bound of (12).

∎

4. ANALYSIS OF THE CSWS, MSWS AND BSWS ALGORITHMS

For convenience of implementation, the Working Set policy can be approximated by measuring the working set periodically instead of at every reference. This replacement algorithm is known as the *Sampled Working Set*, or SWS. We will restrict ourselves to the case where the sampling interval I is a submultiple of the window size T. In other words, $T=kI$, with k integer. The SWS algorithm works then in the following way: Each time a page fault occurs, the missing page is added to the program's resident set of pages. At the end of each sampling interval, all pages that have not been referenced during the last k sampling intervals are expelled from memory. As a result, the program's resident set of pages will then only contain those pages that have been referenced at least once during the last $kI=T$ time units. As program execution resumes, the size of this window will increase linearly with time until it reaches $T+I$ time units at the end of the next sampling period. The Sampled Working Set policy thus behaves as a pure Working Set algorithm whose window size periodically varies between T and $T+I$ with a period I. Let us denote by $\tau(t)$ this instantaneous window size. One has then

$$\tau(t) = T + t \bmod I$$

where $t \bmod I$ is the remainder of the division of t by I.

Assume now that the behavior of the program we want to analyze can be described by a Markov model with a steady-state solution and let us denote by $W_b(t;\tau)$ its resident set of blocks at time t under a pure Working Set policy with window size τ. The resident set of blocks at time t for the SWS policy is then given by

$$R_b(t) = W_b(t;T+t \bmod I) \tag{15}$$

which shows that $R_b(t)$ oscillates between $W_b(t;T)$ and $W_b(t;T+I)$ following a sawtooth curve. A program running under a SWS policy will thus exactly behave as if it were running under a WS policy with a window size $\tau(t)$ varying between T and $T+I$ according to a sawtooth pattern. Once the program reaches the steady state, the probabilities of referencing, or not referencing, any given page do not depend any more on the time elapsed since the program's inception and are thus totally independent of the current value of $\tau(t)$. The probability that block i causes a critical reference at time t is then fluctuating between

$$Pr[i=r_t \cap i \notin W_b(t;T)]$$

and

$$Pr[i=r_t \cap i \notin W_b(t;T+I)]$$

and is thus time-dependent. Following an approach similar to the one of Marshall and Nute [19], one may however introduce a *time-averaged* probability that i causes a critical reference, equal to the average probability that the same block would cause a critical reference if the program were running under a WS policy with window sizes τ uniformly distributed on the interval $[T, T+I)$. In other words, we have

$$Pr[i \text{ causes a critical reference}] = \frac{1}{I}\int_{T}^{T+I} Pr[i=r_t \cap i \notin W_b(t;\tau)]\, d\tau$$

where r_t is the t-th page reference and $W_b(t;\tau)$, contains all blocks that have been referenced during the last τ time units.

Assuming again that each page contains exactly two blocks, the *time-averaged page fault frequency f* is then given by

$$f = \sum_{i=1}^{n} \frac{1}{I}\int_{T}^{T+I} \{Pr[i_1=r_t \cap i_1 \notin W_b(t;\tau) \cap i_2 \notin W_b(t;\tau)]$$

$$+ Pr[i_2=r_t \cap i_1 \notin W_b(t;\tau) \cap i_2 \notin W_b(t;\tau)]\}\, d\tau$$

for any t large enough to offset the influence of the initial conditions.

THEOREM VII: The CSWS algorithm minimizes the page fault rate of all programs whose behavior can be described by a chain having a steady-state solution and which have at most two blocks per page.

Proof:

Assume without loss of generality that each page contains exactly two blocks. Let $C = (c_{ij})$ be the restructuring matrix constructed by the CSWS algorithm. Each element c_{ij} of that matrix is proportional to

$$\frac{1}{I}\int_{t}^{t+I} \{Pr[i=r_\tau \cap i \notin R_b(\tau-1) \cap j \in R_b(\tau-1)]$$

$$+ Pr[j=r_\tau \cap j \notin R_b(\tau-1) \cap i \in R_b(\tau-1)]\}\, d\tau$$

for any t large enough to offset the influence of the initial conditions.

Since

$$R_b(t) = W_b(t; T+t \bmod I),$$

the expression can be rewritten as

$$\frac{1}{I}\int_{T}^{T+I} \{Pr[i=r_t \cap i \not\in W_b(t-1;\tau) \cap j \in W_b(t-1;\tau)]$$

$$+ Pr[j=r_t \cap j \not\in W_b(t-1;\tau) \cap i \in W_b(t-1;\tau)]\}d\tau.$$

By clustering two blocks per page with the objective of maximizing the sum of intra-page affinities, we attempt to find

$$\max \sum_{i=1}^{n} c_{i_1,i_2} =$$

$$\max \sum_{i=1}^{n} \frac{1}{I}\int_{T}^{T+I} \{Pr[i_1=r_t \cap i_1 \not\in W_b(t-1;\tau) \cap i_2 \not\in W_b(t-1;\tau)]$$

$$+ Pr[i_2=r_t \cap i_2 \not\in W_b(t-1;\tau) \cap i_1 \not\in W_b(t-1;\tau)]\}d\tau.$$

This maximum is evaluated on the set of all possible block-to-page mappings, rejecting those where the sum of the sizes of the two blocks would be greater than the page size.

Observing that

$$Pr[i=r_t \cap i \not\in W_b(t-1;\tau) \cap j \in W_b(t-1;\tau)] =$$

$$Pr[i=r_t \cap i \not\in W_b(t-1;\tau)]$$

$$- Pr[i=r_t \cap i \not\in W_b(t-1;\tau) \cap j \not\in W_b(t-1;\tau)]$$

and removing all terms that do not depend on the block-to-page mapping, we can reformulate our objective as

$$\min \sum_{i=1}^{n} \frac{1}{I}\int_{T}^{T+I} \{Pr[i_1=r_t \cap i_1 \not\in W_b(t-1;\tau) \cap i_2 \not\in W_b(t-1;\tau)]$$

$$+ Pr[i_2=r_t \cap i_2 \not\in W_b(t-1;\tau) \cap i_1 \not\in W_b(t-1;\tau)]\}d\tau,$$

which is equivalent to minimizing the program's page fault frequency f.

■

From (15), we can also infer that the program's *time-averaged memory occupancy* $\bar{S}$ is given by

$$\bar{S} = \sum_{i=1}^{n} \frac{1}{I}\int_{T}^{T+I} Pr[i_1 \in W_b(t;\tau) \cup i_2 \in W_b(t;\tau)]d\tau.$$

We then have the following theorem.

THEOREM VIII: The MSWS algorithm minimizes the mean memory occupancy of all programs whose behavior can be described by a chain having a steady-state solution and which have at most two blocks per page.

Proof:

Assume without loss of generality that each page contains exactly two blocks. Each element m_{ij} of the MSWS restructuring matrix is then proportional to

$$\frac{1}{I}\int_{t}^{t+I} Pr[i \in R_b(\tau) \cap j \in R_b(\tau)]d\tau,$$

which can be rewritten as

$$\frac{1}{I}\int_{T}^{T+I} Pr[i \in W_b(t;\tau) \cap j \in W_b(t;\tau)]d\tau.$$

By clustering two blocks per page with the objective of maximizing the sum of intra-page affinities, we attempt to find

$$\max\sum_{i=1}^{n} m_{i_1,i_2} = \max\sum_{i=1}^{n} \frac{1}{I}\int_{T}^{T+I} Pr[i_1 \in W_b(t;\tau) \cap i_2 \in W_b(t;\tau)]d\tau.$$

Observing that

$$Pr[i \in W_b(t;\tau) \cap j \in W_b(t;\tau)] =$$
$$Pr[i \in W_b(t;\tau)] + Pr[j \in W_b(t;\tau)]$$
$$- Pr[i \in W_b(t;\tau) \cup j \in W_b(t;\tau)]$$

and deleting all terms that do not depend on the block-to-page mapping, we can reformulate our objective as

$$\min\sum_{i=1}^{n} \frac{1}{I}\int_{T}^{T+I} \{Pr[i_1 \in W_b(t;\tau) \cup i_2 \in W_b(t;\tau)]\}d\tau,$$

which is equivalent to minimizing the mean memory occupancy $\bar{S}$.

■

THEOREM IX: The BSWS algorithm minimizes a linear combination of the page fault rate and of the mean memory occupancy of all programs whose behavior can be described by a chain having a steady-state solution and which have at most two blocks per page.

Proof:

Similar to the one of Theorem III but based on the proofs and the results of Theorems VII and VIII.

■

Using the same approach, one could also consider the case of programs having an arbitrary number of blocks per page and show that CSWS, MSWS and BSWS then minimize lower and upper bounds of their objectives.

5. EXTENSION TO OTHER MEMORY POLICIES

As the reader has probably noticed, the proofs of the optimality of CWS and MWS did not take into account the composition of the resident set of blocks $R_b(t)$ for the Working Set policy. These proofs thus hold for any strategy-oriented restructuring algorithm minimizing the same performance indices as long as

[i] the probability that a block i belongs to the resident set of blocks at time t, $Pr[i \in R_b(t)]$, has a stationary distribution for all blocks;

[ii] the probability that a page resides in memory is equal to the probability that at least one of the blocks it contains belongs to the current resident set of blocks; in other words,

$$Pr[\text{page } i \text{ in memory}] = Pr[\bigcup_{k \in K_i} k \in R_b(t)].$$

This second condition is the more restrictive: it assumes that the probability that a page resides in memory does not depend on the composition of the other pages. This is not true for the FIFO, LRU, Global LRU and PFF replacement policies and, more generally, for all policies where replacement decisions are (or may be) triggered by the occurrences of page faults.

For the LRU, Global LRU and PFF policies, it is however possible to construct resident sets of blocks $R_b(t)$ such that all pages containing at least one block belonging to the current resident set of blocks will necessarily reside in memory, while some pages residing in memory may not contain any block belonging to the set. One has thus

$$Pr[\text{page } i \text{ in memory}] \geq Pr[\bigcup_{k \in K_i} k \in R_b(t)\ ].$$

As a consequence the page fault rates f generated by these policies have an upper bound $f_{\max}$ given by

$$f_{\max} = \sum_{i=1}^{n} \sum_{j \in K_i} Pr[j = r_t \cap \bigcap_{k \in K_i} k \notin R_b(t)\ ].$$

One has then the following theorems.

THEOREM X: CLRU and CPFF minimize an upper bound of the page fault rate of all programs running under the corresponding memory policy provided that the behavior of the program in that environment can be described by a stochastic chain having a steady-state solution.

Proof:

Similar to the one of Theorem IV but with $f_{\max}$ replacing f.

■

THEOREM XI: CPSI minimizes an upper bound of the page fault rate of all programs running under a Global LRU memory policy provided that the behavior of the program in that environment can be described by a stochastic chain having a steady-state solution, and that the Global LRU environment in which the program is to run can be modeled by Bard's Page Survival Index model [3].

Proof:

Similar to the one of Theorem IV but with f_{max} replacing f.

∎

Theorem X generalizes a similar finding made by Lau [17, 18] for the CLRU algorithm under IRM program behavior assumptions.

Unfortunately, the same approach cannot be applied to minimal algorithms. Since some pages may be resident in memory without containing any block belonging to the current resident set of blocks, one could only compute a *lower* bound for the mean memory occupancy $\bar{S}$. One could therefore only prove that MPSI and MPFF minimize a lower bound of the program's mean memory occupancy. Results for BPSI and BPFF would be even weaker.

6. EXTENSION TO OTHER RESTRUCTURING ALGORITHMS

The same approach can also be applied to non-strategy-oriented restructuring algorithms, provided they define implicitly or explicitly the equivalent of a resident set of blocks.

Hatfield and Gerald's Nearness method is one example of such algorithms [16]: it implicitly assumes that all references are critical and is therefore essentially equivalent to a CWS algorithm with a window size τ equal to one reference, or to a MWS algorithm tuned for a window size equal to two references. One can thus state that the Nearness Method minimizes a very weak upper bound of the program's page fault frequency as well as a very weak lower bound of its mean memory occupancy.

Another non-strategy-oriented restructuring algorithm, proposed by Masuda *et al.* [20], attempts to minimize the *working set size* of the program to be restructured for an arbitrary window size τ^*. This algorithm operates like MWS but with a "wrong" value of the memory policy parameter τ. When applied to a program to be run in a working set environment, it will therefore minimize a upper bound of the program's mean memory occupancy if $\tau^* > \tau$, and a lower bound of this memory occupancy if $\tau^* < \tau$.

7. CONCLUSIONS

We have presented in this paper some analytical results concerning the performance of strategy-oriented program restructuring algorithms in paging environments. These results essentially correlate the performance of a restructuring algorithm with its ability to predict the influence of any block-to-page mapping on the performance of the program to be restructured. These findings corroborate all the experimental evidence collected to date, showing that restructuring algorithms taking into account the characteristics of the environment under which the program will run significantly outperform the restructuring algorithms which ignore that environment.

Acknowledgements

The work reported here was supported in part by the NSF grant MCS 80-12900.

References

[1] A. V. Aho, P. J. Denning and J. D. Ullman, "Principles of Optimal Page Replacement," *J. ACM* 18, 1 (Jan. 1971), 80-93.

[2] J .Y. Babonneau, M. S. Achard, G. Morisset and M. B. Mounajjed, "Automatic and General Solution to the Adaptation of Programs in a Paging Environment," *Proc. 6th. ACM Symp. on Oper. Sys. Prin.* (Nov. 1977), 109-116.

[3] Y. Bard, "Characterization of Program Paging in a Time-sharing Environment," *IBM J. Res. Develop.* 17, (Sept. 1973), 387-393.

[4] E. G. Coffman and P. J. Denning, *Operating Systems Theory*, (Prentice-Hall, Englewood Cliffs, NJ, 1973).

[5] L. Comeau, "A Study of the Effect of User Program Optimization in a Paging System," *ACM Symp. on Oper. Sys. Prin.*, (Oct. 1967), Gatlinburg, Tenn.

[6] P. J. Denning, "Memory Allocation in Multiprogrammed Computer Systems," MIT Project MAC, Computation Structures Group Memo 24, (Mar. 1966).

[7] P. J. Denning, "The Working Set Model for Program Behavior," *Comm. ACM* 11, 5 (May 1968), 323-333.

[8] P. J. Denning, "Working sets Past and Present," *IEEE Trans. Softw. Engrg.* SE-6, 1 (Jan. 1980), 64-84.

[9] D. Ferrari, "Improving Program Locality by Strategy-Oriented Restructuring," *Information Processing 74*, Proc. 1974 IFIP Congress, pp. 266-270.

[10] D. Ferrari "Improving Localities by Critical Working Sets," *Comm. ACM* 17 , 11 (Nov. 1974), 614-620.

[11] D. Ferrari, "Tailoring Programs to Models of Program Behavior," *IBM J. Res. Develop.* 19, 3 (May 1975), 244-251.

[12] D. Ferrari and E. Lau, "An Experiment in Program Restructuring for Performance Enhancement," *Proc. 2nd Int. Conf. on Software Engineering*, San Francisco, Calif. (Oct. 1976), pp.203-206.

[13] D. Ferrari, "The Improvement of Program Behavior," *Computer* 9, 11 (Nov. 1976), 39-47.

[14] D. Ferrari and M. Kobayashi, "Program Restructuring for Global LRU Environment," *Conf. Proc. of Int. Computing Symp.*, Liège, Belgium, April 4-7, 1977.

[15] M. A. Franklin and R. K. Gupta, "Computation of Page Fault Probability from Program Transition Diagram," *Comm. ACM* 17, 4 (Apr. 1974), 187-191.

[16] D. J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory," *IBM Sys. J.* 10 , 11 (Nov 1971), 39-47.

[17] E. Lau, "Performance Improvement of Virtual Memory Systems by Restructuring and Prefetching," Ph. D. Dissertation, Department of EECS, University of California, Berkeley, 1979.

[18] E. Lau J. and D. Ferrari, "Program Restructuring in a Multilevel Virtual Memory," PROGRES Report 81.2 & Memorandum No. UCB/ERL M81/26, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, May 1981.

[19] W. T. Marshall and C. T. Nute, "Analytical Modelling of 'Working Set Like' Replacement Algorithms," 1979 Conf. on Simulation, Measurement and Modeling of Computer Syst., 65-72.

[20] T. Masuda, H. Shiota, K. Noguchi, and T. Ohki, "Optimization of Program Performance by Cluster Analysis," *Information Processing 74*, Proc. IFIP 1974 Congress, 226-270.

[21] J.-F. Paris, "Strategies Optimales en Restructuration de Programmes," R. P. 14/76, Institut d'Informatique, Facultes Universitaires de Namur.

[22] J.-F. Paris, "Program Restructuring in Segmenting Environments," in: D. Ferrari and M. Spadoni eds., *Experimental Computer Performance Evaluation* (North-Holland, Amsterdam, Netherlands, 1981) pp. 249-264.

[23] J.-F. Paris, " Application of Restructuring Techniques to the Improvement of Program Behavior in Virtual Memory Systems," Ph. D. Dissertation, Department of EECS, University of California, Berkeley, 1981. (available as Memorandum No. UCB/ERL M81/44, Electronic Research Laboratory, College of Engineering, University of California)

[24] J. E. Shemer and B. Shippey, "Statistical Analysis of Paged and Segmented Computer Systems," *IEEE Trans. Comp.* EC-15,6 (Dec. 1966), 855-865.