APPLICATION OF RESTRUCTURING TECHNIQUES
TO THE OPTIMIZATION OF PROGRAM BEHAVIOR
IN VIRTUAL MEMORY SYSTEMS

by

Jehan-François Pâris

Memorandum No. UCB/ERL M81/44

18 May 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Application of Restructuring Techniques to the Optimization of Program Behavior in Virtual Memory Systems

By

Jehan Francois Paris

Engineer (Free University of Brussels) 1970
Grad. (University Faculties of Our Lady of Peace, Belgium) 1972
Grad. (University of Paris VI) 1974

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Engineering

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: .... May 18, 1981
Chairman Date

....

D. Brillinger
....

....

Application of Restructuring Techniques to the Optimization of Virtual Memory Systems

Application of Restructuring Techniques to the Optimization

of Virtual Memory Systems

Jehan-François Paris

Ph. D. in Engineering
Electrical Engineering and Computer Science

Professor Domenico Ferrari
Chairman of Thesis Committee

Abstract

One of the most effective ways of obtaining a better performance from virtual memory systems consists of improving the behavior of programs in such environments. Program restructuring attempts to achieve this goal by rearranging the block-to-page mappings of programs.

The best existing restructuring algorithms take into account the *dynamic* behavior of the program to be restructured and attempt to minimize either its page fault frequency or its mean memory occupancy, which are both partial indicators of program performance.

In this thesis, we present a new class of restructuring algorithms that attempt to minimize a global index of program performance, namely its *space-time product*. The primary motivation of these algorithms is to avoid situations where a significant improvement of one index of program performance would be accompanied by a comparably sized deterioration of another index. Hence the name of "Balanced Algorithms" given to our algorithms.

Balanced Algorithms essentially attempt to minimize a restructuring-time estimate of the space-time product of the restructured program. Since they follow a common scheme, they can be easily tailored to a wide range of variable-space memory policies, including Working Set, Sampled Working Set, Global LRU and Page Fault Frequency.

We prove that BWS, the balanced algorithm tailored to Working Set environments, effectively minimizes a linear combination of the number of page faults 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.

In order to evaluate the performance of balanced algorithms under various memory policies and to compare it to those of other restructuring algorithms, a series of trace-driven experiments simulating the behavior of programs before and after restructuring were conducted. These simulations show that BWS, the balanced algorithm tailored to Working Set environments, performs significantly

better than the two best existing restructuring algorithms. Similar results were found with the balanced algorithm tailored to Sampled Working Set environments. BPSI, the balanced algorithm tailored to Global LRU environments, exhibited only a marginal superiority over its rivals, while no clear winner emerged for the Page Fault Frequency environments.

Another consequence of our choice of a global indicator of program performance as restructuring criterion is to allow our approach to be extended to *segmentation environments*, or which no efficient restructuring algorithms have been proposed.

Here too, we prove that BTWWS, the balanced algorithm tailored to Time-Window Working Set environments, minimizes a linear combination of the number of segment faults 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 segment.

Experimental evidence is also presented showing that BTWWS can significantly improve the segment fault frequency of a program without causing any comparable increase of its memory occupancy. On the other hand, our simulations indicate that BSFF, the balanced algorithm tailored to Segment Fault Frequency environments, does not bring any improvement to either indices of program performance.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

It is a pleasure and a privilege for me to thank here Professor D. Ferrari, my thesis advisor, for his untiring support and encouragements. He introduced me to program restructuring and provided an environment in which working has been a pleasure.

I want to thank also Professor A. J. Smith for his numerous suggestions and encouragements.

I am also very grateful to Professor D. R. Brillinger for having served on my thesis committee.

I gratefully acknowledge the support of an "Aspirant" Fellowship and of a travel fellowship, both from the "Fonds National de la Recherche Scientifique," Brussels, Belgium. The support for the typing of an earlier version of Chapter III came from the NSF grant MCS78-24618.

Finally, I would like to thank all my friends, inside and outside the PROGRES group, for their support. Special mentions here--in chronological order--to Edwin Lau, Makoto Kobayashi, Ozalp Babaoglu, Newton Faller and Luis Cabrera.

CHAPTER I

VIRTUAL MEMORY CONCEPTS

1.1. INTRODUCTION

The first full implementation of a virtual memory architecture was the Ferranti ATLAS computer [Foth61] [Kilb62] [Bayl68]. The main feature of this machine was a special address translation hardware that performed dynamic relocation of fixed size blocks of code and/or data in the main memory. This hardware was aided by a set of supervisor routines providing automatic transfer of blocks between the main memory and a secondary store. As a result the ATLAS processor was able to access a two-level storage system consisting of the "real" memory and the secondary store by way of a single linear address space, known today as the "virtual memory" [DenP70] [Dora76].

During the last two decades, virtual memories have progressively emerged as the major tool for managing memory hierarchies. They constitute today a main feature of the majority of medium and large scale computer systems, and play a central role in most data base systems. The reason for this success is simple: Virtual memories allow an efficient use of the memory resources, while taking away from user programs the burden of managing the data transfers between the various levels of the memory hierarchy. From a user's viewpoint, everything happens indeed as if programs were stored in a very large one-level memory.

A consequence of a "transparent" memory hierarchy is that the system's performance can be strongly affected by factors not directly under the user's control. The two most important of these factors are the system's memory management policy and the referencing behaviors of programs.

1.2. VIRTUAL MEMORY MANAGEMENT

In a virtual memory system, programs are typically executed without having their whole address space permanently residing in memory. It will be the responsibility of the virtual memory management routines to bring and to keep in memory all the information currently accessed by each running program. Every time a program attempts to access a portion of its address space not residing in memory at that time, a *fault* condition is said to occur: the execution of the program is interrupted while the required information is brought into memory from the secondary store. If no space is available in memory at that time, some other portions of the program address space will have to be removed from memory and returned to the secondary store.

In the scheme we have just described, any given set of memory locations can contain different portions of a program's address space during the execution of this program. Thus the translation of virtual--i. e. program--addresses into the physical memory addresses will need to be done immediately before the execution of each instruction. This task will require the existence of special-purpose *dynamic relocation hardware*, which will have to consult a mapping table updated every time any transfer of information takes place within the memory hierarchy.

Memory management itself essentially consists of deciding what information to transfer, when to perform this transfer and where to store the transferred information. Some of the specific issues to be considered are

- the size of the blocks transferred within the memory hierarchy (fixed vs. variable size or paging vs. segmentation),
- the algorithm used to decide which blocks should be brought into memory and when (fetch policy),
- the algorithm used to decide which blocks should be removed from memory and when (replacement policy).

1.2.1. Paging v. Segmentation

The most apparent difference between the existing virtual memory systems lies in the choice of the size of their transfer and allocations units. Some systems use only fixed-size entities,better known as *pages*, while other systems leave each programmer free to organize his/her addressing space into as many *segments* as wanted.

1.2.1.1. Paged Virtual Memory

In the ATLAS system, all memory allocation decisions and all exchanges of information between memory and secondary store only involved system-defined fixed-size entities. The address space of each program was logically partitioned into *pages* containing each 512 consecutive words of code or data. Similarly, the physical memory was subdivided into equal size *page frames*. Each time a page was fetched into memory, the system first attempted to find an empty page frame and to allocate it to the incoming page. If no page frame was available at that moment, one had first to be freed, possibly by returning its content to the secondary store.

This form of virtual memory organization is known as *paging*. Its main advantage lies in the much greater ease of performing the memory allocation tasks when one has only to consider equal size entities. Since the page size on a binary machine will normally be a power of two, dynamic translation of virtual addresses into physical addresses is also simplified. On the other hand, the partitioning of the virtual address space into fixed size pages almost never reflects the program's logical structure. Therefore, unrelated blocks of code or data will often share a common page while other blocks will overlap a page boundary.

1.2.1.2. Purely Segmented Virtual Memory

In segmentation architectures, each user program defines the number and the sizes of the *segments* composing its own address space. This partition of the program's address space is totally under the user's control. To reference a particular element within a segment, programs will have to specify the segment name as well as the address of the element within the segment. Segments will not only constitute the transfer units used by the virtual memory management; they will also help programmers to structure their address space and serve as a basis for the implementation of sharing and protection mechanisms.

Segmentation, unlike paging, ensures that the virtual address space of each program will be subdivided into entities reflecting the program's own logical structure. Since segments only contain information that is logically correlated, one may expect a better utilization of memory space by segmentation systems than by paging ones.

On the other hand, memory allocation is a much more difficult job for arbitrary size segments [DenP70] [Knut73]: Each time a segment is brought into memory, it will be placed into a space that is greater than or equal to its size. This will result in the fragmentation of the free memory space into domains often too small to be allocated to incoming segments and thus lead to inefficient use of the memory. As a result, paging still remains today the more widely used organization for virtual memory systems.

1.2.1.3. Paged and Segmented Virtual Memory

As we said before, motivations for providing a segmented address space extend well beyond the need for defining transfer and allocation units in virtual memory systems. A possible solution for avoiding the fragmentation problems associated with pure segmentation architecture, while keeping its advantages, will thus be to subdivide segments into fixed-size pages. The best known example of this paged and segmented architecture is the MULTICS system [Orga72] [DenJ65].

The scheme unfortunately complicates the address translation process. Each segment name will have first to be transformed into a segment number that will be used as an offset in a segment table pointing to the corresponding page table. The virtual address within the segment will then be mapped into a page number and a page offset. In this scheme, segment sizes will always be multiples of the page size and will tend to be much larger than in a purely segmented virtual memory.

1.2.2. Fetch Policy

In both paging and segmentation systems, the purpose of the fetch policy is to determine which page(s) or segment(s) should be brought into memory and when this task should be performed. At least three techniques may be used to perform the fetching task; they are demand fetch, prefetch, and initially loaded demand fetch [Lau79].

Because of its simplicity, *demand fetch* remains still today the most popular fetch policy. It consists of bringing into memory pages or segments one at a time and only when a missing page or segment causes a fault. (In paged segmentation architectures, only the the missing page is brought into memory.)

One may envision other policies aimed at reducing the frequency of faults by prefetching pages or segments before they are referenced and cause a fault. These policies would have essentially to predict future program behavior. The most general of these prefetch policies would be able to anticipate future page faults and decide when it would be time to bring a given page or segment in memory. Such policies would require collecting a lot of information on the behavior of each program and would thus be quite difficult to implement. Their rate of success in terms of the number of faults correctly predicted and anticipated remains still to be determined.

A more feasible approach would be to prefetch page(s) or segment(s) only at fault times. This technique is known as *demand prefetching*. It has been implemented in some systems--among which VAX VMS [DEC 78]--but experimental data are still inconclusive about its merits [Jose70] [Smit78b] [Smit78d] [Lau79].

Finally, one can restrict prefetching to the time before execution begins. The corresponding policy, called *initially-loaded demand fetch* or *warm start*, consists of bringing into memory before the program's execution begins a set of prespecified pages or segments. Once execution has begun, a pure demand fetch or a prefetch policy is followed [Kubo76] [East78].

1.2.3. Replacement Policy

The purpose of the *replacement policy* is to decide which page(s) or segment(s) should be removed from memory and when should this event occur. Replacement policies are without any doubt the area of virtual memory management that has been studied the most extensively. Among the several replacement policies that have been proposed or implemented, one may distinguish the local fixed partition policies, the global policies and the local variable partition policies.

1.2.3.1. Local Fixed Partition Policies

A first set of replacement policies apply when the system allocates to each active program a fixed partition of memory. Each time a fault condition occurs, a page (or one or more segments depending on their sizes) must thus be removed from the partition assigned to the faulting program in order to free enough space for the page (or the segment) that will be fetched.

One example of such policies is the First-In-First-Out (FIFO) algorithm, which removes from memory the pages or segments that have been residing in memory for the longest time interval. This requires the system to maintain a page ordering based on the memory loading times of the pages or segments. Since this ordering is updated only upon a fault, the procedure can be performed efficiently by software and the algorithm requires no special hardware. Unfortunately, the time that a page or a segment has spent residing in memory is a poor indicator of the future reference behavior of that page or segment.

Considerably more efficient is the Least-Recently-Used (LRU) policy, which selects as candidates for removal the page(s) or segment(s) that have not been referenced for the longest time interval. For this algorithm, the system must maintain, for each running program, a stack ordering of all pages or segments residing in memory according to the time of their last reference. This stack must be updated each time the page or the segment referenced is not the same as the last one. Efficient execution of this operation requires thus special stack updating hardware.

The size of the partition allocated to each running program is an important factor of the performance of all fixed-partition replacement algorithms. Since the memory requirements of programs may widely differ within a typical workload, proper tuning of the virtual memory normally requires a different partition size for each program--or each class of programs.

A more important problem occurring with fixed partition policies is that the memory requirements of a program often vary during its execution. A better solution should thus consist of allowing the memory space allocated to each program to vary dynamically during its execution. This can be realized either by global policies or by local variable partition policies.

1.2.3.2. Global Policies

Global policies allow any faulting program to obtain more space by reclaiming space previously allocated to itself or to any other program. Examples of such strategies are Global LRU and Global FIFO. As these algorithms manage the whole memory as a single pool, they actually implement a variable partition scheme.

Like its local counterpart, the Global LRU algorithm requires a special stack updating hardware; such a feature was indeed developed for the CDC STAR system, which implemented a Global LRU replacement policy [Requ78]. Because of this requirement, other system designers have rather chosen to develop alternative replacement policies similar to Global LRU but easier to implement [East79].

The best known of these policies is the Clock-1 replacement policy implemented in MULTICS [Corb68] [East79]. In this system, all page frames have a *use bit* set by hardware each time any information contained in the page frame is referenced. When a page fault occurs, a software pointer starts scanning the use bits of all page frames. If the bit is on, it is turned off; if the bit is off, the scan stops and the page frame content is returned to the secondary store. The current location reached by the pointer is remembered and will be used as the starting point for the next scan.

Unlike their local counterparts, global policies provide no direct way for controlling the amount of memory allocated to each individual program. The only possible way for tuning the system is by acting on the number of jobs allowed to compete at any time for memory, i. e., on the multiprogramming level.

1.2.3.3. Local Variable Partition Policies

Local variable partition algorithms are somewhat more ambitious: They attempt to evaluate the memory requirements of each program and to ensure that these requirements will be met for all currently active programs.

The best known example of such algorithms is the *Working Set* policy developed by Denning for paging environments [DenP66] [DenP68a] [DenP72] [DenP80]. This policy uses a control parameter τ, known as the *window size*, and defines the *working set* of a program at time t as the set of all pages that have been referenced in the interval $[t-\tau+1,t]$. It stipulates that all pages members of the current working set must reside in memory while the others may be returned to the secondary store.

Several extensions of the working set algorithm to segmentation environments have been proposed by Denning and Slutz [DenP78]. The *Time-Window Working Set* policy (TWWS), for instance, removes segments when they have not been referenced for T time units; it is quite similar to the original paged working set policy with a window size τ=T+1 [DenP78]. Another such extension is the *Space-Time Working Set* strategy (STWS), under which any segment of size s_x is removed from memory as soon as the duration of the interval since the last reference to it reaches T/s_x time units.

The original working set policy and all its extensions to segmentation environments need special hardware to detect when a page or a segment has not been referenced for a given time interval. Some variants of the working set policy have been aimed at systems lacking this special hardware but having use bits. Let us mention here the *Sampled Working Set* policy [Rodr73] [Ferr75] which samples periodically the use bits of all pages residing in memory and expels then all pages that have not been referenced for a given number of sampling intervals.

Another example of a local variable partition policy is the Page Fault Frequency Algorithm (PFF) developed by Chu and Opderbeck [ChuO72] [Opde74] [Chu76]. This algorithm defines its control parameter T as a critical inter-fault interval, and states that pages will be expelled at fault time if and only if

(a) they have not been referenced since the last fault time, and

(b) the time interval between the two faults is greater than T.

Thus, the PFF algorithm automatically increases the memory occupancy of programs at fault time as long as their fault frequency exceeds 1/T. On the other hand, its attempts to reduce their memory occupancy when this frequency falls below 1/T. The purpose of this feedback mechanism is to allow the algorithm to adapt to dynamic changes in program behavior during execution. As we will see later, this scheme has also some unfortunate consequences on the stability of the algorithm. In particular, Franklin, Graham and Gupta have found that an increase in T may sometimes increase the page fault frequency of programs, rather than decreasing it as expected [Fran78].

1.3. PERFORMANCE CONSIDERATIONS

1.3.1. The Locality Principle

Virtual memory systems essentially provide users with a computing environment where the memory requirements of each program are reduced at the cost of an increase of its I/O activity and a slowdown of its execution. The quality of the trade-off obtained will of course depend on the effectiveness of the system's memory management policies. A second factor, less obvious but even more important, is the referencing behavior of the programs constituting the system's workload.

Consider, for instance, the case of a program accessing in a random way all portions of its address space. Suppose that all program addresses have the same probability to be the next reference. Let V be the size of the program's address space and S the size of the memory space allocated to it. Then, for any realizable combination of fetch and replacement policies, the probability that a given reference will cause a fault will be equal to $V-(S/V)$. Thus, if 30% of the program's address space resides in memory, one may expect that 70% of its references will result in a fault condition. Such fault rates would bring the average access time of the virtual memory very close to the one of the secondary store and are thus not acceptable.

Fortunately enough, the vast majority of programs do not exhibit this kind of referencing behavior. They tend rather to concentrate their references in a relatively small, reasonably stable subset of their address space and this subset is largely made up of addresses in close spatial proximity. This property has been described by Denning [DenP68b] as the *locality principle*. It implies basically two things. First, there is a high probability that the next reference of a program will be to an address in close spatial proximity with the addresses that have been recently referenced. If the program is executed in a virtual memory system, this address will be probably contained in a page or a segment already fetched into memory. This is termed *spatial locality*. Second, the composition of this subset of preferred addresses will remain rather stable over any short interval of time. Thus, the same pages or segments will tend to remain referenced during relatively long periods of time. This is known as *temporal locality*.

1.3.2. Sources of Performance Degradation

The locality principle states that efficient operation of a virtual memory system can only be achieved if each active program is allowed to keep in memory all the pages or segments it is currently referencing. If this condition is not fulfilled, the rate of transfer of pages or segments between the memory and the secondary store will rapidly increase and eventually exceed the capacity of the I/O channel. The phenomenon has been described by Denning as *thrashing* [DenP68b]. It results in an underutilization of the CPU because the system will spend a considerable amount of its time in a state where all active program will be waiting for a missing page or segment and thus will be unable to run. Prevention of trashing will then essentially consist of ensuring that a sufficient amount of memory can be allocated to each active program.

Another source of performance degradation can be overgenerous allocation of memory to programs. If an excessive amount of memory is allocated to one or more programs, then the total number of programs allowed to reside simultaneously in memory will be reduced below the optimum. This, in turn, can cause CPU underutilization either directly or by increasing the amount of job swapping.

1.3.3. Evaluating Virtual Memory Performance

Because of the strong influence of program behavior on virtual memory performance, one cannot evaluate the performance of any virtual memory without relating it to the characteristics of the programs constituting its workload. One cannot compare, for instance, two replacement policies without defining first the set of programs for which the comparison will be carried out. Moreover, the validity of the comparison will strongly depend on the extent to which these programs are representative of the system's real workload.

Ideally, the performance of any virtual memory should always be evaluated in terms of its contribution to the overall performance of the system as measured by the system's throughput, its response time for a given job or a given class of jobs, and so forth. This approach, however, raises several problems. First, it supposes that we know what constitutes a "typical" workload for the system under study. Second, it requires that we are able to reproduce, for each experiment, all environmental factors like the system's initial state, the pattern of job arrivals and so forth. Finally, the multiplicity of factors that must be taken into account complicates the interpretation of the results of the evaluation study [Smit80] [Smit81].

A simpler alternative exists. It consists of evaluating the performance of the virtual memory on a per program basis. We will thus monitor--or simulate--the execution of a given program in the virtual memory environment under study and measure the performance of the virtual memory for that job. The procedure will be repeated for several "typical" programs and the obtained values will then be used as indicators of the global performance of the virtual memory. The approach has one main drawback. It assumes implicitly that all memory management decisions can be made on a per program basis and that interactions between concurrently executing programs do not affect significantly the overall performance of the virtual memory. This assumption holds more or less for local replacement policies but is totally unrealistic for the global ones. On the other hand, we get a much clearer picture of the individual behavior of each program. This is especially valuable when one wants to compare the performances of several versions of the same program.

1.3.4. Local Indices of Virtual Memory Performance

When a program executes in a virtual memory environment, the performance of the virtual memory for that program can be directly expressed in terms of the memory space allocated to the program and the cost of the information transfers between memory and secondary store.

Evaluating the memory space allocated to a program running under a fixed partition replacement policy is a trivial matter: the space allocated to the program is given by the size of the partition in which it is running. Problems arise with variable partition policies when one attempts to define an "average" memory occupancy. First, no average value of memory occupancy can account for the dynamics of the memory allocation process. Smith has indeed pointed out [Smit76] that the presence of sharp peaks in the memory occupancy curve can trigger unnecessary deactivations of other processes by the system. The presence and the intensity of these peaks is thus a factor of the system's overall performance that does not show up in the mean memory occupancy. Second, one can even argue on the procedure to be used for computing this mean memory occupancy. Should it take into account only the times during which the program is running or include the times during which it is waiting for a missing page or segment? The first alternative at least guarantees that the final value will not depend on any estimates of page or segment wait times.

Evaluating the cost of information transfers between the memory and the secondary store will also involve some approximations. A universal one is to neglect the cost of transfers from memory to the secondary store. These transfers occur each time the replaced page or segment has been modified during its residence in memory. Yu's study [Yu76] has shown that these "write-backs" made practically no difference when comparing memory management policies.

In paging systems without prefetching, each page fault brings exactly one page into memory. Since all pages have the same size, the number of faults occurring during the execution of the program--or their frequency--constitutes a natural indicator of the cost of information transfers within the memory hierarchy. Because of unequal segment sizes, the same assumption is not true for segmentation systems where the average time required to service a fault is indeed a linear function of the size of the segment causing the fault. More precisely, if s_i is the size of the faulting segment, the average time T_w required to service the fault will be given by

$$T_w = T_l + T_t . s_i \qquad (1.1)$$

where T_l is the mean access time of the secondary store and T_t the mean time to transfer one unit of data--for example, one byte.

The *swapping load* L_s will then be defined as the sum of all delays occurring at segment fault times and caused by the secondary store latency or the segment transfer times. Representing by r the number of faults occurring during the execution of the program and by $N_{b,in}$ the total number of bytes brought into memory, one can thus write

$$L_s = r \,.\, T_l + N_{b,in} \,.\, T_t \qquad (1.2)$$

This expression also applies to systems implementing demand prefetching as long as all pages or segments fetched at any fault time are fetched in one I/O operation.

1.3.5. Two-Dimensional Representations

Memory occupancy and swapping load express two complementary aspects of the local performance of a virtual memory. It has thus become customary to represent this performance by a two-dimensional curve $L=L(S)$ relating the swapping load L to the corresponding value of the program's memory occupancy. Figure 1.1 represents one such curve collected for an APL interpreter running under a pure Working Set policy with a page size of 2 Kbytes. As it is customary in paging environments, the swapping load has been expressed in terms of the program's average page fault rate$f$.

Another popular representation of the local performance of a virtual memory is the program's *lifetime curve*, which is the function $g(S)=1/f(S)$ defined for all $S > 0$ and returning the mean time between faults when the mean memory occupancy is S. Figure 1.2 displays the lifetime curve of the same APL interpreter, running again under a Working Set policy with a page size of 2 Kbytes. Typical lifetime curves exhibit *knees*, i. e., points at which $g(x)/x$ is locally maximum. The most pronounced one, known as the *primary knee*, is geometrically defined as the highest point of tangency between a ray from the origin and the curve; it corresponds also to the global maximum of $g(S)/S$.

By putting together the curves corresponding to the same program and two different memory policies, one can decide which one is the better and for which range of memory occupancies. Similarly, one can compare the performances of two versions of the same program.

This leaves, however, open the problem of comparing two operating points of a program. Suppose, for instance, that we want to compare two versions of the same program under the same memory policy with the same value of the control parameter (for instance, the window size for the Working Set policy or the number of page frames for the Local LRU policy). We will run the two versions of the program and obtain a memory occupancy S_1 and a swapping load L_1 for the first version of the program and corresponding values S_2 and L_2 for its second version.

If we have $S_1 \leq S_2$ and $L_1 \leq L_2$ (but not $S_1 = S_2$ and $L_1 = L_2$), we can state that version 1 of the program performs better. Similarly, we will say that version 2 is better if we have $S_1 \geq S_2$ and $L_1 \geq L_2$ (but not $S_1 = S_2$ and $L_1 = L_2$). In all other cases, we cannot say that one version of the program is clearly better than the other. To do so, we would need to be able to express the performance of the virtual memory by a single performance index. This index indeed exists and is called the *Space-Time Product*.

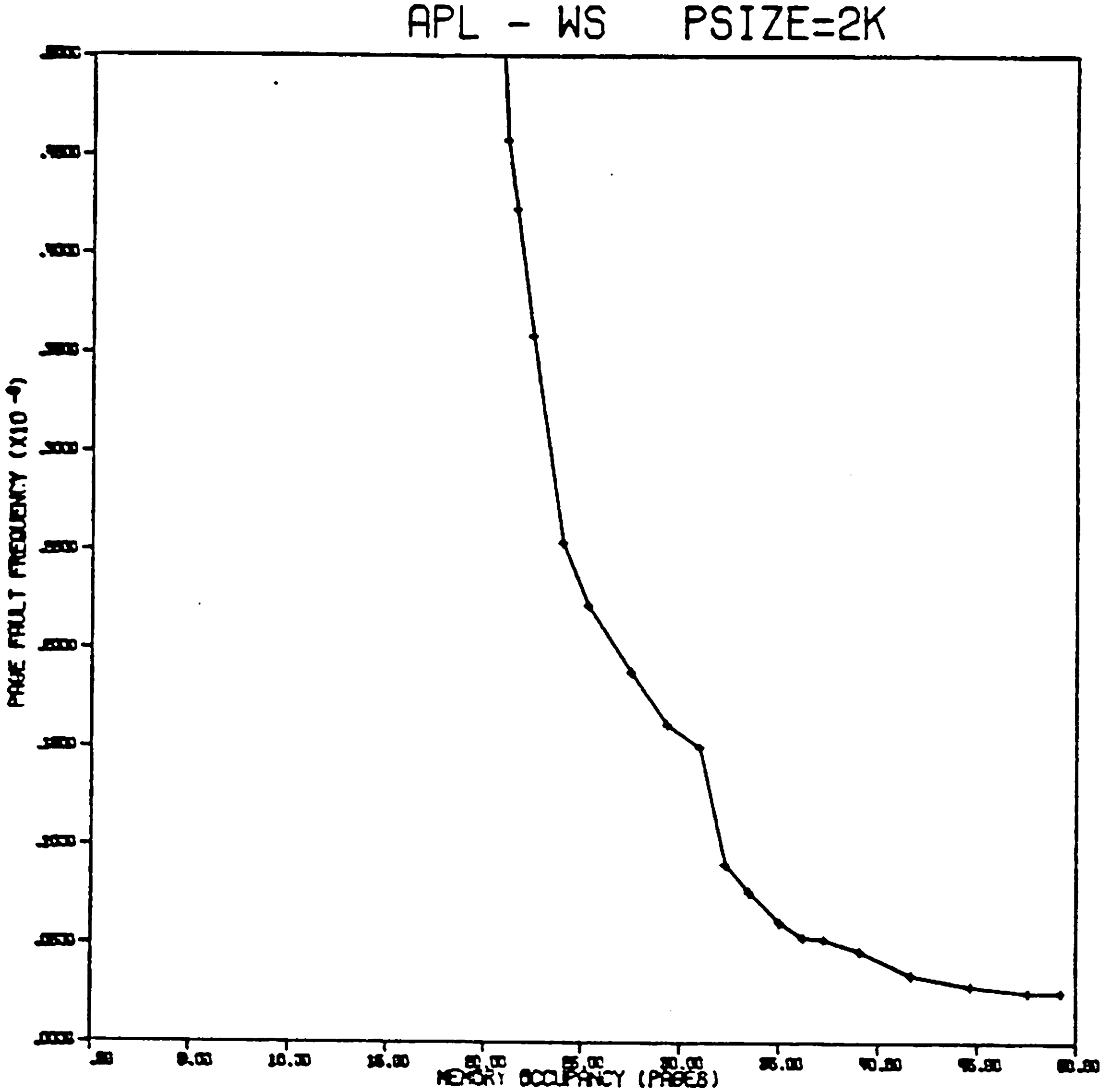

Figure 1.1

1.3.6. The Space-Time Product.

By definition [Bela69], the space-time product C characterizing the behavior of a program running in a virtual memory environment during a real time interval (0,t') is given by the integral

$$C = \int_0^{t'} S(u)du, \qquad (1.3)$$

where $S(u)$ is the number of memory page frames occupied by the program at time u.

As we have just defined it, this criterion expresses the *storage costs* corresponding to the effective memory occupancy of the program during the interval $(0,t')$.

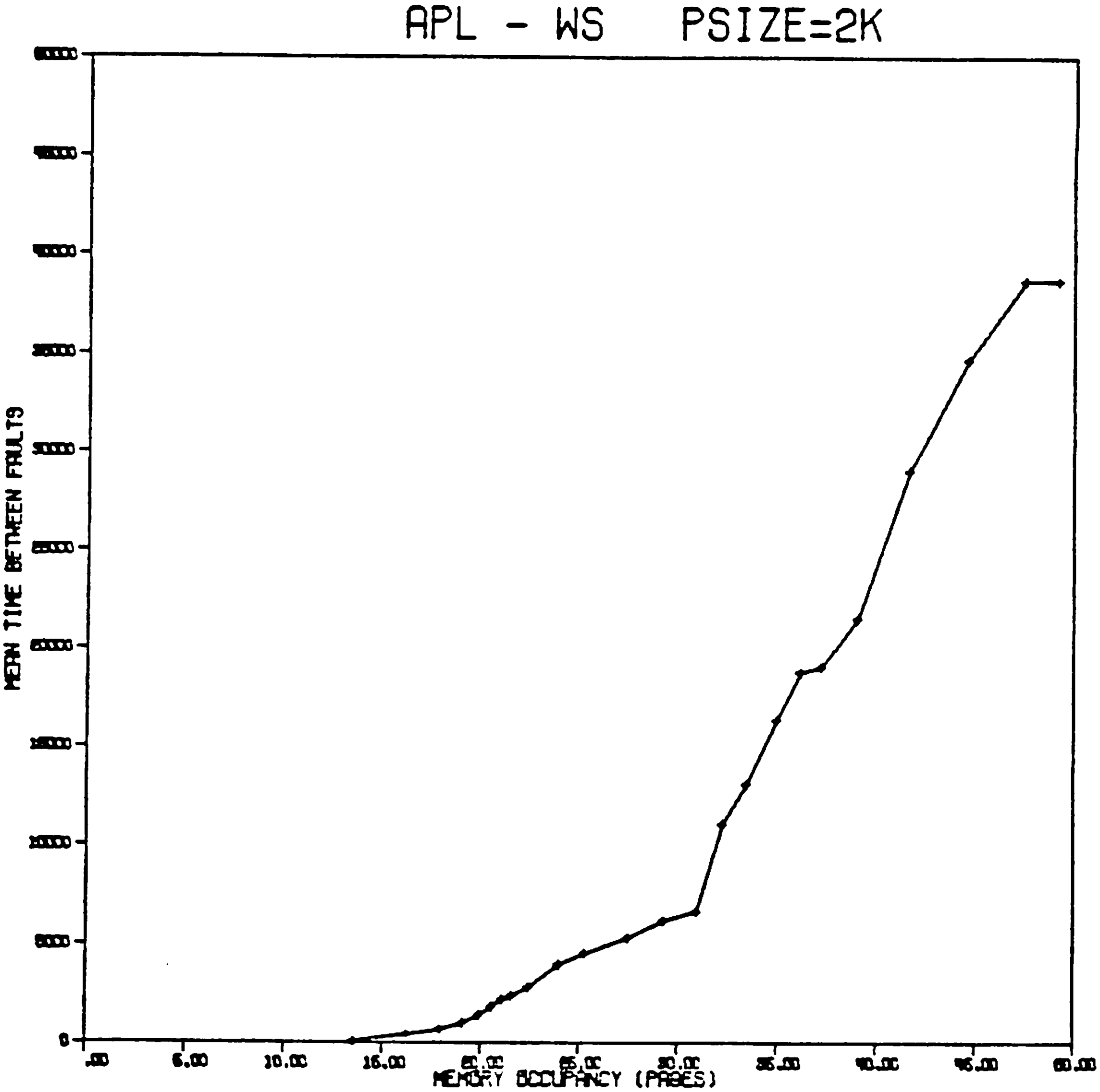

Figure 1.2

Being defined as a real time interval, $(0,t')$ includes the times during which the program was actually running as well as the *dead times* during which its execution was suspended. In a multiprogramming environment, these dead times may result from two causes: the faults that may occur during program execution and the parallel execution of other programs.

In order to eliminate this extraneous influence and to keep only the intrinsic factors characterizing the program behavior in a given environment, we should only take into account the delays caused by faults [Chu072].

The space-time product characterizing the behavior of the program being executed in a segmentation environment during a *virtual* time interval $(0,t)$ will thus be given by

$$C = \int_0^t S(u)du \; + \sum_{j=1}^{r} S(t_j).(T_l + T_t.s_j), \tag{1.4}$$

where r is the total number of faults occurring during $(0,t)$, t_j the time of the j-th fault and s_j the size of the segment(s)-- or the page(s)--brought into memory at time t_j.

In paging environments without prefetching, a single page is brought into memory at each page fault and the expression reduces to

$$C = \int_0^t S(u)du \; + \sum_{j=1}^{r} S(t_j).T_w \tag{1.5}$$

Especially in paging environments, the space-time product is often approximated by

$$C = \bar{S}.(1 + f.T_w), \tag{1.6}$$

where $\bar{S}$ is the mean memory occupancy, t the virtual time and $f = r/t$ the fault frequency. Though easy to compute, this approximation is not very accurate: Graham [Grah76] and Lau [Lau 76] have found that it is not consistently higher or lower than the true value of C, and can sometimes be in error by as much as 20%.

1.3.7. Virtual Memory Tuning

Another problem, closely linked with these performance issues, is how to adjust the control parameters of a virtual memory in such a way as to maximize the system performance for a given workload.

As we said earlier, virtual memories implementing a global replacement policy have only one control parameter, namely the system's multiprogramming level. Tuning the virtual memory will thus consist of determining an optimal multiprogramming level for the system. In general this optimum varies with the workload; hence an adaptive control scheme is needed to readjust it periodically. Several of these have been proposed so far. Among them, the *L=S criterion*, which states that the optimal multiprogramming level is reached when the system's overall mean-time-between-fault is equal to the mean fault service time [DenP76a], and the *50% criterion*, which recommends to keep the utilization of the paging device around $(50+d)$%, where d is a constant less than 10 [Lerou76] [DenP76b].

Local replacement policies, on the other hand, permit individual tuning of the memory space allocated to each program residing in memory. This indeed allows each program to be run at its minimum space-time product for that policy, but poses also the problem of selecting the proper value of the control parameter for each running program.

An elegant construction has been proposed by Denning and Kahn for solving that problem. Since the primary knee of a program's lifetime curve $g(S)$ corresponds to a maximum of $g(S)/S$, it also minimizes $S/g(S)$. By definition of the lifetime curve, we have $g(S)=1/f(S)$ and $S/g(S)$ can be rewritten as $S.f(S)$, where S stands for the program's mean memory occupancy and $f(S)$ for its fault frequency $f = r/t$. Recalling equation (1.6), one can thus see that operating a program at the primary knee of its lifetime curve will minimize the component of its space-time product due to segment or page faults. When T_w is large, this will also approximately minimize the program's space-time product.

Define the ***virtual-to-real time ratio*** of a program as the quotient

$$h(S)=t/(t+L_s) \qquad (1.7)$$

where t stands for the total virtual execution time and L_s for the swapping load. (Recall that, in paging environments, L_s is equal to $r.T_w$.) Consider now the curve $h(S)$ relating the virtual-to real time ratio of a program to its mean memory occupancy. This curve has a shape similar to the one of the lifetime curve. Its primary knee will correspond to a maximum of $h(S)/S$, i. e., to a minimum of $S.(t+L_s)$, which is proportional to the approximate expression of the space-time product given by equation (1.6). Thus, running a program at the primary knee of h(S) will ensure that the program is running very close to its optimum space-time product.

In order to ensure that the exact maximum is reached, one should take into account for the computation of the program's mean memory occupancy the time intervals during which the program has been waiting for a missing page or segment. Let us represent by $\tilde{S}$ the mean memory occupancy computed this way. Then the primary knee of $h(\tilde{S})$ will correspond to a minimum of $\tilde{S}.(t+L_s)$, which is by definition the space-time product C of the program.

1.4. CONCLUDING REMARKS

Despite all advances made in understanding the problems of virtual memory management, obtaining an acceptable level of performance from a virtual memory remains a non-trivial task. The classical approach to this problem has essentially consisted of tuning the virtual memory management policies. As we have seen, this approach has resulted in the design of better replacement policies, the implementation of better load control mechanisms, and so forth.

Another approach is possible. It consists of improving the behavior of programs and adapting program structure to the requirements of virtual memory systems. From a user's viewpoint, this approach offers the main advantage of not requiring any modifications to the operating system. As we will see, this does not prevent it from being quite often surprisingly efficient.

- *Phase II:* A *restructuring algorithm* produces a graph model of the program. Each node i of this graph will represent a block and each edge joining two nodes i and j will have a label a_{ij} that will express some measure of the desirability of placing the two blocks into the same page. These a_{ij} will be referred to as *affinities*. The precise definition of the affinities will depend on each particular restructuring algorithm.
- *Phase III:* A *clustering algorithm* is applied to the restructuring graph. This algorithm will attempt to minimize the sum of the affinities between blocks belonging to different clusters while enforcing the condition that each cluster must fit into one single page. Thus blocks with the strongest mutual affinities will tend to be gathered into the same page.
- *Phase IV:* Blocks are relocated in the program's virtual address space according to the results of the clustering algorithm. At the end, one will normally attempt to fill gaps left by clusters the size of which is not exactly equal to one page.

2.3. ANALYSIS OF THE CLUSTERING PHASE

The aim of the clustering phase is to find a partition of blocks into pages that will group together the blocks exhibiting the strongest mutual affinities. The search for this ideal partition will lead to two main problems. First, one has to precisely define the notion of mutual affinity between n blocks when n is larger than 2. Secondly, one has to find a relatively efficient algorithm to perform the task of finding the partition that will effectively maximize the intra-cluster affinities.

2.3.1. Defining affinities between more than two blocks

Like in many other clustering problems, the problem of defining affinities--or any other nearness index--between more than two blocks remains an open question, even complicated, in this case, by the fact that the restructuring graph is not a totally accurate model of the program behavior.

Consider, for instance, the case of a restructuring procedure whose objective is to minimize the number of page faults occurring during the execution of the program. It is then natural to define the affinity between two blocks i and j as the number of times a page fault could be avoided if the two blocks i and j were stored in the same page.

Suppose that we have now to evaluate the affinity between a block i and the cluster containing the two blocks j and k. The affinity between i and $\{j,k\}$ must express the number of times a page fault can be avoided by storing block i and cluster $\{j,k\}$ into the same page. Now it can happen that storing blocks j and k into the same page did not avoid any of the page faults that would be avoided if j or k were stored with i. Then the affinity $a_{i\{jk\}}$ can be defined as

$$a_{i\{j,k\}}=a_{ij}+a_{ik}$$

However, it can also happen that some of the page faults that can be avoided by storing i with j or k have been already eliminated by storing j and k together. Then

$$a_{i\{j,k\}}<a_{ij}+a_{ik}$$

and we have no ways of estimating the number of these page faults. The only alternative would be to drop our graph representation of program behavior in favor of a more complex model taking into account interactions between more than two blocks. This would greatly increase the time and space requirements of the restructuring procedure, which explains why no attempts have been made to pursue any further this approach.

CHAPTER II

THE RESTRUCTURING APPROACH

2.1. INTRODUCTION

As we have seen, the vast majority of the efforts devoted to the optimization of virtual memory systems have been directed to the design of more efficient, and often more complex, memory management algorithms. Considerably less attention has been paid, on the other hand, to the alternative way of obtaining a better performance of virtual memory systems, namely by increasing the degree of locality of the programs to be executed [Braw68] [Braw70] [Baer72].

An obvious solution should be to teach programmers to write more local programs. This approach has proved quite effective in areas like numerical analysis. For instance, one can rewrite matrix manipulation algorithms having in mind to increase the degree of locality of references within each matrix. This approach is systematically used in the LINPACK mathematical subroutines package [Dong79].

A similar technique consists of modifying the implementations of large data structures. For instance, one may replace the traditional column--or row--storage of matrices by a submatrix storage where each matrix is partitioned into equal-size square submatrices occupying each one page or one segment [McKe69] [Fisc79].

Rewriting programs in order to improve their locality is not always that easy. Besides, it offers the serious drawback of going against the objective of keeping the memory hierarchy transparent to the system's users.

A possible alternative is then offered by *program restructuring* [Come67] [Tsao72] [Hatf71] [Ferr73] [Ferr78]. Unlike other methods, program restructuring deals with programs already written and essentially consists of rearranging the order in which the various blocks of code and data constituting a program are stored in the program's virtual address space. Program restructuring thus operates in a way that is totally transparent to the program authors, maintainers and users. It even tends to work better for programs consisting of many relatively small modules.

For several reasons, primarily historical, program restructuring has been almost exclusively applied to programs running in paging environments. The two following chapters will thus discuss program restructuring as a tool for improving the behavior of programs in paging environments, while a special chapter will be devoted to the specific problems of segmented virtual memories.

2.2. ORGANIZATION OF PROGRAM RESTRUCTURING PROCEDURES

With very few exceptions [Babo77], all restructuring procedures share the same organization in four phases [Ferr74c]:

- *Phase I:* The program to be restructured is partitioned into *blocks*, the size of which should ideally be less than or equal to one half of the page size. Each of these blocks can be any piece of code or data. The relocation process is however much simplified if they are chosen to be the various relocatable object modules constituting the program.

Several attempts have been made to find experimentally the best definition of the affinity between two clusters of blocks. Masuda et al. [Masu74] found that a "modified average" method taking into account the sizes of the two clusters performed best. Achard et al. [Acha78] experienced even better results using a Jacquard index defined as

$$S(C_1,C_2) = s(C_1,C_2) \,/\, (u(C_1,C_2)+v(C_1,C_2))$$

where $s(C_1,C_2)$ is the sum of all a_{ij} such that $i \in C_1$ and $j \in C_2$, $u(C_1,C_2)$ the sum of all a_{ij} such that $i \in C_1$ and $j \notin C_2$, and $v(C_1,C_2)$ the sum of all a_{ij} such that $i \notin C_1$ and $j \in C_2$.

One should however remember that the performance of a clustering algorithm using any of these definitions of the affinity between two clusters will strongly depend on the idiosyncrasies of the affinities produced by the restructuring algorithm.

Recall, for instance, our argument on the fact that the affinity between two clusters is not merely the sum of affinities between all elements of the first cluster and all elements of the second. This argument was introduced assuming that the affinity between two blocks i and j was the number of page faults that could be avoided by storing i and j into the same page. It does not necessarily hold for other definitions of inter-block affinities, like, for instance, that which calls the affinity between two blocks the number of times the two blocks have been consecutively referenced during the execution of the program.

2.3.2. Algorithm Complexity Issues

Assume, for the sake of simplicity, that the affinity between two clusters of blocks has been defined as the sum of the affinities between all blocks of the first cluster and all blocks of the other one. Now, the problem of finding the partition of blocks into pages grouping together the blocks with the strongest mutual affinities reduces to the problem of finding the partition maximizing the sum of affinities between blocks sharing the same page. In the general case where more than two blocks can reside in the same page, this problem can be proven to be NP-hard.

Consider indeed a restricted case where each node of the graph represents a block of size 1 and each edge has a label equal to 1. Assume now that we want to maximize the sum of inter-cluster edges and that the maximum cluster size is 3. This problem is known as partitioning a graph into triangles and has been found to be NP-complete [Karp75]. It follows then that the general clustering problem must be NP-hard.

One can thus safely conjecture that no polynomial time algorithm will ever be able to determine an optimal block to page mapping for an unrestricted restructuring graph. Polynomial time algorithms however exist if we assume that there will never be more than two blocks per page; the clustering problem then becomes a weighted matching problem, which can be solved in $O(n^{2.5})$ time [Even75].

Because of the high cost of optimal clustering algorithms, program restructuring procedures uniformly resort to near-optimal algorithms. Almost all algorithms utilized are in fact variants of the same hierarchical clustering algorithm which starts by assigning a separate cluster to each block and merges at each step the two clusters having the strongest mutual affinities until all clusters have reached their maximum size [Ferr75d] [Masu75] [Acha78].

2.3.3. Conclusion

The clustering phase raises several problems for which no satisfactory answers exist. As a result, we will be generally unable to obtain an optimal block-to-page mapping.

Fortunately enough, this shortcoming does not seem to affect unduly the global performance of the restructuring procedure. Ferrari has pointed out that there is typically a small subset of blocks characterized by large mutual affinities and playing a crucial role in the clustering phase (as reported in [Lau 79]). The bulk of the improvement of program performance obtained by the restructuring procedure depends on the arrangement of these blocks and one may conjecture that only slight additional improvements could result from optimal clustering with respect to any sub-optimal solution properly grouping these crucial blocks.

2.4. ANALYSIS OF THE RESTRUCTURING PHASE

Being responsible for constructing the restructuring graph, the restructuring algorithm constitutes the key part of any program restructuring procedure. Quite often, it will indeed be the most time-consuming part of the procedure; in any case, it will be the key factor of the overall performance of the restructuring procedure.

2.4.1. Static Algorithms

Earlier algorithms based their definitions of affinities on an analysis of the static structure of the program: blocks calling or referencing each other, blocks nested inside loops, and so forth. These algorithms were the natural offspring of algorithms previously developed for the automatic generation of overlay structures [Rama66] [Lowe70] [VerH71].

A more recent example of such *static* algorithms is the AFFINF algorithm proposed by Snyder [Snyd78]. This algorithm uses a directed graph and defines the affinity a_{ij} between two nodes as the sum of the raw number of source references from i to j plus ten times the number of source references inside a loop in block i to block j. References that occur in a nested loops count geometrically, i. e. as 10^n, where n is the depth of the nesting.

The main interest of these static algorithms resides in their ease of implementation and their relatively low cost. On the other hand, a static restructuring graph misses the whole dynamic behavior of the program, e. g. how often and when a block is effectively referencing and calling another block. For this reason, static algorithms appear now to be outclassed by the so-called *dynamic* algorithms which take into account the behavior of programs at execution time, as represented, for instance, by their reference strings.

2.4.2. Dynamic Algorithms

Dynamic algorithms base their definition of affinities on data collected during one or several "typical" runs of the program to be restructured. In general dynamic algorithms are more expensive than static ones but work much better provided it is possible to define such thing as a "typical" execution of the program to be restructured, that is to come up with "typical" input data. This task can be very difficult for certain types of programs, the behavior of which is strongly data dependent. Strong experimental evidence exists however showing that the behavior of many interesting types of programs--like compilers--is reasonably insensitive to input data as far as the restructuring process is concerned.

The primary cost of a dynamic restructuring algorithm is the one of collecting data on the dynamic behavior of the program to be restructured. Gathering of such information normally involves simulating or monitoring one or more executions of the program one wants to restructure. The cost of this procedure essentially depends on the hardware tools available--generally none--and on the accuracy of the measurements. A trace of procedure calls and returns can be obtained by instrumenting the program to be restructured either at the source or at the linking level. On the other hand, obtaining a full trace of program execution including all data references will normally require running the program through a software interpreter.

These cost considerations have the unfortunate effect of reducing the field of application of dynamic restructuring algorithms to programs which are often executed, like compilers, text processors, and so forth. This restriction is however much less severe than it appears because these programs constitute the bulk of the system load in many installations. Moreover, this limitation can be somewhat relaxed for hybrid restructuring algorithms like Babonneau and Achard's RELIEUR [Acha75] [Babo77] [Acha78], which also takes into account the static structure of the program.

2.4.3. Review of Dynamic Algorithms

An early example of dynamic algorithms is the *Nearness Method* developed by Hatfield and Gerald [Hatf71]. Like most other dynamic algorithm, it assumes that we have collected a *block reference string* of the program to be restructured. This block reference string will consist of the sequence of all blocks $b_1, b_2, \ldots, b_n$ referenced during an execution of the program.

Suppose that block i often appears after block j in this block reference string. This means that the two blocks are often referenced one after the other. A block-to-page mapping in which the two blocks would be stored into the same page would thus increase the locality of the program and probably avoid several page faults. A possible measure of the affinity--or nearness--between two blocks i and j is then given by the number of times these blocks have been consecutively referenced during the execution of the program.

The matrix $A = (a_{ij})$ representing the restructuring graph can thus be constructed in three steps:

```
(i)   For all $i$ and $j$ do   $a_{ij} := 0$   od;
(ii)  For all $t$ from 2 to $n$ do
          $a_{b_{t-1},b_t} := a_{b_{t-1},b_t} + 1$
      od;
(iii) For all $i$ and $j<i$ do   $a_{ij} := a_{ji} := a_{ij} + a_{ji}$   od.
```

The last step of the algorithm ensures that the matrix is symmetrical and represents thus a non-directed graph.

The most obvious flaw of the Nearness Method consists of only taking into account interactions between blocks that are referenced directly one after the other. Suppose, for instance, that blocks a, b and c are successively referenced in the order aabbbaaabbc. One may then expect that grouping blocks a and c together would have the same beneficial effect as grouping b and c together. This is not recognized by the Nearness Method, which does not detect any affinity between a and c.

In order to overcome this limitation, Ryder [Ryde74] has proposed a scheme where the affinity between two blocks i and j is incremented by a_0 when the two blocks are referenced immediately one after another, by a_1 when references to blocks i and j are separated by a reference to another block, by a_2 when they are separated by references to two other ones, and so forth. Ryder reports good results obtained with increments a_i decreasing linearly with the number of blocks separating the two references up to a distance of 4, i. e. a_0=5, a_1=4, a_2=3, a_3=2 and a_4=1, but admits that these values are absolutely arbitrary.

The Nearness Method and Ryder's algorithm both attempt to store within a single page the blocks that are the most often referenced one after the other. The beneficial effect of the restructuring process should thus be a reduction of the page fault frequency of the restructured program and, perhaps, some decrease of its memory occupancy.

An alternative solution consists of aiming the restructuring algorithm at reducing the *working set size* of programs for some window size τ more or less arbitrarily chosen. This approach has been followed by Masuda, Noguchi and Okhi [Masu74]. Their algorithm evaluates periodically the working set of blocks of the program to be restructured during one or more executions. This working set of blocks is defined as the set of all blocks that have been referenced during the last τ references. The affinity a_{ij} between two blocks is then the number of times the two blocks have been members of the same working set.

When Masuda's algorithm is applied to a program being executed under a working set replacement strategy, any reduction in the program's working set size will automatically cause an equal reduction of its memory occupancy, provided that the restructuring algorithm and the replacement policy have used the same window size. Masuda's experiments, backed by our own, indicate that similar beneficial effects may be expected from the restructuring procedure as long as the window size used by the replacement policy remains larger than the one selected by the restructuring algorithm. The picture becomes however less clear when the replacement policy behavior significantly departs from the one of a working set strategy with a large window size.

A significant contribution was thus made by Ferrari [Ferr73] [Ferr74a] [Ferr74b] [Ferr74c] [Ferr75] [Ferr76a] [Ferr76b] [Ferr77a], who introduced the concept of *strategy-oriented restructuring* and proposed a method to define affinities which

- is explicitly based on a *measurable indicator* of the program's performance, like its page fault frequency or its mean memory occupancy, and
- takes into account the *memory management strategy* of the system in which the program will be run.

2.5. STRATEGY-ORIENTED RESTRUCTURING

Unlike other restructuring algorithms, each strategy-oriented algorithm is characterized by the performance index it attempts to optimize and by the replacement policy for which it is tailored.

2.5.1. Critical Algorithms

Critical Algorithms attempt to minimize the page fault frequency of programs being executed under several replacement policies. The Critical Working Set algorithm (CWS) [Ferr74b], which is probably the best known example of these algorithms, attempts for instance to minimize the page fault frequency of programs assuming that they will run under a working set replacement policy.

Under a working set policy with window size τ, all pages that have been referenced at least once during the last τ references are kept in memory. Thus the only references that are susceptible to cause a page fault are the ones referring to a block that has *not* been referenced during the last τ references. We will call these references *Critical References*. The set of blocks that are guaranteed to be present in memory when the t-th reference is issued will be called the *Resident Set of Blocks* $R_b(t)$ of the program at time t. (We will assume that $R_b(0)=\emptyset$ and that $R_b(1)$ contains the first block referenced. This definition is slightly different from the one adopted by Ferrari and Kobayashi, according to which $R_b(1)=\emptyset$.) Thus, under a working set strategy, $R_b(t)$ contains all blocks that have been referenced at least once during the τ last references, including the current one.

Since the purpose of the restructuring algorithm is to reduce the page fault frequency of programs, the best measure of the affinity between two blocks i and j is given by the number of page faults that could be avoided by storing the two blocks together. This quantity can be estimated by counting the number of times a critical reference to i or j occurs while the other is a member of the Resident Set of Blocks at time $(t-1)$.

Let thus $b_1, b_2, \ldots, b_n$ be a block reference string collected during one execution of the program we want to restructure. The matrix $C=(c_{ij})$ representing the restructuring graph will have initially all zero entries and will be 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 one all $c_{ij}$'s such that $i \in R_b(t)$ and $j = b_t$
    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 [Ferr76b]),FIFO (CFIFO [Ferr76b]), Sampled Working Set (CSWS [Ferr75][Ferr76a]) and global LRU environments (CPSI [Ferr77a]). They can be derived from the CWS algorithm by modifying in an appropriate manner the definition of the Resident Set of Blocks $R_b(t)$. For instance, the Critical LRU algorithm, which applies to programs running under a Local LRU policy, is essentially identical to the CWS algorithm with the only difference that the Resident Set of Blocks for a LRU policy with a partition size of m pages is made of the last m blocks that have been referenced [Ferr75].

2.5.2. Minimal Algorithms

Unlike Critical Algorithms, Minimal Algorithms [Ferr76] attempt to minimize the *memory occupancy* of restructured programs. To achieve this goal, they attempt to store within a common page blocks that will be often simultaneously residing in memory. Thus, the algorithm will evaluate at fixed sampling intervals during a simulated execution of the program its current Resident Set of Blocks and increment by one all edges of the restructuring graph corresponding to a pair (i,j) of blocks simultaneously members of $R_b(t)$.

Let $b_1, b_2, \ldots, b_n$ represent again a block reference string collected during a run of the program to be restructured. Assume that the algorithm will update the restructuring graph each K references. Then, the matrix $M=(m_{ij})$ representing the restructuring graph 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) [Ferr76b], Sampled Working Set (MSWS) [Ferr76b] and Global LRU (MPSI) [Ferr77a]. Like their Critical counterparts, they differ from each other only in the way their Resident Set of Blocks is defined.

2.5.3. Influence of the Memory Management Strategy

As we said above, the only difference between two Critical or two Minimal Algorithms tailored to two different memory policies lies in the way their Resident Sets of Blocks are defined. Thus, obtaining the Critical or the Minimal Algorithm tailored to a given memory policy only requires the update of the routine evaluating $R_b(t)$.

This task is quite easy when the memory policy allows one to predict, at restructuring time, which blocks will be guaranteed to be present in memory at any time of a program's execution. Examples of such policies are the Local LRU and the Sampled Working Set policies.

We have already mentioned that the Resident Set of Blocks of a program run under a Local LRU policy with a partition size m contains the m last distinct blocks referenced by the program. Now, in the case of the Sampled Working Set Policy, we recall that the window size is an exact multiple of the time interval at which the page use bits are sampled. Let us assume a sampling interval of I references and a window size τ equal to KI references. Then the Resident Set of Blocks at time t can be defined as the set of all blocks that have been referenced at least once during the the last K sampling intervals excluding the current one. Thus $R_b(t)$ is made of all blocks referenced within the time interval [I(t div I - K), t], where the symbol div represents integer division.

The problem becomes somewhat more difficult when the memory policy does not allow us to predict accurately the Resident Set of Blocks of a given program. Such is the case with all global policies, including Global LRU and its variants.

Global policies allow indeed any faulting program to obtain more space by claiming space previously allocated to itself or to other programs. Thus, the paging activities of programs that execute concurrently may strongly interfere with each other [Oliv74][Smit80]. As a result, programs that require many pages in a short time interval tend to steal page frames from other programs. The Resident Set of Blocks of a given program will then depend on the paging behaviors of all programs concurrently residing in memory.

Fortunately enough, the resultant of these influences for Global LRU environments can be expressed by a simple model due to Bard, who has successfully tested it with the CP-67 system [Bard73] [Bard75]. Bard's model is based on the following observations: Under a global policy, programs can lose pages only when their execution is suspended. Interruptions of program execution take place when a page fault occurs, when the program issues an I/O and waits for its completion, or when the program's time quantum has expired. When the overall system paging activity is low, the inactive pages of a program will tend to remain in memory even after a relatively large number of interruptions. On the other hand, higher system paging activities will result in faster removal of unreferenced pages. One can thus summarize the global effect of these influences on the paging behavior of a given program by a single parameter Ψ, called the *Page Survival Index* (PSI), which is defined as the average number of interruptions that an unreferenced page can "survive" before being expelled from memory.

Ferrari and Kobayashi thus proposed [Ferr77a] to define the Resident Set of Blocks of a program being executed under a Global LRU policy as the set of all blocks that have been referenced at least once during the time interval covering the last Ψ interruptions of program execution. (Since one does not know, at restructuring time, which *block* faults will cause a *page* fault, one must assume that each block fault will result in a page fault and thus an interruption. This pessimistic assumption is consistent with the definition of $R_b(t)$ as the set of blocks whose presence in memory at time t is guaranteed.)

It becomes thus possible to compute the Resident Set of Blocks of a program running under a Global LRU policy with a given Ψ and to define critical and minimal restructuring algorithms aimed at programs to be run in Global LRU environments. Ferrari and Kobayashi found that the performances of these two algorithms--respectively named Critical PSI and Minimal PSI --were excellent but not appreciably better than those of CWS and MWS.

2.5.4. Extension to PFF Environments

As we said before, the PFF policy bases all its memory allocation decisions on the page fault frequencies of programs. Whenever the page fault frequency of a program exceeds a given critical level 1/T, all pages causing faults are brought into memory without replacing any other pages. On the other hand, once a page fault occurs after an interfault interval larger than T, all the program's pages that have not been referenced during this interval are returned to the secondary store.

Contrarily to what happens with other local memory policies, one cannot compute $R_b(t)$ by simulating what would happen if the replacement policy would apply directly to the program's blocks, rather than to its pages: One would then get a block fault frequency that would generally be much larger than the program's page fault frequency and the Resident Set of Blocks of the program would be grossly overestimated. Our approach will then be somewhat different. Since the PFF algorithm always keeps in memory all *pages* that have been referenced since the last *page fault*, all *blocks* that have been referenced since the last *block fault* must also reside in memory. Moreover, the PFF algorithm expels pages only when a page fault occurs after an interfault interval larger than T. Thus, all pages--or blocks--that have been referenced during the last T time units will always reside in memory. The Resident Set of Blocks of a program running under a PFF policy will thus contain all blocks that have been referenced at least once since the last block fault or during the time interval [t-T, t].

Using this definition, one can then design critical and minimal restructuring algorithms tailored to the PFF memory policy. The performances of these algorithms will be discussed in the next chapter.

2.5.5. The Choice of the Performance Indicator

Although they are very sound indicators of program performance in virtual memory environments, page fault frequency and mean memory occupancy are not the only criteria that can be chosen for constructing restructuring algorithms. One could envision ,for instance, algorithms aimed at improving some dynamic aspects of the program performance, like smoothing the peaks of its memory occupancy curve or distributing more evenly page faults during the total virtual execution time.

We experimented briefly with such algorithms, getting quite disappointing results, especially with the algorithm attempting to spread more evenly page faults. This can retroactively be explained by the fact that the program investigated--a WATFIV compiler--had the vast majority of its page faults occurring at locality transitions. Clusters of page faults were thus a "natural" result of the program organization and almost impossible to break.

The real problem with page fault frequency and mean memory occupancy is different: they share the common drawback of being only partial indicators of the global performance of a program in a paging environment. This basic flaw is enhanced by the well known fact that these two indicators vary in opposite directions when the program's performance is not too far from its optimum. Hence, attempts to minimize one indicator may worsen the performance of the other.

The solution adopted by Ferrari when using a critical algorithm has been to take the page fault frequency as the optimization criterion for the restructuring procedure, while checking afterwards for a possible increase in the mean memory occupancy. (Similarly, when running a minimal algorithm, one has to check afterwards for a possible increase in the page fault frequency.) In the majority of cases (see [Ferr76b]), this very simple procedure has been found to perform quite satisfactorily.

We feel, however, that this procedure does not constitute a complete solution to the problem and does not allow any control over what is happening during the restructuring process. A more satisfactory solution would be to choose, as optimization criterion for our restructuring algorithm, a performance indicator that would depend *simultaneously* on the page fault frequency and the mean memory occupancy of the program it characterizes.

Among the possible criteria, the most interesting one appeared to us to be [Pari76] the space-time product criterion. We will show how this criterion can be used to construct a new family of strategy-oriented restructuring algorithms—the so-called *Balanced Algorithms* and, later, how the same approach can be extended from paging to segmentation environments.

CHAPTER III

BALANCED ALGORITHMS FOR PAGING ENVIRONMENTS

3.1. THE MOTIVATIONS OF BALANCED ALGORITHMS

The Balanced Algorithms constitute a new family of program restructuring algorithms aimed at reducing the space-time product of the programs. Like their Critical and Minimal counterparts, they differ from each other in the replacement strategy for which they are tailored. Here too, the replacement strategy affects only the way the Resident Set of Block is defined within each algorithm. Unlike other strategy-oriented restructuring algorithms, Balanced Algorithms attempt to optimize a global index of program performance in a paging environment, namely the space-time product of the program. As we said before, this criterion was selected because it offers a natural way to combine the page fault frequency and the mean memory occupancy criteria into a single performance indicator, and has, in addition, a direct physical interpretation of its own.

Let now $S(u)$ denote the memory occupancy of a program at a given time u and $(0,t)$ be the program execution interval. Our optimization criterion will then be (see equation 1.5)

$$C = \int_0^t S(u)du + \sum_{j=1}^{r} S(t_j).T_w \qquad (3.1)$$

where r is the total number of page faults occurring during (0, t), t_j the time of the j-th page fault and T_w the average page wait time.

Note that this expression only depends on the instantaneous memory occupancy $S(x)$ of the program and on the page faults which occur during the virtual time interval considered.

3.2. DERIVATION OF AN ALGORITHM SCHEME

The easiest, and probably most natural way, to infer a program restructuring algorithm from a performance indicator taken as a cost function is to consider marginal costs. These marginal costs represent the contribution of undesirable events, like a page fault, to the performance indicator; conversely, they express also the quantity by which the performance indicator is decreased each time one of such events is avoided thanks to the restructuring process.

In our case, we have to consider two kinds of events susceptible of influencing our performance indicator, namely the occurrence of a page fault and the need for an additional page frame during a given time interval. From (3.1), we can deduce

- that the marginal cost of a page fault is $S(t_f).T_w$ where $S(t_f)$ is the number of memory page frames occupied by the program during the page wait interval, and
- that the marginal cost of one additional page frame during a time interval Δt is equal to Δt.

Given these marginal costs, the affinity between two blocks i and j may simply be defined as the sum of the marginal costs of all undesirable events that would not occur if blocks i and j were stored in the same page.

For instance, it is obvious that, if blocks i and j are simultaneously present in main memory during a time interval Δt, storing both blocks in the same page will tend to reduce the program's memory occupancy. Similarly, if block i is currently residing in memory while a reference to a previously inactive block causes a "block fault" condition, one way to avoid the page fault that could result from this reference to an inactive block is to store blocks i and j in the same page.

Hence the following general scheme, which applies to *all* program restructuring algorithms based on the space-time product criterion.

Let us denote by

$(b_1, b_2, \ldots, b_n)$ a block reference string corresponding to one run of the program to be restructured,

$S(t)$ the number of memory page frames allocated to the program after execution of the t-th reference,

T_m the mean inter-reference time interval, and

T_w the mean page-wait time.

Here again, $R_b(t)$ will represent the resident set of blocks, i.e. the set of blocks that will be present while the t-th reference is processed.

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 increment by $\alpha = S(t).T_w$ all $a_{ij}$'s such that $i \in R_b(t-1)$ and $j = b_t$;
        increment by $\beta = T_m$ all $a_{ij}$'s such that $i \in R_b(t)$ and $j \in R_b(t)$
    od.
(b) For all $i$ and $j<i$ do
        $a_{ij} := a_{ji} := a_{ij} + a_{ji}$
    od.
```

Let us remark that what we have defined is not a single program restructuring algorithm but rather a family of restructuring algorithms, the *Balanced Algorithms*, each of which will be tailored to a particular memory management strategy. For instance, there will be a *Balanced Working Set* algorithm aimed at programs to be run under a "pure" Working Set policy, a *Balanced Sampled Working Set* algorithm aimed at programs to be run under a Sampled Working Set policy, a *Balanced Page Fault Frequency* algorithm aimed at programs to be run under a Page Fault Frequency policy, and so forth.

3.2.1. Implementation Considerations

A few problems arise when one attempts to implement the above scheme. In general, it will not be possible to evaluate the quantities $S(t)$ during the matrix construction phase since these quantities depend on the final block mapping. The simplest solution will then be to replace these $S(t)$ by a constant value $\hat{S}$ that will be an estimate of the average number $\bar{S}$ of memory page frames occupied by the program. Note that a similar solution has been adopted by Prieve and Fabry in their optimal variable-space page replacement algorithm VMIN [Prie76].

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

Let m represent the number of blocks constituting the program being restructured. Then, the processing of each reference in the program's block reference string will require $O(m^2)$ operations, which lead to a total 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 affinity between the two blocks will be incremented will become $K.T_m$. The approximation remains acceptable as long as the sampling interval $T_s=K.T_m$ is relatively small compared to the average memory residence time of a page.

Keeping the same notations as before, the final version of our algorithm will then become:

```
(a) For all t from 1 to n do
        if $b_t \not\in R_b(t-1)$ then    (* block fault *)
            increment by $\alpha = \hat{S}.T_w$ all $a_{ij}$'s such that $i \in R_b(t-1)$ and $j=b_t$;
        fi;
        if $t$ mod $K$=0 then  (* sampling time *)
            increment by $\beta=K.T_m$ all $a_{ij}$'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
        $a_{ij} := a_{ji} := a_{ij} + a_{ji}$
    od.
```

Since α and β are now constant, the modified algorithm scheme then has the nice property of defining a restructuring matrix that is a linear combination of the matrices obtained by the corresponding Critical ($\alpha=1,\beta=0$) and Minimal Algorithms ($\alpha=0,\beta=1$).

3.3. ANALYTICAL STUDY OF BALANCED ALGORITHMS

3.3.1. Stochastic Models of Program Behavior

Several probabilistic models of program behavior in virtual environments have been developed during the last fifteen years. These models differ by their complexity, their accuracy and their tractability. All these models were aimed at describing the behavior of programs in terms of page references. Since we are here primarily interested in *block* references, we will apply these models to block rather than to page references as it is usually done.

Let us assume that a program consists of a given number m of blocks whose indices are denoted by 1, 2, ..., m. M will then be the set of block indices, i. e. M = {1, 2, ..., m}. Let ..., b_{t-1}, b_t, b_{t-1}, ... be an infinite block reference string generated by the program. b_t thus represents the index of the block referenced at discrete virtual time t.

The simplest model of program behavior one can envision is probably the so-called Independent Reference Model (IRM) [DenP66] [Aho71] [King71] [Coff73]. The IRM assumes that each block i of a program is referenced with a fixed probability p_i which does not depend on the previously referenced blocks. Obviously, one must have

$$\sum_{i=1}^{m} p_i = 1.$$

The main interest of the IRM is its simplicity and its tractability. The p_i parameters are relatively easy to evaluate and the model is well suited to analytical treatment. For instance, Ferrari [Ferr80] (see also [Lau79]) has shown that CWS and MWS were effectively optimal with respect to programs whose behavior can be described by an independent reference model and which have at most two blocks per page. The simplicity of the IRM has however one drawback: Since it assumes that the probability of referencing a given block does not depend on the past history of the program, the model does not conform to the locality principle and thus is not a realistic representation of program behavior [Coff73].

Another popular model of program behavior is the LRU Stack Model (LRUSM) originally proposed by Shemer et al. [Shem66] [Coff73]. This model assumes that the probability of referencing a block at a given time only depends on the number of distinct blocks that have been referenced since the block was referenced last. LRUSM thus maintains a stack ordering of blocks according to the times they were referenced last. Let $(s_1, s_2, \ldots, s_m)$ denote the content of this stack and s_i the i-th most recently referenced block. To each level i in the stack is associated a given probability d_i that the block at that level will be the next reference. If we have $d_1 > d_2 > \cdots > d_m$, blocks that have been recently referenced will tend to be also the most often referenced ones in the near future. The LRUSM thus conforms to the locality principle. On the other hand, it cannot model the behavior of a program during transitions between phases or keep track of the individual behaviors of blocks. This last limitation makes the LRUSM virtually useless in modeling program behavior for restructuring purposes.

A better solution is to use a first-order Markov chain whose states correspond to the indices of the last referenced blocks. The transition probability matrix of the chain can then be written as $P = (p_{ij})$, where $p_{ij} = Pr\,[b_t=j \mid b_{t-1}=i]$. We naturally have

$$\sum_{j=1}^{m} p_{ij} = 1, \qquad i=1,2,\ldots,m.$$

Assuming that the chain is homogeneous, irreducible and aperiodic, one can compute its limiting state probability vector $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_m)$. This vector λ is the eigenvector of matrix P and each λ_i represents the steady-state probability of referencing block i.

This first-order chain--often referred to as the Markov model--can simulate the behavior of programs having one or more disjoint phases with different degrees of locality [Fran74] [Cour76]. It has been used by Lau to model the behavior of programs being restructured by the CWS algorithm [Lau79]. Lau has indeed proved that CWS is optimal with regard to all programs whose behavior could be described by the Markov model and which have two blocks per page.

Despite its modeling power, the Markov model has however the major drawback of requiring m^2 parameters. This prohibits practical application of the model to programs consisting of numerous pages--or blocks. To circumvent this difficulty, Easton has introduced a special first-order Markov model that takes only into account consecutive references to the *same* block and requires only n+1 parameters. The transition probabilities of Easton's model are given by

$$p_{ii} = r + (1-r)\lambda_i$$

$$p_{ij} = (1-r)\lambda_j, \quad i \neq j,$$

where $0 \leq r < 1$ and $\lambda_i > 0$ for $i = 1, \ldots, m$. Since

$$\sum_{j=1}^{m} p_{ij} = 1, \quad i=1,...,m,$$

one must have necessarily

$$\sum_{i=1}^{m} \lambda_i = 1.$$

Note that r represents the probability that $b_t = b_{t-1}$. The λ_i are the eigenvalues of the transition probability matrix and are thus the steady-state probabilities of referencing a given block.

The relative simplicity of Easton's model does not prevent it from being sometimes surprisingly accurate. For instance, Kobayashi has found that Easton's model could be almost as good as the Markov model for estimating the average and the distribution of working set sizes [Koba79].

3.3.2. Analysis of the BWS Algorithm

Because of the approximations introduced in the computation of inter-block affinities, the BWS algorithm does not effectively attempt to minimize an exact expression of the space-time product but rather a linear combination of page fault frequency and mean memory occupancy. We will examine here the version of the algorithm for which the coefficients α and β are kept constant during the whole restructuring process and the sampling interval T_s is taken equal to T_m in order to avoid sampling errors.

As we pointed out in chapter II, 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. Because of this limitation, we have to restrict our analytical study of the Balanced Working Set Algorithm to the case of programs which contain *at most* two blocks per page.

Let us consider one of these programs. Assume that it consists of m blocks occupying a total of n pages. We must have necessarily $m \le 2n$. Note that $m = 2n$ would only hold if there were exactly two blocks per page. This will not be true in general since m can be odd or some blocks can be too large to share a page with another block.

For convenience purposes, 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, they 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 numbered i_1 and i_2. The infinite sequence b_1, ..., b_{t-1}, b_t, b_{t+1}, ... will represent an infinite block reference string produced by the program. In a Working Set environment, page fault frequency and 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 that the program's behavior can be described by a stochastic model that has 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 causes a fault at time t] =

$$Pr[i_1{=}b_t \,|\, i_1 \notin R_b(t-1) \text{ and } i_2 \notin R_b(t-1)]$$
$$+ \; Pr[i_2{=}b_t \,|\, i_2 \notin R_b(t-1) \text{ and } i_1 \notin R_b(t-1)]\}$$

and the page fault rate f is given by

$$f \;=\; \sum_{i=1}^{n} \{Pr[i_1{=}b_t \,|\, i_1 \notin R_b(t-1) \text{ and } i_2 \notin R_b(t-1)]$$
$$+ \; Pr[i_2{=}b_t \,|\, i_2 \notin R_b(t-1) \text{ and } i_1 \notin R_b(t-1)]\}$$

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

$$Pr[i_1 \in R_b(t) \text{ or } i_2 \in R_b(t)]$$

The mean memory occupancy of the program is then given by

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

LEMMA 3.1: The CWS algorithm minimizes the number of page faults 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

$$Pr[i{=}b_t \,|\, i \notin R_b(t-1) \text{ and } j \in R_b(t-1)]$$
$$+ \; Pr[j{=}b_t \,|\, j \notin R_b(t-1) \text{ and } i \in R_b(t-1)]$$

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

$$\max \sum_{i=1}^{n} c_{i_1,i_2} =$$
$$\max \sum_{i=1}^{n} \{Pr[i_1{=}b_t \,|\, i_1 \notin R_b(t-1) \text{ and } i_2 \notin R_b(t-1)]$$
$$+ \; Pr[i_2{=}b_t \,|\, i_2 \notin R_b(t-1) \text{ and } i_1 \notin R_b(t-1)]\}$$

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

Observing that

$$Pr[i{=}b_t \,|\, i \notin R_b(t-1) \text{ and } j \in R_b(t-1)] =$$
$$Pr[i{=}b_t \,|\, i \notin R_b(t-1)]$$
$$- \; Pr[i{=}b_t \,|\, i \notin R_b(t-1) \text{ and } j \notin R_b(t-1)]$$

we can thus rewrite our objective function as

$$\max \sum_{i=1}^{n} \{Pr[i_1{=}b_t \,|\, i_1 \notin R_b(t-1)]$$
$$+ \; Pr[i_2{=}b_t \,|\, i_2 \notin R_b(t-1)]$$

$$- Pr[i_1{=}b_t \mid i_1 \notin R_b(t-1) \text{ and } i_2 \notin R_b(t-1)]$$

$$- Pr[i_2{=}b_t \mid i_2 \notin R_b(t-1) \text{ and } i_1 \notin R_b(t-1)]\}$$

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

$$\min \sum_{i=1}^{n} \{Pr[i_1{=}b_t \mid i_1 \notin R_b(t-1) \text{ and } i_2 \notin R_b(t-1)]$$

$$+ Pr[i_2{=}b_t \mid i_2 \notin R_b(t-1) \text{ and } i_1 \notin R_b(t-1)]\}$$

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

■

LEMMA 3.2: 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, each element m_{ij} of the restructuring matrix is then proportional to

$$Pr[i \in R_b(t) \text{ and } j \in R_b(t)]$$

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

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

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

Observing that

$$Pr[i \in R_b(t) \text{ and } j \in R_b(t)] =$$

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

$$- Pr[i \in R_b(t) \text{ or } j \in R_b(t)]$$

we can thus rewrite our objective function as

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

$$- Pr[i \in R_b(t) \text{ or } j \in R_b(t)]\}$$

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

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

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

■

THEOREM 3.1: The BWS algorithm minimizes a linear combination of the number of page faults 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. In the version of BWS we analyze, each element a_{ij} of the restructuring matrix is then proportional to

$$\hat{S}.T_w.Pr[i=b_t \mid i \not\in R_b(t-1) \text{ and } j \in R_b(t-1)]$$
$$+ \hat{S}.T_w.Pr[j=b_t \mid j \not\in R_b(t-1) \text{ and } i \in R_b(t-1)]$$
$$+ T_m.Pr[i \in R_b(t) \text{ and } j \in R_b(t)]$$

If c_{ij} and m_{ij} represent the corresponding CWS and MWS restructuring matrices, we have then

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

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

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

Using the results of Lemmas 3.1 and 3.2, we can rewrite our objective as

$$\max \sum_{i=1}^{n} \{\hat{S}.T_w.Pr[i_1=b_t \mid i_1 \not\in R_b(t-1)]$$
$$+ \hat{S}.T_w.Pr[i_2=b_t \mid i_2 \not\in R_b(t-1)]$$
$$- \hat{S}.T_w.Pr[i_1=b_t \mid i_1 \not\in R_b(t-1) \text{ and } i_2 \not\in R_b(t-1)]$$
$$- \hat{S}.T_w.Pr[i_2=b_t \mid i_2 \not\in R_b(t-1) \text{ and } i_1 \not\in R_b(t-1)]$$
$$+ T_m.Pr[i_1 \in R_b(t)]$$
$$+ T_m.Pr[i_2 \in R_b(t)]$$
$$- T_m.Pr[i_1 \in R_b(t) \text{ or } j_2 \in R_b(t)]\}$$

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

$$\min \sum_{i=1}^{n} \{\hat{S}.T_w.Pr[i_1=b_t \mid i_1 \not\in R_b(t-1) \text{ and } i_2 \not\in R_b(t-1)]$$
$$+ \hat{S}.T_w.Pr[i_2=b_t \mid i_2 \not\in R_b(t-1) \text{ and } i_1 \not\in R_b(t-1)]$$
$$+ T_m.Pr[i \in R_b(t) \text{ or } j \in R_b(t)]\}$$

which is equivalent to

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

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

■

Consider now a two-dimensional representation of program performance $(\bar{S}, f)$, where $\bar{S}$ stands again for the program's mean memory occupancy and f for its page fault rate. Define as the *BWS curve* the set of points in the $(\bar{S},f)$ plane corresponding to the performance of a given program restructured by the BWS algorithm with the same window size but different $\hat{S}.T_w/T_m$ ratios. We have then the following corollary.

COROLLARY 3.1: If the behavior of a program can be described by a chain having a steady-state solution and if the program has at most two blocks per page, then the BWS curve of this program is the set of all points corresponding to block-to-page mappings for which it is impossible to improve either the page fault frequency or the mean memory occupancy without worsening the other criterion.

Proof:

Let $(\bar{S}_1, f_1)$ be a point on the BWS curve of a given program. By definition of the BWS curve we know that $\hat{S}.T_w.f_1 + T_m.\bar{S}$ is minimum for some values of $\hat{S}.T_w$ and T_m.

Assume now that it is possible to restructure the program so as to obtain a better page fault frequency $f' < f_1$ while keeping the same memory occupancy $\bar{S}$. We would thus have

$$\hat{S}.T_w.f' + T_m.\bar{S} < \hat{S}.T_w.f_1 + T_m.\bar{S}$$

which contradicts our hypotheses. Since the same conclusion would hold for $\bar{S}$, it is indeed impossible to improve either the page fault frequency or the mean memory occupancy of the program without worsening the other criterion.

■

This situation is known as a Paretian optimum with respect to the page fault frequency and the mean memory occupancy.

3.3.3. Extension to Other Memory Policies

As the reader has probably noticed, the proofs of the optimality of CWS, MWS and BWS did not take into account the composition of the Resident Set of Blocks $R_b(t)$ for the Working Set policy. These proofs would 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,

$$Pr[\text{page i in memory}] = Pr[i_1 \in R_b(t) \text{ or } i_2 \in R_b(t)].$$

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 the page fault timing triggers the replacement decisions.

For LRU, Global LRU and PFF policies, we have seen that one can construct a Resident Set 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 resident in memory may not contain any block belonging to the set. One has thus

$$Pr[\text{page } i \text{ in memory}] \geq Pr[i_1 \in R_b(t) \text{ or } i_2 \in R_b(t)].$$

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

$$f_{\max} = \sum_{i=1}^{n} \{Pr[i_1 = r_t \mid i_1 \notin R_b(t-1) \text{ and } i_2 \notin R_b(t-1)]$$

$$+ Pr[i_2 = r_t \mid i_2 \notin R_b(t-1) \text{ and } i_1 \notin R_b(t-1)]\}$$

Using the same proof as for Lemma 3.1, one could then prove that CLRU, CPSI and CPFF minimize an upper bound of the number of page faults of all programs running under the corresponding memory policy provided that the behavior of the program in that environment can be described by a Markov chain having a steady-state-solution and that the program has at most two blocks per page. (For the CPSI algorithm, one must add the supplementary condition that the Global LRU environment in which the program is to run can be modeled by Bard's Page Survival Index Model.) These results generalize a similar finding made by Lau [Lau79] 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 relative to Balanced Algorithms would be even less interesting.

3.4. EMPIRICAL STUDY OF BALANCED ALGORITHMS

A series of trace-driven simulations were conducted in order to evaluate the performance of Balanced Algorithms and to compare their performance with those of the corresponding Critical and Minimal restructuring algorithms.

The traces we used for our experiments were full traces (instruction and data references) of a WATFIV compiler, an FFT program and an APL interpreter, all collected on an IBM 360/91 at the Stanford Linear Accelerator Center. Block sizes were 1024 bytes for the WATFIV and APL traces, 512 bytes for the FFT trace. Trace lengths varied between one and three million references.

Besides the program, other factors considered in our study were

- the memory policy ("pure" Working Set, Sampled Working Set, Global LRU and Page Fault Frequency)
- the control parameter of the policy (window size for Working Set and Sampled Working Set, estimated page survival index for Global LRU and critical interfault interval T for PFF),
- the α and β coefficients used by each balanced algorithm,
- the number of blocks per page (2 or 4).

Preliminary experiments convinced us that we would need at least three or four different values of the control parameter to cover each memory policy. Besides, the "interesting" values of this control parameter--i.e. these for which the program will not thrash or execute without paging-- were also dependent on each individual trace. These considerations made a full factorial experiment unpractical. Rather than selecting an a priori incomplete factorial design, we decided in favor of a more empirical "exploratory" design where already performed experiments would influence the design of the next ones.

In order to limit the cost of our simulations, it was also necessary to use compressed versions of the traces. The reduction algorithm used to generate the compressed traces replaced each trace by a sequence of "reference sets", each containing the blocks being referenced at least once during a sampling interval of 1,173 references. Since the algorithm preserves the ordering of the first references to each block within each sampling interval, the quality of the reduced trace [Lau79] is somewhat superior to the quality of the reduced traces obtained by the "Snapshot Method" analyzed by Smith [Smi77].

All simulations of balanced algorithms were performed using fixed values for α and β for each run of the restructuring procedure. The algorithm's sampling interval T_s was always taken equal to the sampling interval of the compression algorithm, i. e., 1,173 references.

3.4.1. Empirical Study of the Balanced Working Set Algorithm.

Three distinct restructuring algorithms applicable to a working set environment were thus considered: the Critical Working Set (CWS) algorithm , the Minimal Working Set (MWS) algorithm, and the Balanced Working Set Algorithm (BWS). The assumed page sizes for our experiments were 1024, 2048 and 4096 bytes, the first value applying only to the FFT trace.

We have conducted our experiments with window sizes varying between 10 and 150 sampling intervals. Each of these intervals corresponded to 1,173 references from the original trace. For each window size, we have measured the performance of the non-restructured program (NR) and those of the program restructured by CWS, MWS and BWS. In order to evaluate experimentally the optimal combination of α and β corresponding to each window size, we have simulated repeated executions of the BWS algorithm for each window size using α/β ratios varying between zero and infinity.

The results of our simulations for WATFIV with a page size of 2048 bytes are summarized in Figure 3.1, where the mean memory occupancies are represented on the horizontal axis and the page fault frequencies on the vertical axis.

As can be seen, the set of points corresponding, for a given window size, to the performance of the program after restructuring is shaped like a segment of a hyperbola, the two extremes of which respectively correspond to the program restructured by CWS and MWS and where the intermediate points correspond to BWS. For all four window sizes, the point corresponding to the Minimal Algorithm is the uppermost one, while that one corresponding to the Critical Algorithm, is the rightmost one. One can also notice that the lower portions of the four curves arer almost horizontal. Thus, the marginally better performance of the CWS algorithm in terms of page fault frequency appears to go together with large increases of the memory occupancy and BWS appears to be clearly superior to CWS for all four window sizes. The result of the comparison between MWS and BWS is somewhat mitigated by the fact that MWS seems to operate fairly well for large window sizes.

We decided thus to measure the space-time products of all versions of the program, assuming a mean inter-reference time T_m equal to 1 ms and a page-wait time T_w of 20 ms. Looking at the space-time product curves corresponding to each window size (see Fig. 3.2), one can see that BWS perform significantly better than CWS and MWS for the four windows considered. One should however note one discrepancy between these results and the theory. When we derived our algorithm, we found the optimal α/β ratio to be equal to $\bar{S}.T_w/K.T_m$. With an average memory occupancy

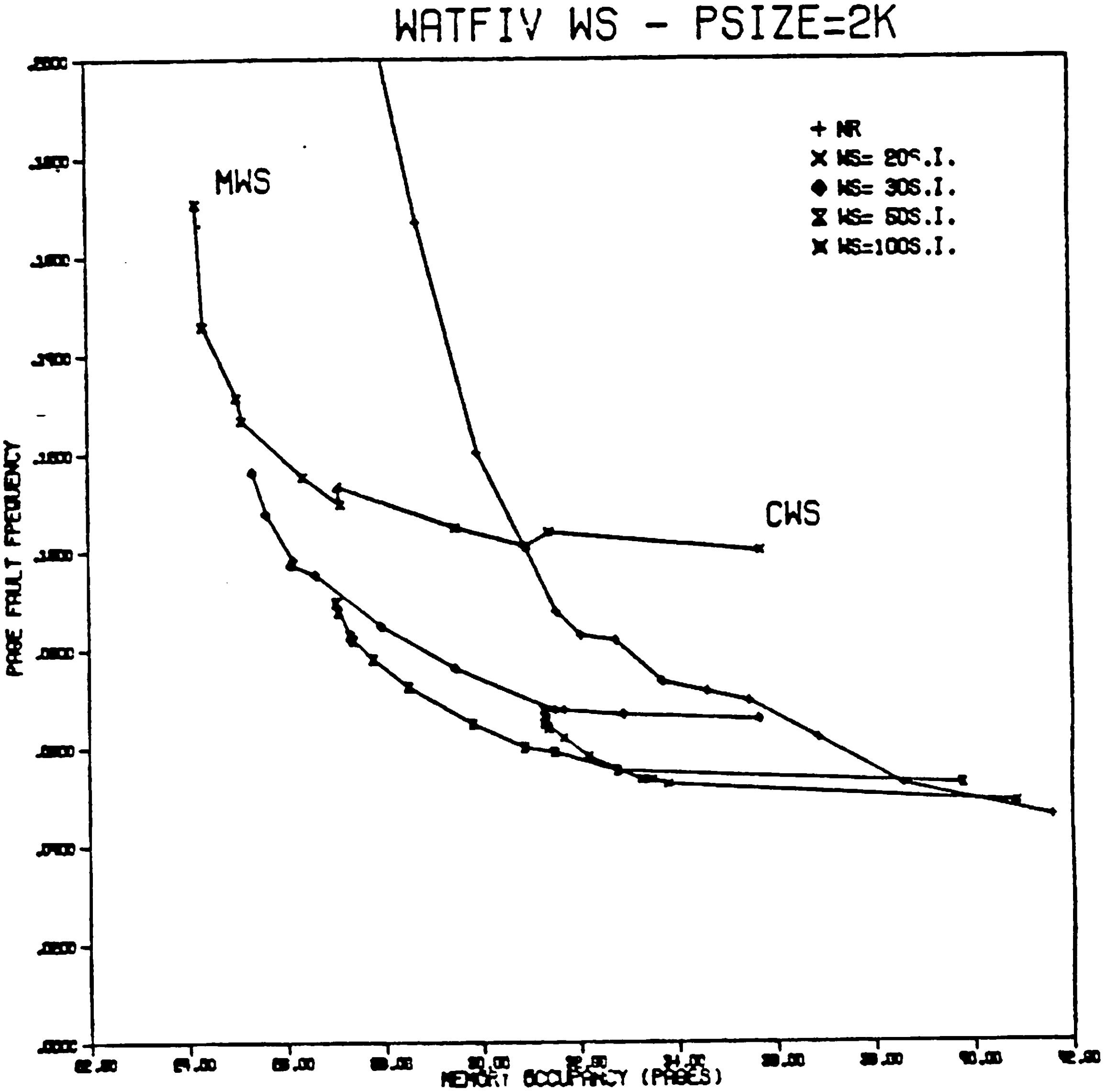

Figure 3.1

varying between 25 and 40 pages, a mean inter-reference time of 1ms, and a page wait time of 20ms, one would thus expect to find the minimum of each space-time curve to correspond to values of α/β varying between 500 and 800. In fact, these minima occurred at α/β ratios in the range between 75 and 250. A possible explanation for this phenomenon could be that the BWS algorithm, like all Balanced Algorithms, does not take into account the fact that minimizing the mean memory occupancy $\bar{S}$ also minimizes the individual contribution $S(t_f).T_w$ of each page fault to the space-time product. Thus the part of the algorithm attempting to minimize the program's memory occupancy has a stronger influence than expected on the space-time product and this will be reflected by smaller α/β ratios.

In order to check if the same conclusions would hold if there were more than two blocks per page, we repeated the same simulations assuming now a page size of 4 Kbytes, and thus four blocks per page. We observed basically the same program

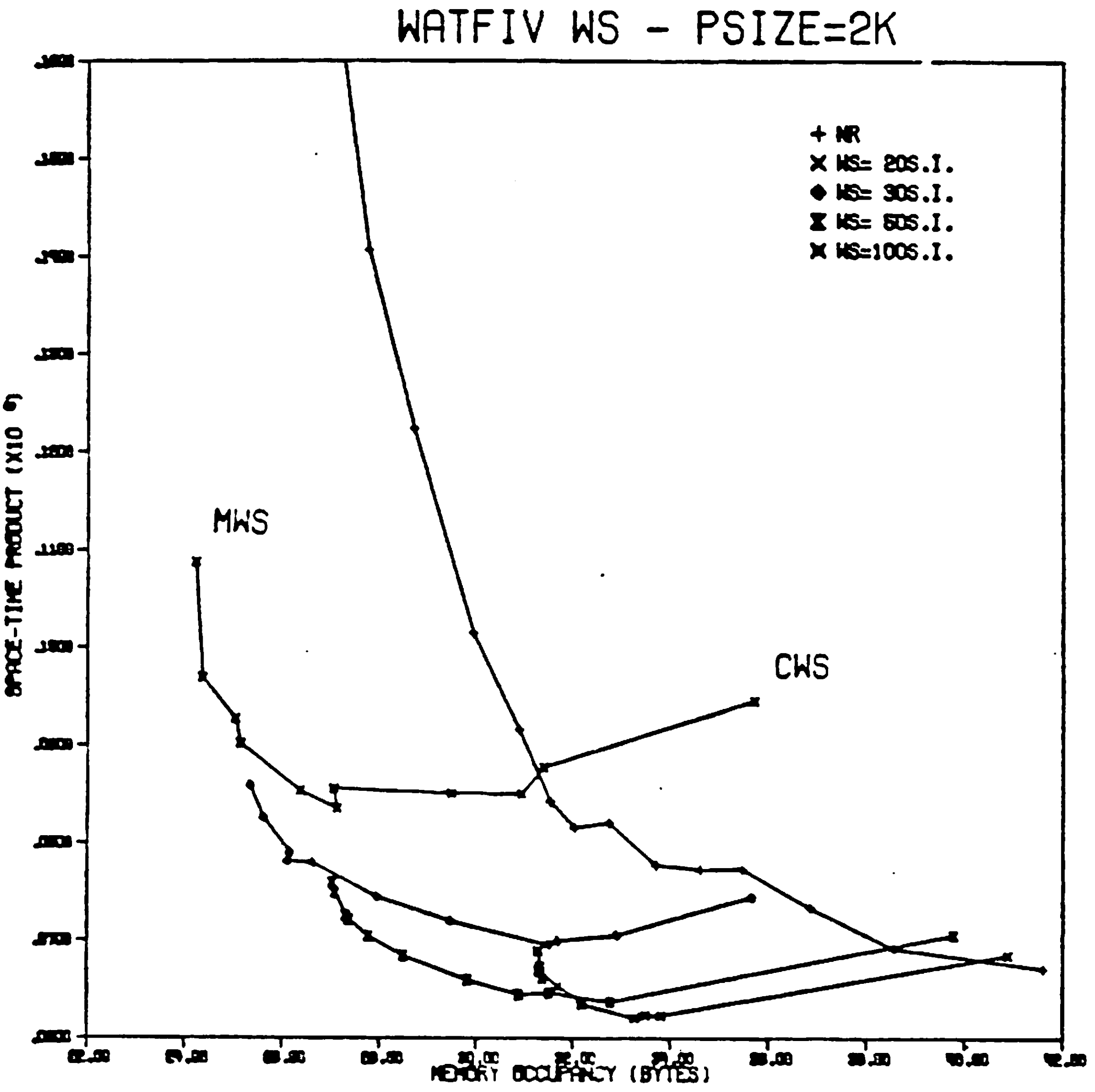

Figure 3.2

behavior, although the beneficial effects of the restructuring process were much stronger for BWS, CWS and MWS (see Fig. 3.3).

Results obtained for the APL trace and a page size of 2048 bytes are very similar to the ones obtained with the WATFIV trace (see Fig. 3.4 and 3.5). Here too, the observed minima of the space correspond to values of α/β varying between 75 and 250 pages, thus well below the theoretical optimum $\bar{S}.T_w / K.T_m$.

Results obtained with the FFT trace and a 1024 byte page size (see Fig. 3.6) indicate a very good performance of MWS and a rather disappointing performance of BWS, which is only outperforming MWS for two window sizes (20 and 100 sampling intervals). In order to see if this could be attributed to the smaller page size, we repeated our experiments with a page size of 2048 bytes (see Fig. 3.7). The results of these experiments were much flattened curves showing no clear superiority of any method, except for the smallest window size where BWS dominates MWS and CWS. Looking at

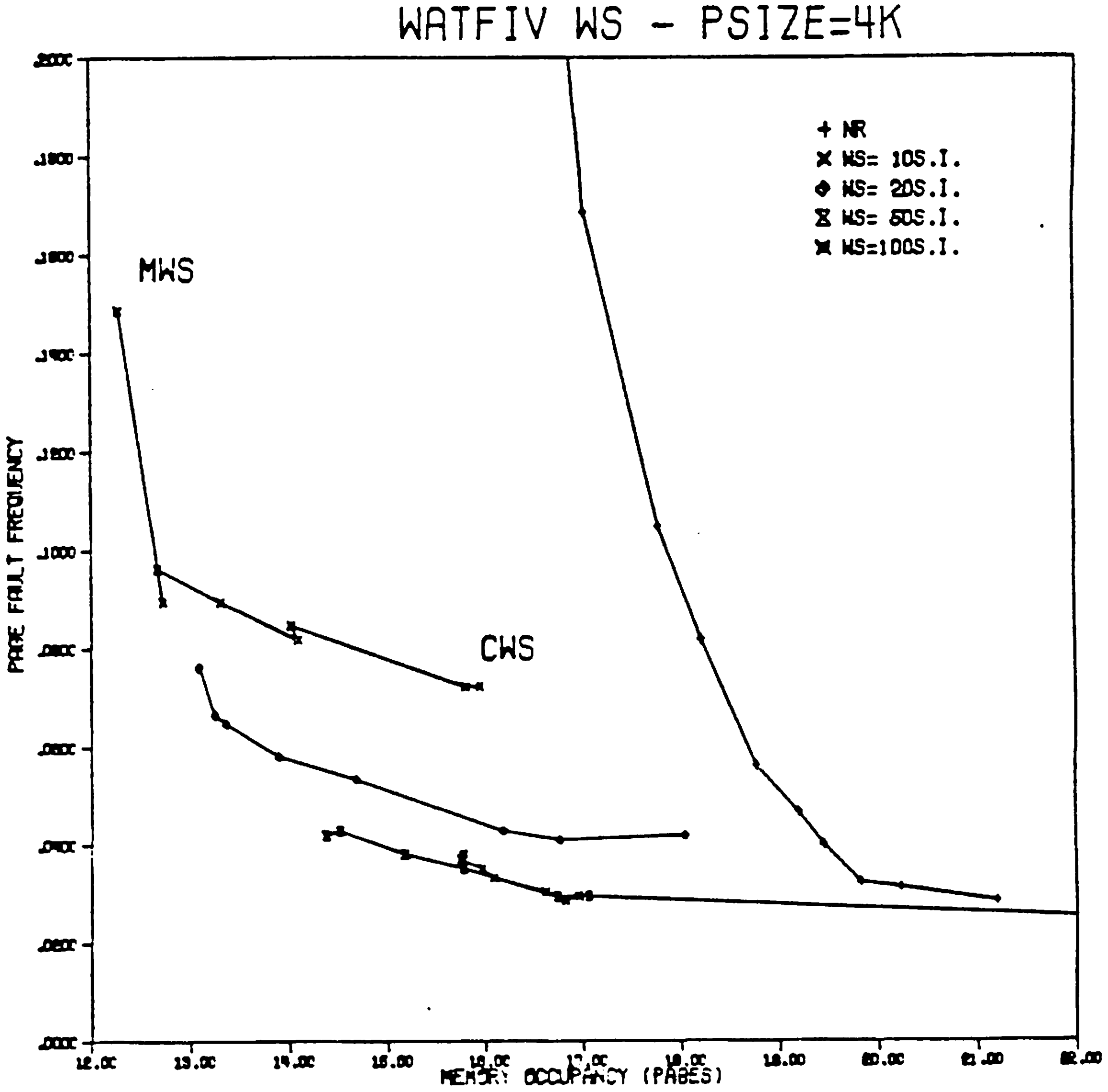

Figure 3.3

space-time product figures (see Fig. 3.8), BWS appears however to perform significantly better than CWS for all window sizes while its advantage over MWS at 20, 50 and 100 sampling intervals is only marginal.

To summarize our findings, BWS appears to clearly outperform both MWS and CWS for small window sizes--i.e. up to 50 sampling intervals for WATFIV and APL, up to 10 sampling intervals for FFT. It remains generally superior to CWS and MWS for larger window sizes although its superiority over MWS becomes much less significant.

3.4.2. Empirical Study of the Balanced Sampled Working Set Algorithm

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. For convenience, assume that the sampling interval I is a submultiple of the window size τ. In other

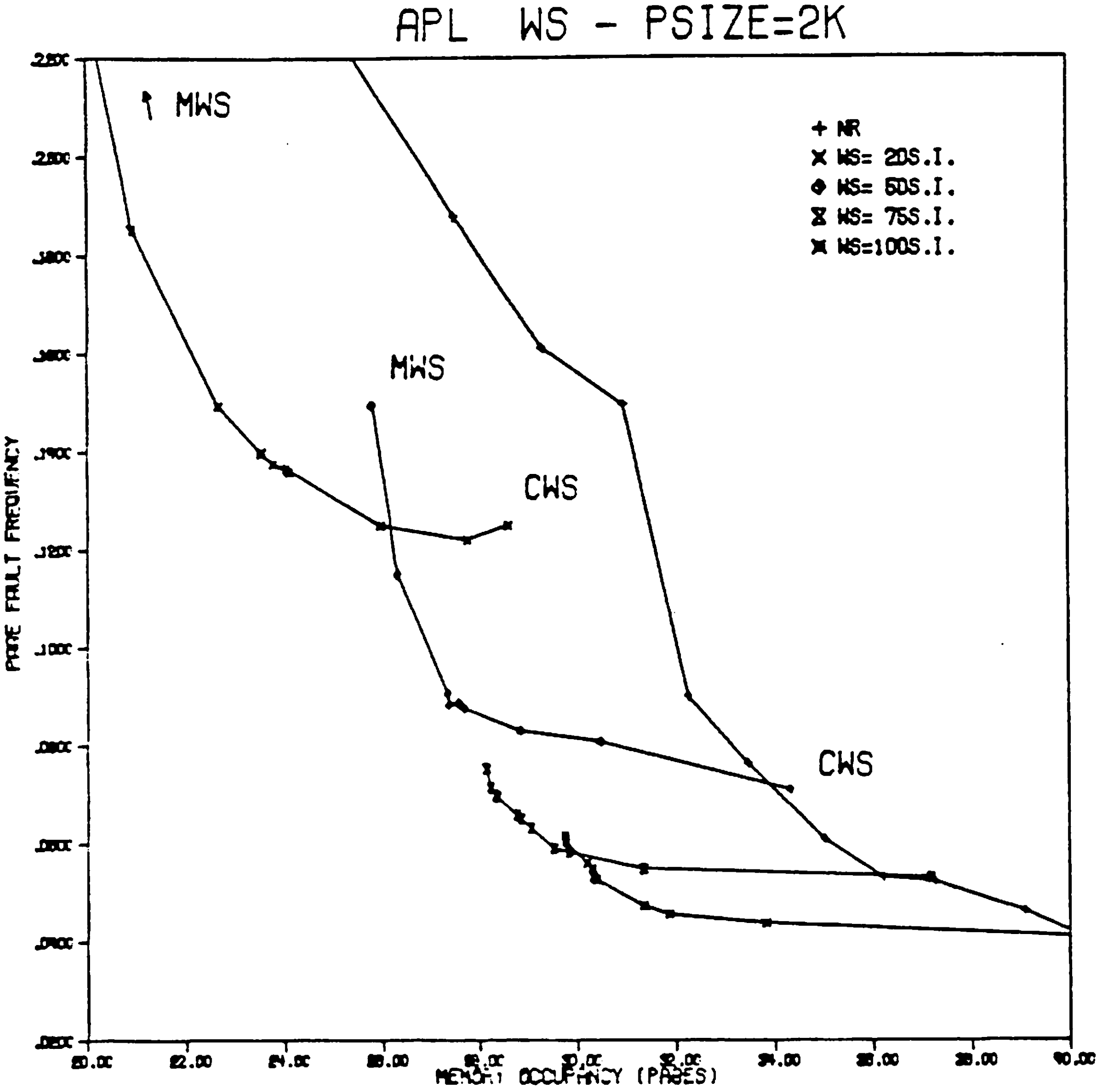

Figure 3.4

words, τ=k.I, 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 k.I=τ time units. As program execution resumes, the size of this window will increase linearly with time until it reaches τ + I time units at the end of the next sampling period. The SWS algorithm is thus essentially equivalent to a "pure Working Set algorithm with a window varying periodically between k.I and (k+1).I time units.

We decided to run our simulations of the BSWS algorithm assuming that the sampling interval of the replacement algorithm was always equal to its window size. This ensured that we would observe program behaviors as different as possible from those

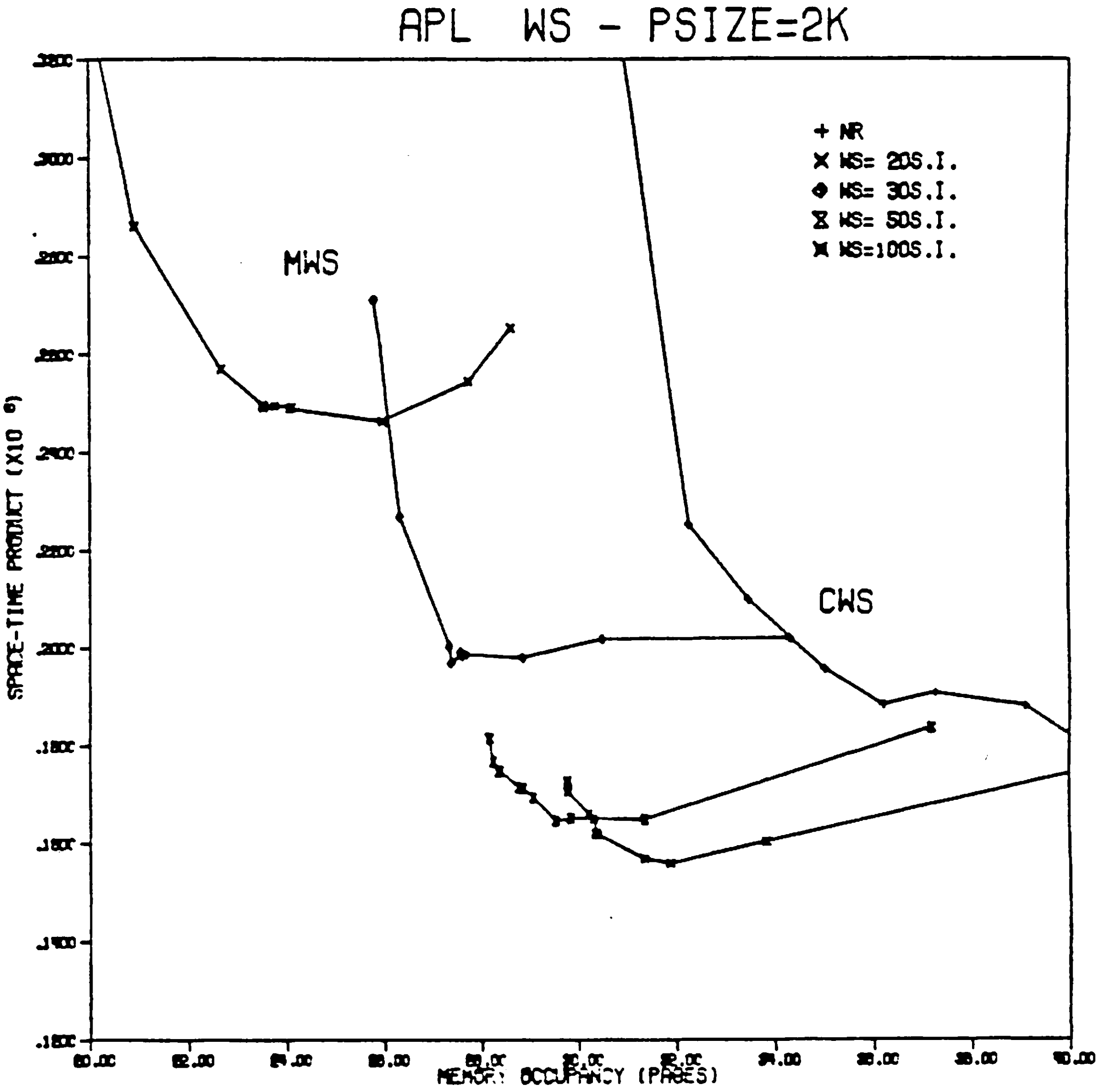

Figure 3.5

observed under a "pure" Working Set policy. Besides, the I=τ hypothesis makes the SWS algorithm much easier to implement: The algorithm only requires then one *use bit* per page that is automatically set to one each time the page is referenced. (As a last extremity, this use bit can be simulated by software [Baba79].) At the end of the sampling interval, all pages with a use bit set to one will have their use bit reset to zero, while other pages will be expelled from memory.

We ran our simulations of the BSWS algorithm using the WATFIV trace and assuming a page size of 2048 bytes (see Fig 3.9). As one can see, comparing Fig. 3.1 and 3.9, the switch from a pure Working Set to a Sampled Working Set Policy does not alter the basic behavior of the restructuring algorithm. A program running under a SWS replacement policy seems indeed to behave as if it were running under a pure Working Set policy with a window size oscillating between τ and τ + I time units.

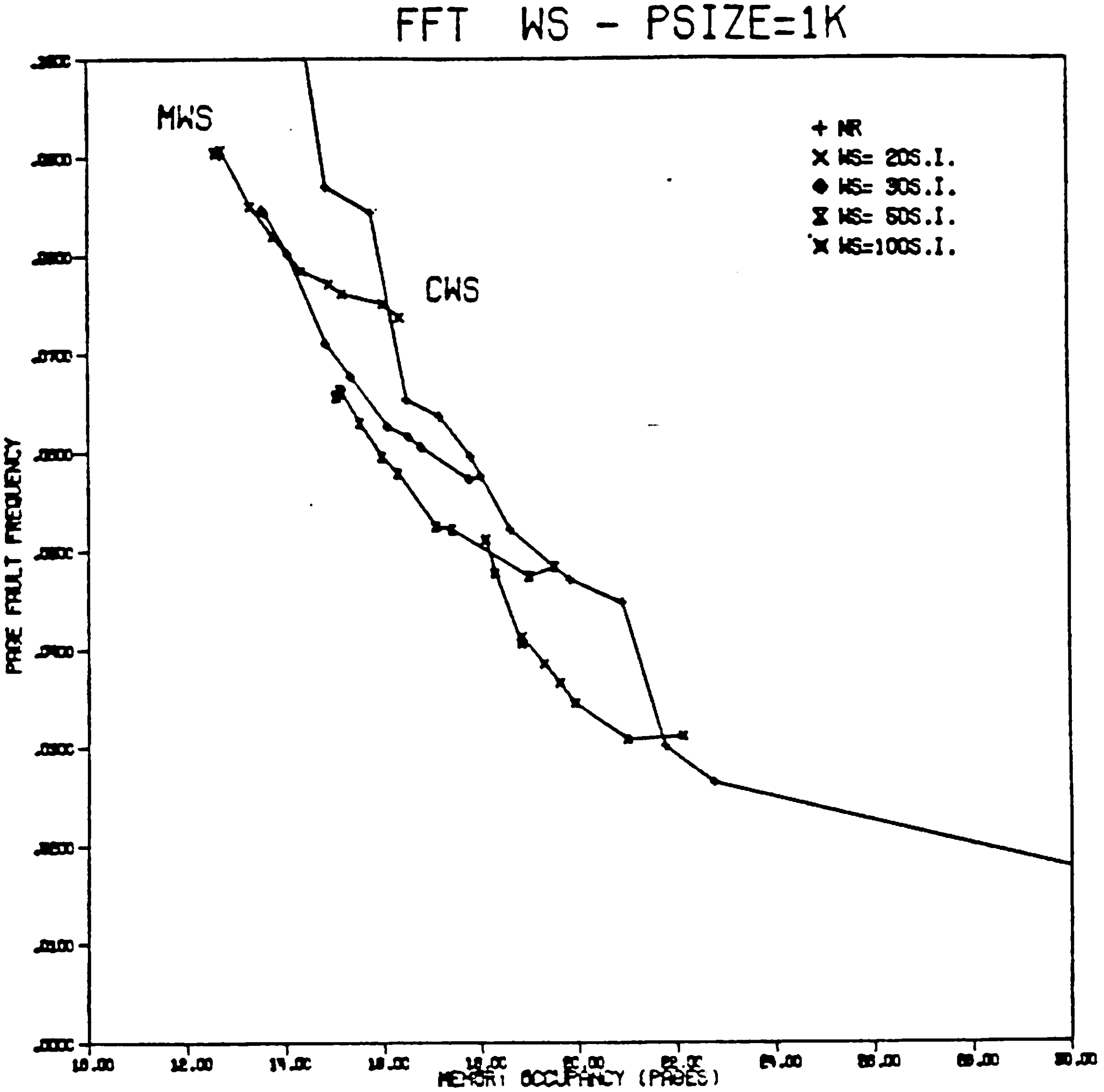

Figure 3.6

We thus felt to have collected enough evidence to assert that the BSWS algorithm is not behaving differently from its "pure" Working Set counterpart--BWS, and decided against any further simulations.

3.4.3. Empirical Study of the Balanced PSI Algorithm

Despite the clear superiority of local variable-space strategies, global policies, like Global LRU and its variants, still remain widely used because of their simplicity [Oli74]. As we said before, the paging behavior of a program running in a Global LRU environment can be approximately described by a single parameter Ψ, which represents the average number of interruptions that an unreferenced page can "survive" before being expelled from memory. This Page Survival Index (PSI) allows us in turn to define the Resident Set of Blocks of a program as the set of all blocks that have been referenced at least once during the time interval covering the last Ψ

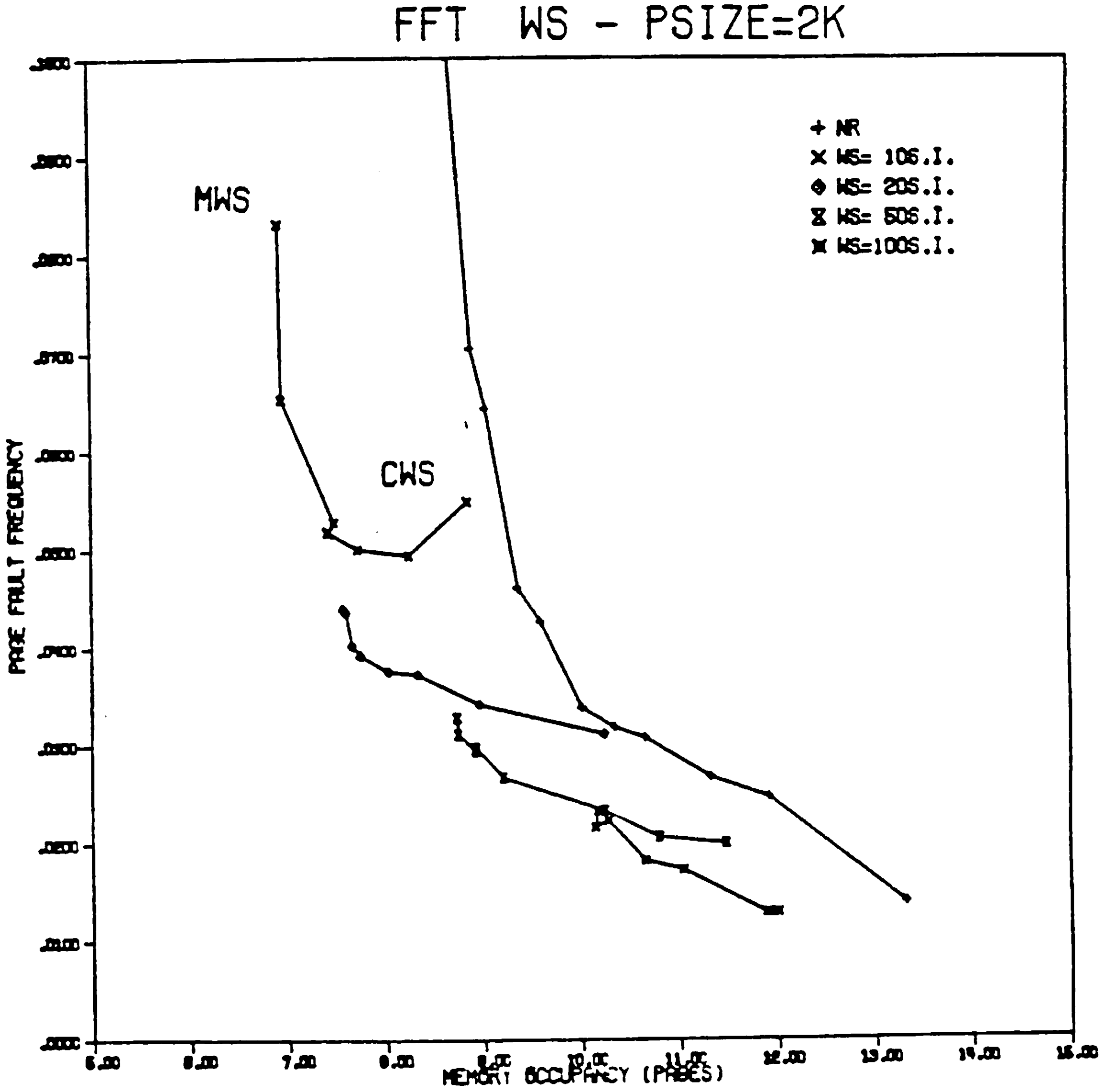

Figure 3.7

interruptions of program execution.

Exact simulation of the PSI model assumes that we possess all the necessary information for scheduling properly all interruptions. Since this was not the case, we had to make several assumptions. As our traces did not contain any information on the I/O activity of each program, we decided not to take into account program interruptions caused by I/O waits. Interruptions resulting from time quantum expiration were assumed to be uniformly distributed between 0 and 400 sampling intervals. Because of the compression process, the traces only contained the first references to each block within each sampling interval. We had thus to guess how further references to the same block would be distributed within each sampling interval. We decided that the best would be to assume that every block referenced during any given sampling interval would be continuously referenced during that sampling interval. In other words, a block would be assumed to reside in memory as long as less

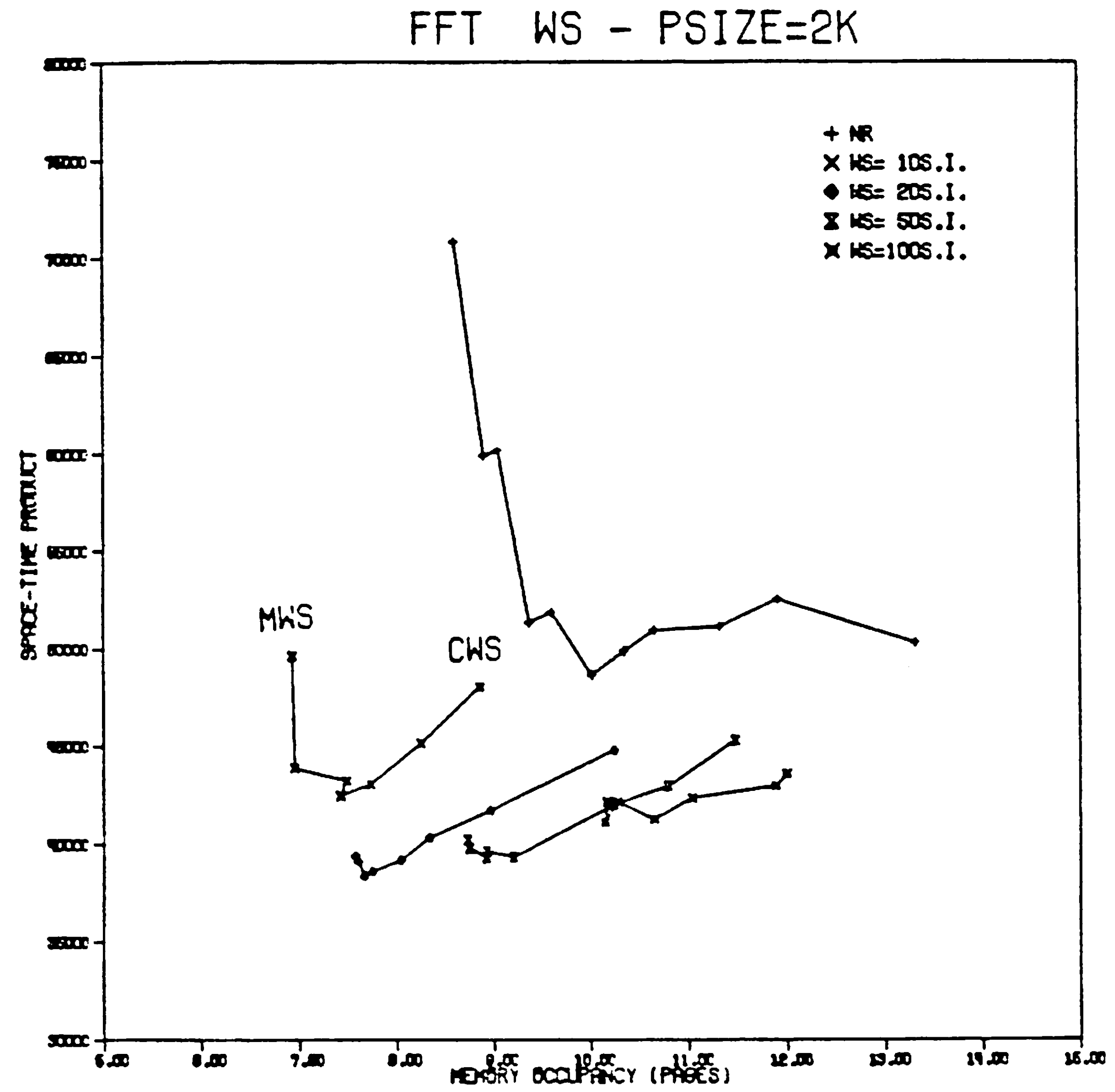

Figure 3.8

than Ψ blocks faults or time quantum expirations had occurred since the last sampling interval during which the program had been referenced.

We ran our simulations of BPSI, CPSI and MPSI algorithms for values of the Page Survival Index Ψ varying between 8 and 15 interruptions. This range was chosen in accordance to Bard's measurements of a CP-67 system, which had found 13 to be a reasonable value for Ψ [Bard73]. Results of these simulations for the WATFIV trace and a page size of 2048 bytes are summarized in Fig. 3.10. They show a mediocre performance of BPSI, CPSI and MPSI without any evidence that any of the three methods clearly dominates the other two. The same conclusions also hold for the APL trace which exhibits again a behavior similar to the one of WATFIV.

Results obtained with the FFT trace and a page size of 2 Kbytes were not very different although the three restructuring algorithms performed somewhat better (see Fig. 3.11). One should however point out the erratic performance of PSI for Ψ=8,

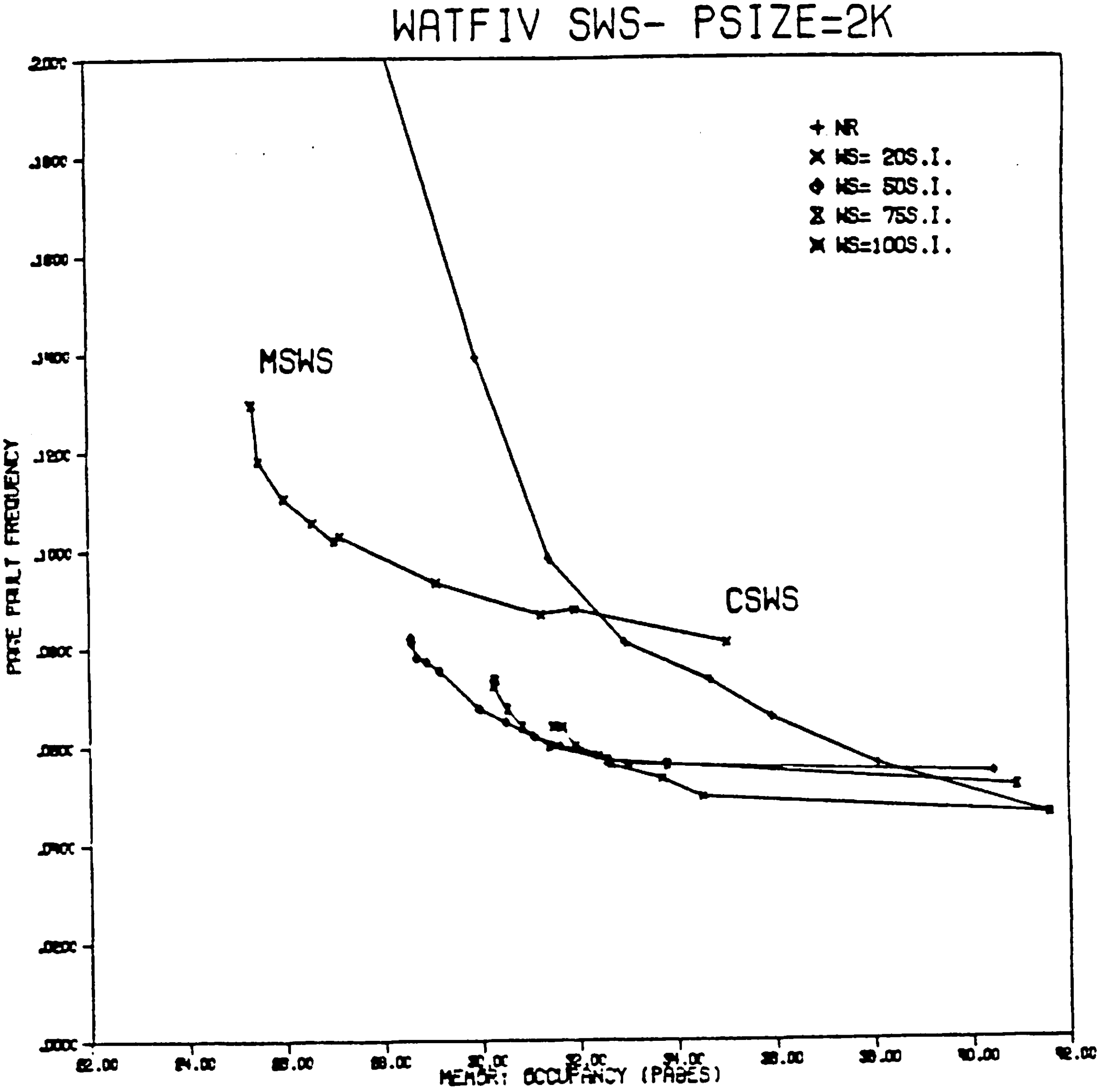

Figure 3.9

probably due to the clustering phase. These results were not totally unexpected since Ferrari and Kobayashi had already found no clear winner in their comparison of CPSI and MPSI applied to a Pascal compiler [Ferr77a].

3.4.4. Empirical Study of the Balanced PFF Algorithm

The last environment we considered in our simulation study was the page Fault Frequency policy [Chu072] [Opde74] [Chu076]. The PFF policy generated a great deal of interest at its introduction because it is easy to implement and it also claims to be less sensitive than Working Set to a detuning of the control parameter. Since then, this latter claim has been disputed by Graham [Grah76] and by Franklin and Gupta [Fran78]. Moreover, PFF has also been found to exhibit--like FIFO--anomalous behaviors where an increase in the memory occupancy of a program could also

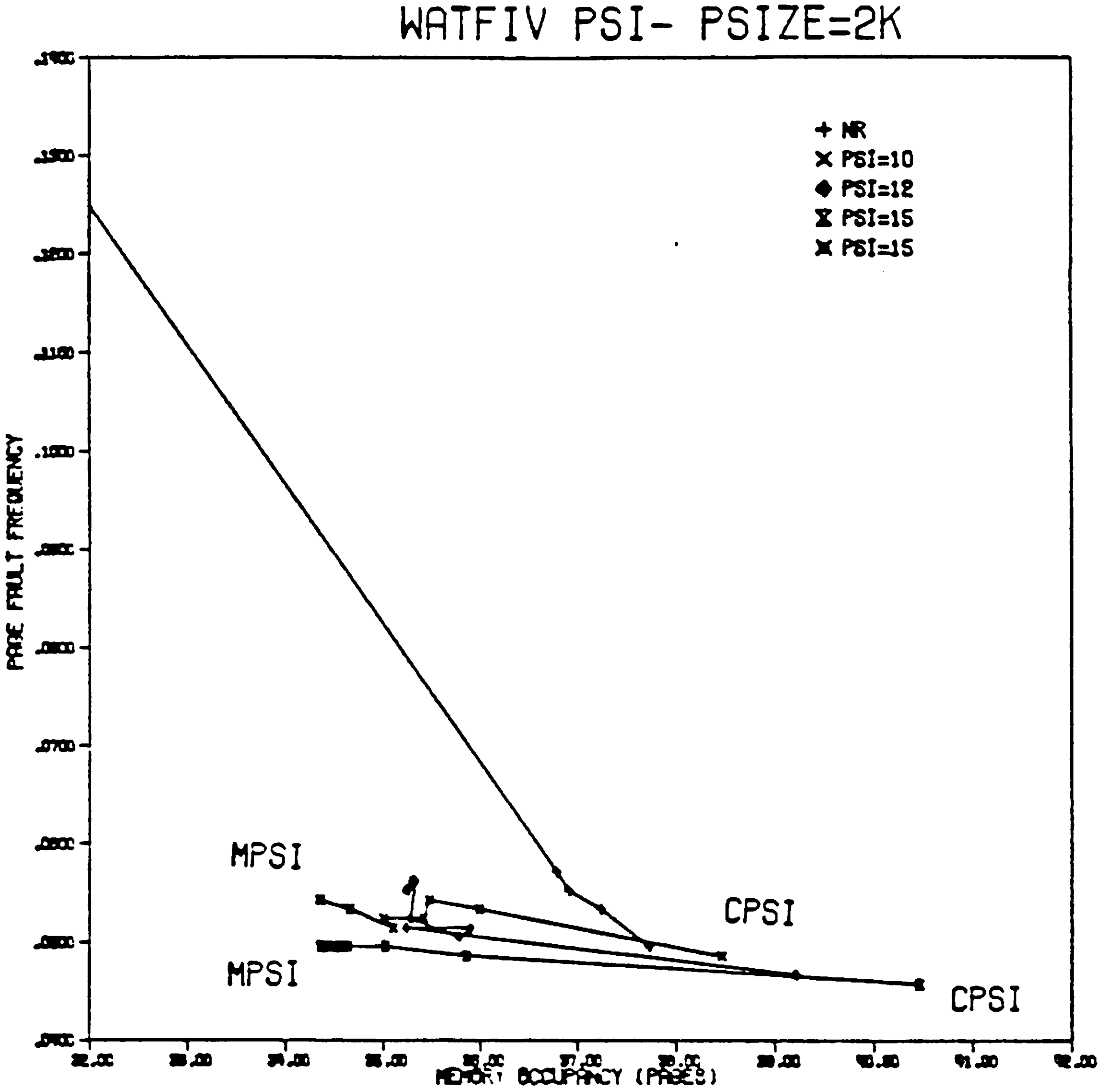

Figure 3.10

increase the page fault frequency of the program. Despite these facts, the simplicity of PFF makes it a strong candidate for implementation on all machines having hardware use bits.

In our simulations of the Critical PFF, Minimal PFF, and Balanced PFF Algorithms, the Resident Set of Blocks was always defined as the set of all blocks that had been referenced at least once during the last T time units or since the last *block* fault. We experimented with more sophisticated definitions of $R_b(t)$ and found that they did not result into any improvement in the performance of any of the three restructuring algorithms. As in our simulations of a PSI environment, we had to take into account the fact that our traces only contained the first reference to each block within each sampling interval. Here too, we assumed that every block referenced during any sampling interval would be continuously referenced during the whole sampling interval.

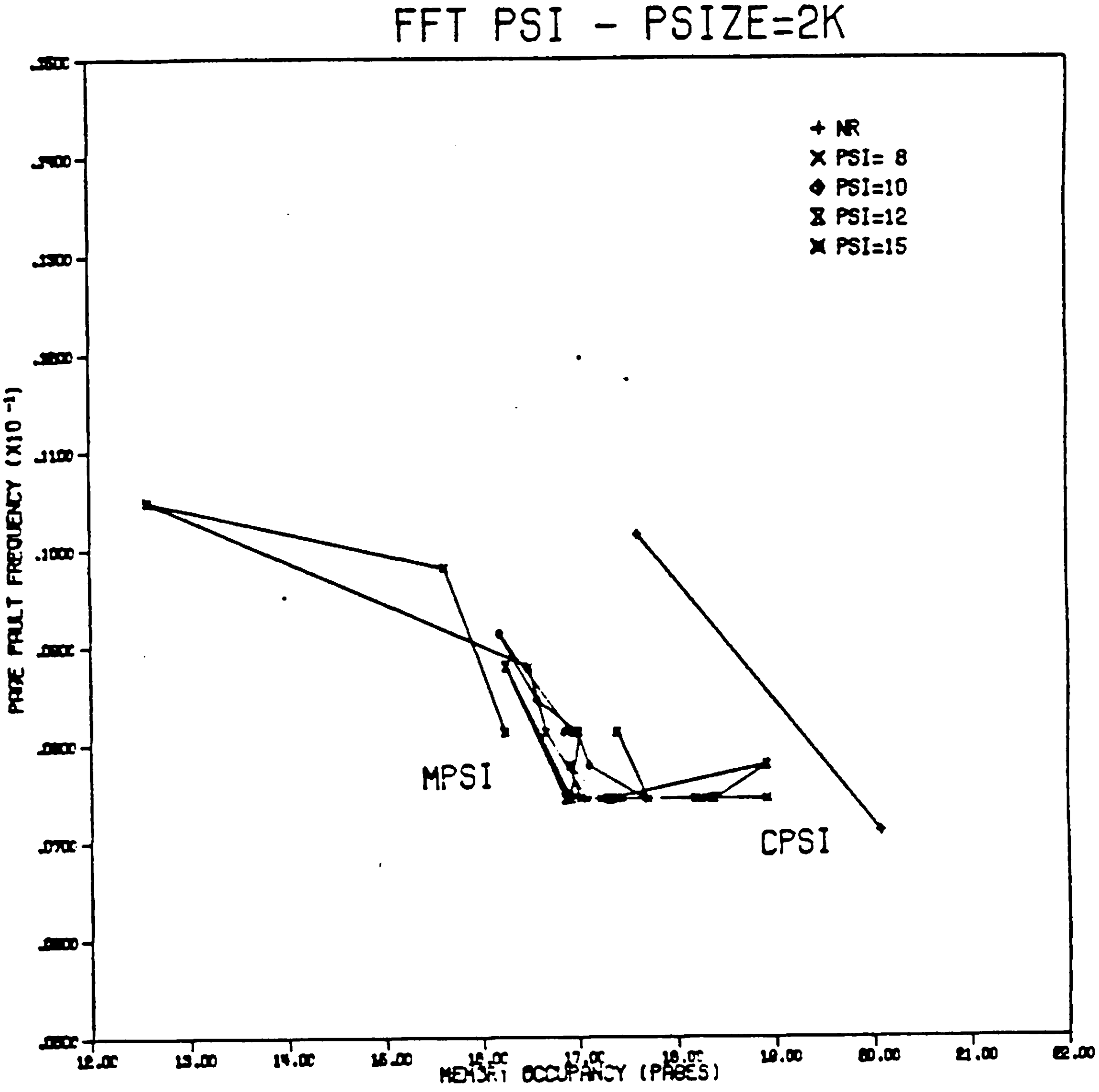

Figure 3.11

The results of our simulations of BPFF, CPFF and MPFF for the WATFIV trace and a page size of 2048 bytes are summarized in Fig. 3.12. The main surprise was the excellent behavior of CPFF for all values of T: our measurements indicated that CPFF is indeed better than BPFF for T equal to 10 sampling intervals and only marginally worse than BPFF for T equal to 20 and 50.

Results obtained with the FFT trace and a page size of 1 Kbyte were even more surprising (see Fig. 3.13): Two out of the three curves corresponding to the program restructured by BWS at various α/β ratios are almost identical to the demand curve for the non-restructured program. For the third, corresponding to T equal to 10 sampling intervals, the leftmost point of the curve, which minimizes simultaneously the page fault frequency and the memory occupancy, corresponds to the program restructured by CPFF while MPFF leads to a higher fault frequency and a much higher memory occupancy. The same simulations were repeated assuming a page size of 2

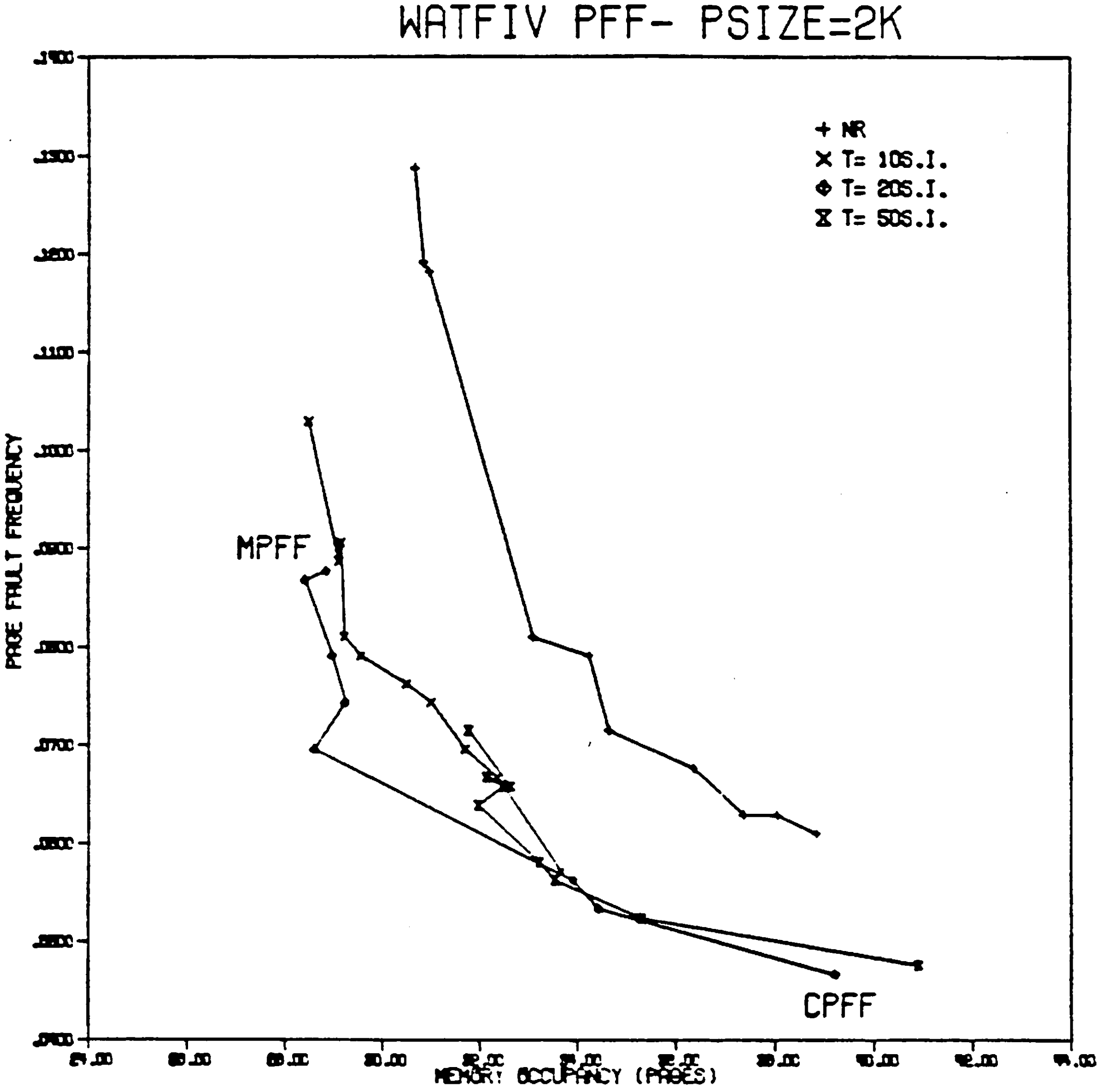

Figure 3.12

Kbytes, and thus four blocks per page. We observed then almost identical curves for BPFF at the three values of T and no clear winner among MPFF, CPFF and BPFF. (see Fig. 3.14).

This abnormal behavior of CPFF, MPFF and BPFF becomes less paradoxical if one considers the mechanism used by the PFF algorithm to reclaim memory space: Idle pages are only expelled at fault time and this occurs if and only if the last fault occurred more than T time units ago and the page considered for possible expulsion has not been referenced during that interval.

Suppose now that the new block-to-page mapping produced by the restructuring process results in the suppression of the k-th page fault previously occurring during the execution of the program. Depending on the timing of this page fault, several things can happen. First, this page fault could have been initiating the expulsion of several idle pages. The removal of the fault would then result in an increase of the

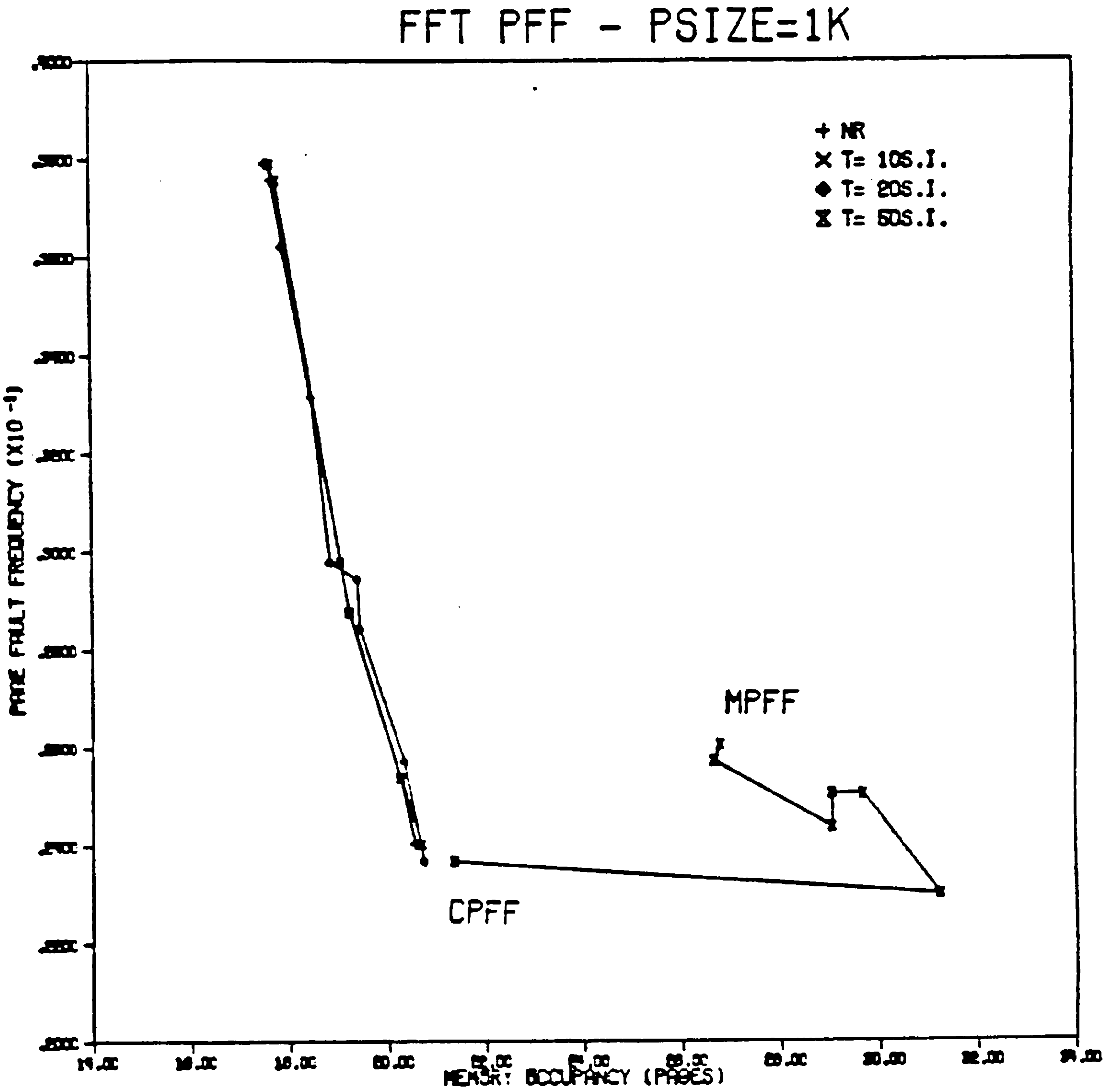

Figure 3.13

program's memory occupancy since the idle pages would stay in memory at least until the next page fault. In other cases, the disappearance of a page fault can cause an increase of the interfault time interval over the trigger level T. This would then result in the possible expulsion of several pages at the next page fault, causing then a reduction of the program's memory occupancy and the possible occurrence of additional page faults later in the future. Finally, it may also happen that the page fault eliminated by the restructuring process caused the resetting of the use bits of several pages that would not be referenced again until the next page fault. Should this page fault occur more than T time units later, the removal of the preceding page fault would have actually prolonged by at least T time units the survival of all these pages in memory.

In conclusion, it appears than the strong coupling existing in the PFF algorithm between the timing of page faults and the memory allocation decisions makes any

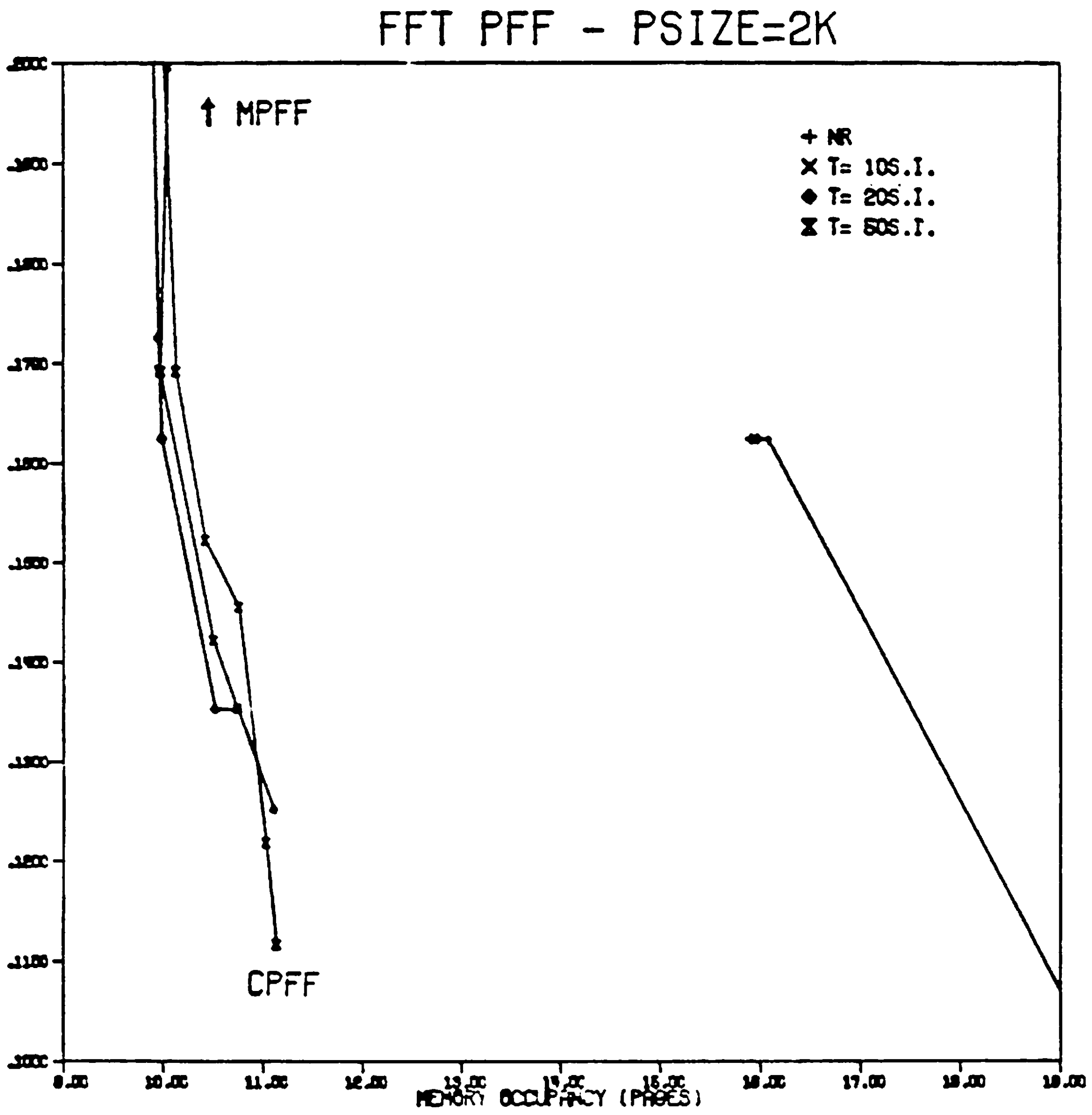

Figure 3.14

consistent prediction of the performance of a restructuring algorithm very difficult. The restructuring process must then be performed on a trial and error basis and the search for a near-optimal solution may require a lot of tuning. It should be pointed out, however, that these problems appear to be more linked to the nature of the PFF algorithm than to the restructuring process itself.

3.4.5. The Tuning of Balanced Algorithms.

Unlike their critical and minimal counterparts, Balanced Algorithms have one control parameter, namely their α/β ratio. This gives the algorithm much more flexibility but requires also some tuning. As we pointed already out for the BWS algorithm, the values of the α/β ratio that actually correspond to a minimum space-time product for a given value of the policy control parameter are not those predicted by the theory. Moreover, they appear to depend mostly on the relative performances of the corresponding critical and minimal algorithm.

In general, it will then be necessary to try several α/β ratios before selecting the proper one. This procedure is much less cumbersome than it appears at first glance because

- the most expensive part of a dynamic restructuring algorithm is the gathering of the block reference trace;
- any efficient implementation of a balanced algorithm will construct simultaneously the corresponding critical and minimal restructuring matrices and use these two matrices to construct each balanced matrix;
- space-time product curves obtained by varying the α/β ratio for a given value of the control parameter are rather flat near the optimum, making an exact determination of the optimum ratio unnecessary;
- the optimum ratio observed for each program and each policy seems to be relatively insensitive to variations in the policy's control parameter.

3.5. CONCLUDING REMARKS.

We have presented here a new family of program restructuring algorithms aimed at reducing the space-time product of programs being executed under various memory management policies. The overall performance of these algorithms has been found to be significantly better than those of the two best existing restructuring algorithms. Since the new algorithms are not more difficult to implement than Critical and Minimal algorithms, they should be the first technique to be considered for improving the behavior of all programs having a sufficient frequency of use to justify the cost of the procedure.

CHAPTER IV

BALANCED ALGORITHMS FOR SEGMENTATION ENVIRONMENTS

4.1. INTRODUCTION

As we have seen in the two last chapters, program restructuring can substantially improve the behavior of programs in paged virtual memory systems. On the other hand, very little effort has been devoted to the extension of this approach to segmented virtual memory systems and the results obtained so far have been rather disappointing [Chen74].

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 segmentation 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 *pages*. 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 restructuring matrix expressing the costs of keeping each pair of blocks i and j within 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--or affinities-- 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. 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.

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 into main memory and keeping it there. There will thus be cases where merging two blocks will actually decrease the program's performance. As a first consequence, affinities cannot be considered as being essentially positive any more. 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 one 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 segmentation 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 [Chen74]. 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 segmented environments would be to base the definition of interblock costs on some *global* index of the program's performance. A well known example of such performance index is the *Space-Time Product* criterion proposed by Belady and Kuehner [Bela69]. We have already shown that this criterion could be used for constructing various "Balanced" restructuring algorithms tailored to different paging environments. We want to show here how the same approach can be extended to segmentation environments and how efficient strategy-oriented restructuring algorithms can be derived from the space-time product criterion and tailored to various segment replacement policies.

4.2. DERIVATION OF AN ALGORITHM SCHEME

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}'s 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 referencing behavior during the program's execution. This will enable us to predict under what 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 segmentation environments lies in the fact that, in a segmentation 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

$$T_w = T_l + T_t . s_i$$

where T_l 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 segmentation environment during a *virtual* time interval $(0,t)$ is given by

$$C = \int_0^t S(u)du + \sum_{j=1}^{r} S(t_j).(T_l + T_t . s_{x_j}) \qquad 4.1$$

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 recently referenced block, the segment would have probably 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

$$\alpha=S(t)\ .\ (T_l + T_t.s_j)$$

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 the other blocks in 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

$$\beta=s_i.\Delta t$$

space-time units.

Similarly, each time the segment will be brought into 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

$$\gamma=S(t_f).T_t.s_i$$

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.

4.2.1. 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* [Ferr74c] [Ferr76b]. 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 $R_b(t-1)$.

Like in paging environments, evaluating the resident set of blocks of a program at time t is a more or less difficult task depending on the system's memory policy. To each memory policy, one can associate a specific definition of $R_b(t)$ and thus 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 Window Working Set Algorithm (BSTWS), a Balanced Segment Fault Frequency Algorithm (BSFF), and so forth.

4.2.2. Formal Definition of Balanced Algorithms.

Let us denote by

$(b_1, b_2, \ldots, b_n)$ a block reference string collected during one run of the program to be restructured,

s_i the size of block i,

S(t) the memory space occupied by the program while processing the t-th reference (this size obviously depend on the block-to-segment mapping),

$R_b(t)$ the Resident Set of Blocks at time t, i. e. while processing the t-th reference (we assume $R_b(1)=\{b_1\}$),

T_m the mean inter-reference time,

T_l 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 $\alpha=S(t).(T_l + T_t.s_{b_t})$ 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 into 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 corresponding to the pairs of blocks one of which 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 Time-Window Working Set policy or the Balanced Segment Fault Frequency for the Segment Fault Frequency policy, one has only to specify the proper expressions for $R_b(t)$ and S(t).

4.2.3. 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 $\hat{S}$ that will be some estimate of the program's 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 [Prie76].

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 requiring that, at each reference, all the elements a_{ij}'s of the restructuring matrix corresponding to a block $i \notin R_b(t)$ and to 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 once every K memory references. In this case, the running time of the algorithm becomes $O(n.m^2/K)$ 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 $\alpha = \hat{S}.T_l$ 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.
```

4.3. ANALYTICAL STUDY OF THE BTWWS ALGORITHM

As we did before for pages, we will assume that segments can contain a maximum of two blocks. In order to simplify our proof, we will add to the m blocks of sizes $s_1, s_2, \ldots, s_m$ constituting the program m fictitious blocks of sizes $s_{m+1} = \ldots = s_{2m} = 0$ that will *never* be referenced. Since these blocks will never cause a segment fault or occupy any memory space, their presence will not alter the performance of the program.

Because of these dummy blocks, we may assume that our program will consist of exactly m segments and that each segment i will contain two blocks i_1 and i_2. The infinite sequence $b_1, \ldots, b_{t-1}, b_t, b_{t+1}, \ldots$ will represent an infinite block reference string produced by the program. The segment fault rate in a TWWS environment can be written in terms of block reference probabilities and of the probability that a given block is in the Resident Set of Blocks $R_b(t-1)$. Assuming that we use a stochastic model that has a steady-state solution, the segment fault rate f is given by

$$f = \sum_{i=1}^{m} [Pr(i_1 = b_t \mid i_1 \notin R_b(t-1) \text{ and } i_2 \notin R_b(t-1))$$

$$+ Pr(i_2 = b_t \mid i_2 \notin R_b(t-1) \text{ and } i_1 \notin R_b(t-1))]$$

Similarly, the mean memory occupancy of the program is given by

$$\bar{S} = \sum_{i=1}^{m} (s_{i_1} + s_{i_2}) \,.\, Pr(i_1 \in R_b(t) \text{ or } i_2 \in R_b(t))$$

THEOREM 4.1: The BTWWS algorithm minimizes a linear combination of the number of segment faults and of the mean memory occupancy of all programs whose behavior can be described by a Markov chain having a steady-state solution and which have at most two blocks per segment.

Proof:

In the version of BTWWS we analyze, all elements a_{ij} of the restructuring matrix are proportional to

$$\hat{S}.T_l.Pr(i = b_t \mid i \notin R_b(t-1) \text{ and } j \in R_b(t-1))$$

$$+ \hat{S}.T_l.Pr(j = b_t \mid j \notin R_b(t-1) \text{ and } i \in R_b(t-1))$$

$$- s_i.T_m Pr(i \notin R_b(t) \text{ and } j \in R_b(t))$$

$$- s_j.T_m Pr(j \in R_b(t) \text{ and } i \notin R_b(t))$$

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

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

$$\max \sum_{i=1}^{m} \{\hat{S}.T_l.Pr(i_1 = b_t \mid i_1 \notin R_b(t-1) \text{ and } i_2 \in R_b(t-1))$$

$$+ \hat{S}.T_l.Pr(i_2 = b_t \mid i_2 \notin R_b(t-1) \text{ and } i_1 \in R_b(t-1))$$

$$- s_{i_1}.T_m.Pr(i_1 \notin R_b(t) \text{ and } i_2 \in R_b(t))$$

$$- s_{i_2}.T_m.Pr(i_1 \in R_b(t) \text{ and } i_2 \notin R_b(t))\}$$

Observing that

$$
\begin{aligned}
Pr(i=b_t \mid i \notin R_b(t-1) \text{ and } j \in R_b(t-1)) = {} & \\
& Pr(i=b_t \mid i \notin R_b(t-1)) \\
& - Pr(i=b_t \mid i \notin R_b(t-1) \text{ and } j \notin R_b(t-1))
\end{aligned}
$$

and

$$
\begin{aligned}
Pr(i \notin R_b(t) \text{ and } j \in R_b(t)) = {} & \\
& s_i . T_m . Pr(i \in R_b(t) \text{ or } j \in R_b(t)) \\
& - s_i . T_m . Pr(j \in R_b(t))
\end{aligned}
$$

we can rewrite our objective function as

$$
\begin{aligned}
\max \sum_{i=1}^{m} \{ & \hat{S}.T_l.Pr(i_1=b_t \mid i_1 \notin R_b(t-1)) \\
& + \hat{S}.T_l.Pr(i_2=b_t \mid i_2 \notin R_b(t-1)) \\
& + s_{i_1}.T_m.Pr(i_2 \in R_b(t)) \\
& + s_{i_2}.T_m.Pr(i_1 \in R_b(t)) \\
& - \hat{S}.T_l.Pr(i_1=b_t \mid i_1 \notin R_b(t-1) \text{ and } i_2 \notin R_b(t-1)) \\
& - \hat{S}.T_l.Pr(i_2=b_t \mid i_2 \notin R_b(t-1) \text{ and } i_1 \notin R_b(t-1)) \\
& - s_{i_1}.T_m. Pr(i_1 \in R_b(t) \text{ or } i_2 \in R_b(t)) \\
& - s_{i_2}.T_m. Pr(i_1 \in R_b(t) \text{ or } i_2 \in R_b(t)) \}
\end{aligned}
$$

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

$$
\begin{aligned}
\min \sum_{i=1}^{m} \{ & \hat{S}.T_l.Pr(i_1=b_t \mid i_1 \notin R_b(t-1) \text{ and } i_2 \notin R_b(t-1)) \\
& + \hat{S}.T_l.Pr(i_2=b_t \mid i_2 \notin R_b(t-1) \text{ and } i_1 \notin R_b(t-1)) \\
& + s_{i_1}.T_m.Pr(i_1 \in R_b(t) \text{ or } i_2 \in R_b(t)) \\
& + s_{i_2}.T_m.Pr(i_1 \in R_b(t) \text{ or } i_2 \in R_b(t)) \}
\end{aligned}
$$

which is equivalent to

$$\min(\hat{S}.T_l.f \ + \ T_m.\bar{S})$$

or

$$\min(K_1.f \ + \ K_2.\bar{S})$$

where K_1 and K_2 are constants.

4.4. 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 [Ferr76a].

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. The 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. The 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 or 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 [Lau 79] and is essentially a variant of Smith's "Snapshot Method" [Smit77]. 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 *on the average* to 0.28494 ms of execution time for the non-instrumented version of the program.

4.4.1. Balanced Time-Window Working Set Algorithm.

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 $\hat{S}.T_l$ in our experiments but rather to repeat each simulation for selected fault costs varying between 5000 and 10^7 bytes * sampling intervals, i. e. between 1.425 and 28,494 bytes * seconds.

For each window size and for each segment fault cost selected, we simulated the application of a BTWWS 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 [Ferr74c]. The only significant adjustment we made to the algorithm consisted of removing any limitation related to cluster sizes. Former experiments with restructuring algorithms in paging environments had convinced us that more sophisticated clustering algorithms would not necessarily perform better.

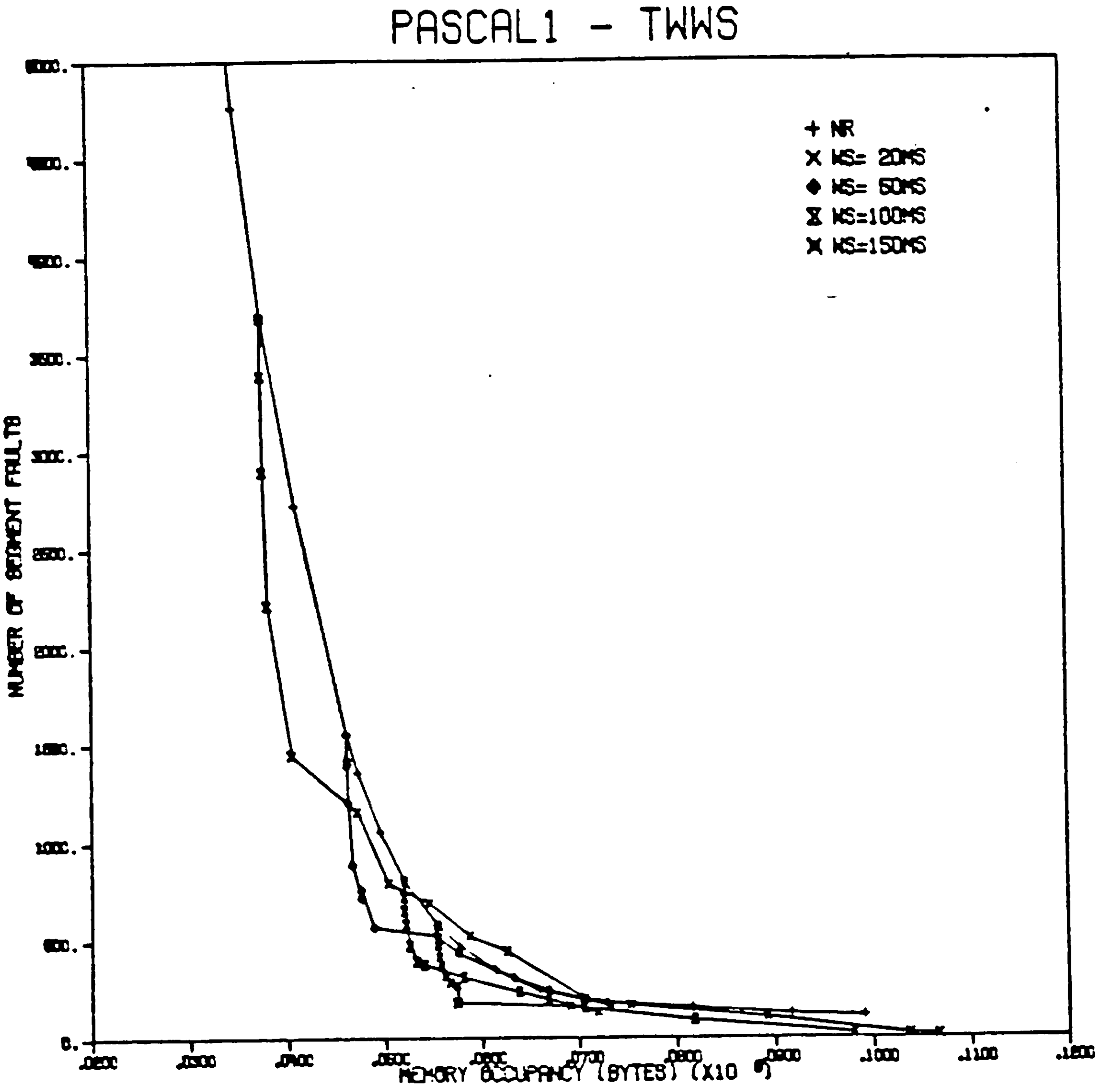

Figure 4.1

For each run, we measured the number of segment faults, the total number of bytes brought into memory and the mean memory occupancy of the program before and after restructuring. Figures 4.1 and 4.2 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 cost equal to zero. For that particular value, the structure of the program remains unchanged during the restructuring process.

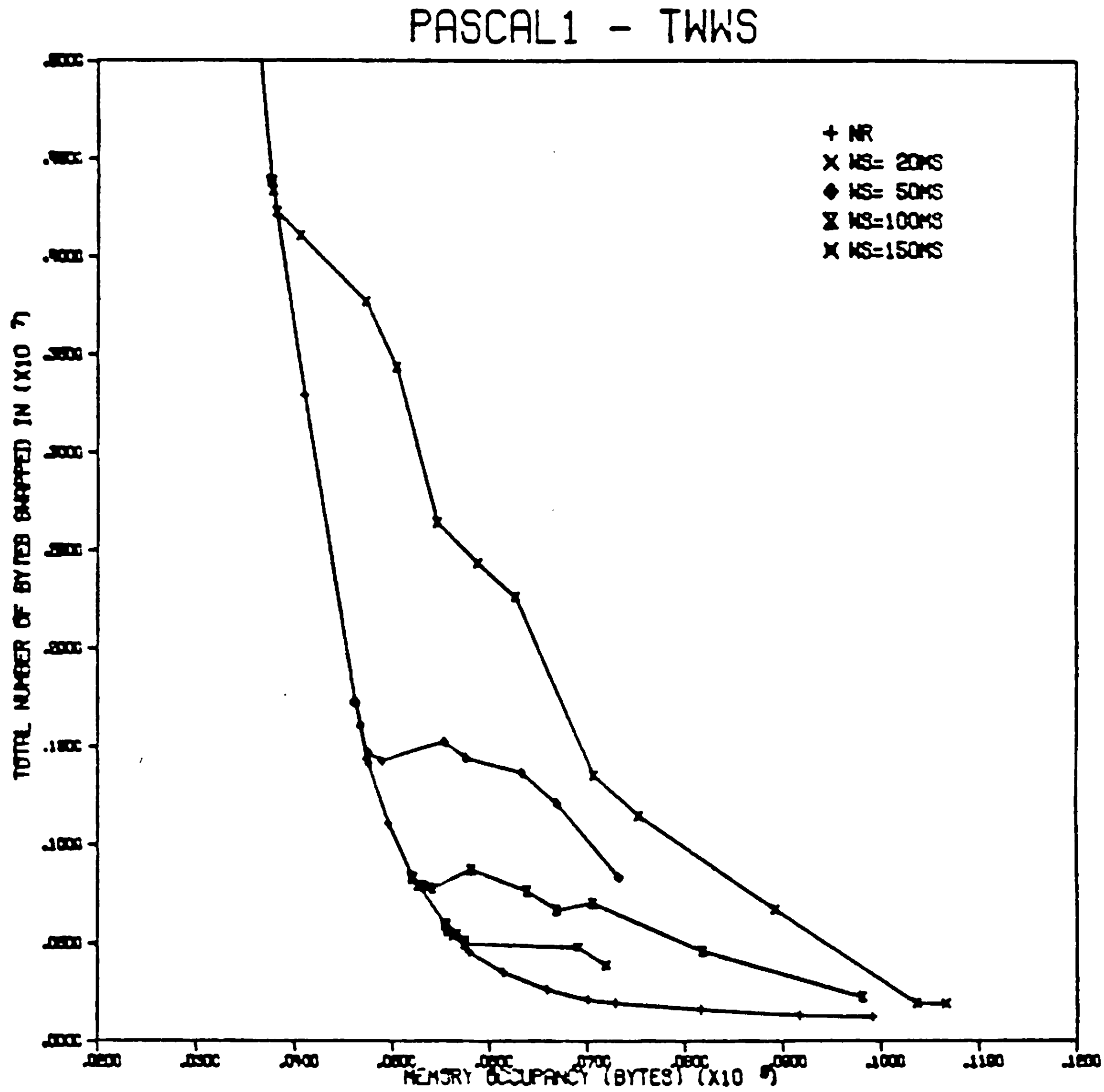

Figure 4.2

Looking at Figure 4.1, 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 α becomes superior or equal to 10^5 bytes * reference set, i. e. 28,494 bytes * ms. Figure 4.2, 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 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 4.2.2 and representing by $N_{b,in}$ the total number of bytes brought into memory during the execution of the program, one can thus write

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

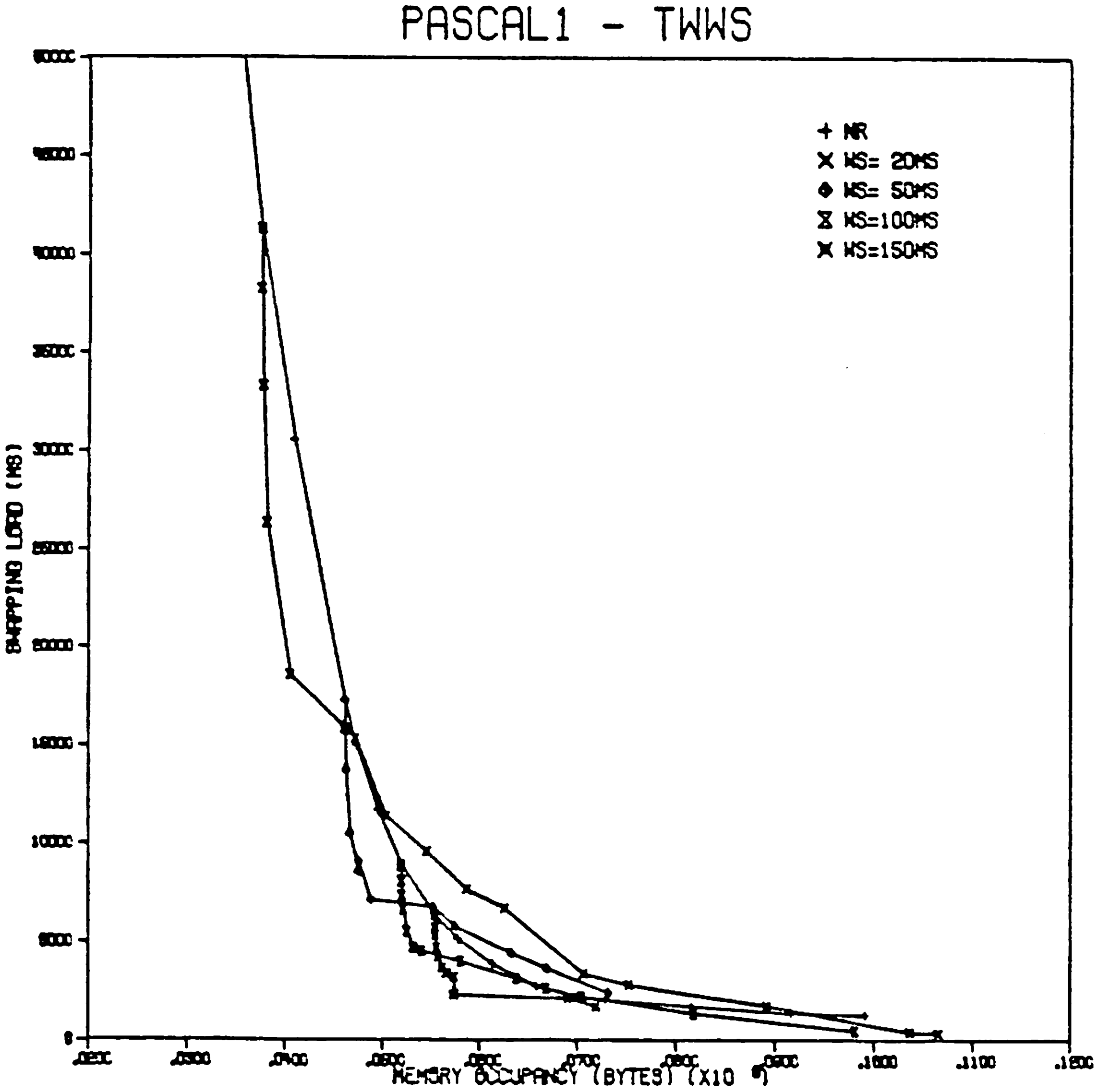

Figure 4.3

Figure 4.3 displays the values of this swapping load computed for a latency time $T_l = 10 ms$ 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.

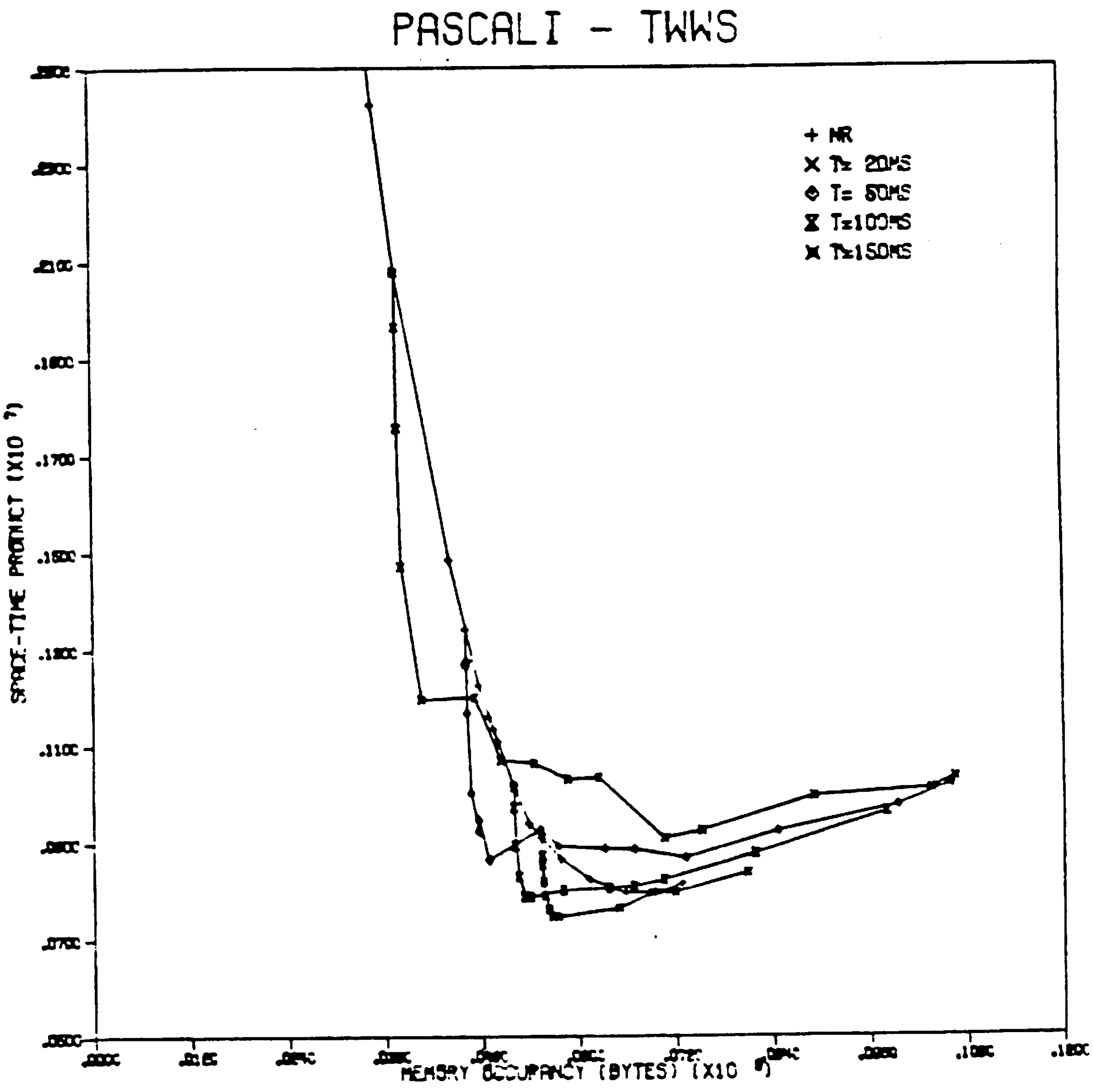

Figure 4.4

Similar conclusions could be reached by computing the program *space-time product C* as given by equation 4.1. Figure 4.4 displays the values of this space-time product computed for a latency time $T_l = 10ms$ and a transfer time $T_t = 10^{-6}$ s/byte.

4.4.2. Balanced Segment Fault Frequency Algorithm

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.

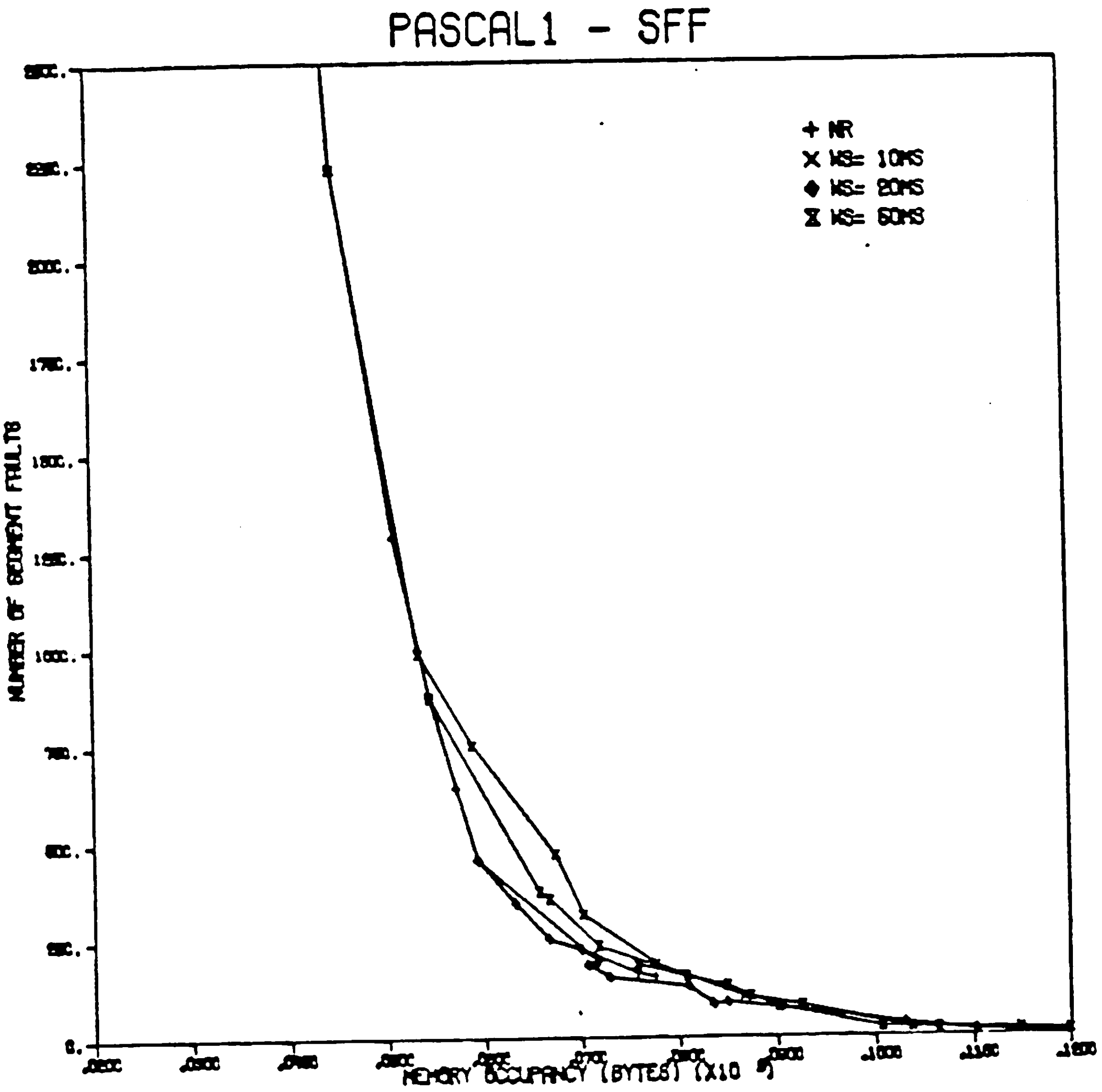

Figure 4.5

In this case, however, the results were quite different. As Figure 4.5 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 4.6. In conclusion, one can safely affirm that the restructuring process has no beneficial effects on the overall behavior of the program.

As we pointed already out in the preceding chapter, the Page Fault Frequency algorithm is known to exhibit some anomalies [Fran74]. In this case, however, we think that a much simpler explanation exists. Since the Segment Fault Frequency algorithm expels idle segments only at segment fault times [ChuO72], any decrease of the segment fault frequency below 1/T will result in an increase of the program's memory occupancy.

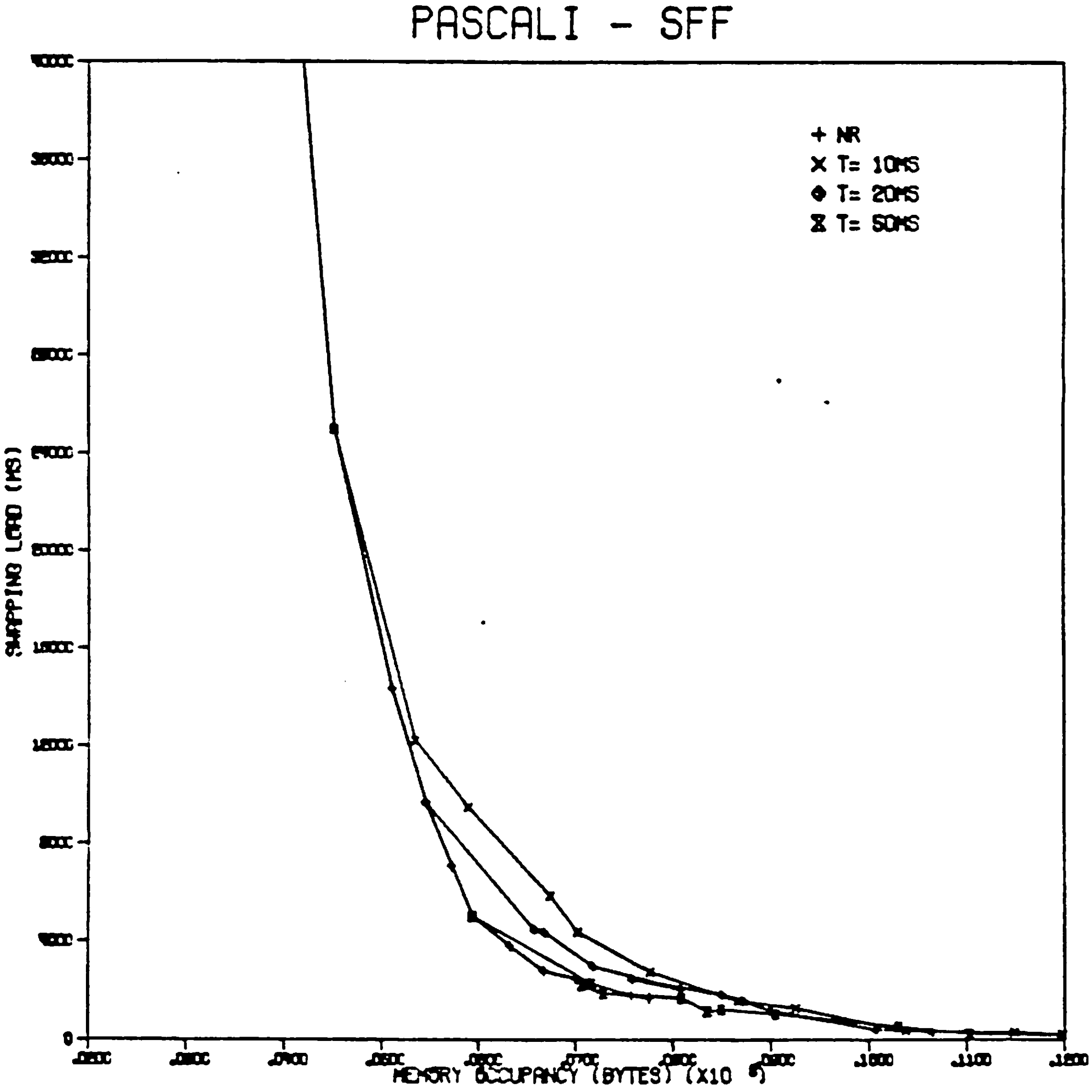

Figure 4.6

4.5. CONCLUDING REMARKS

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 segmentation environments should basically involve:

-- the gathering of more experimental evidence;

-- the study of possible modifications in the definition of interblock costs;

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

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

CHAPTER V

CONCLUSION

5.1. SUMMARY

One of the most effective ways of obtaining a better performance from virtual memory systems consists of improving the behavior of programs in such environments. Program restructuring attempts to achieve this goal by rearranging the block-to-page mappings of programs. Considerable experimental evidence exists supporting the effectiveness of this approach for paged virtual memory systems. This evidence also suggests that the most critical part of a program restructuring procedure is the criterion used to decide which blocks should be mapped to the same page and which not: The most efficient restructuring algorithms base their decisions on the run-time behavior of the program to be restructured and take also into account the memory policy under which the program will run.

Existing restructuring algorithms either had no quantifiable objectives or attempted to minimize a partial indicator of the performance of the restructured program--like its page fault frequency or its mean memory occupancy. Our contribution to field has been to introduce a new class of restructuring algorithms that attempt to minimize a global index of program performance, namely the program's *space-time product*. Our primary motivation was to avoid situations where a significant improvement of one index of program performance would be accompanied by a comparably sized deterioration of another index. Hence the name of "Balanced Algorithms" given to our algorithms.

Balanced Algorithms essentially attempt to minimize a restructuring-time estimate of the space-time product of the restructured program. Since they share a common algorithm scheme, they can be easily tailored to a wide range of variable-space memory policies, including Working Set, Sampled Working Set, Global LRU and Page Fault Frequency.

We were able to prove that BWS, the balanced algorithm tailored to Working Set environments, effectively minimizes a linear combination of the number of page faults 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. Arguments were also presented showing why the same claim would not hold for the balanced algorithms tailored to the Global LRU and Page Fault Frequency policies.

In order to evaluate the performance of balanced algorithms under various memory policies and to compare it to those of other restructuring algorithms, we conducted a series of trace-driven experiments simulating the behavior of programs before and after restructuring under several experimental conditions. The parameters studied were the system's memory policy, the control parameter of this memory policy and the page size. In each case, the two other restructuring algorithms simulated were a strategy-oriented algorithm minimizing the page fault frequency of programs running under that policy and another one minimizing their mean memory occupancy.

These simulations show that BWS, the balanced algorithm tailored to Working Set environments, performs significantly better than the two other algorithms. Similar results were found with the balanced algorithm tailored to Sampled Working Set environments. BPSI, the balanced algorithm tailored to Global LRU environments, exhibited only a marginal superiority over its rivals, while no clear winner emerged for the Page Fault Frequency environments.

Another consequence of our choice of a global indicator of program performance as restructuring criterion is to allow the extension of our approach to *segmentation environments*, for which no efficient restructuring algorithms were known.

Here too, we were able to prove that BTWWS, the balanced algorithm tailored to Time-Window Working Set environments, minimizes a linear combination of the number of segment faults and of the mean memory occupancy of all programs whose behavior can be described by a Markov chain having a steady-state solution and which have at most two blocks per segment.

Experimental evidence was also presented showing that BTWWS can significantly improve the segment fault frequency of a program without causing any comparable increase of its memory occupancy. On the other hand, our simulations indicated that BSFF, the balanced algorithm tailored to Segment Fault Frequency environments, did not bring any improvement to either indices of program performance.

5.2. DIRECTIONS FOR FURTHER RESEARCH

Several questions concerning the behavior of Balanced Algorithms remain open, among which the analytical modeling of the Balanced Sampled Working Set Algorithm and the sensitivity of Balanced Algorithms to changes in the program's input data or to readjustments of the memory policy control parameter. In the latter case, previous experimental data concerning other strategy-oriented restructuring algorithms suggest however that the performance of Balanced Algorithms would not be too strongly affected by these two factors [Ferr76a] [Ferr76b].

Another way of looking at the problem would be to design *strategy-independent* balanced algorithms following the approach pioneered by Kobayashi [Koba77]. It is however not yet clear which underlying model of program behavior should be chosen and how localities should be defined.

Future research in the field of program restructuring algorithms tailored to segmentation environments should first be concerned with the gathering of more experimental evidence. Other directions of research would be essentially similar to the ones sketched above for the paging environments and have been enumerated at the end of chapter IV.

Another goal for further research would be the design of restructuring algorithms tailored to fixed-space segmentation policies. The only optimization criteria in this case are the segment fault frequency and the swapping load. The most difficult problem lies in the fact that the restructuring algorithm should then be able to evaluate which segments risk to be expelled from memory each time two blocks are merged in order to form a larger segment. Approximate solutions probably exist for LRU environments but it is difficult to assess the performance of these algorithms without any experimental evidence.

A last research direction--and probably the most promising one-- would be to apply Balanced Algorithms to the problem of *optimal prefetching* in paging environments. Rather than attempting to decide at run-time which pages should be fetched when a fault occurs, one could define off-line *clusters* of pages that would always be fetched into memory and returned to the secondary store as a single entity. This problem is essentially equivalent to the one of finding the best block-to-segment mapping for a program to be executed in a segmentation environments and very similar techniques could be used.

BIBLIOGRAPHY

[Acha75] Achard, M.S., J. Y. Babonneau and G. Morisset, "Segmentation Automatique des Programmes Indépendamment des Langages de Programmation," Rapport de Recherches No. 125, IRIA-LABORIA, Le Chesnay, France, May 1975.

[Acha78] Achard, M. S., J. Y. Babonneau, M. Carpentier, G. Morisset and M. B. Mounajjed, "The Clustering Algorithms in the OPALE Restructuring System," in *Performance of Computer Installations* (D. Ferrari ed.) North Holland, Amsterdam, Netherlands, 1978, pp. 137-163.

[Aho 71] Aho, A. V., P. J. Denning and J. D. Ullman, "Principles of Optimal Page Replacement," *J. ACM* 18, 1 (Jan. 1971), 80-93.

[Baba79] Babaoglu, O., W. Joy and J. Porcar, "Design and Implementation of the Berkeley Virtual Memory Extension to the Unix Operating System," University of California, Berkeley, (1979).

[Babo77] Babonneau, J. Y., 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.

[Baer72] Baer, J.-L. and G. R. Sager, "Measurement and Improvement of Program Behavior under Paging Systems," in *Statistical Computer Performance Evaluation* (W. Freiberger ed.), Academic Press, New York, 1972,pp. 241-264.

[Baer76] Baer, J.-L. and G. R. Sager, "Dynamic Improvement of Locality in Virtual Memory Systems," *IEEE Trans. Softw. Engrg.* SE-2, 1 (Mar. 1976), 54-62.

[Bard73] Bard, Y., "Characterization of Program Paging in a Time-sharing Environment," *IBM J. Res. Develop.* 17, (Sept. 1973), 387-393.

[Bard75] Bard, Y., "Application of the Page Survival Index (PSI) to Virtual Memory System Performance," *IBM J. Res. Develop.* 19, 3 (May 1975), 212-220.

[Bats76] Batson, A., "Program Behavior at the Symbolic Level," *Computer* 9, 11 (Nov. 1976), 21-28.

[Bayl68] Baylis, M. H. J., D. G. Fletcher and D. J. Howarth, "Paging Studies made on the I.C.T. Atlas Computer," *Information Processing 68*, Proc. 1968 IFIP Congress, pp. 831-837.

[Bela66] Belady, L. A., "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Sys. J.* 5, 2 (1966), 78-102.

[Bela69] Belady, L. A. and C. J. Kuehner, "Dynamic Space Sharing in Computer Systems," *Comm. ACM* 12, 5 (May 1969), 282-288.

[Benn77] Bennett, B. T. and P. A. Franaczek, "Permutation Clustering: An Approach to On-Line Storage Reorganization," *IBM J. Res. Develop.* 21, (Nov. 1977), 528-533.

[Braw68] Brawn, B. and F. Gustavson, "Program Behavior in a Paging Environment," 1968 AFIPS FJCC, *AFIPS Conf. Proc.*, Vol. 33, 1019-1032.

[Braw70] Brawn, B. S., F. G. Gustavson and E. S. Mankin, "Sorting in a Paging Environment," *Comm. ACM* 13, 8 (Aug. 1970), 483-494.

[Chen74] Chen, P. S. and A. Gallo, , "Optimization of Segment Packing in Virtual Memory," in *Computer Architecture and Networks* (E. Gelenbe and R. Mahl Eds.), North Holland Publ., 1974.

[Chu72] Chu, W. W. and H. Opderbeck, "The Page Fault Frequency Paging Algorithm," 1972 AFIPS FJCC, *AFIPS Conf. Proc.*, Vol. 41, Pt. 1, 597-609.

[ChuO76] Chu, W. W. and H. Opderbeck, "Program Behavior and the Page-Fault-Frequency Replacement Algorithm," *Computer* 9, 11 (Nov. 1976), 29-38.

[Coff73] Coffman, E. G. and P. J. Denning, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[Come67] Comeau, L., "A Study of the Effect of User Program Optimization in a Paging System," *ACM Symp. on Oper. Sys. Prin.*, (Oct. 1967), Gatlinburg, Tenn.

[Corb68] Corbato, F. J., "A Paging Experiment with MULTICS System," Memo Mass.C-M-384, Project MAC, M.I.T., Cambridge, MA, 1968.

[Cour76] Courtois, P. J. and Vantilborgh, H. "A Decomposable Model of Program Paging Behaviour," *Acta Informatica* 6 (1976), 251-275.

[DEC78] Digital Equipment Corporation, *VAX 11/780 Technical Summary*. Maynard, Mass., 1978.

[DenP66] Denning, P. J. "Memory Allocation in Multiprogrammed Computer Systems," MIT Project MAC, Computation Structures Group Memo 24, Mar. 1966.

[DenP68] Denning, P. J., "The Working Set Model for Program Behavior," *Comm. ACM* 11, 5 (May 1968), 323-333.

[DenP68b] Denning, P. J., "Thrashing: Its causes and Prevention," 1968 AFIPS FJCC, *AFIPS Conf. Proc.*, Vol 33, 915-922.

[DenP70] Denning, P. J., "Virtual Memory," *Computing Surveys* 2, 3 (September 70), 153-189.

[DenP72] Denning, P. J. and S. C. Schwartz, "Properties of the Working Set Model," *Comm. ACM* 15, 3 (Mar. 1972), 191-198. Corrigendum: *Comm. ACM* 16, 2 (Feb. 1973), 122.

[DenP76a] Denning, P. J., "An L=S Criterion for Optimal Multiprogramming," *Proc. Int. Symp.Computer Performance Modeling, Measurement and Evaluation*, ACM SIGMETRICS and IFIP WG 7.3, Mar. 1976, pp 219-229.

[DenP76b] Denning, P. J., K. C. Kahn, J. Leroudier, D. Potier and R. Suri, "Optimal Multiprogramming," *Acta Informatica*, 7, 2 (1976), 197-216.

[DenP78] Denning, P. J. and D. R. Slutz, "Generalized Working Sets for Segment Reference Strings," *Comm. ACM* 21, 9 (Sept. 1978), 750-759.

[DenP80] Denning, P. J., "Working sets Past and Present," *IEEE Trans. Softw. Engrg.* SE-6, 1 (Jan. 1980), 64-84.

[DenJ65] Dennis, J. R., "Segmentation and the Design of Multiprogrammed Computer Systems," *J. ACM* 21, 4 (Oct. 1965), 589-602.

[Dida79] Diday, E., "Problems of Clustering and Recent Advances," Rapport de Recherches No. 337,IRIA-LABORIA, Le Chesnay, France, Jan. 1979.

[Dong79] Dongara, J. J., J. R. Bunch, C. B. Moler and G. W. Steward, *LINPACK User's Guide*. SIAM, Philadelphia, PA, 1979.

[Dora76] Doran, R. W., "Virtual Memory," *Computer* 9, 10 (Oct. 1976), 27-37.

[East75] Easton, M. C., "Model for Interactive Data Base Reference Strings," *IBM J. Res. Develop.*, 19, (Nov. 75), 550-556.

[East77] Easton, M. C. and R. Fagin, "Cold-Start v. Warm-Start Miss Ratios," *Comm. ACM* 21, 10 (Oct. 1978), 866-872.

[East78] Easton, M. C. , "Model for database Reference Strings Based on Behavior of Reference Clusters," *IBM J. Res. and Dev.* 22, 2 (March 1978), 197-202.

[East79] Easton, M. C. and P. A. Franaczek, "Use Bit Scanning in Replacement Decisions," *IEEE Trans. Comput.*, C-28, 2 (Feb. 1979), 133-141.

[Ferr73] Ferrari, D., "A Tool for Automatic Program Restructuring," *Proc. 1973 ACM National Conf.*, Atlanta, GA, 223-228.

[Ferr74a] Ferrari, D., "Improving Program Locality by Strategy-Oriented Restructuring," *Information Processing 74*, Proc. 1974 IFIP Congress, pp. 266-270.

[Ferr74b] Ferrari, D., "Critical-Set Algorithms for Program Locality Improvement," *Proc. 12th Atherton Conf. on Circuit and Systems Theory*, Monticello,. IL, (Oct. 1974), 641-648.

[Ferr74c] Ferrari, D. "Improving Localities by Critical Working Sets," *Comm. ACM* 17 , 11 (Nov. 1974), 614-620.

[Ferr75] Ferrari, D., "Tailoring Programs to Models of Program Behavior," *IBM J. Res. Develop.* 19, 3 (May 1975), 244-251.

[Ferr76a] Ferrari,D. 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.

[Ferr76b] Ferrari, D., "The Improvement of Program Behavior," *Computer* 9, 11 (Nov. 1976), 39-47.

[Ferr77a] Ferrari, D. and M. Kobayashi, "Program Restructuring for Global LRU Environment," *Conf. Proc. of Int. Computing Symp.*, Liège, Belgium, April 4-7, 1977.

[Ferr77b] Ferrari, D. "An Approach to the Design of a Learning Memory Manager," *Proc. 1977 SIGMETRICS / CMG VIII Conf. on Computer Performance: Modeling, Measurement and Management*, Washington, D. C., Nov. 29-Dec. 2.

[Ferr78] Ferrari, D. *Computer Systems Performance Evaluation*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Ferr80] Ferrari D., private communication (June 1980).

[Fisc79] Fischer, P. C. and R. L. Prower, "Storage Reorganization Techniques for Matrix Computation in a Paging Environment," *Comm. ACM* 22, 7 (July 1979), 405-414.

[Foth61] Fotheringham, J., "Dynamic Storage Allocation in the ATLAS Computer, Including an Automatic Use of Backing Store," *Comm. ACM* 4, 10 (Oct. 1961), 435-436.

[Fran74] Franklin, M. A. and R. K. Gupta, "Computation of Page Fault Probability from Program Transition Diagram," *Comm. ACM* 17, 4 (Apr. 1974), 187-191.

[Grah76] G. S. Graham, "A Study of Program and Memory Policy Behavior," Ph. D. Dissertation, Dept. of Comp. Sci., Purdue U., W. Lafayette, Ind., Dec. 1976.

[Haik78] Haikala, I., "Ohjelman Uudeleenryhmittely Segmentoivassa Ymparistossa" (Program Restructuring in a Segmented Environment), TKOL (Department of Computer Sciences) series C 70/78, University of Helsinki, Finland.

[Hatf71] Hatfield, D. J. and J. Gerald, "Program Restructuring for Virtual Memory," *IBM Sys. J.* 10 , 11 (Nov 1971), 39-47.

[Hoag79] Hoagland, A. S., "Storage Technology: Capabilities and Limitations," *Computer* 12, 5 (May 1979), 12-18.

[Jose70] Joseph, M., "An Analysis of Paging and Program Behavior," *Computer J.* 13, 1 (Feb. 1970), 48-54.

[Karp75] Karp, R. M., "On the Computational Complexity of Combinatorial Problems," *Networks* 5 (1975), 45-68.

[Kilb62] Kilburn, T., D. Edwards, M. J. Lanigan and F. Sumner, "One-level Storage Systems," *IRE Trans.* EC-11, Vol. 2 (April 1962), 223-235.

[King71] King, W. F. III, "Analysis of Demand Paging Algorithms," *Proc. IFIP Congress 71*, Ljubjana, Yugoslavia (Aug. 1971), TA-3-155 to TA-3-159.

[Koba77] Kobayashi, M., "A Set of Strategy Independent Restructuring Algorithms," *Software–Practice and Experience* 7, 5 (1977),585-594.

[Koba79] Kobayashi, M. "The Working Set Distribution of the Markov Program Behavior Model," Memorandum No. UCB/ERL M79/46, Electronics Research Laboratory, University of California, Berkeley, Calif., July 1979.

[Knut73] Knuth, D. E., *The Art of Computer Programming*, Vol. 1: Fundamental Algorithms, 2nd. ed., Addison-Wesley, Reading, Mass., 1973.

[Kubo76] Kubo, H. and M. Kobayashi, "Evaluation of Optimal Page Size and Initial Loading under a Systemwide Criterion," *NEC Res. Develop.* 41 (Apr. 1976), 27-37.

[Lau 79] Lau, E., "Performance Improvement of Virtual Memory Systems by Restructuring and Prefetching," Ph. D. Dissertation, Department of EECS, University of California, Berkeley, 1979.

[Lerou76] Leroudier, J. and D. Potier, "Principles of Optimality for Multiprogramming," *Proc. Int. Symp. Computer Performance Modeling, Measurement, and Evaluation*, ACM SIGMETRICS and IFIP WG7.3, Mar. 1976,pp. 221-218.

[Lowe70] Lowe, T. C. "Automatic Segmentation of Cyclic Program Structures Based on Connectivity and Program Timing," *Comm. ACM*, 13, 1 (Jan. 1970), 3-9.

[Madi76] Madison, A. W. and A. P. Batson, "Characteristics of Program Localities," *Comm. ACM* 19, 5 (May 1976), 285-294.

[Mars79] Marshall, W. T. and C. T. Nute, "Analytical Modelling of 1979 Conf. on Simulation, Measurement and Modeling of Computer Syst., 65-72.

[Masu74] Masuda, T., H. Shiota, K. Noguchi, and T. Ohki, "Optimization of Program Performance by Cluster Analysis," *Information Processing 74*, Proc. IFIP 1974 Congress, 226-270.

[Masu79] Masuda, T., "Methods for the Measurement of Memory Utilization and the Improvement of Program Locality," *IEEE Trans. Softw. Engrg.* SE-5, 6 (Nov. 1979), 618-631.

[Matt70] Mattson, R. L., J. Gecsei, D. R. Slutz and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Sys. J.* 9, 2 (1970),78-117.

[McKe69] McKellar, A. C. and E. G. Coffman, "Organizing Matrices and Matrix Operations for Paged Memory Systems," *Comm. ACM* 12, 3 (Mar. 1969), 153-165.

[Oliv74] Oliver, N. A. "Experimental Data on Page Replacement Algorithm," 1974 AFIPS NCC, *AFIPS Conf. Proc.*, Vol. 43, 179-184.

[Opde74] Opderbeck, H. and W. W. Chu, "Performance of the Page Fault Frequency Algorithm in a Multiprogramming Environment," *Information Processing 74*, Proc. IFIP 1974 Congress, 235-241.

[Orga72] Organick, E. I. *The Multics System*, MIT Press, Cambridge, Mass., 1972.

[Pari76] Paris, J.-F., "Stratégies Optimales en Restructuration de Programmes," R. P. 14/76, Institut d'Informatique, Facultés Universitaires de Namur.

[Pari78] Paris, 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.

[Pari81] Paris, J.-F., "Program Restructuring in Segmenting Environments," in *Experimental Computer Performance Evaluation* (D. Ferrari and M. Spadoni eds.) North-Holland, Amsterdam, Netherlands,pp 249-264.

[Prie76] Prieve, B. G. and R. S. Fabry, , "VMIN--An Optimal Variable Space Page Replacement Algorithm," *Comm. ACM* 20, 5 (May 1976), 295-297.

[Rama66] Ramamoorthy, C. V., "The Analytical Design of a Dynamic Look-Ahead and Program Segmenting System for Multiprogrammed Computers," *Proc. 1966 ACM National Conf.*, 229-239.

[Rand69] Randell, B., "A Note on Storage Fragmentation and Program Segmentation," *Comm. ACM* 12, 7 (July 1969),365-369 and 372.

[Requ78] Requa, J. A. "Virtual Memory Design Reduces Program Complexity," *Computer Design* 17, 1 (Jan. 1978), 97-106.

[Rodr73] Rodriguez-Rosell, J. and J. P. Dupuy, "The Design, Implementation, and Evaluation of a Working Set Dispatcher," *Comm. ACM* 16, 4 (Apr. 1973), 556-560.

[Rodr76] Rodriguez-Rosell, J., "Empirical Data Reference Behavior in Data Base Systems," *Computer* 9, 11 (Nov. 1976), 9-13.

[Ryde74] Ryder, K. D. "Optimizing Program Placement in Virtual Systems," *IBM Sys. J.* 13, 4 (April 1974), 292-306.

[Salt75] Saltzer, J. H. and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE* 63, 9 (Sept 1975), 1278-1308.

[Shem66] Shemer, J. E. and B. Shippey, "Statistical Analysis of Paged and Segmented Computer Systems," *IEEE Trans. Comp.* EC-15,6 (Dec. 1966), 855-863.

[Smit76] Smith, A. J., "A Modified Working Set Paging Algorithm," *IEEE Trans. Comp.* C-25, 9 (Sept. 1976), 907-914.

[Smit77] Smith, A. J., "Two Simple Methods for Efficient Analysis of Memory Trace Data," *IEEE Trans. Softw. Engrg.* SE-3 , 1 (Jan. 1977), 94-101.

[Smit78a] Smith, A. J., "A Comparative Study of Set Associative Memory Mapping and Their Use for Cache and Main Memory," *IEEE Trans. Softw. Engrg.* SE-4, 2 (Mar. 1978), 121-130.

[Smit78b] Smith, A. J., "Sequentiality and Prefetching in Data Base Systems," *ACM Trans. DBS* 3, 3 (Sept. 1978), 223-247.

[Smit78c] Smith, A. J., "Bibliography on Paging and Related Topics," *Oper. Syst. Review* 12, 4 (Oct. 1978), 39-56.

[Smit78d] Smith, A. J., "Sequential Program Prefetching in Memory Hierarchies," *Computer* 11, 12 (Dec. 1978), 7-21.

[Smit80] Smith, A. J., "Multiprogramming and Memory Contention," *Software—Practice and Experience*, 10 (1980), 531-552.

[Smit81] Smith, A. J., "Internal Scheduling and Memory Contention," *IEEE Trans. Softw. Engrg.* SE-7, 1 (Jan. 1981), 135-146.

[Snyd78] Snyder, R. G. "On A Priori Program Restructuring for Virtual Memory Systems," in *Operating Systems Theory and Practice* (D. Lanciaux ed.), Proc. Second Int. Sympos. on Operating Systems Theory and Practice, Rocquencourt, France, October 2-4, 1978, North Holland, Amsterdam, The Netherlands, 1979, pp. 207-224.

[Spir72] Spirn, J. R. and P. J. Denning, "Experiments with Program Locality," 1972 AFIPS FJCC, *AFIPS Conf. Proc.*, Vol. 41, 611-622.

[Spir76] Spirn, J. R., "Distance Strings Models for Program Behavior," *Computer* 9, 11 (Nov. 1976), 14-20.

[Spir77] Spirn, J. R., *Program Behavior: Models and Measurements*, Elsevier North-Holland, New York 1977.

[Triv77] Trivedi, K. S., "On the Paging Performance of Array Algorithms," *IEEE Trans. Comp.* C-26, (October 1977), 938-947.

[Tsao72] Tsao, R. F., L. W. Comeau and B. H. Margolin, "A Multi-Factor Paging Experiment: I. The Experiment and the Conclusion," in *Statistical Computer Performance Evaluation* (W. Freiberger ed.), Academic Press, New York, 1972,pp. 103-134.

[VerH71] Ver Hoef, E. W., "Automatic Program Segmentation Based on Boolean Connectivity," 1971 AFIPS SJCC, *AFIPS Conf. Proc.*, Vol. 38, 491-495.

[Yu 76] Yu, F. S., "Modeling the Write Behavior of Computer Programs," Ph. D. thesis, Dept. of Computer Sci., Stanford U., Palo Alto, Calif., May 1976.