Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1980

Program Restructuring in Segmenting Environments

Jehan-François Päris

Report Number:

80-344

Päris, Jehan-François, "Program Restructuring in Segmenting Environments" (1980). *Department of Computer Science Technical Reports.* Paper 274.
https://docs.lib.purdue.edu/cstech/274

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.

TR #344

Experimental Computer Performance and Evaluation
D. Ferrari and M. Spadoni (eds.)
© SOGESTA, 1981
North-Holland Publishing Company

PROGRAM RESTRUCTURING IN SEGMENTING ENVIRONMENTS

Jehan-François Pâris

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907
U.S.A.

We present here a family of program restructuring algorithms aimed at programs to be executed in segmented virtual memory systems. These algorithms attempt to minimize the space-time product of restructured programs and can be tailored to various memory management policies like Time-Window Working Set and Segment Fault Frequency. Preliminary experiments with the trace of execution of a PASCAL compiler seem to indicate that the algorithm may significantly improve the behavior of programs in segmenting environments with a disk-like secondary storage and a time window working set policy.

1. INTRODUCTION

One of the major problems facing the designer of virtual memory systems is achieving a reasonable level of performance in the system's memory hierarchy. To a very large extent, this problem can be attributed to the presence in the system's workload of programs exhibiting scattered reference patterns. Virtual memory systems, indeed, run efficiently when their workload is composed of so-called "local" programs, i. e. programs that, at any given time, access only a small, slowly varying, subset of their total addressing space. At the limit, the presence in the workload of one program accessing in a purely random way all portions of its address space would slow down the average access time of the virtual memory to that of its slowest component.

Improving the behavior of programs has been recognized quite early as being one of the most efficient ways of enhancing the performance of virtual memory systems [1]. The most obvious way to achieve this goal would be to teach programmers to write more local code. This approach, however, has two drawbacks: it complicates significantly the task of programmers and cannot be applied to existing programs.

2. THE RESTRUCTURING APPROACH

Program restructuring constitutes an attempt to overcome these limitations [2-7]. It deals with programs already written and operates by rearranging the various blocks of code or data of a

This research has been carried while the author was at the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. He was then supported by a travel grant from the "Fonds National de la Recherche Scientifique," Brussels, Belgium.

program in its virtual address space. This process is absolutely transparent to the system's users. Considerable experimental evidence has been accumulated proving that program restructuring can substantially improve the behavior of programs in paged virtual memory systems [3]. On the other hand, very few efforts have been devoted to the extension of this approach to segmented virtual memory systems and the results obtained so far have been rather disappointing [8].

The reasons for this situation are quite simple. In a paging environment, the linear output of compilers is often a block-to-page mapping that destroys the locality naturally present in the block reference string. Since nothing similar happens in segmenting environments, there is not the same need for a corrective action. Also, existing program restructuring algorithms rely heavily on the fact that, in paged virtual memory systems, all exchanges of information between the main memory and the secondary store involve only fixed size <u>pages</u>. Therefore, the problem of finding a better arrangement of blocks in the program's address space is essentially a matter of finding a better block-to-page mapping. This can be done by constructing first a <u>restructuring matrix</u> --or cost matrix-- expressing the costs of keeping each pair of blocks i and j in separate pages and then applying a clustering algorithm to this matrix. The result of the clustering algorithm will be a new block-to-page mapping that will group together blocks having the highest interblock costs and, therefore, minimize the sum of costs corresponding to blocks actually stored in distinct pages.

The various restructuring algorithms differ essentially from each other in the way they define these interblock costs. The most sophisticated of them, the so-called "strategy-oriented" algorithms [3], take into account the page replacement strategy of the system in which the restructured programs will eventually be executed and express interblock costs in terms of some index measuring the program's performance in this peculiar environment. In all cases, there is never any penalty associated to the storing of two blocks in the same page; thus, interblock costs are essentially positive quantities.

3. <u>SPECIFIC PROBLEMS OF SEGMENTING ENVIRONMENTS</u>

The same basic assumptions cannot be made for segmented virtual memory systems. Segment sizes, and their number, can arbitrarily vary. Therefore, the decision of storing two blocks in the same segment is bound to affect the segment size; this will in turn have an influence on the costs of bringing the segment in main memory and keeping it there. There will thus be cases where merging two blocks will actually decrease the program performance. As a first consequence, interblock costs cannot be any more considered as being essentially positive. Any program restructuring algorithm neglecting this fact will produce unacceptable block-to-segment mappings.

Consider, for instance, the case of a restructuring algorithm having as objective to minimize the number of segment faults occurring during the execution of the program. This algorithm would be a segment-oriented version of the so-called "Critical Algorithms," which are among the best known restructuring algorithms for paging environments. Applied to a program being executed in a segmenting environment, this algorithm will lead to the trivial solution of gathering all blocks constituting the program into a single segment.

On the other hand, any algorithm attempting to minimize the main memory occupancy of the restructured program will lead to the fragmentation of the program into as many segments as feasible.

The failure of the two approaches we have just sketched can be explained by the fact that, in both cases, we attempted to optimize only one indicator of the program's performance. While being quite successful in that regard, we achieved an unacceptable overall result because of the drastic deterioration of other program performance indicators. A possible solution could be to introduce some additional constraint on the new block-to-segment mapping obtained by the clustering algorithm that will ensure that no unacceptable mapping will ever be produced by the restructuring algorithm. This was indeed the solution adopted by Chen and Gallo [8]. Their algorithm attempts to minimize the total number of cross-references between segments while enforcing the condition that the total number of segments must remain constant. This condition ensures that none of the pathological block-to-segment mappings we have discussed above will ever occur. On the other hand, it introduces also an artificial constraint on the block ordering produced by the restructuring algorithm. It is intuitively clear that this constraint will lead to the rejection of otherwise perfectly acceptable block orderings and, thus, may significantly degrade the algorithm's performance.

A more sensible approach to the problem of program restructuring in segmenting environments would be to base the definition of interblock costs on some <u>global</u> index of the program's performance. A well known example of such performance index is the <u>Space-Time Product</u> criterion proposed by Belady and Kuehner [9]. This criterion has been already used for constructing various program restructuring algorithms tailored to different paging environments [6-7]. For all memory management policies investigated, these so-called "Balanced Algorithms" have been found to perform significantly better than the other existing strategy-oriented restructuring algorithms. We want to show here how the same approach can be extended to segmenting environments and how efficient strategy-oriented restructuring algorithms can be derived from the space-time product criterion and tailored to various segment replacement policies.

4. BALANCED ALGORITHMS AND THE SPACE-TIME PRODUCT

Balanced Algorithms differ essentially from other program restructuring algorithms in the way the elements of the restructuring matrix A are computed. Each element a_{ij} of the restructuring matrix will represent the increase of the space-time product that would result from the decision of keeping blocks i and j in separate segments; a negative entry in the matrix will then correspond to the situation where storing the two blocks in the same segment would have a detrimental effect on the space-time product of the restructured program. The procedure used to evaluate these a_{ij} will essentially consist of using a trace of memory references, collected during a previous run of the program, in order to simulate, as closely as possible, block behavior during the program's execution. This will enable us to predict under which circumstances the storing of two blocks in the same segment could have beneficial or detrimental effects on the space-time product of the restructured program; the algebraic sum of these effects for each pair of blocks will be, by definition, the entry of the restructuring matrix corresponding to

that pair of blocks.

In terms of space-time product, the main difference between paging and segmenting environments lies in the fact that, in a segmenting environment, the average time required to service a segment fault is a linear function of the size of the segment causing the fault. More precisely, if s_i is the size of that segment, the average time T_w required to service the fault will be given by

$$T_w = T_1 + T_t . s_i$$

where T_1 is the mean access time of the secondary store and T_t the mean time to transfer one unit of data.

Let now S(u) denote the memory occupancy of a program at a given time u. The space-time product characterizing the behavior of the program being executed in a segmenting environment during a <u>virtual</u> time interval (0, t) is given by

$$C = \int_0^t S(u)\,du + \sum_{j=1}^{r} S(t_j)\ .(T_1+T_t.s_{x_j})$$

where r is the total number of segment faults occurring during (0, t), t_j the time of the j-th segment fault and x_j the segment causing that fault.

As we said before, the decision of storing two blocks in the same segment can have both beneficial and detrimental effects on the performance of the program. These effects will be directly reflected by corresponding variations of its space-time product. The resultant of these variations can be evaluated for each pair of blocks i and j by examining the program's reference patterns. That value will be, by definition, the element a_{ij} of the restructuring matrix.

Suppose, for instance, that block j is referenced after a long interval of inactivity. Suppose also that block j is stored in a segment containing only blocks that have also been inactive for a while. Then, the segment containing block j will probably not be present in memory and a segment fault will occur. On the other hand, should block j have been stored in a segment containing at least one block currently referenced at the time considered, the segment would have been present in memory and the potential segment fault avoided. This would be reflected in the space-time product of the restructured program as a saving of

$$S(t)\ .\ (T_1 + T_t.s_j)$$

space-time units, where S(t) is the current memory occupancy of the program and s_j the size of block j.

Suppose now that block i has been stored in a segment k containing other blocks and that some of these blocks are active during a time interval Δt during which block i is inactive. Then, block i will be resident in memory, along with segment k, during that time interval although its presence in memory is not necessary. This will be reflected in the space-time product of the program as a waste of

$$s_i . \Delta t$$

space-time units. Similarly, each time that the segment will be brought in memory because some block of that segment, different from x_i, is referenced after having been inactive for a while, there will be a need for transferring s_i data units and this will result in an increase of the program's space-time product by

$$S(t_f) \cdot T_t \cdot s_i$$

additional space-time units. However, when the secondary storage is a disk-like device, i. e. a device characterized by a significant access time and a high transfer rate $1/T_t$, this increase remains limited.

5. INFLUENCE OF THE SYSTEM'S MEMORY POLICY

So far, we have carried our discussion assuming that a segment containing only blocks that have been inactive "for a while" will be no more resident in memory To be more specific, we have to take into account the memory policy of the system in which the restructured program will be executed and introduce the concept of the Resident Set of Blocks [3,4]. By definition, the resident set of blocks $R_b(t)$ of a program at a given time t of its execution in a well defined environment is the set of all blocks that will be present in memory at time t regardless of the block-to-segment mapping. As a corollary of this definition, any segment containing at least one block member of that resident set at time t will be necessarily present in memory at that time. By analogy with the concept of segment fault, we will say that a block fault occurs at time t when the referenced block at time t is not a member of $R_b(t-1)$.

Evaluating the resident set of blocks of a program at time t is a more or less difficult task depending on the system memory policy. The Time-Window Working Set memory policy (TWWS), for instance, removes a segment when it has been unreferenced for T time units; it is quite similar to the original paged working set policy with a window size $\tau = T+1$ [10,11]. Under the TWWS policy, the resident set of blocks $R_b(t)$ will be simply the set of all blocks that have been referenced during the last T time units.

One of the variants of the TWWS policy is the so-called Space-Time Working Set policy (STWS), under which a segment i is removed when it has not been referenced for T' / s_i time units [11]. Suppose now that S_{max} is the maximum segment size, then $R_b(t)$ will contain only the blocks that have been referenced during the T' / S_{max} last time units, where S_{max} is the maximum segment size. This expression represents obviously a worst case estimate and, therefore, does not provide a reliable estimator of the set of blocks that will probably be present in memory at any given time.

Another memory policy that has been applied to segmenting environments is the Segment Fault Frequency policy (SFF) which is essentially a segmented version of the Page Fault Frequency policy developed by Chu and Opderbeck for paging environments [12]. The SFF policy removes segments from memory at segment fault time, if and only if an interval of more than T time units has elapsed since the last segment fault; the segments removed are those that were unreferenced during that interval. One can therefore assume [7] than $R_b(t)$ will contain at least all blocks that have been referenced during the time interval $(t_{lf}-T, t)$ where t_{lf} is the time of the last block fault before the current reference.

To each of these memory policies one can associate a particular balanced restructuring algorithm that will be tailored to that policy. Therefore one can speak of a Balanced Time-Window Working Set Algorithm (BTWWS), a Balanced Space-Time Working Set Algorithm (BSTWS), a Balanced Segment Fault Frequency Algorithm (BSFF), and so forth. All these algorithms share the same structure and differ only in the way their resident sets of blocks $R_b(t)$ are defined.

6. FORMAL DEFINITION OF BALANCED ALGORITHMS

Let

$(r_1, r_2, \ldots, r_n)$ be a reference string collected during one run of the program to be restructured,

$b(t)$ the block containing the t-th reference,

s_i the size of block i,

$S(t)$ the memory space occupied by the program while processing the t-th reference,

$R_b(t)$ the resident set of blocks at time t, i. e. while processing the t-th reference (we assume $R_b(1)=\{r_1\}$),

T_m the mean inter-reference time,

T_1 the mean access time of the secondary store,

T_t the mean time to transfer one data unit,

The restructuring matrix $A = (a_{ij})$ has all zero entries initially and is constructed in the following way:

(a) For all t from 1 to n do

if $b(t) \notin R_b(t-1)$ then (* block fault *)

increment by $\alpha = S(t).(T_1+T_t.s_{b(t)})$ all a_{ij}'s such that $i \in R_b(t)$ and $j=b(t)$;

decrement by $\gamma = S(t).T_t.s_i$ all a_{ij}'s such that $i \notin R_b(t)$ and $j=b(t)$

fi;

decrement by $\beta = s_i.T_m$ all a_{ij}'s such that $i \notin R_b(t)$ and $j \in R_b(t)$

od;

(b) For all i and all j<i do

$a_{ij} := a_{ji} := a_{ij} + a_{ji}$

od.

In other words,

[a] each time a block fault occurs, the algorithm

- -- attempts to avoid the occurrence of a segment fault by incrementing all the entries of A that correspond to the pairs of blocks containing a block already in memory and the block causing the block fault, and

- -- attempts to avoid any increase in the size of the segment to be brought in memory by decrementing all the entries of A that correspond to the pairs of blocks containing a block not residing in memory and the block causing the block fault;

[b] at each reference, the algorithm decrements all the entries of A that correspond to the cases where one block resides in memory and the other does not.

Note that the algorithm we have described can be applied to all memory policies for which it is possible to construct the resident set of blocks $R_b(t)$ and to determine the memory space $S(t)$ occupied by the program at time t. To obtain the restructuring algorithm tailored to a specific memory policy, like the Balanced Time Window Working Set for the TWWS policy or the Balanced Segment Fault Frequency for the SFF policy, one has only to specify the proper expressions for $R_b(t)$ and $S(t)$.

7. IMPLEMENTATION CONSIDERATIONS

A few problems arise with the above scheme when the implementation of a specific balanced restructuring algorithm is attempted. First, it will be generally impossible to evaluate $S(t)$ at restructuring time as the program memory occupancy depends on the final block-to-segment mapping produced by the restructuring algorithm. The simplest solution is then to replace $S(t)$ by a constant value $\bar{S}$ that will be some estimate of the program mean memory occupancy $\bar{S}$. This approximation is essentially the same as the one adopted by Prieve and Fabry in their optimal variable-space page replacement algorithm VMIN [13].

Another problem concerns the cost of running the algorithm. One can expect, from any reasonable memory strategy, that the number of block faults will be considerably lower than the total number of references. One can therefore neglect, as a first approximation, the contribution of the fault handling routine to the running time of the algorithm. The critical part of the scheme is then the one that requires that, at each reference, all the elements a_{ij}'s of the restructuring matrix corresponding to a block $i \notin R_b(t)$, and a block $j \in R_b(t)$, be decremented by $s_i . T_m$.

Let m represent the number of blocks constituting the program being restructured. Then, the processing of each reference of the program's execution trace will essentially require $O(m^2)$ operations and one can assume a running time of $O(n.m^2)$ for the algorithm. In order to reduce this cost, one can resort to a sampling technique and perform the aforementioned routine each K memory references. In this case, the running time of the algorithm will become $O(n.m^2 / K)$ and the quantity by which the interblock cost of the two blocks will be decremented becomes $K.T_m$ times the size of the block not included in $R_b(t)$. The approximation remains acceptable as long as the sampling interval $K.T_m$ is relatively small compared to the average stay

of a segment in memory.

A third modification can be made whenever the secondary store is a disk-like device. These devices are essentially characterized by a significant access time T_l and a high transfer rate $1/T_t$. One can thus neglect, as a first approximation, the contributions of the segment sizes to the costs of segment faults.

Keeping the same notations as in the last section, the new version of our algorithm will then be:

(a) For all t from 1 to n do

if $b(t) \notin R_b(t-1)$ then (* block fault *)

increment by $\alpha = \bar{S}.T_l$ all a_{ij}'s such that $i \in R_b(t)$ and $j=b(t)$;

fi;

if t mod K = 0 then (* sampling time *)

decrement by $\beta = s_i.K.T_m$ all a_{ij}'s such that $i \notin R_b(t)$ and $j \in R_b(t)$

fi

od;

(b) For all i and all j<i do

$a_{ij} := a_{ji} := a_{ij} + a_{ji}$

od.

8. EXPERIMENTAL RESULTS

In order to evaluate the performance of balanced algorithms under two different memory policies, we developed trace-driven program behavior simulators for time-window working set and segment fault frequency policies. The trace used in our experiments was a block reference string that had been obtained from an instrumented PASCAL compiler by Ferrari and Lau [5].

The PASCAL compiler from which the traces were obtained is running on a CDC 6400 at the University of California, Berkeley. It is 17,945 60-bit words large and counts 139 procedures. Assuming that a 60-bit word corresponds roughly to eight bytes, its size, expressed in bytes, would then be 143,560 bytes. Sizes of the procedures vary between a maximum of 665 words (5,320 bytes) and a minimum of 18 words (144 bytes) with an average of 129 words (1,032 bytes).

The reference string we used in our experiments was collected while the compiler was compiling parts of its source code. Total execution time, including instrumentation overhead, was 163.508 s, which corresponds to a run time of 9.318s for the standard, non-instrumented version of the compiler. Because of the instrumenting procedure utilized, only instruction references were collected. The lack of data references is not, however, a major drawback since the

instruction and the data portions of a program can be restructured independently provided that instructions and data are stored in different segments. Besides, working-set environments have the property that the presence of one segment in memory at any time does not depend on the behavior of other segments and, therefore, the block-to-segment mappings and the performance improvements obtained by restructuring the instruction portion of a program do not depend on the reference patterns and on the organization of its data portion.

In order to reduce the cost of our simulations, we decided to use a compressed version of the original trace for driving our two simulators. The trace reduction algorithm utilized to produce the compressed trace has been described by Lau [14] and is essentially a variant of Smith's "Snapshot Method" [15]. It replaced the original reference string by a sequence of 32,702 "reference sets", each containing the instruction blocks referenced during a 5 ms interval; because of the instrumenting overhead, each of these sampling interval corresponds <u>on the average</u> to 0.28494 ms of execution time for the non-instrumented version of the program.

We performed our simulations of a Balanced Time-Window Working Set (BTWWS) algorithm for four window sizes between 20 and 150 ms. For each simulation, the resident set of blocks $R_b(t)$ at time t before processing the t-th reference was thus defined as the set of all blocks that have been referenced at least once during respectively the last 20, 50, 100 or 150 ms. The algorithm's sampling interval for evaluating the negative components of interblock costs --$K.T_m$-- was set to 18 reference sets, i. e. approximately 5 ms. Since the restructuring process primarily involves the gradual merging of the program's original segments into larger units, we were interested in measuring the algorithm's performance at various stages of this merging process. Therefore, we decided not to use one fixed segment fault cost $S.T_1$ in our experiments but rather to repeat each simulation for selected fault costs varying between 5000 and 10^7 bytes * sampling intervals, i. e. 1.445 and 28,494 bytes * seconds.

For each window size and for each segment fault cost selected, we simulated the application of a BTTWS algorithm to the PASCAL compiler and evaluated the performance of the restructured program under the same set of inputs. Being primarily interested in the phase of the restructuring process where the restructuring matrix was built, we decided to use a simple, but efficient, clustering algorithm analogous to the one described by Ferrari in [3]. The only significant adjustment that we made to the algorithm consisted of removing any limitation related to cluster sizes. Former experiments with restructuring algorithms in paging environments [7] had convinced us that more sophisticated clustering algorithms would not necessarily perform better.

For each run, we measured the number of segment faults, the total number of bytes brought in memory and the mean memory occupancy of the program before and after restructuring. Figures I and II summarize these results. On both figures, the curve labeled "NR" corresponds to the non-restructured version of the program and each individual point of the curve represents a different window size. Each of the four other curves on each figure corresponds to a given window size and varying segment fault costs. The uppermost point of each curve corresponds to the limit case of a segment fault

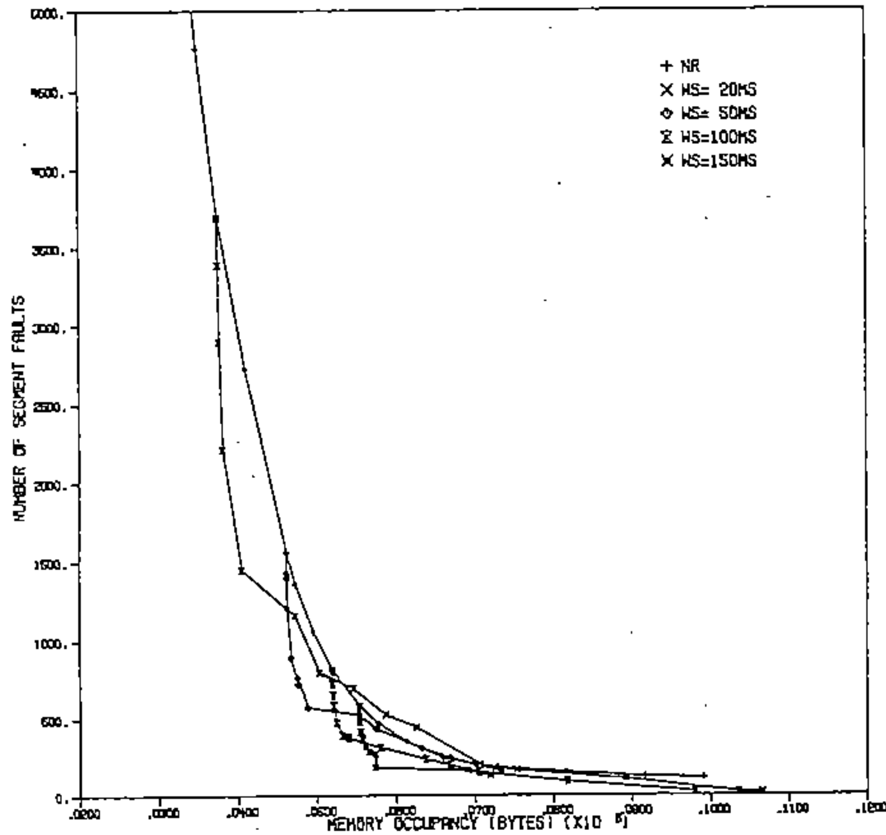

Figure I

cost equal to zero. For that particular value, the structure of the program remains unchanged during the restructuring process.

Looking at Figure I, one can see that the restructuring process can decrease the number of segment faults observed during an execution of the program by at least 50% without causing any significant increase of its memory occupancy; this increase becomes appreciable only when the segment fault cost parameter becomes superior or equal to 10^5 bytes * reference set, i. e. 28,494 bytes * ms. Figure II, on the other hand, shows clearly that the total number of bytes swapped in decreases much more slowly than the number of segment faults. This observation is easy to explain if one remembers that the restructuring process consists essentially of merging the

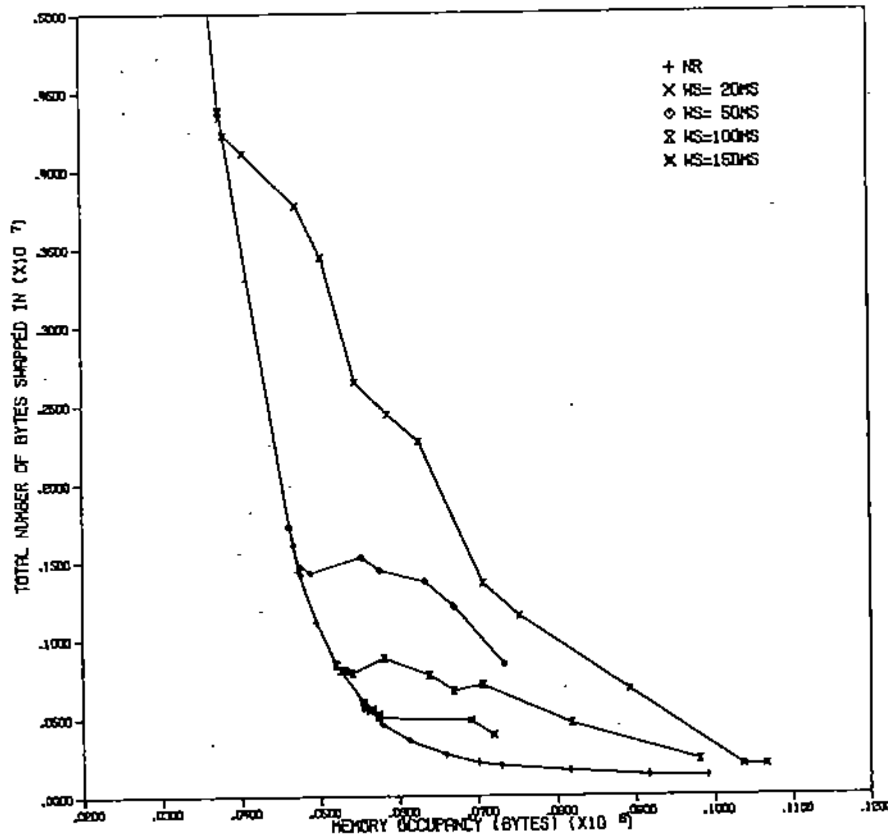

Figure II

program's original segments into larger units. Therefore one can expect to have, for a given memory occupancy, less segment faults but a higher byte traffic between the memory and the secondary store.

The global effect of this reduction in the number of segment faults and this increase of the byte transfer rate for a given memory occupancy can be evaluated by computing the swapping load L_s of the program. By definition, this swapping load L_s is the sum of all delays occurring at segment fault times and caused by the secondary store latency or the segment transfer times. Keeping the same notations as in section 6 and representing by $N_{b,in}$ the total number of bytes brought in memory during the execution of the

program, one can thus write

$$L_s = r \cdot T_l + N_{b,in} \cdot T_t$$

where r is again the total number of segment faults occurring during the time interval considered.

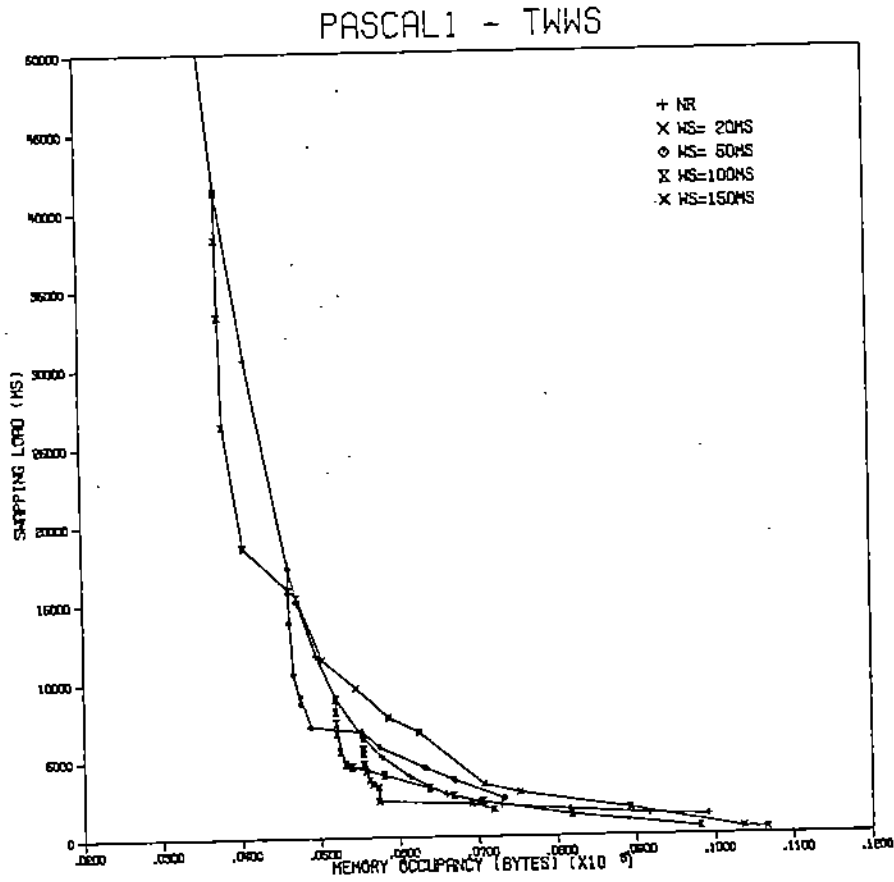

Figure III

Figure III displays the values of this swapping load computed for a latency time T_l=10ms and a transfer time T_t=10^{-6}s/byte.

For these values, which correspond to a reasonably fast secondary store, the contribution of the latency times to the swapping load is so preponderant that one could almost neglect the influence

of the segment transfer times and assume a swapping load L_s proportional to the number of segment faults r. Since this phenomenon will only grow stronger when the latency time increases, one can safely assume that the beneficial effects of the restructuring process will remain as important for a wide range of secondary stores.

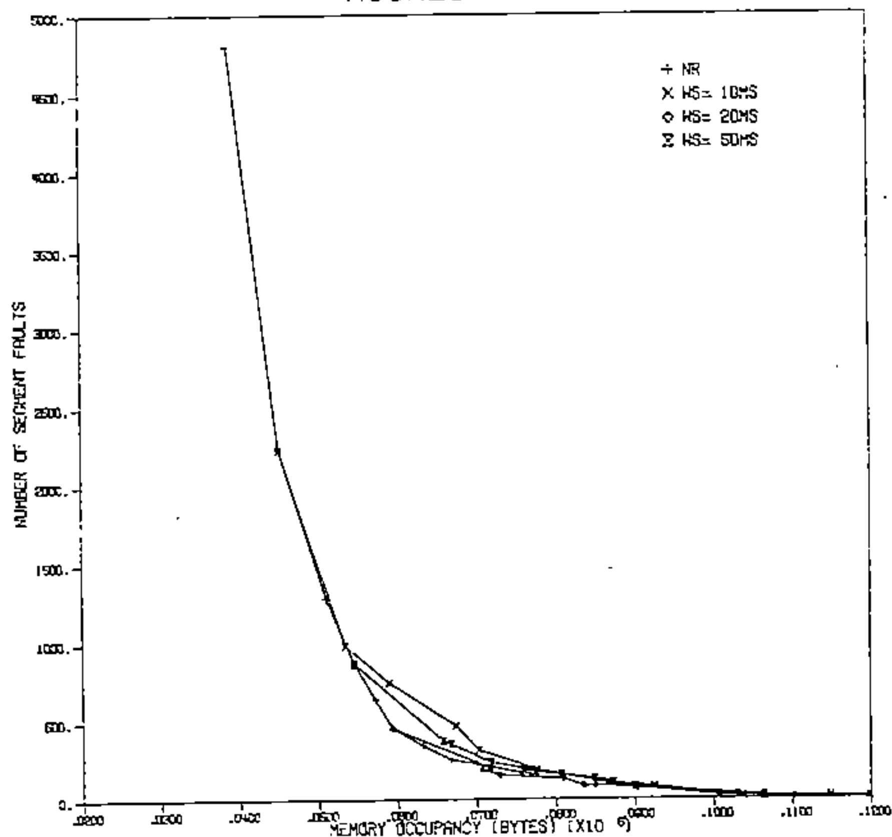

Figure IV

The same experiments were repeated for a Segment Fault Frequency memory policy using the same PASCAL compiler. We ran our simulations of a Balanced Segment Fault Frequency restructuring algorithm (BSFF) for various values of the segment fault cost and three values of the SFF T parameter, namely 10, 20 and 50 ms.

In this case, however, the results were quite different. As

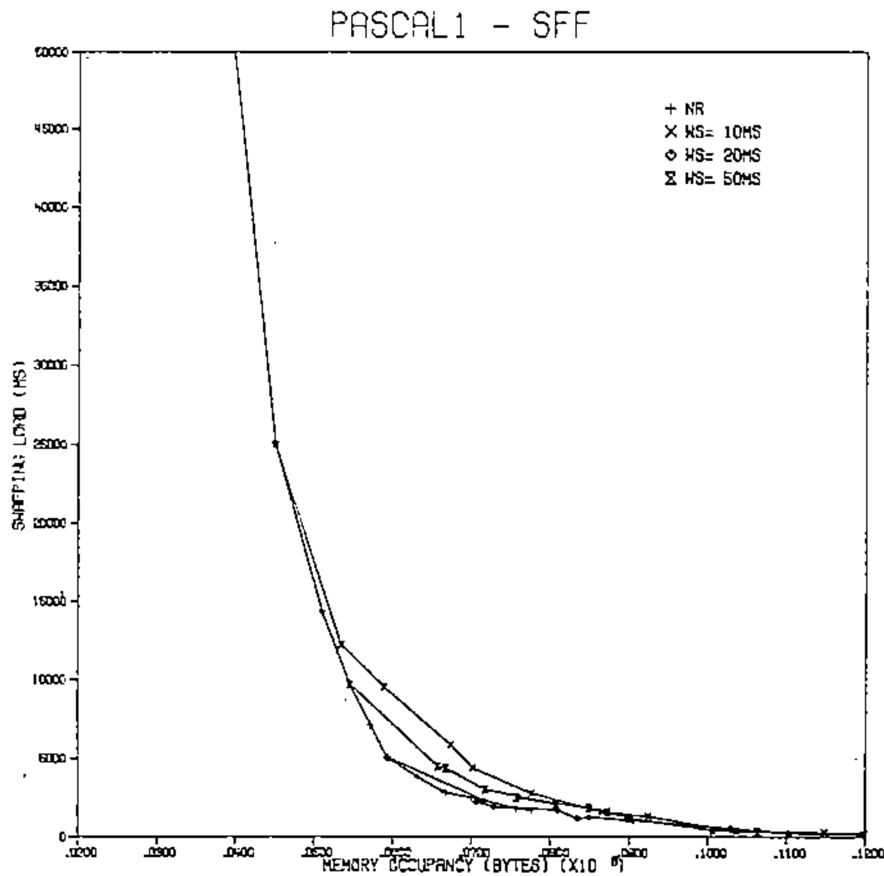

Figure V

Figure 1V shows, the number of segment faults achieved by the various restructured versions of the program were never much better than the ones obtained, for the same memory occupancy, by the non-restructured program. These results are even more disappointing if we compute the various swapping loads --see Figure V--. In conclusion, one can safely affirm that the restructuring process has no beneficial effects on the overall behavior of the program.

The Page Fault Frequency algorithm is known to exhibit some anomalies [17]. In this case, however, we think that a much simpler explanation exists. Since the Segment Fault Frequency algorithm expels idle segment only at segment fault times [12], any decrease of the segment fault frequency below 1/T will result in an increase

of the program's memory occupancy.

Conclusion

The limited experimental evidence we have gathered seems to indicate that program restructuring can significantly improve the performance of programs executed in a segmented environment characterized by a time-window working set policy and a disk-like secondary store. Further investigations in the field of restructuring algorithms for segmenting environments should basically involve:

-- the gathering of more experimental evidence;

-- the study of possible modifications in the definition of inter-block costs;

-- investigations on the influence of the clustering algorithm on the performance of restructured programs;

-- investigations on the portability of restructuring algorithms (what would happen if some parameters of the system's memory policy were to change?) and on their data dependence (to which extent will the behavior of the restructured program be influenced by its input data?).

Acknowledgements

The author wants to thank here Professors D. Ferrari and A. J. Smith from the University of California, Berkeley for their numerous suggestions and encouragements as well as his friends, inside and outside the PROGRES group, for their support. He would also like to express his thanks to Professor P. J. Denning whose comments helped to make the paper clearer.

References

[1] Brawn, B. and Gustavson, F. Program Behavior in a Paging Environment, AFIPS Conf. Proc. Vol. 33 (1968 FJCC), 1019-1032.

[2] Hatfield, D. J. and Gerald, J. Program Restructuring for Virtual Memory, IBM Sys. J. 10 , 11 (Nov 1974), 39-47.

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

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

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

[6] Pâris, J.-F. Application of the Space-Time Product Criterion to the Definition of a New Family of Program Restructuring Algorithms, R. P. 4/78, Institut d'Informatique, Facultés Universitaires de Namur (1978).

[7] Pâris, J.-F. Improving the Behavior of Programs in Virtual Memory Systems,, Ph. D. Dissertation, University of California, Berkeley (in preparation).

[8] Chen, P. S. and Gallo, A. Optimization of Segment Packing in Virtual Memory, in Computer Architecture and Networks (E. Gelenbe and R. Mahl Eds.), North Holland Publ., 1974.

[9] Belady, L. A. and Kuehner, C. J. Dynamic Space Sharing in Computer Systems, Comm. ACM 12 , 5 (May 1969), 282-288.

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

[11] Denning, P. J. and Slutz, D. R. Generalized Working Sets for Segment Reference Strings, Comm. ACM 21 , 9 (Sept. 1978), 750-759.

[12] Chu, W. W. and Opderbeck, H. The Page Fault Frequency Paging Algorithm, AFIPS Conf. Proc. Vol. 33 (1972 FJCC), Pt. 1, 597-609.

[13] Prieve, B. G. and Fabry, R. S. VMIN--An Optimal Variable Space Page Replacement Algorithm, Comm. ACM 20, 5 (May 1976), 295-297.

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

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

[16] Smith, A. J. Two Methods for Efficient Analysis of Memory Trace Data, IEEE Trans. Softw. Engrg. SE-3, 1 (Jan. 1977), 94-101.

[17] Franklin, M. A., Graham, G. S. and Gupta, R. K. Anomalies with Variable Partition Paging Algorithms, Comm. ACM, 21, 3 (March 1978), 232-236.