# THE IMPROVEMENT OF PROGRAM BEHAVIOUR IN PAGED COMPUTER SYSTEMS

C. J. PAVELIN



Ph.D.

University of Edinburgh

August 1970

# THE IMPROVEMENT OF PROGRAM BEHAVIOUR IN PAGED COMPUTER SYSTEMS


## SUMMARY

The thesis initially considers two questions: what is meant by program behaviour in a paged computer system, and what is meant by its improvement. The former, especially, demands a general discussion on paging and its uses and abuses in modern operating systems. Reference is made to a brief study of the behaviour of some KDF9 programs. The problem of restructuring a program in order to improve its paging behaviour, is then investigated; a solution using clustering techniques is suggested. A scheme to perform such a restructuring automatically, on the basis of monitored information from the program at run-time, has been implemented on the ICL 4-70. This scheme is described and some results presented; these show that considerable improvement can be obtained.



This work was supported by International Computers Ltd.

CONTENTS

utilisation whatever the degree of multiprogramming.

2) A 'virtual store' (see introduction) can be realised, a great convenience for the user.

3) Use of memory is made flexible; a program can be loaded without the need for a large area of contiguous vacant store. (The mapping mechanism solves all relocation problems in physical store.)

In paging systems we have in addition:

4) extreme simplicity, owing to the fact that a page can be loaded arbitrarily into any available page frame. If allocation is in non-constant size units, as these are loaded and unloaded the vacant store rapidly fragments into blocks of varying lengths. When a storage request is made, the system has to search for a suitable unoccupied section or perhaps reshuffle the code around the store to create one. Such problems, and the considerable software overheads associated with their solution, do not arise in the paging situation. In addition the constancy of size means one less set of variable parameters in the system.

Disadvantages

1) Mechanisms which map from the program name space into actual address space can be expensive (the hardware components), and result in an increase in average addressing time.

2) The tables containing information about units necessary to the address mapping mechanism (e.g. page tables) take up valuable space in main memory.

3) There is greater software (and possibly hardware) overhead in moving several small areas of data between working store and backing store, than one large area. (Thus the total overhead in

transferring a complete program into store is greater for smaller units.)

And finally the additional factor against paging systems.

4) Since page sizes are fixed, their boundaries will normally be imposed more or less arbitrarily on a program. This makes it more difficult to reap the profit from advantage 1 than in the case where variable size units of allocation can be chosen to reflect the structure of the program to some degree.

Paging thus gives the gain of advantage 4 while foregoing part of the potential benefits expressed in advantage 1. The realisation of the latter is in any case far from easy; but this is particularly true in the paging environment. However it was not simply the failure to achieve these benefits that led to unfortunate results in some early paging systems, but a lack of appreciation of the fact that this is where the problem lay.

(We note in passing that the choice of ideal page size is a compromise between the overheads of 2 and 3 above, and the gains of advantage 1. For the latter to exist at all, the size clearly has to be fairly small compared to the average program requirements; this gives an upper bound, but in practice it is not clear what lower bound the disadvantages impose. The commonly chosen page size of about 1000 words may be dictated more by tradition than measured efficiency considerations. Shemer and Shippey (ref.4) detail the effects of page size.)

## 2. THE ALLOCATION PROBLEM

The problem arises from the fact that there is no sure way of knowing which parts of a program will be referenced in a particular

run or time-slice. Short of loading the whole program, which obviously extracts none of the potential benefits of advantage 1, there must be at least some occasions when reference is made to a page not in main memory, i.e. a page-fault occurs. It is at these times that whatever action the system takes introduces a possible cause of inefficiency.

1) The normal course is to load the referenced page (loading only after a faulting reference is known as demand paging). If there is an available page-frame, i.e. one not occupied by pages of an active program, there is no difficulty (but even then it should be noted that the I/O overhead involved in loading a set of pages is less when requests are made altogether than when each page is loaded singly after a page-fault.)

More significant is the likely case that there is no available space when the page-fault occurs. This makes it necessary to over-write - after writing back if necessary - a page, either of the program concerned, or of another. This in itself is not significant, but it becomes so if the overwritten page is required again in the current run or time-slice of its program. Every reloading represents a possible loss of efficiency over whole program roll-in roll-out methods.

One could reserve for the program before it is allowed in store a sufficient space allocation to ensure that the above situation never occurred, but this extreme would probably lead to reservation of almost the total program size, and again scarcely realise one of the main benefits of the small allocation unit.

2) The other alternative, in a time-sharing system, is to use the faulting reference to terminate the time-slice, the program

being unloaded to make room for others. This is no solution though; there will be an increase in the proportion of time spent loading and unloading programs, and a decrease in CPU utilisation. An upper bound to the length of the time-slice is a constraint arising from the demands of a good response time for a large number of users; a system is unwise to reduce a time-slice more than these inefficient demands of time-sharing already dictate.

Thus the attempt to take some of advantage 1 of the last section may lead to more movement of pages between backing store and main memory (page swapping). A very simple model can demonstrate at least the possibility of this increased I/O swamping the multi-programming advantage which apparently arises from having more programs in store.

Suppose program i is allocated store size $s_i$ over a long period, and that as faults occur, pages of i (chosen according to some strategy) are overwritten. Take as a unit of time, the average page fetch time from immediate backing store. (Page write time is not explicitly included as only those pages which have changed have to be written back - this can be allowed for in the average fetch time if necessary.) For rotating backing store devices, the unit is typically a few thousand instruction cycles.

Let $r_i(s_i)$ be the average number of page faults that occur from program i during a unit of its own processing time (this is the program's page-fault rate). An upper limit to the ratio of its CPU time : real time is $1:1+r_i(s_i)$ ——(1.1)
(this is an upper limit because time when the program is ready to use the CPU but it is unavailable, is ignored). Then the total

CPU utilisation is bounded above by:

$$\sum_{\text{all progs.}} \frac{1}{1+r_i(s_i)}$$

If the total core size is M, and a set of similar programs are each allocated store s (giving fault rates r(s) ), the number of such programs that can be accommodated is M/s; the CPU utilisation is then at most:

$$\frac{M}{s(1+r(s))} \qquad \text{——(1.2)}$$

Suppose each program is subject to processing time-slices of length T (between which its pages are removed), and it references S(T) distinct pages in such a time-slice. Assuming wholly demand paging, then r as defined above will achieve its minimum value of S(T)/T for s=S(t). For allocations greater than S(T), the efficiency will clearly fall. But consider a very small s, perhaps just two or three pages; little knowledge of program behaviour is required to see that the interval between faults can easily be no more than a few instructions. The value of r (in the above units) would then be over a thousand and the CPU utilisation practically zero, except with a vast main memory. It is obvious how irrelevant the multiprogramming advantage (effectively the multiplier M/s) can become if page-faults occur with great rapidity. (Note also how a small time-slice - $T \ll 1$ in the defined units - imposes an immediate bound on achievable efficiency).

Somewhere between the extremes of store allocation, there will be an optimum. The difficulty lies in finding it, or equivalently finding the optimum number of the programs currently competing for service, to allow in store. It will depend at any moment on the

behaviour of all the programs, the replacement strategy being used, and other system variables. The best allocation will be continuously varying and modern systems adopt adaptive strategies (refs.5,6); these respond to changing program demands and try and maintain page-faulting at an acceptable level, even if some programs have to be given a large proportion of their total size.

The overwhelming significance of the page fetch time should be quite clear; it is because our time unit is extremely long compared to instruction times (and therefore to likely intervals between page faults) that efficiency problems can so easily arise (see e.g. Denning ref.7). Bulk core store cheap enough to replace the rotating drum normally used for immediate backing store at present, could revolutionize the efficiency of time sharing systems.

Inexplicably, some early system designers allowed the concept of paging to obscure the problems of storage allocation. All active programs fought simultaneously for the available store with the inevitable results that excessive page swapping led to congestion and very low CPU utilisation. However programs were believed to behave under paging, there would seem to be no reason to have adopted such a policy.

Notes on formula 1.2

This formula only purports to give insight into gross aspects of system performance, but since it is referred to later in this chapter, a mention of the principal limitations is given here.

1) Ratio 1.1 is an upper limit only attained when there is a small number of programs and the CPU utilisation is fairly low. In practice the situation is worse than the model suggests - there

will be occasions when a program is held up not for I/O but because another process is using the CPU. (Even a moderately accurate treatment of multiprogramming is complex, and depends on probability models for aspects of program behaviour - see ref.8.)

2) The page fetch time is not constant. Worse still, its average will lengthen when queueing for I/O channels starts to take place. This may be due to heavy page faulting of active programs, or pages filtering through lower levels of store (disc to drum etc.), via the main memory.

3) CPU time lost due to system use or I/O cycle stealing is not allowed for.

On all these counts, formula 1.2 gives an optimistic view of the situation.

## 3. PROGRAM BEHAVIOUR

A program which references a large proportion of its pages within short time intervals is generally said to be 'badly behaved' (or have a high 'vagrancy'). A good operating system must expect, and be able to cope with, programs displaying any behaviour pattern; a badly behaved program simply yields less of the advantages of small-unit allocation. However a knowledge of general characteristics of average programs, or perhaps of particular system software, is useful for performance prediction, aspects of design, and for evaluation of paging as an allocation method. Since the much cited and pessimistic study by Fine, Jackson and McIsaac (ref.9), there have been numerous other empirical investigations of paging behaviour (e.g. refs. 10,11,12).

Unfortunately it is not easy to decide just what to measure;

the complete page reference history of a process is very difficult to sum up briefly but usefully in a quantitative manner, although qualitative statements (such as 'well-behaved') can be made. Many studies have avoided the difficulty by examining not how the program behaves in absolute page reference terms but terms relating to its performance when run under a given strategy of allocation and replacement (refs.11,13). A typical measure is the total number of page faults or sometimes a highly system dependent statistic such as total elapsed time. Such investigations are very useful if there is some relation to real operating systems, but it must be remembered that the results may say more about the particular allocation schemes than about the program behaviour under paging. Other workers (refs.9,12) have simply made direct measurements of possibly relevant statistics - e.g. the average number of instructions obeyed in a page before a branch to another page, or the average number of pages accessed between supervisor calls. There has been general agreement that programs in general display a higher vagrancy than originally hoped (ref.9 implies that the conception had been of 'a high speed memory filled with a page or two from each of many programs requiring processing') but more specific conclusions are rarely reached.

Appendix A contains a brief description of a pilot study made on an English Electric K.D.F.9 computer using an interpretive method and imposing a page grid on the programs. This is a typical procedure for gathering data from programs actually running on a non-paged machine. The investigation was designed simply to provide data for preliminary assessment of the restructuring

techniques of chapter III, but the opportunity was taken to examine more general aspects of behaviour of the few programs tested. Some results are quoted in the appendix and later on in this chapter.

Before considering empirical behaviour in detail, a mention is made of the analytic approach. This attempts to characterise the reference behaviour of a program in terms of a few easily defined patterns - cyclic, sequential, random etc. (refs.4,14). (See also III.2, III.3). However as yet there seems to be no published evidence, that real programs can be successfully modelled in this way - it may be that the parameters of each pattern vary much too rapidly to be amenable to analysis. It is difficult enough to spot gross behaviour patterns without also requiring that they be deduced from more basic assumptions.

## 4. RESTRICTED STORE BEHAVIOUR

One area of study, and an important factor in some, perhaps badly designed, operating systems, is the way programs behave when constrained to run in a constant and limited quantity of store. The usual measure is the average number of page-faults per unit of computation time of the program (or its inverse, the expected processing time interval between page faults). The variables are:

a) the number of page frames of memory allocated (s),

b) the replacement strategy: once the allocated store is full, this is the method of choosing which of the s pages is replaced when an out-of-store page is referenced. In fact although much effort has gone into empirical studies of replacement strategies

(e.g. refs.11,15), results seem to imply they are not such a significant factor as might be expected. Belady (ref. 15) compared some well known strategies with the theoretical optimum: that attained by a best decision based on a knowledge of the whole future reference pattern. His published graphs indicate that there is little consistent and significant difference between any of the better replacement algorithms, which on average generated rather more than twice the theoretical minimum number of page-faults. Even a purely random page replacement decision gave only about three times the minimum. The following sections will assume the 'Least Recently Used' (LRU) strategy - that page is removed which has not been referenced for the longest time. (This is one of the best, although there may be implementation difficulties. It is sometimes said that LRU fares badly on programs displaying a 'cyclic character', i.e. the mere fact that a page has not been accessed for a long period is indication that it will be required again shortly. In practice this consideration can hardly apply with the precision required to affect the strategy adversely, although artificial examples are easily constructed.)

The significant relationship is thus that between the average fault rate (r) and the store allocated (s). r will be defined as in formula 1.2 (section 2).

Form of r/s curve

No knowledge of program behaviour is required to deduce the following:

a) For sufficiently large s (i.e. the total program size), r will be zero (neglecting page-faults arising from initial loading).

b) r will increase (or more strictly not decrease) as s

increases. For supposing the same program is considered under store allocations $s_1$ and $s_2$ where $s_1 > s_2$. Under the LRU strategy, a page-fault in the $s_1$ system represents a reference to a page not in the last $s_1$ referenced - and so obviously not in the last $s_2$ referenced. So the fault is reflected in the $s_2$ system which thus has at least the page-fault rate of $s_1$.

It is intuitively reasonable and experimentally verifiable that b holds almost invariably with any replacement strategy. However, perverse reference strings can be constructed for which it will not be true. Belady et al. (ref. 14) give an example with the 'First-in First-out' (FIFO) algorithm and quote a case of such an anomaly arising in practice with two close values of s for a particular program.

Suppose a program at mean intervals 'c' makes a random reference to one of a set of 'a' pages. If 's' pages are allowed in core ($1 \leqslant s \leqslant a$), the probability of a reference being external is (a-s)/a. The resulting page-fault rate would thus be:

$$r = \frac{1}{c}\left(1 - \frac{s}{a}\right)$$

If c is very small, say comparable with the time of one machine instruction, then for any s less than a, the fault rate will be intolerably large. (E.g. if the time unit - the page fetch time - is 5000c, s=19, a=20, then r=250. This would mean 250 faults for an amount of processing equal to the time of a single page-fetch).

Now the reference behaviour of a program to the whole set of its pages is normally nothing like this; however within any period of a few milliseconds there is usually a nucleus of pages being continually referred to, with frequencies of no more than a few

instructions. Such a nucleus may consist of only the current instruction page and data page but it may be much larger. There is no real practical point, though, in examining how the program behaves in less store - the significant behaviour area must be when at least this amount of store is allocated and the efficiency of the program itself is becoming at least tolerable.

If the remaining pages of the program were being accessed randomly, but with some longer period, there would again be a linear decrease of page-fault rate with increasing store. However in practice, some pages being more favoured than others, the probability of external reference drops more rapidly for lower values of s as these more popular pages become more likely to be accommodated. (This fact of locality of information references forms a central part of the working set notion of Denning (ref.6,16) - see next section). Fig.1-1 shows the general form of the curve. Critical parts of the curve of two examples from the KDF9 study (Appendix A) are given in fig.1-2. These show the area where the average fault-rate, over periods shown on the graph, is between 1 and 10 faults per 10K instructions. Coffman and Varian (ref.11) give examples of this curve for some programs on the IBM 360/50, although much of their data appears to be with very limited store leading to extremely high page-fault rates.

Belady and Kuehner (ref. 17) state to have found that a section of the curve can often be approximated by: $r=b/s^k$ where the value of k for many programs is approximately 2. (In their notation $e=as^k$, where e is the expected interval between faults. Belady in ref.15 states this formula in what appears to be a slightly different connection; the page size M/s is varying and the total memory size M is constant.) This would imply that log r

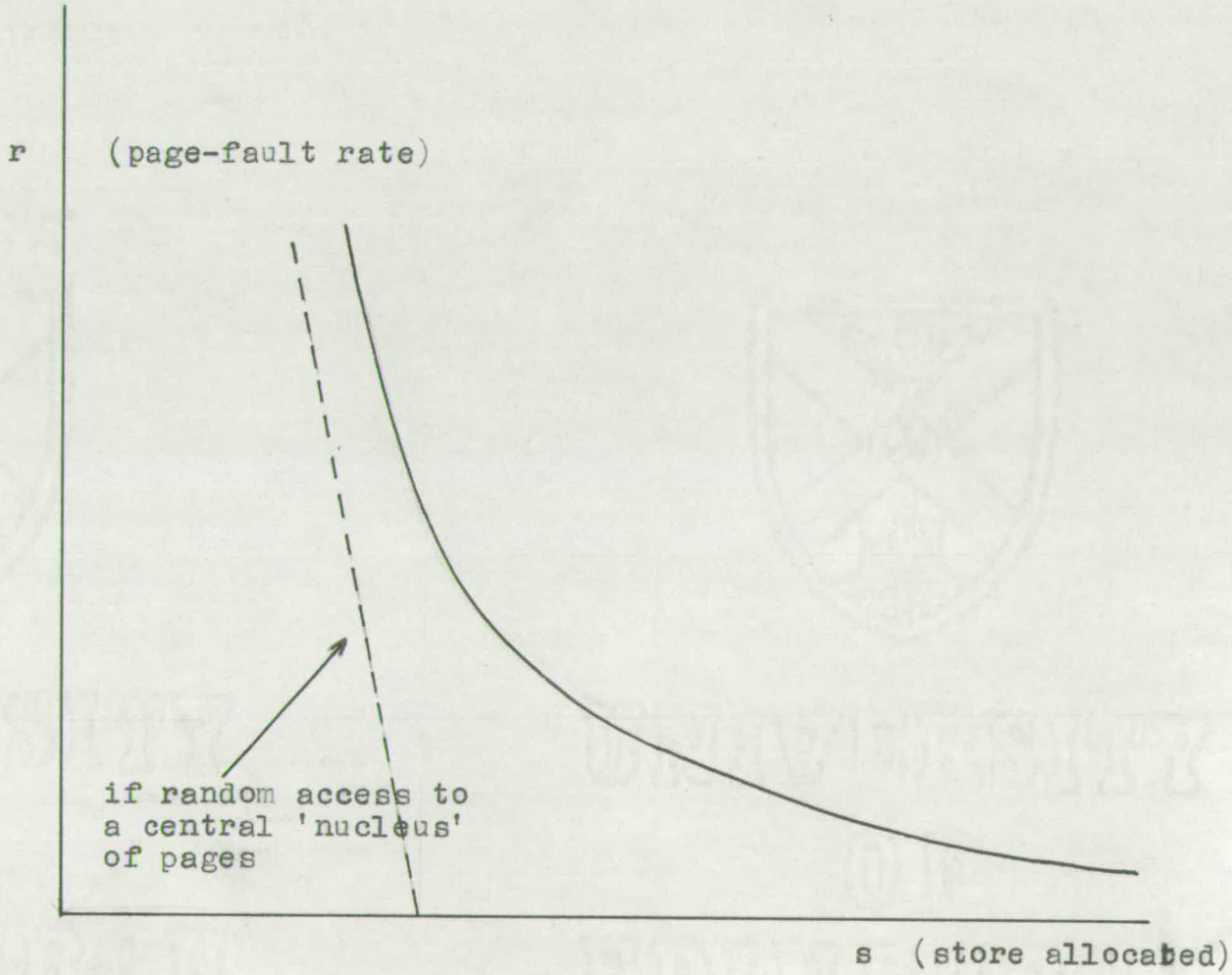
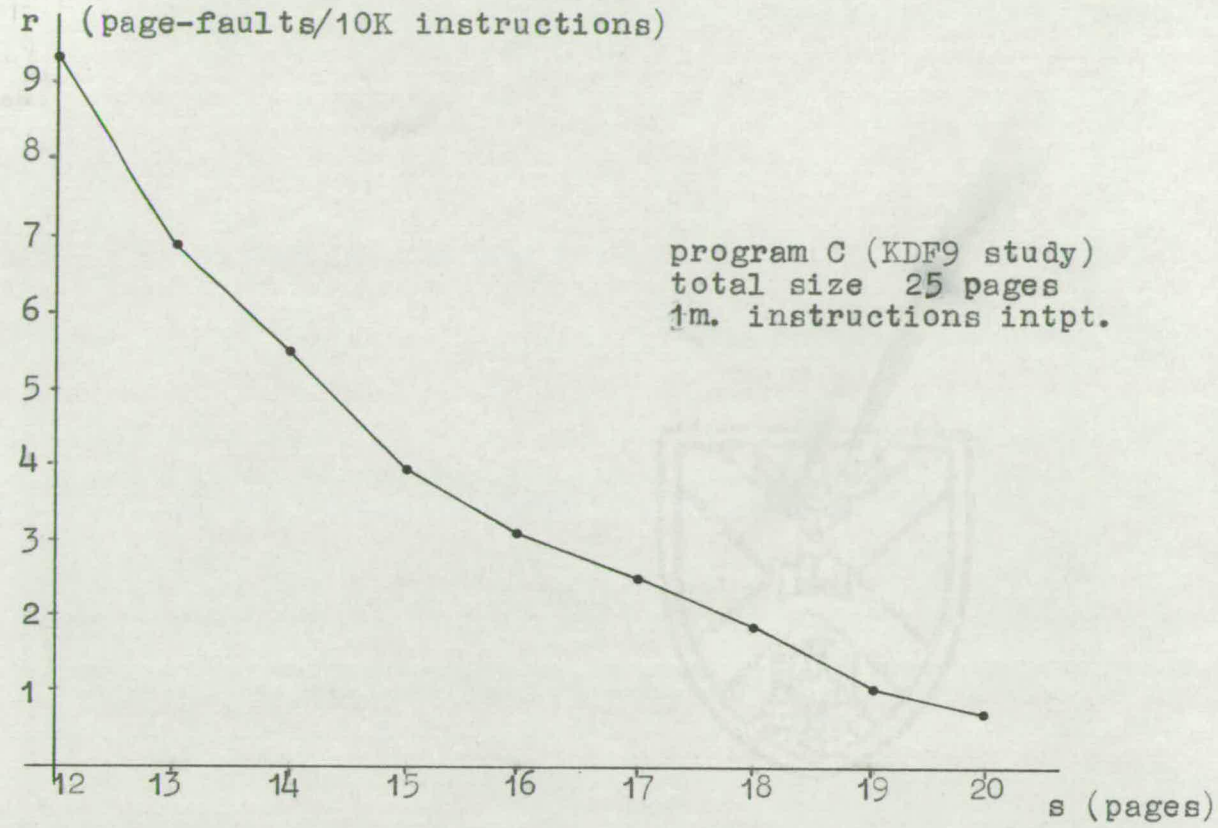



Fig 1-1 Character of restricted store behaviour



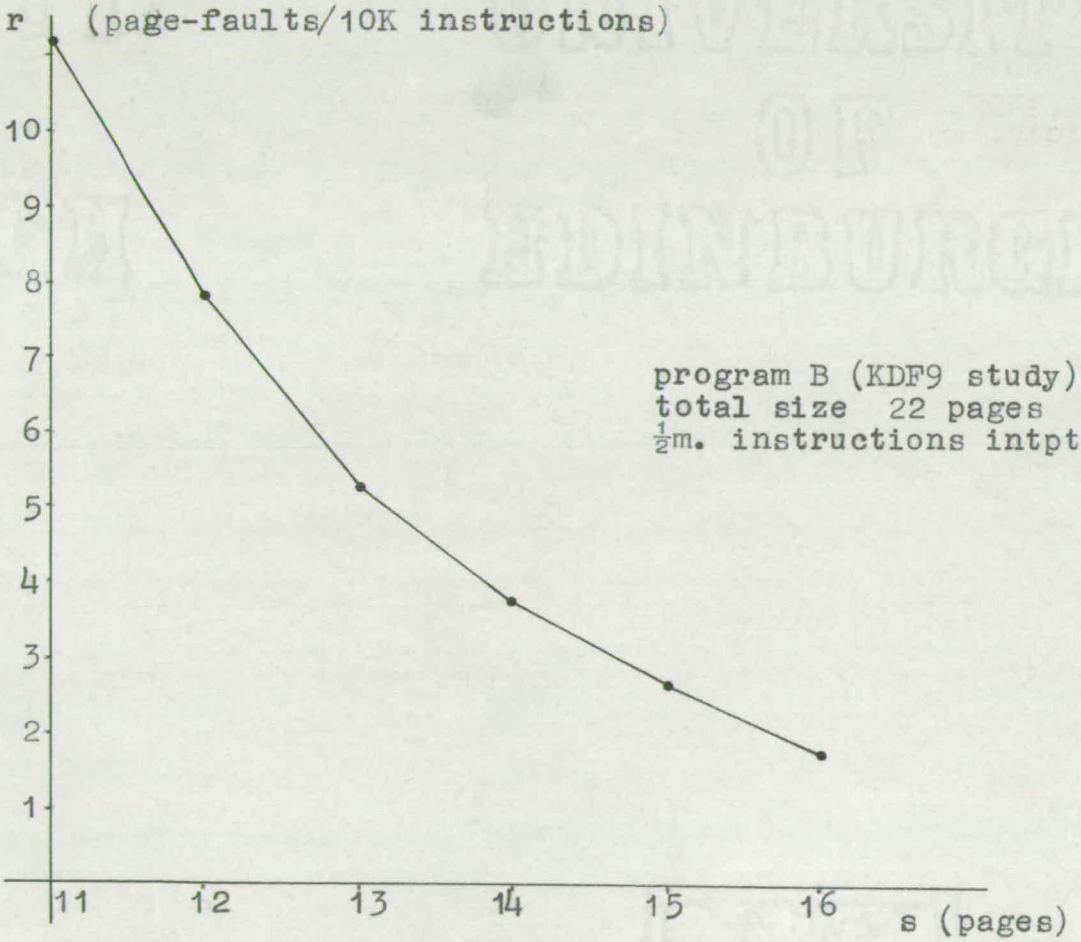
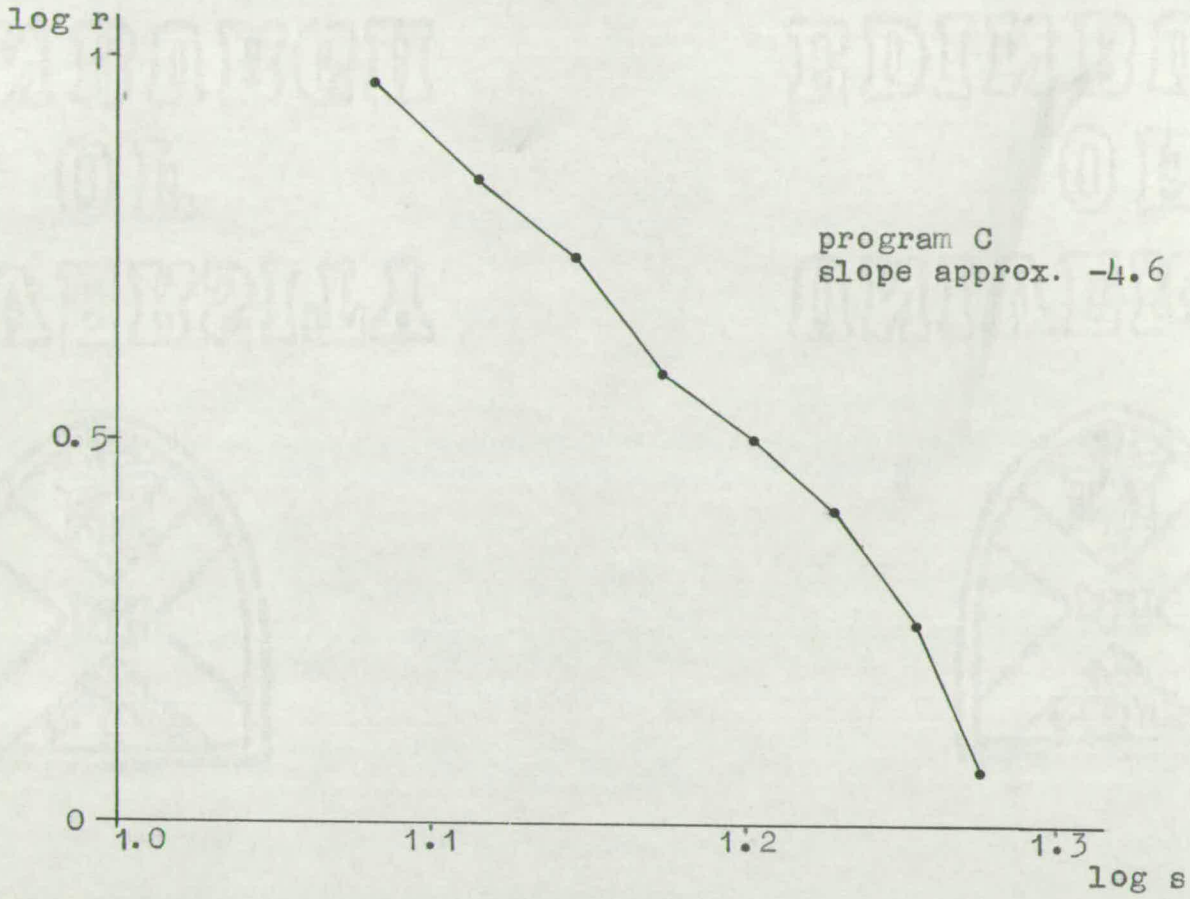




Fig 1-2 Examples of restricted store behaviour



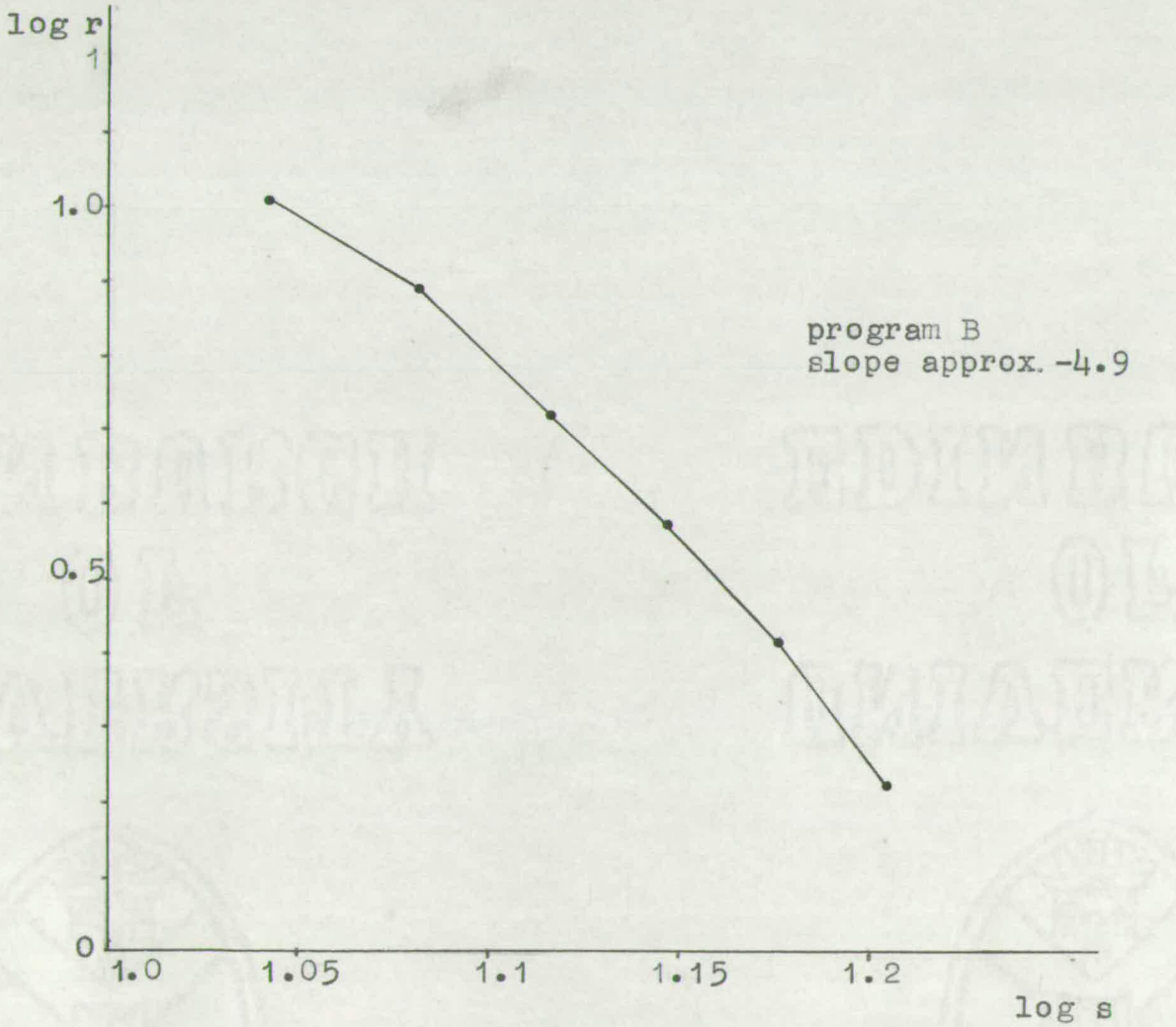




Fig 1-3 Test of hypothesis $r=as^{-k}$ for programs of fig 1-1

plotted against log s were a straight line of slope -k. Fig.1-3 shows this relation plotted for the examples of fig.1-2. A linear approximation with both these programs gives values of k much greater than two - almost five in fact. However it seems likely that the sections of the curves to which ref. 17 refers were in the lower ends of store allocation - the KDF9 curves represent areas of relatively lower fault rates, although probably the important area from the practical scheduling point of view. It is in any case worth examining this formula and its implications briefly, to give insight into its parameters.

Take as before the page fetch time (typically about 5ms) as time unit. Writs $b=s_o^k$ in the above formula, and we have:

$$r = (s_o/s)^k \qquad \text{——(1.3)}$$

Then $s_o$ represents that store allocation with which the program will fault, on average, once during an amount of processing time equal to the page fetch time. With, say, half this allocation, the rate will be $2^k$ faults per unit processing time.

If we substitute for r in formula 1.2 (section 2), we get:

$$\frac{M}{s(1+s_o^k s^{-k})}$$ as an upper bound to efficiency.

This has a maximum where $1+s_o^k(1-k)s^{-k} = 0$

i.e. at: $s = s_o(k-1)^{1/k}$ or $r=1/(k-1)$ (if $k>1$)

The optimal allocation is thus slightly greater than $s_o$ (for any value of $k>2$), and the expected CPU interval between faults will be rather greater than the page fetch time. Note that in both the examples of fig 1-2, to reduce the mean fault rate to, say, 1 in 5K instructions, an allocation of about 3/4 of the total program

size is required. In short time-slices the initial loading becomes a significant part of the total page movement, i.e. the value of r in the formula 1.2 will not be the same as the restricted store rate. We look at this situation at the end of the next section.

The r/s graph (or, if formula 1.3 is considered acceptable, simply the values of $s_0$ and k) is a way of quantitively summing up certain reference characteristics of a program under paging - the problem of section I.3. However it is not considered a very satisfactory description of program behaviour for the following reasons.

a) Limited store behaviour is not a factor relevant to modern operating systems, which do not attempt to compress too many programs into main store but allow the program demands to determine a variable store allocation.

b) The replacement strategy must be specified, and can add its own discontinuous characteristics to the result.

c) The fault rate can very greatly from one interval to another, even during what would be regarded as a single phase of a program. As defined above, r is of course an average, a quantity which may have little meaning in this context.

The next section considers a system independent and more suitable measure of program behaviour.

## 5. THE WORKING SET FUNCTION

The working set W(t,T) of a process at an instant t was defined by Denning (ref.6,16) to be that set of pages referenced in the preceding processing time interval of length T, where T is known as

the working set parameter. The working set size, s(t,T), is the number of pages in W(t,T), i.e. the number of distinct pages referenced during the last T seconds of execution.

Denning suggested that recently used pages will constitute a good prediction of the immediate requirements, and that for efficient running these pages, once loaded should be preserved in core. He proposed an allocation scheme based upon the following:

1) A program is not run unless there is sufficient space in main memory for its working set (for some value of T).

2) Until the process blocks, or the time-slice terminates, a page in W(t,T) must not be removed from memory.

Denning proves (ref.16) the superiority of this, in terms of store utilisation for the same rate of page faults, over certain fixed store strategies.

If $I(t_1,t_2)$ is a processing interval, we can define for a given T, $S^I(T)$ as the average working set size over the interval I, i.e.:

$$S^I(T) = \int_{t_1}^{t_2} s(t,T)dt$$

If I is taken over the whole program run, or a large number of runs if it is data dependent, we shall write S(T); this simply means the average number of pages the program accesses in all possible processing intervals of length T. As a function of T, $S^I(T)$ has the following properties.

1) It is continuous and right and left differentiable. This is perhaps momentarily surprising since s(t,T) is a step function both in t and T (store references occurring at discrete instants).

2) S(0)=0 and S(T) is non-decreasing and concave downwards.

3) The slope at T is the mean rate at which pages outside

W(t,T) are referenced, i.e. the average page-fault rate that would result if the strict working set allocation policy were observed (by average we simply mean total page-faults/total time I).

Fig. 1-4a shows the general character of the curve.

Denning (ref.16) using slightly different definitions proves 2 and a theorem equivalent to 3. We give in Appendix C alternative proofs from first principles; these are interesting in their emphasis on the continuity of S(T).

If I is a period over which the working set size $s(t,T_o)$ is constant, we can state as a corollary to 3: the slope of $S^I(T)$ at $T=T_o$ will give the average fault rate if the program is run in a restricted store under the 'least recently used' replacement strategy. This follows because the contents of the store under this strategy form a working set for some value of T. If the size of this is constant for $T=T_o$, the LRU strategy is equivalent to the strict working set allocation strategy.

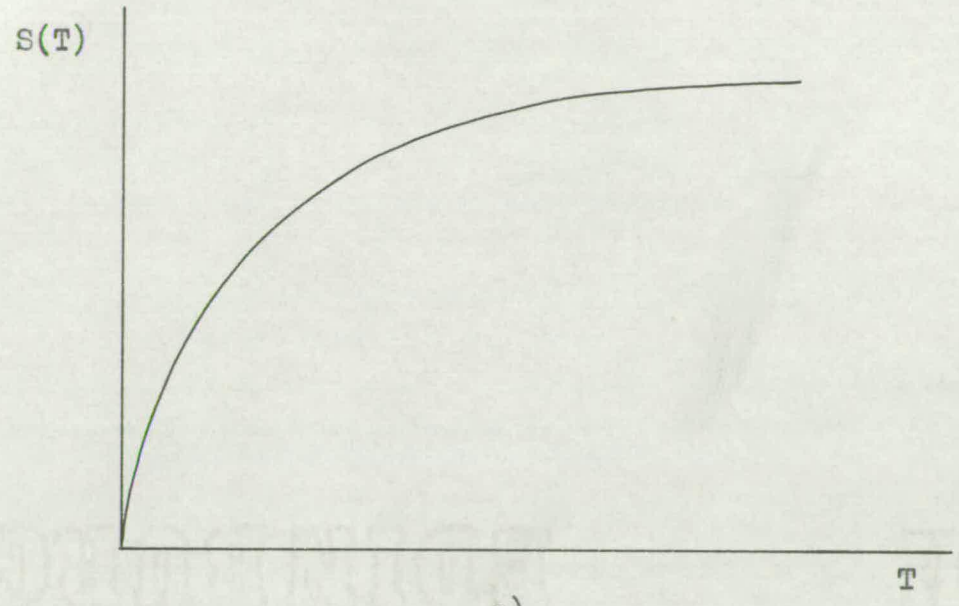
The average working set (WS) curve appears to be a useful description of paging behaviour not only because of its immediate definition but also property 3. Denning's strategy depends on the fact that recently used pages are a good predictor of immediate requirements: just how good s(t,T) is as a predictor is measured by the rate at which new pages are entering it. This is not just of relevance to a system which uses the working set strategy - many systems rely on a certain amount of prepaging at the beginning of a time-slice in order to reduce the amount of demand paging. The assumption is normally that recently used pages - perhaps those of the last time-slice - are likely to be used in the current one. Thus the slope of the working set curve at T has a general

interpretation: the average value as a predictor of the pages accessed in time-slices T. One might say that a really badly behaved program running in time-slices of length $T_1$ would not only access large numbers of its pages in such intervals, giving a large value of $S(T_1)$ but will change its set of accessed pages from one interval to the next, giving a large $dS/dT$ at $T_1$.

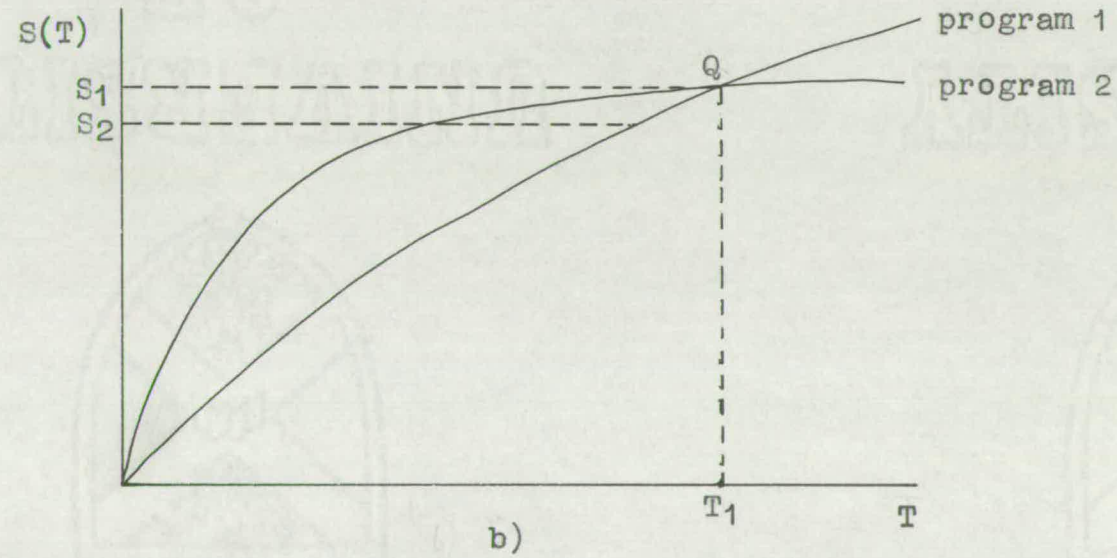
Suppose fig 1-4b represents part of the S curves of two programs. In intervals of length $T_1$ they both access on average the same number of pages $S_1$. Suppose both were being run subject to time-slices of $T_1$. In many systems they would not appear equally well behaved. The steep slope of the curve for program 1 at $Q(S_1,T_1)$ indicates that the pages of one interval are not on average a good predictor for the next, a preloading strategy would lead to more page-faults than with program 2. The slope of the latter is almost flat at $T_1$ indicating this program makes its references to the same set of $S_1$ pages in successive intervals, and these will be successfully preloaded.

If the working set sizes had been fairly constant over the program runs, the effect of running them in restricted store can be seen. In, say, an amount of store $S_2$, program 1 will run at a much higher fault rate even though the store is initially filled up more quickly with program 2. (This can be produced by program 2 making rapid access to a limited number of pages, while program 1 travels slowly but sequentially through a very large number.)

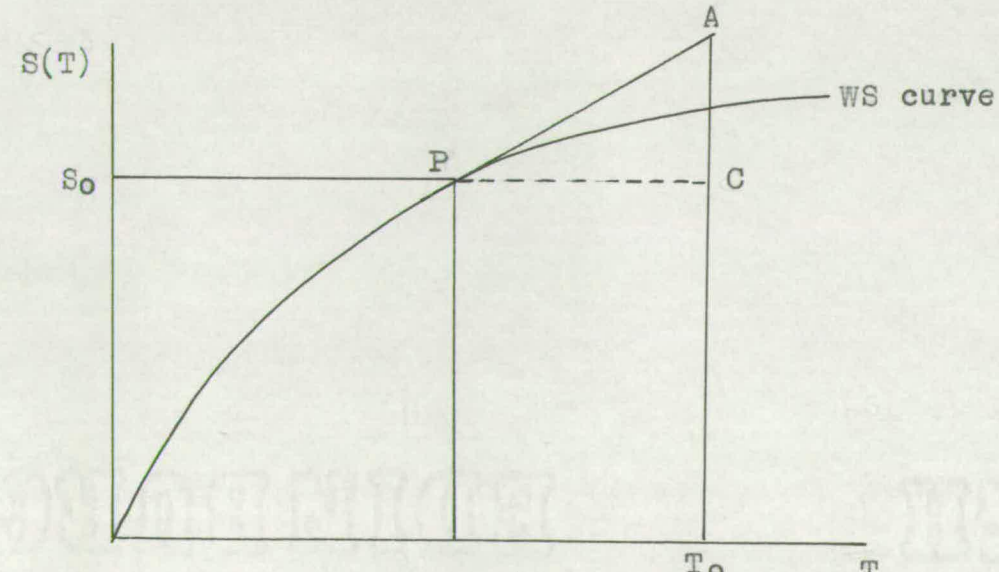
As a description of program behaviour, S(T) suffers from one defect of the restricted store curve; it is an <u>average</u> and cannot reflect slow changes of behaviour over periods long compared to the range of values of T examined. A program might consist of two



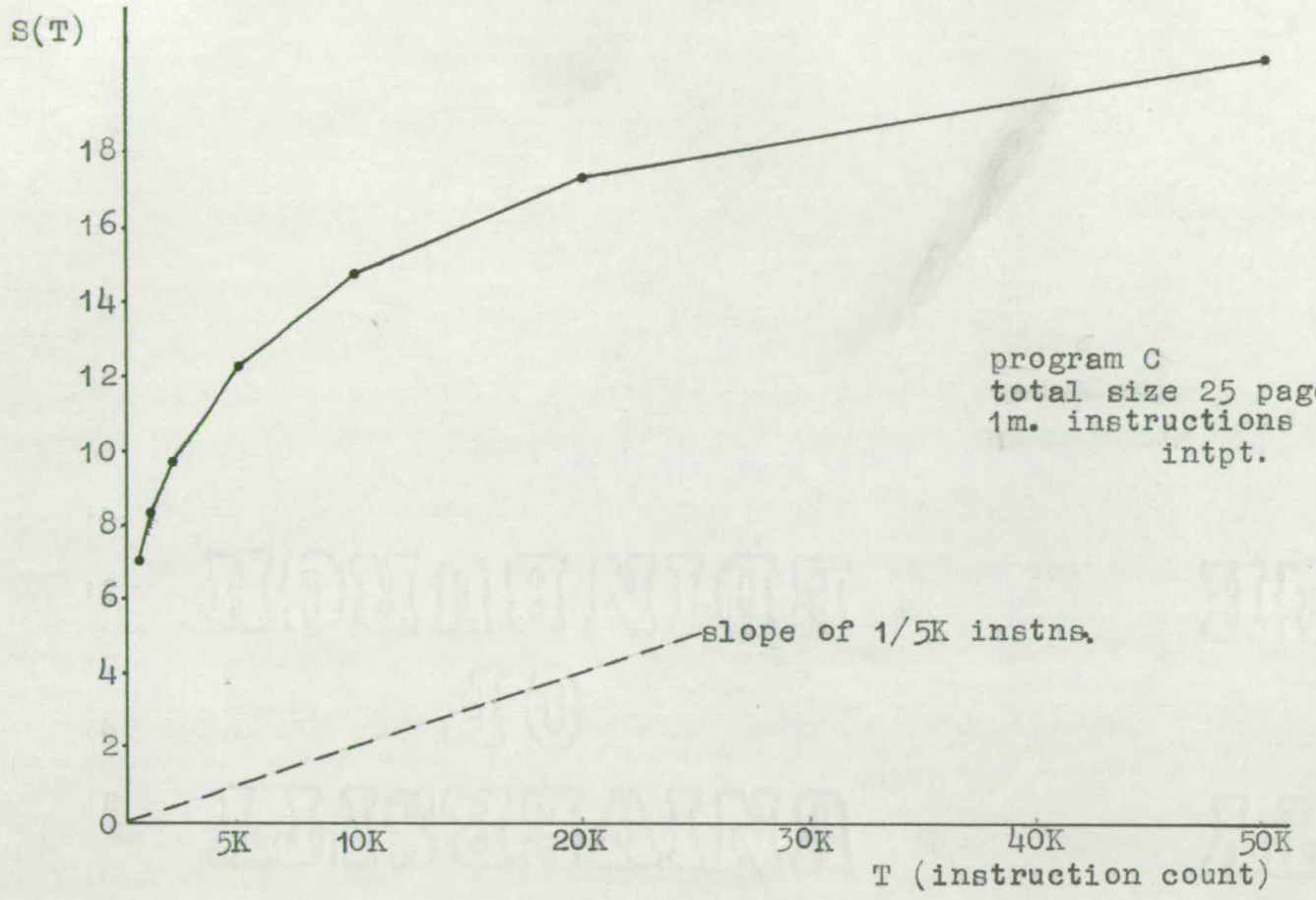

a)



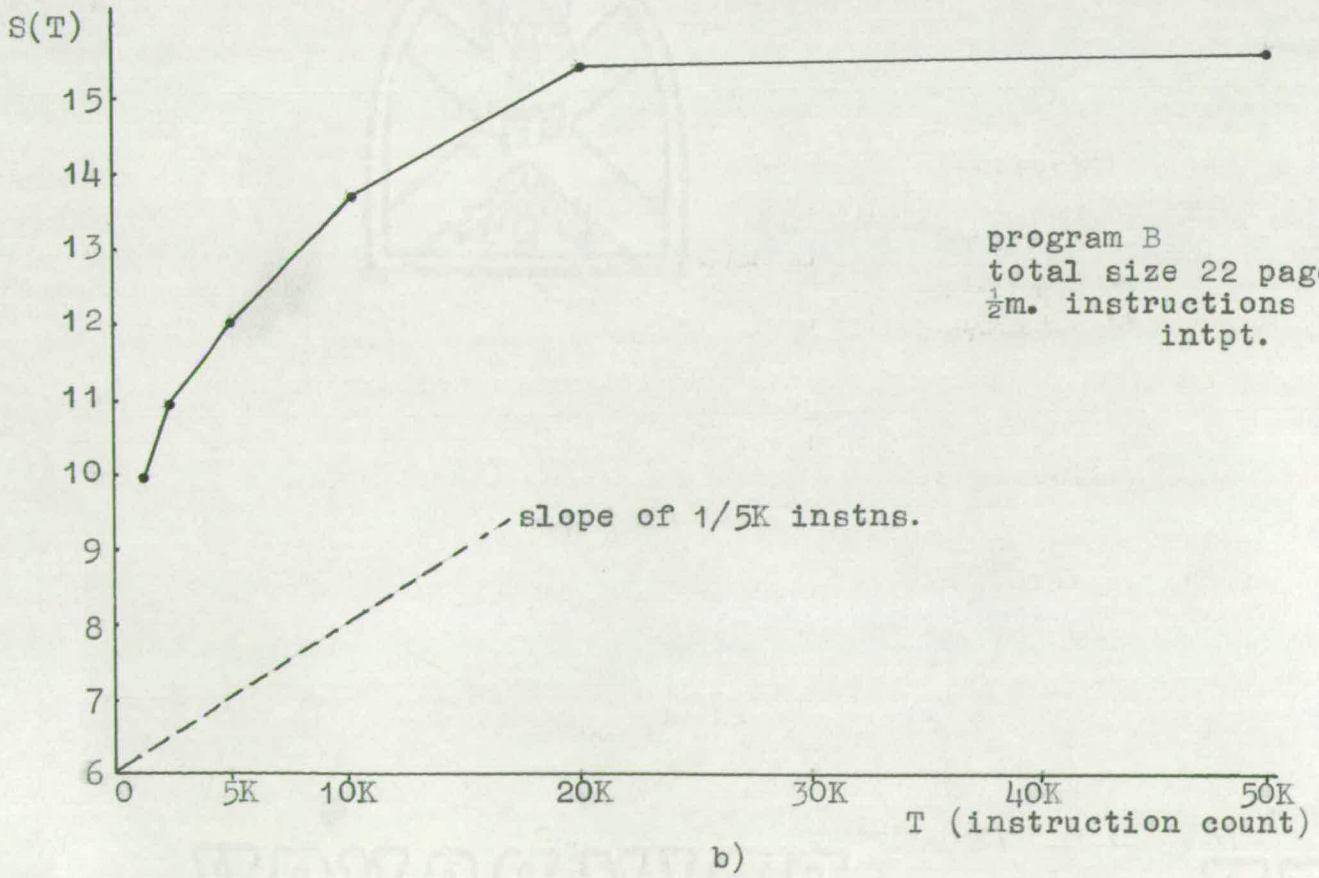

b)




c)




a)




b)

Fig 1-5 Examples of average WS curves

phases in one of which two pages are constantly accessed, and in the other twenty. The average working set size of 11 for intervals small compared to the phase length, is not very enlightening. (However even in this case, the slope will be very small, indicating a generally good predicting behaviour. This is of course true, except at the time when control goes from one phase to the other - it is only this which causes any slope at all.) In such circumstances, the behaviour of the program is best summed up by giving working set curves for, and the duration of, each phase separately.

There is one type of program about which additional information is required to make much use of the WS curve: one that is highly interactive. The time-slice is necessarily determined by the program if the processing interval between console interactions is very short. To deduce anything useful about the program, the order of magnitude of these processing intervals is needed in order that attention may be directed on to the relevant portion of the WS curve. It is of course large highly interactive programs which are the bane of any time-sharing system, paged or otherwise.

Determination of WS curves

In practice it is impossible to measure $s(t,T)$ at every reference instant t, or for every length of interval T. One chooses a sequence of values of T in a relevant region (this may depend on the system - a sufficient range will probably be between half the page fetch time and twice the maximum time-slice to which the program will be subjected), and simply measure the average number of pages accessed in as many intervals of each length as possible (see Appendix A).

Figs. 1-5a, 1-5b show points on the S(T) curve for two programs on the KDF9 (see Appendix A) with simulated page sizes of 500 (48-bit) words. T is measured as instruction count. The periods I consisted of one million and half a million instructions, respectively. The total program sizes given are actually the total extents of reference during the test periods; the declared store requirements in a partitioned environment may well have been more (especially since a run-time stack was involved). Obviously such requirements would have to be known if a comparison with non-paged performance were to be made.

The following comments about 1-5a are given as an example of what is illustrated by the WS curve. Assume a page fetch time p of 5000 instructions.

1) On average, the program accesses half its pages in very short processing intervals, of length just over p.

2) The working set allocation to achieve an average fault rate of 1/p - perhaps considered acceptable - is about 16 pages (envisage 'smoothing out' the curve), not much over half the total program size. This corresponds to a WS parameter of 15K instructions, or 3p. Alternatively one could say that if the time-slices were greater than 3p, preloading the pages referenced during the previous time-slice would be successful to the extent that an interval of at least p would be expected to elapse before a new page is referenced.

3) If the program were highly interactive with processing intervals of, say, 2K instructions, it would be very badly behaved. On average 10 pages are accessed in even such short intervals, and the steep slope at T=2K indicates that it will be of little use trying to reduce demand paging by preloading the referenced pages

of the previous interval.

Additional Notes

We can now continue the calculation of the last section, examining the effect of short time-slices. Assuming the page-fault rate is constant and given by the formula 1.3: $r=(s_o/s)^k$, we have by integration, WS size $S(T)=((k+1)s_o{}^k T)^{1/k+1}$.

Thus the time to access s distinct pages is:

$$\frac{s^{k+1}}{s_o{}^k(k+1)} = \frac{s}{r(k+1)}$$

If a program is wholly demand-paged in a time-slice T, the total number of faults would be:

$$s + (T - \frac{s}{r(k+1)})r$$

This is the first s faults to fill the allocated store, followed by the steady fault-rate r for the remaining part of the time-slice (assumed positive).

The average fault rate as defined in formula 1.2 is then:

$$\frac{s}{T} + r - \frac{s}{(k+1)T} = r + \frac{s}{T}\frac{k}{k+1}$$

and the upper bound to efficiency is thus:

$$\frac{M}{s(1 + \frac{s}{T}(\frac{k}{k+1}) + s_o{}^k s^{-k})}$$

Fig 1-6 shows the value of this function for T=1 and 1/3, $s_o$=20, k=3, M=100, up to the value of s=S(T), the total number of pages referenced during the time-slice, The optimum allocation is seen to be not far below this level, and the efficiency drops

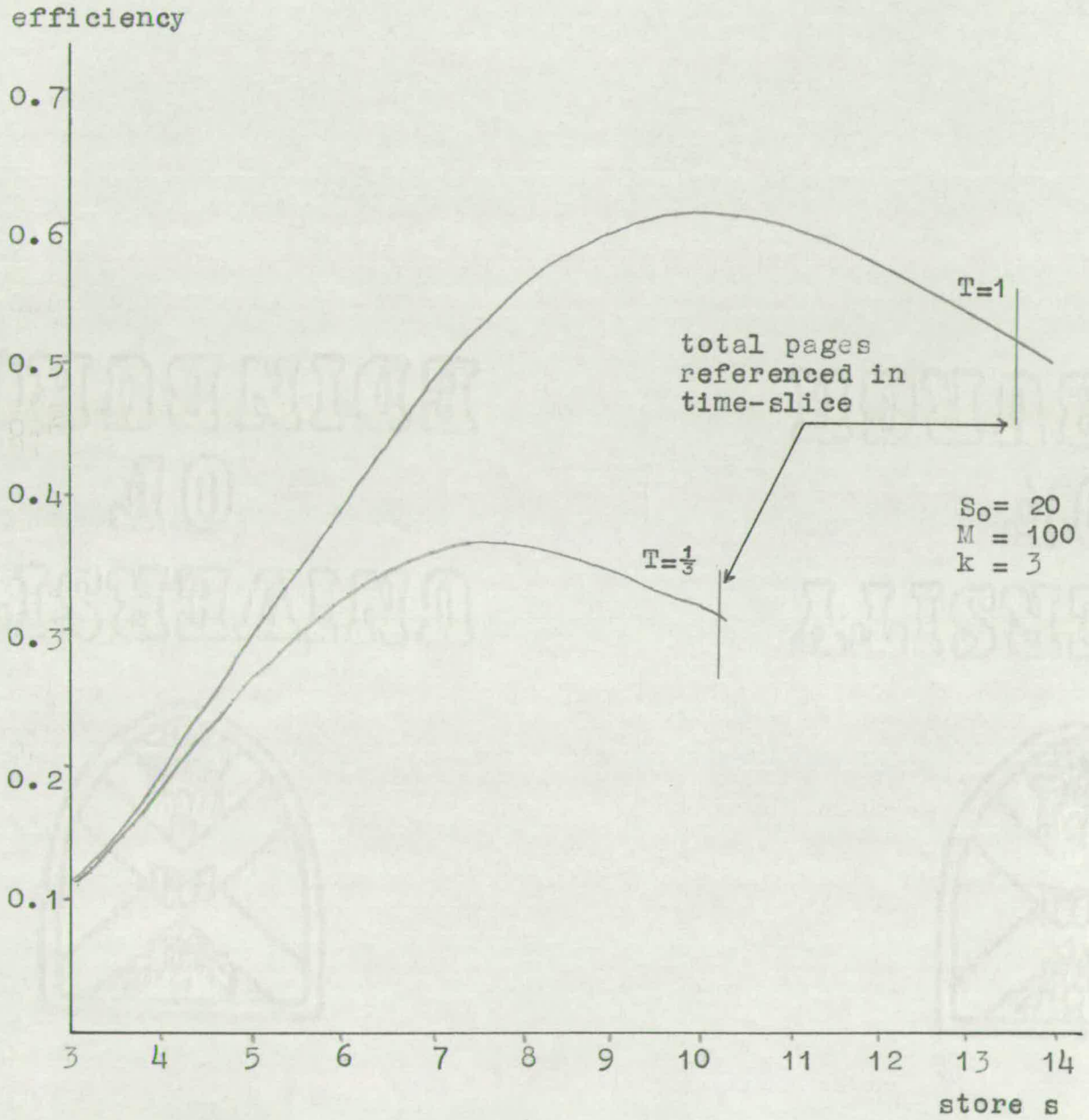



Fig 1-6 Total efficiency in short time-slices

rapidly for values of s less than the optimum. Even at optimum, efficiencies are low for small values of the time-slice, well known to be true in any system. In such circumstances some degree of pre-paging, to cut down the total load time, is desirable if it can be done accurately.

Note that for a program in 'steady state' (i.e. constant working set sizes), fig. 1-4 c gives an interesting method of determining the total number of page-faults if a program is restricted to store $S_o$ and runs for time $T_o$, demand paging from nothing to begin with, as in the above example. The horizontal at $S_o$ intersects the WS curve at P; the tangent here intersects the ordinate at $T_o$ in A. Then $AT_o$ is the total number of faults.

## 6. CONCLUSIONS

This chapter has attempted to identify precisely the advantages inherent in paging and the problems involved in the extraction of a real benefit. It is seen that the vital issue is not the strategy for the choice of page replacement following a page-fault, although this is significant. It is rather the control of these page-faults by the choice of which programs to allow to run and how much store to allocate to each at any moment. The shortsightedness of some early paging allocation strategies is now generally accepted, and paging schemes tend to be designed at least with Denning's general principles in mind.

However many investigations, both empirical and theoretical, of program behaviour have until recently been dominated by early ideas; most effort has gone into replacement strategies and

restricted store behaviour, and few general conclusions have been reached. It is of course difficult to judge the importance of paging overheads (points 2 and 3 in section 1) without much information from real systems. But the reference behaviour of average programs is very important, and it is suggested that studies of mean working set curves for a large range of programs operating in virtual memories would form a good basis for judgment of the potential efficacy of paging. It was not amongst the aims of this thesis to do this: the results from the KDF9 study are presented for example only. The KDF9 is not in any case a paged machine, and mainly owing to the expense of the interpretive method used, the results are limited. It is hoped, however, that they give some insight into relevant aspects of program behaviour under paging.

## II IMPROVEMENT OF PROGRAM BEHAVIOUR

### 1. INTRODUCTION

In a single-line batch system, the only feature of a program's behaviour relevant to its own, or the system's, performance is its total elapsed time: that is its processing time + such I/O time which cannot be overlapped with processing. To optimise a program's behaviour under these conditions simply means to minimise the CPU time required to perform the job, and by suitable buffering to achieve as much I/O in parallel as possible. In a multiprogramming scheme where programs are loaded into variable size partitions, the program size becomes an additional factor, since the total number of jobs which can be accommodated in main store affects the multiprogramming facility. A user demanding a large partition is likely to find his job is delayed to run in slack periods; alternatively (or also) he will incur a greater charge. If a decreased size could be achieved at the expense of greater processing time, a complex optimisation problem may arise. The situation is still further complicated in modern dynamic allocation systems where only parts of the program need be in working store at any moment. The pattern of reference within a program becomes a factor highly relevant to performance, particularly in a time-sharing environment where a continual swapping between main memory and backing store is necessary. Optimization is now a process far removed from simply minimizing the total processing time.

The possibility of designing or reorganising programs to improve their behaviour under paging has been at least suggested since the time when paging studies were first made. Fine et al.

(ref.9) made some general suggestions (although with little conviction) and more recently Kuehner and Randell (ref.18) laid down some intuitively reasonable rules of program design, the foremost advocating not to access a wide variety of pages in rapid succession. Experiments have shown that in certain systems, redesign of algorithms (ref.10), a change in data storage methods (ref.19), or a fairly arbitrary repackaging of parts of a program (ref.20) can give an enormous improvement in program performance.

If there is a potential for such improvement, it is desirable to devise a systematic approach to optimisation in a paging environment. This chapter discusses the fundamental difficulties which unfortunately arise at every level of consideration of this problem.

## 2. THE AIMS OF OPTIMISATION

At a theoretical level, a significant problem is the definition of the ultimate aim of program optimisation. One can distinguish three distinct quantities to each of which some degree of attention might be directed.

a) program performance

b) 'cost' to the system

c) charge incurred

The performance of a non-interactive program must be judged as the total time from the first presentation of the program to the system (i.e. including the queueing time in a batch system), until completion. This measure may also be considered satisfactory in the case of tasks which interact with the programmer, his own response times being deducted from the total. However a

subjective element arises in that machine response times for small amounts of processing are generally considered more significant, time for time, than for lengthy computational work; an appropriate weighting could allow for this.

The 'cost' to the system is a rather less readily defined concept. The physical wear and tear on the hardware is obviously of no significance here; the cost must be regarded as the disruptive effect which the program causes to the system, i.e. the effect on the performance of all the other users. A numerical expression of the system cost of a program P could thus reasonably be defined as the difference between observed mean completion time of other users, and that which would have been observed had P never entered the system. If there are no other users this is zero. In a single stream batch system where jobs are run in the order they are presented, a program causes a cost proportional to its own elapsed time on the machine - subsequent jobs are displaced by this amount. In this case the optimisation of program performance and of system cost would be equivalent processes. However this equivalence need not occur in more complex systems. An idealised example follows and demonstrates this fact; it is hoped this will also give insight into system cost.

Example of system cost

We envisage a simple multiprogrammed batch system with two fixed size partitions; the two programs in store at any moment compete for the use of the single CPU, but they use separate I/O channels (whose store-cycle stealing effect is ignored). The only factors affecting performance at any moment are the ratios of

the processing time to the extra I/O time (i.e. not overlapped by its own processing) for each program.

Denote these by $n_i:m_i$ $(=1-n_i)$, $i=1,2$, for partitions 1 and 2. Each program obtains the CPU for a real time proportional to its demand, thus in a period in which prog.1's CPU time is $n_1$, prog.2's is $n_2$. We also assume that during the I/O period $m_1$ of program 1, there is a period $m_1m_2$ in which neither program can use the CPU. Thus the assumptions of the model are:

Prog. 1 has the CPU $n_1/(1+n_1n_2)$ of total time

Prog. 2 " " " $n_2/(1+n_1n_2)$ " " "

The CPU is idle $m_1m_2/(1+n_1n_2)$ " " " (Note $1+n_1n_2=n_1+n_2+m_1m_2$)

Suppose the average program in the system has characteristics $(n,m)$, and a program P of characteristics $(n_2,m_2)$ enters partition 2. Assume its total processing time is $En_2$ and I/O time $Em_2$, so E would be the elapsed time of P if it were running on its own. We require to calculate the cost of P in an average environment.

The proportion of total time spent processing P when there is an average program in the other partition is found from the formulae above. Since the total processing requirement of P is $En_2$, its elapsed time must be:

$$e = E(1+nn_2) = E+Enn_2 \qquad \text{——(2.1)}$$

Thus half the programs in the system - those in the queue for this partition - have been displaced by this amount.

The programs in the other partition which are running during this elapsed time e would, <u>had an average program been in place of P</u>, have lasted:

$$e(1+n^2)/(1+nn_2) = E(1+n^2)$$

Note their performance has worsened or improved according as the

CPU demand of P ($n_2$) is greater or less than average (n). Thus the queue for partition 1 has been displaced:

$$e-E(1+n^2) \qquad \text{——(2.2)}$$

The total system cost is the average of (2.1) and (2.2), i.e.

$$\tfrac{1}{2}E(1-n^2) + Enn_2 \qquad \text{——(2.3)}$$

compared to a program performance (excluding time in batch queue over which the program has no control here) of the elapsed time 2.1.

Suppose, say, that n is near unity, i.e. the average program has high processing content. Intuitively, then, a program which is I/O bound ($n_2$ small) will have little cost to the system - this is confirmed by (2.3) where $Enn_2$ (= n X program processing time) is the dominant term. The displacement of jobs in its own partition is largely counterbalanced by the extra progress in the other partition to which the CPU is more often available. Thus suppose a program could be redesigned at the expense of considerable I/O time but making a saving of some processing time (for example writing out data to backing store might save having to pack it in a complex way to fit into store). This would increase E but decrease $En_2$. Then (remembering we have assumed n near one), the cost to the system (2.3) may be decreased while the program elapsed time (2.1) is increased.

This example also shows the dependence of both program performance and cost upon the environment, i.e. on the characteristics of all the active programs. A knowledge of the system without an idea of the average load on it may be insufficient for judgement of optimisation criteria.

It should be clear that it may achieve nothing to define a

cost function directly in terms of program utilisation of some key resource, or as the sum of such functions, unless the result can be shown to bear a relation to practical performance measures. The store utilisation - the (elapsed) time-space integral over main memory - of a program is an example of such a function. It may indeed often accurately reflect aspects of cost, but this needs to be shown, otherwise any other than gross manipulations on it may be empty mathematical juggling. (Note that in this connection, the model of section I.2 is relevant. The assumptions there were equivalent to taking the total efficiency of a set of similar programs as bounded above by M/R, where R is the rate of store utilisation per unit processing time of each program. Under the assumptions of this model, then, store utilisation indeed bears a close relation to efficiency and therefore (see below) to system cost. But one should bear in mind the severe limitations, listed previously, of the model.)

System efficiency has been used earlier to mean 'effective CPU rate' (e.c.r.), i.e. the rate at which instructions of user programs are obeyed. This is a reasonable expression of system performance; the cost of a program could therefore be defined in terms of its effect on the total e.c.r. of all the other programs in the system. E.c.r. is not a wholly satisfactory measure however, since delays to I/O bound programs will not be reflected so largely as delays to highly CPU bound processes. Also in the time-sharing situation there is again the complication of response times being considered a key factor in performance. The odd situation arises that it is the less important factor to which attention must be directed in the measurement of program cost. Up to a certain

level of saturation, most time-sharing systems will consider the maintenance of a good response time for highly interactive jobs more important than a high overall efficiency; it is thus the latter whose degradation becomes apparent when a 'high cost' program enters the system.

In a sense the optimal program performance should coincide with minimum system cost; this will ensure that a user's efforts to improve his own program are entirely in the 'right' direction from this system point of view. However this point becomes almost irrelevant in the 'ideal' system which minimizes system cost by reflecting it back into the performance of the program. A program which would normally be considered 'badly behaved' is given lower priorities on system queues and will have control less often (equivalent in a batch system to rescheduling a long job to run in a slack period). The performance of a badly behaved program suffers, but other users are shielded from its effects. The total system performance is only degraded in as much as it includes the performance of the errant program.

The third target suggested for possible optimisation was the charge made to the user. It seems intuitively obvious that the variable part of such a charge should be related to the system cost and nothing else. This means that a job run at a quiet period - overnight, say - will be charged less than the same job at a time of heavy demand. It also means that if the system genuinely succeeds in reducing the system cost of a 'bad' program, resulting in poor performance of the latter, there is no need to make the additional penalty of a high charge. Charging algorithms in general for multiprogrammed systems tend to be fairly arbitrary

and sometimes utterly wrong, but the matter will not be pursued here.

This discussion, although probably raising more questions than it has answered, has tried to illuminate the problems arising from what is meant by 'optimisation of performance'. We have seen that in the ideal case program performance is the significant variable; the apparent conflict between optimisation of this and of system cost, disappears. Realisation of this ideal situation is a task for the system designer.

## 3. EXPLICIT MINIMISATION FUNCTION

At a practical level, the first task in a systematic optimisation is the expression of some approximation to performance as a function of program definable characteristics over which there is some degree of control. In a single-line batch system the function is of course the total CPU time + non-overlapped I/O time; in the fixed partitioned multiprogrammed case modelled in the last section, a function giving more weight to the CPU time would be necessary. However the problem is more difficult in a dynamic allocation system. It is not enough to define system characteristics (e.g. store utilisation, number of page faults) which themselves depend on the program behaviour; we require a function controllable and measurable at the level of program.

The minimisation scheme described later in this thesis takes as its function the average working set size, S(T) (see section I.5), the processing time being unaltered. Any value of T may be chosen - the most suitable choice may vary from program to program as well as depend on the system strategy. In a system using

Denning's working set allocation scheme (ref 6), S(T) is closely related to performance. For this system attempts to reduce page-fault rates to an acceptable level by ensuring that any program in core has a store allocation equal to its current working set size: programs are only allowed in core when such space is available (note that if there is a priority rule favouring small programs, system costs are equalised). If the strategy works, total system performance depends mainly on the number of programs that can simultaneously exist in core, i.e. the multiprogramming facility. The average space allocated to a program is the mean working set size and clearly is a key feature.

However it is suggested that under any adaptive allocation strategy, a highly beneficial effect will result from a significant reduction of the above function for a suitable value of T. This simply makes explicit the general precept that one should attempt to reduce the number of pages referenced in short intervals (ref.18).

## 4. METHODS OF OPTIMISATION

Techniques of optimising page reference behaviour can be divided roughly into two classes; we call these 'design level' and 'coding level' methods.

Almost certainly, the largest improvements in program behaviour can come from an intelligent design (particularly of data layouts), aimed at reducing the spread of references over short periods. This has been impressively demonstrated by Brawn and Gustavson (ref.10)who reprogrammed three problems (matrix inversion, data correlation, sorting), involving large-scale data reference on the M44/44X experimental system. Under a FIFO page replacement strategy, the amount of store required for efficient performance

was reduced very substantially: by a factor of six in one case. These were obviously particular (and initially very badly behaved) programs; a more general approach is that of McKellar and Coffman (ref.19) who examine the paging rate (under a fixed store strategy) that results from matrix operations under different methods of matrix storage.

However the redesign of programs to improve paging behaviour is necessarily an imprecise procedure and impossible to examine in a general way. A definitive approach to optimisation must confine itself to 'coding level'; by this is meant either the choice of page boundaries across existing code and data, or a complete repackaging of parts of the program among pages. There has been some work dealing with methods of locating reasonable segment boundaries (refs. 21,22,23) and more recently such studies have suggested paging systems as an area of possible application (refs. 24,25). The only work to which the author has discovered reference relating to repackaging solely for the improvement of paging behaviour has described the experiments of Comeau (ref.20). A brief and pessimistic summary of the immediately apparent problems appears in ref. 18 (p1013).

The great advantage of optimisation at this level is its amenity to automation; the possibility exists of repackaging being performed by a compiler, perhaps using the results of automatic monitoring software. Whether the whole process is worthwhile must be judged by the same criteria as with any optimising compiler. The degree of improvement obtained must be balanced against the time and effort in producing it. The results obtained from the scheme described later suggest that optimisation would be well

justified on the large, often used system programs (compilers, etc.) which are often a principal cause of poor system performance.

Theoretical aspects of program restructuring are examined in the next chapter.

## III FORMAL RESTRUCTURING OF PROGRAMS

### 1. INTRODUCTION

For restructuring purposes a program is envisaged as being partitioned into a collection of blocks - henceforth termed chunks. This partitioning may extend over the whole program, or simply over part of it: perhaps just the instructions and not the data. The assumption implicit throughout this chapter is that the chunks can be rearranged in any order and can thus be packed into pages in any desired manner. Such a rearrangement may require alterations or additions to be made to the chunks - for instance extra instructions could be necessary to maintain the correct flow of control. It is assumed that any changes in the dynamic pattern of reference to the chunks (with respect to process run time) thus caused can be regarded as insignificant. The sizes of the chunks are also assumed to be given and to include any increases that the above alterations may cause.

A simple but coarse partitioning would be such that each chunk was an entire loadable module; relocation is then an easy matter. At a very fine level, one might take the small instruction sequences between one branch instruction and the next, or individual data items. Normally as the chunk sizes become smaller, the difficulties associated with their quantity increase but the potential for the improvement of program behaviour also increases considerably. The few actual examples given in this chapter are for clarification and necessarily deal with a very few chunks, but it must be remembered that in practice there may be many hundreds. Practical difficulties concerning choice of chunks, repackaging etc. can be

very considerable but all such details are left to the next chapter.

The following two sections comment on some theoretical work which has already been done in areas based on the above aspect, and which has possible relevance in the paging situation.

## 2. THE STATIC GRAPH

The simplest representation of program structure is as a directed graph whose nodes are the program chunks. Suppose the chunks are numbered from 1 to n. We make a directed link from chunk i to chunk j if:

a) An instruction in i can be directly followed by an instruction in j (this covers not only jump instructions but a 'drop through' from i to j).

b) An instruction in i can reference data in j.

c) An instruction in j can reference data in i.

Thus a data reference is represented by a directed link in each direction (alternatively this can be regarded as a non-directed link).

The resulting graph is equivalent to a (n x n) matrix S with $s_{ij} = 1$ if there is a link from chunk i to j, $s_{ij} = 0$ otherwise. S is known as a Boolean connectivity matrix.

This is a construct based purely on the static structure of the program and could be easily generated at compile time. The mere presence of a reference or branch from chunk i to j justifies a link; the frequency with which this path is taken, or whether it is taken at all at run time, is not relevant.

Graph theoretical techniques can then be applied to yield certain information about the program structure (refs. 21,22).

Chunk j is said to be reachable from i if there exists a set of links forming a directed path from i to j. Ramamoorthy (ref.21) has shown how to use S to form the reachability matrix R, ($r_{ij} \neq 0$ if and only if j is reachable from i). This shows up redundant chunks (which can never be entered) and blind alley errors (sets of chunks which once entered can never be left).

Consider the relation between nodes i and j defined by: 'i is reachable from j and j is reachable from i ($r_{ij} = r_{ji} = 1$)'. This is easily shown to be an equivalence relation; the nodes of the graph can therefore be partitioned into disjoint equivalence classes. Those consisting of more than one element are known in graph theory as maximal strongly connected subgraphs. In programming terms, such a subset represents a set of chunks which cannot be revisited once control has passed out of it; the partition is therefore a reasonable initial segmentation of the program. Ramamoorthy describes how to identify these subsets using the reachability matrix.

Although interesting and elegant, the graph-theoretical approach is probably of very limited application. The MSC subgraphs will often be very large - perhaps the whole program excluding the initialisation section (this will be true of any program with a central main loop). They are in any case likely to represent phases which are quite clear cut at the design level and form natural and obvious elements of segmentation without recourse to formal techniques. More significantly very little can possibly be deduced about the run-time flow of control from this static model of program structure.

## 3. THE DYNAMIC GRAPH

The above model can be simply extended to store some information about dynamic behaviour. Interchunk links are given a numerical weight proportional to the number of times the link is traversed when the program is run, i.e. define:

$s_{ij} \propto$ the number of times control is transferred from chunk i to chunk j (or reference made to data chunk j from i).

The actual number of times links are traversed in particular runs is not important here; only the ratios of the $s_{ij}$ over long periods. We define $s_{ii}$ to be zero for each i.

Two particular subsets can be identified: 'entry chunks', those which can be directly entered from the external world, and 'exit chunks' from which control can pass directly out. It is convenient to represent the external environment by an additional chunk. This is linked by appropriate weights to all entry and exit chunks, and the system is now closed. Assuming control begins in the external environment and eventually ends there, the sum of the weights on links entering any chunk is equal to the sum of the exit weights, i.e. for all i:

$$\sum_{j=1}^{N} s_{ji} = \sum_{j=1}^{N} s_{ij} = n_i \text{ (say)}$$ where N is the number of chunks.

The $n_i$ are proportional to the total number of times each chunk is entered.

$$\text{Put } p_{ij} = \frac{s_{ij}}{n_i} \qquad \text{all } i,j$$

Then $p_{ij}$ is the proportion of the total number of exits from i which go to j. Given that control is in chunk i and no other information, $p_{ij}$ may be regarded as the probability of j being the

next chunk.

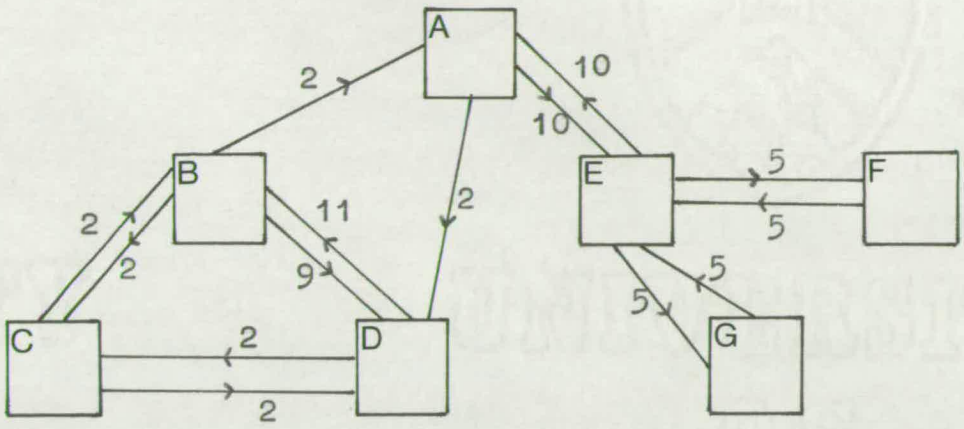
Then $\sum_{i=1}^{N} n_i p_{ij} = \sum_{i=1}^{N} s_{ij} = n_j$ for each j

i.e. $\underline{n}(n_i)$ is a left eigenvector of the matrix $P(p_{ij})$.

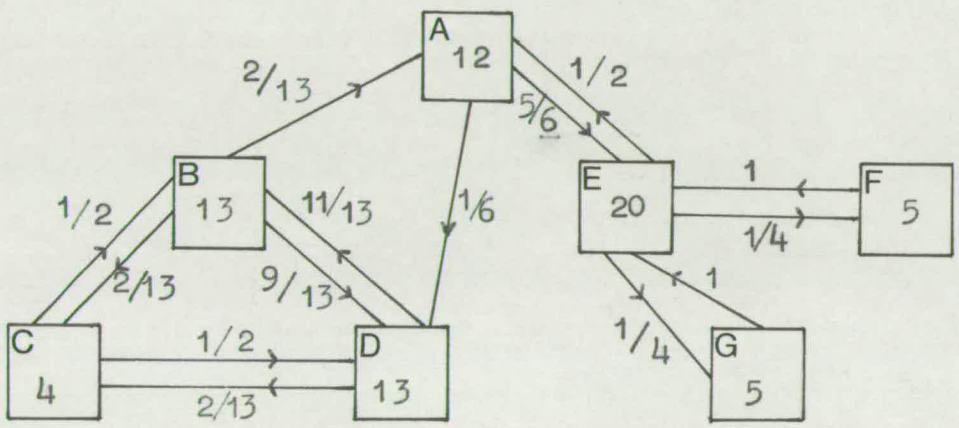
This fact could be of practical use in the construction of the S matrix of a program. If data is gathered by direct monitoring at process run-time, the $s_{ij}$ and $n_i$ would be the easiest quantities to measure. However if an approximation were being sought by estimation, especially in a large program, the $p_{ij}$ would be easier to guess. These are more local quantities than the $s_{ij}$, to which the constant of proportionality adds a global effect. From the estimated 'probability' matrix P, the $n_i$ can be calculated (using customary matrix methods for the determination of eigenvectors), and hence S.

Fig. 3-1a gives an example of a graph with possible weights ($s_{ij}$), and fig. 3-1b the corresponding graph with $p_{ij}$ and $n_i$. (Any multiple of the $s_{ij}$ and $n_i$ gives equivalent graphs).

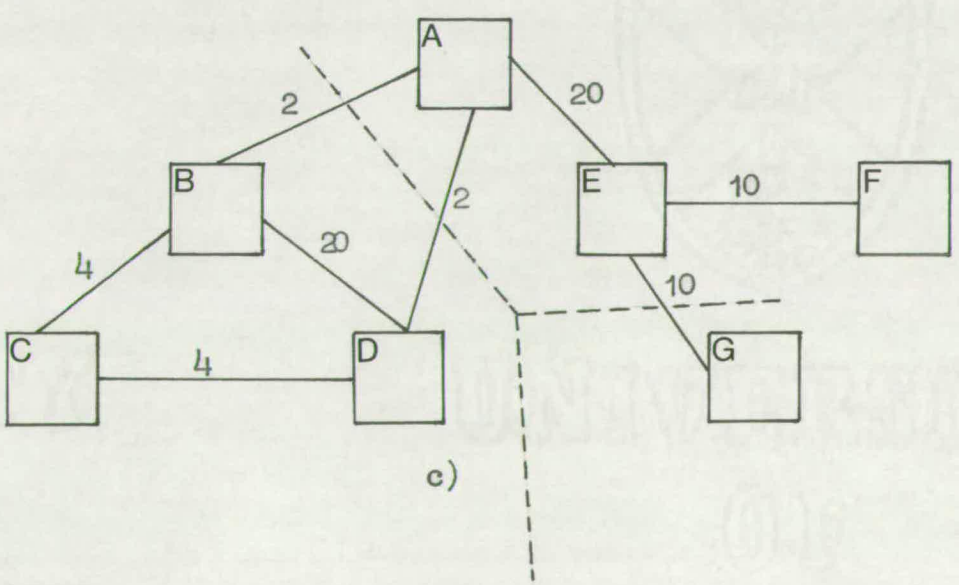
It should be noted that the above relationship between P and the use frequency vector $\underline{n}$ is a purely arithmetical consequence of the definitions; the derivation of $p_{ij}$ is not supposed to imply that at any moment the next chunk to be entered depends solely, or even at all, on the current chunk (although it is true that the derivation is pointless if the dependence is slight). Thus in the example of fig.3-1, it is quite consistent with the given information that C be a subroutine called from either B or D and returning thereto. The successor to C is determined entirely by its predecessor - given that the chunk sequence has been BC, the probability is 1 that the next is B again. The fact that $p_{CB} = \frac{1}{2}$




a)




b)




c)

Fig 3-1 Dynamic Graphs

simply means that over a long period, one half the exits from C go to B (because in this case one half the entries come from B).

Approaches to the dynamic graph of a program (refs.21,23,26) have invariably used a Markov model (ref. 27), taking the chunks to represent the states of a homogeneous Markov chain. This makes the assumption that the probability of the next reference being to chunk j, say, depends only on the current chunk i, and is independent of time or the previous chunk history. All the results from the theory of Markov processes can then be used (in particular the limiting state probability vector $\underline{\pi}$, if it exists, satisfies $\underline{\pi}'P=\underline{\pi}'$ where P is the transition probability matrix). However it seems unlikely that any particular program can be validly modelled in this way; the presence of procedures, loops etc. immediately add a highly 'non-Markovian' element. This is not to say that such a model cannot be useful in examining broad aspects of program behaviour or in simulation experiments, but it is probably of little use in practical restructuring problems.

The segment problem

A repackaging problem which has been considered (refs. 21,23) is what is termed here the 'segment' problem. This has possible use in the paging situation, and also demonstrates an application of the $s_{ij}$ graph (uncluttered with notions of probability); we therefore examine it briefly.

It is required to group the chunks into segments of some maximum size L, so that the number of inter-segment jumps during a long period is minimized. The obvious application is to divide up a large program to run in a smaller machine with a minimum number of complete overlays.

Write $c_{ij} = s_{ij} + s_{ji}$. This is proportional to the number of times control passes between chunks i and j. If A is a subset of the chunks, define:

external connectivity of A = $\sum_{i \in A, j \notin A} c_{ij}$

This is proportional to the number of times that the boundary of A is crossed; so the segment problem is to partition the chunks so that the sum of the external connectivities of each subset is minimised. (It is trivial that this is equivalent to maximising the sum of the internal connectivities. $\sum_{i,j \in A} c_{ij}$)

Note that the intersegment jumps are only minimised over many program runs. Over any particular run, and certainly over any part of one, it need not be true unless the program exhibits the stationary behaviour that the Markov model assumes.

Fig. 3-1c shows the connectivities $c_{ij}$ for the previous example. If a maximum of three chunks per segment were allowed, the dotted lines show the best partitioning (F and G may be interchanged). The sum of the weights crossing boundaries is 14. The sum of the external connectivities is 4+14+10=28, each link being counted twice.

The problem is then quite precisely defined. Given that the original $s_{ij}$ are accurately proportional to inter-chunk transfer frequencies over a long period, the sum of the external connectivities of a set of segments is in the same proportion to twice the inter-segment transfer frequency. Nothing else about the program is required or need be assumed.

Unfortunately in the general case the discovery of the optimum grouping is extremely difficult. To enumerate all the valid partitions and work out the result for each one would be quite

out of the question for any but a very few chunks. We have a problem of at least the order of the travelling salesman problem (ref. 28) with the additional difficulty that no dynamic programming techniques to reduce the amount of computation without immense storage problems, suggest themselves. (It is not obvious how to avoid the repeated computation of the external connectivity of any particular subset). Any practical solution must involve a heuristic technique, hopefully to lead to a near optimal result. We do not discuss this here (ref. 21 suggests a method), although the method of section 7 dealing with a slightly related problem would probably be effective. Two points are worth making for the unwary, though.

a) In fig.3-1c, the chunk in a segment by itself in the best grouping, i.e. G, is not the <u>least used</u> chunk which is C.

b) The best partitioning has not necessarily the least number of segments. If, in fig.3-2, three chunks can be packed to a page, the three segment packing is obviously far better than any into just two segments.

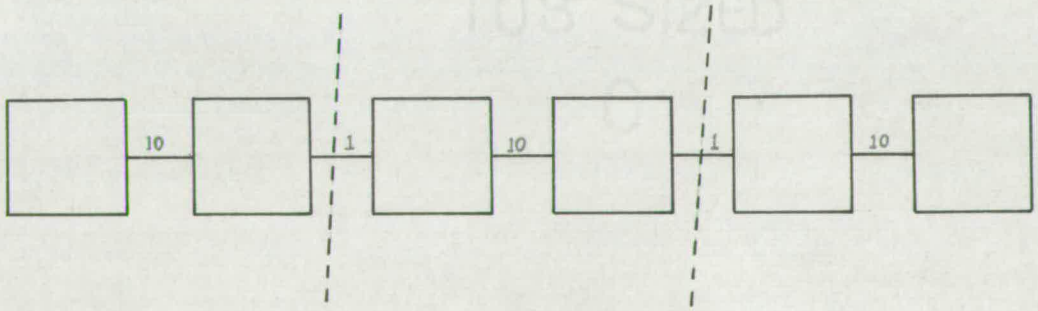



Fig.3-2

## 4. RELEVANCE TO THE PAGING PROBLEM

First thoughts on the question of the improvement of program behaviour under paging might suggest an approach like that above: suppose we restructure into pages so as to minimise the inter-page transfers, would this have a highly beneficial effect?

Intuitively one would expect an improvement: control would remain in any one page for longer periods, and rarely used chunks would tend to be removed from the main paths, reducing the effective size of the program. However it is suggested that an emphasis on crossing boundaries is completely misplaced in the paging situation.

For instance, a tight loop over a page boundary, lasting say ½ms. and involving fifty jumps is likely to be no more significant than a single branch. Once both pages are in store, and no reasonable allocation strategy would involve the removal of either within ½ms. of being used (unless the end of a time-slice intervened), no further cost is incurred by further transfers from one to the other. A single reference to each of several pages during a short period will normally represent far worse behaviour than repeated boundary crossings of two or three pages. This means that the ability to choose where a page boundary should lie across the code of a program may not be very useful. One could of course use it to separate two adjacent but dynamically widely separate sections (perhaps a subroutine from its casually following code), but the superficially attractive idea of going to great lengths to avoid loops over page boundaries has possibly little validity.

An extreme example of the lack of importance of direct connectivity in the paging context is shown in fig.3-3. This

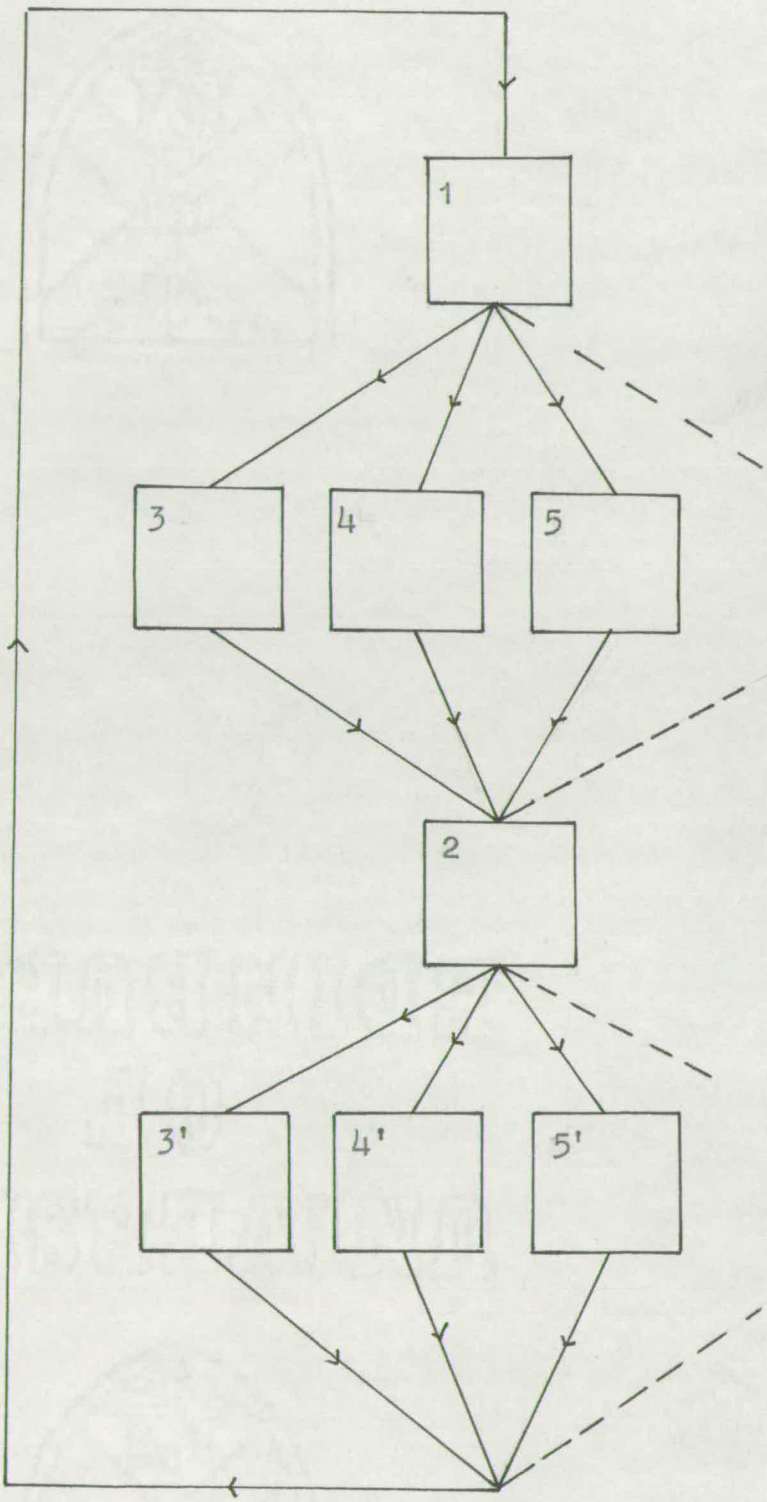



Fig 3-3 Irrelevance of Direct Connectivity

shows chunks which can be packed two to a page. Control passes from chunk 1 to a random one of 3,4,5,..., then to 2 which switches to the corresponding one of 3',4',5',... and then back to 1 again. A possible sequence is thus 1525'1323'1.... Suppose the processing time interval between two entries to 1 is fairly short, compared to the average time-slice. Then the two chunks 1 and 2 are always going to be referenced, and within short periods of each other; they are obviously best packed together. Chunk 3 may rarely be referenced, but when it is (and only then) a call of 3' follows; 33' should be packed together, similarly 44' etc. Thus the most effective packing, in the majority of paging environments at least, would give a zero external connectivity to each page and would be the worst solution to the segment problem.

Generally in the paging situation one is interested in program behaviour over periods which, although only measured in milli-seconds, still contain hundreds of references to perhaps many pages. In these circumstances the connectivity matrix contains information on too fine a scale to be of much use in restructuring. However it does form a fairly compact way of expressing elements of program structure. It is obviously tempting to redefine the coefficients as a form of 'dynamic distance'; thus 3 and 3' in the above example will have a connection since they are always called within a short time of each other. We shall see how consideration of a precise paging problem leads to just this approach.

## 5. THE WORKING SET PROBLEM

As stated in II.4, the paging function which is our target for minimisation is the average working-set size for some given time interval. We seek a packing of the chunks into pages which

minimises the mean number of pages referenced in intervals of the given length.

It should be noted that this function is additive over various parts of the program; if it is only feasible to restructure part (e.g. the instructions and not the data), this can be treated on its own, the contribution of the rest to the total working-set size not being relevant. In practice this property is essential: large programs will often consist of many modules only linked together at run-time and containing library and system routines; it would not be practicable to demand that the whole program be restructured at once (unless chunking were done at the gross modular level). For example, a scheme which sought to minimise page-faulting when a program was run in a given n pages of store would be of academic interest; one could do nothing without information about the total program.

Intuition (and the continuity of S(T) ) suggests that a good restructuring for an interval length $T_1$ will be good for any T in the neighbourhood of $T_1$. However fig.3-4 shows a possible situation for two widely different values $T_1$ and $T_2$. Any restructuring is likely to make a large improvement throughout the range, but that for $T_1$ is a long way from optimal for $T_2$ and vice versa. Two simple examples (allied to those in the last section) are given to demonstrate the importance of the time-scale.

Example 1

Fig 3-6 shows chunks which may be packed up to three to a page. Control resides in A and B for a time $T_1$, then C and D for the same period, then E and F, and then the cycle begins again. If $T_1$ is large compared to the WS parameter T, the optimal grouping is

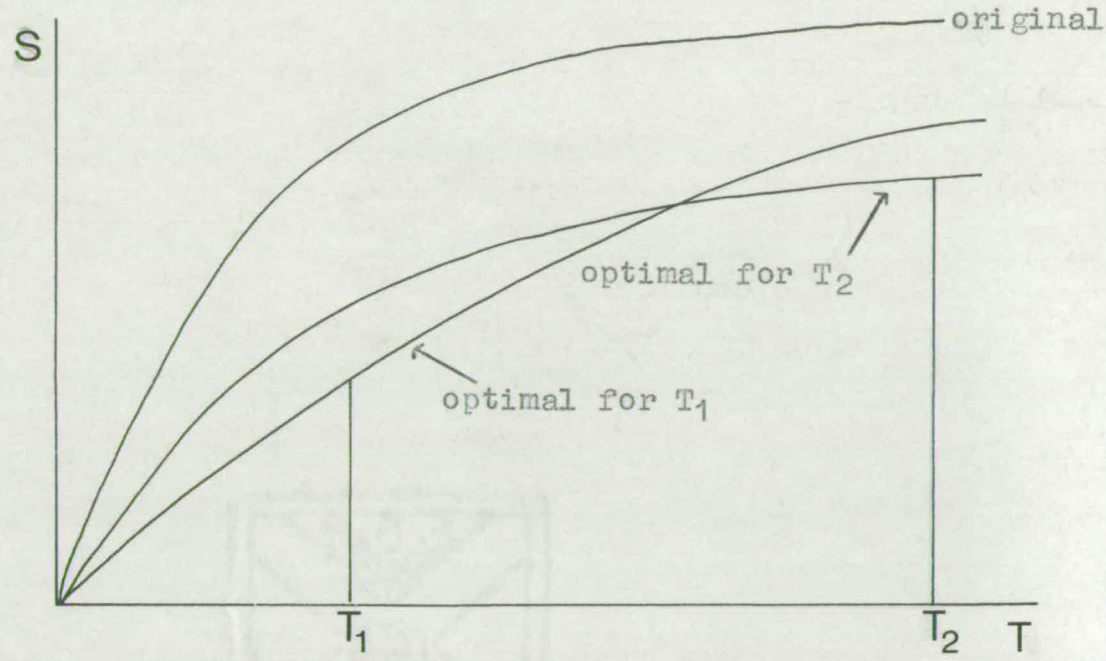



Fig 3-4 Effect of Restructurings on Working-set Curves

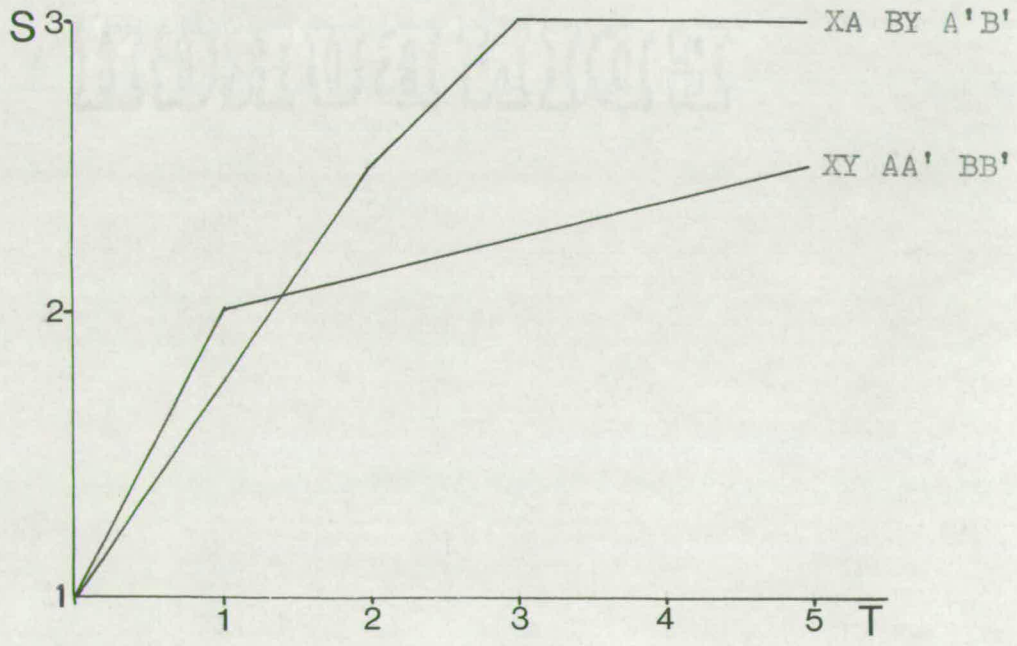



Fig 3-5 Working-set curves of Example 2

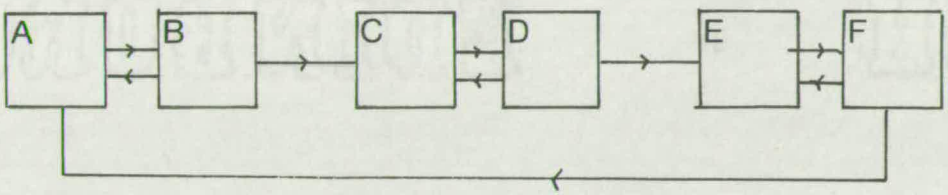



Fig 3-6 Importance of Time Scale, Example 1

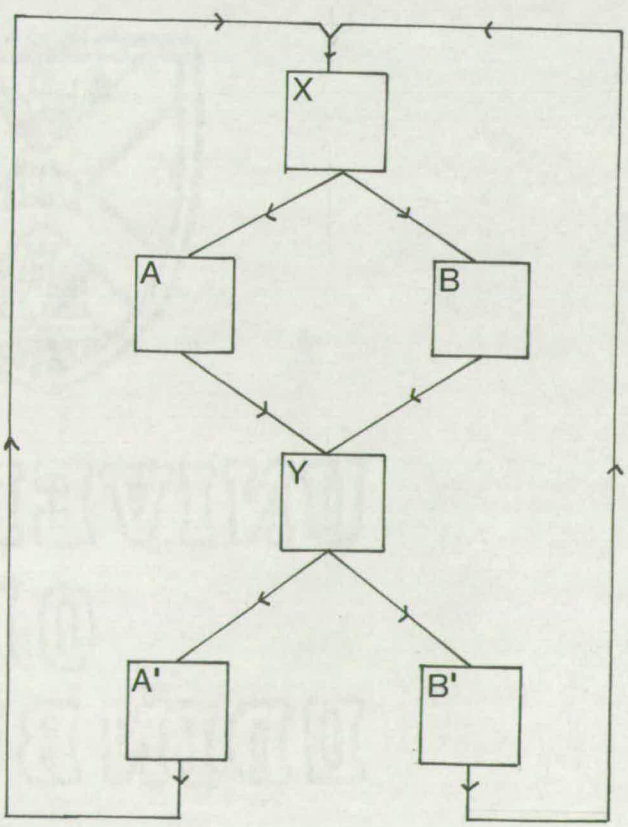



Fig 3-7 Importance of Time Scale, Example 2

AB CD EF. In most intervals (a proportion $1-T/T_1$) only one page will be referenced, while with any packing into just two pages, both will be referenced in at least 1/3 of the intervals. On the other hand, if $T_1$ is small, all the chunks will be referenced in any WS interval, and they must be packed as densely as possible, ABC DEF. For $T=T_1$ both these groupings give the same result.

Example 2

This is shown in fig 3-7. The chunks can be packed two to a page, control residing in each chunk for unit time. Control flows XAYA' or XBYB' with equal probability. As in the example of section 4, if T is large the best packing is XY AA' BB', but for small T the direct connections become significant, and a packing such as XA BY A'B' is better (A' and B' can equally well be given a page each). Actual values of the working set size are easily calculated (by considering the likelihood of each possible chunk sequence occurring in an interval); fig 3-5 shows the WS curves for each of the structurings above.

As in the case of the dynamic graph and the segment problem, we adopt an empirical approach to the solution of this problem; rather than trying to construct a model we simply attempt to measure data directly related to the specified WS parameter and work with this. This makes no simplifying assumptions about program behaviour, but we must guard against the risk that the practical effort of obtaining data outweighs the gain caused by restructuring.

## 6. AN EXPLICIT FORMULATION

Suppose the program is monitored over a long period (perhaps

over many runs), and a record is made of which chunks are referenced in each of a large number of processing intervals of length T. After N such intervals, this record is contained in a (n x N) array R where $r_{ij}$ =1 if chunk i is called in the j th. interval, $r_{ij}$ = 0, otherwise.

Given any grouping of the chunks into pages, the number of pages referenced in any interval is easily seen from R, and therefore the average number for all the monitored intervals. If for any restructuring this average is close to the actual average working set size of the program (which is of course a theoretical measure defined over all possible intervals of all possible runs of the program), R is said to be a representative record. For most large programs it is probably enough that every chunk has been entered a few times, although this may take many thousands of intervals if the chunks are small (see Chapter 4). (For a set of intervals to be truly representative of reference behaviour, it would have to contain every possible chunk combination repeated according to the frequency of its occurrence in all runs, an utterly impractical requirement with more than a very few chunks. R is sufficient for the limited information needed in this context.)

The optimal restructuring is then taken as that which gives the best results based on the tested intervals.

Example

| chunks \ intervals | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| A | 0 | 0 | 1 | 0 | 1 |
| B | 0 | 1 | 0 | 0 | 0 |
| C | 1 | 0 | 0 | 0 | 0 |
| D | 1 | 0 | 1 | 0 | 1 |
| E | 1 | 0 | 0 | 1 | 0 |
| F | 0 | 1 | 0 | 1 | 0 |

The six chunks A-F can be packed up to three to a page, and we make the unlikely supposition that the matrix above, containing the reference information of just 5 intervals, is representative. Consider for example the grouping ABC DEF. The first page is called in every interval except the fourth (none of chunks A,B or C being referenced then). The other page is referenced in all five intervals. The mean number of pages per interval is thus (4+5)/5 = 1.8. The optimal clustering, easily found here by trial and error, is ACD BEF. The result for this is (3+3)/5 = 1.2.

Given sufficient monitoring data, then, we have a simply and precisely defined problem, just as in the case of the segment problem. As with the latter however, there is no obvious solution procedure: exhaustive enumeration of all valid structurings cannot be considered. A further difficulty here lies in the large quantity of data, growing with the number of intervals tested; in the segment case all the necessary information is stored in the constant size (n x n) array (although with several hundred chunks this is not small). A practical solution will depend on the development of a heuristic method which yields a good restructuring, and a way of avoiding the problems associated with the storage and access of reference data from perhaps many thousands of intervals.

## 7. THE SIMILARITY ARRAY

With each chunk in the above problem is associated its (1 x N) reference vector. It is intuitively clear that two chunks which are normally referenced together in intervals are likely to be both in the same page in a good structuring - we are trying to minimise the wastage caused by only small parts of pages being referenced during an interval. Thus it seems reasonable to attempt to group

the chunks according to the 'similarity' of the reference vectors. For instance, in tackling the example of section 6 above, an obvious first move would be to group chunks A and D together on this basis.

Expressed in this way, some resemblance is seen to the well-known clustering problems which arise in classification theory (or numerical taxonomy) (refs. 29,30), and more recently in certain other fields such as pattern recognition (ref. 31). In these studies it is assumed that there is a given set of elements each defined by the values of an associated set of attributes. The values are usually binary, indicating simply presence or absence of an attribute. For example if the elements are diseases, they might be characterised by a set of symptoms; if insects, by various physical characteristics. In general terms, it is required to form clusters of elements such that those in the same cluster appear to look alike, and are dissimilar to elements in other clusters. The success of the result is judged by criteria relevant to the problem area; thus it might be hoped that all diseases in one cluster be amenable to similar treatment.

The first step in such clustering techniques is always to construct a similarity array S, whose elements $s_{ij}$ measure a defined coefficient of similarity between elements i and j. The choice of a satisfactory definition of this depends on the particular problem. Examples are:

1) number of attributes present in i and j

or 2) number of attributes either present or absent in both i and j.

Coefficients are often normalised to give a result between zero and one; thus if 2 above is divided by the total number of attributes,

the similarity will be unity for elements with identical attribute sets. The similarity array is then taken as summing up all the relevant information about the relationship of the elements to each other; it is used as the sole data to some clustering algorithm which groups the chunks into classes.

The resemblance of the above to the working set problem is obvious if we interpret reference to a chunk in a given interval as the possession by that chunk of the appropriate attribute. The difference is in the final objective: the clusters of chunks which will form pages are strictly limited in size.

A great attraction of defining a pairwise 'closeness' between chunks, and working purely with this, is that the similarity array is a compact way of storing data - it may be built up during the monitoring process, and the problem of handling an indefinitely large quantity of interval reference data is avoided. If there are n chunks, an (n x n) array S is initialised to zero. At the end of each interval, put (for all i,j):

$s_{ij} = s_{ij} + 1$ if and only if chunks i and j have both been referenced during the interval

A count is also kept of the total number of intervals N.

Noticing that the diagonal elements will contain the frequencies of use of each chunk, we can work out from S any pairwise relationship between two of the original reference vectors, and therefore any similarity coefficient.

E.g $s_{ij}$ number of intervals i and j both called

$2s_{ij}+N-s_{ii}-s_{jj}$ number of intervals i and j the same (both called or both not)

It is important to notice that the similarity array does not of course preserve all the original information (it is as though we

replaced the co-ordinates of a set of points by the distance between them, the elements here being defined in a very large dimensional space with an unusual measure of distance). Thus consider two possibilities for the reference histories of three chunks in four intervals:

| | | | | |
|---|---|---|---|---|
| A | 1001 | | A | 1001 |
| B | 1010 | and | B | 1010 |
| C | 1100 | | C | 0011 |

Both situations would give the same set of pairwise similarity coefficients (on any definition of the latter). The difference in the mutual relation between A, B and C, which is somewhat closer in the second case (since the references to all are confined to three intervals), cannot be expressed.

Choice of similarity coefficient

Various coefficients were used with data obtained from the KDF9 study (see appendix A), the most consistently successful being simply $s_{ij}$, the number of intervals in which both chunks are referenced. This puts no weight on zero matches: intervals in which neither chunk is called. These, then, are no more significant than intervals in which one chunk is referenced and not the other, thus:

| | |
|---|---|
| A | 11100100100 |
| B | 11100100100 |

would have the same similarity as

| | |
|---|---|
| A | 11101101100 |
| B | 11110110111 |

In the first case, A and B should obviously go in the same page; in the second, the high number of mismatches may make this a bad policy. However in practice, the density of 1's in the reference vectors of most chunks is quite low, and the positive match seems to be the

significant factor; attempts to weight zero matches or put a negative weight on mismatches gave in general less good results. Given any coefficient, a situation can usually be constructed for which it will be perfect or disastrous; what is required is the one suitable for the sort of behaviour that computer programs, chunked in the chosen manner, display. It is of course quite a simple matter to change the similarity coefficient in the clustering program.

The definition of the similarity array is the first step in a suggested heuristic solution to the explicit problem of section 6. But by continuing the approach at the end of section 4, we might have reached this point without ever defining the problem in precise terms. Connectivity as used there could be regarded as the frequency with which two chunks are called within a very short time of each other. Having decided that this was too immediate a relation to be relevant, we might have simply extended this to 'the frequency with which the two chunks are referenced within a period T of each other', and thus defined the similarity coefficient.

## 8. CLUSTERING ALGORITHMS

The second stage of the heuristic solution of the restructuring problem is a clustering procedure which, taking the similarity array as input, forms the chunks into groups of total size less than a page. The KDF9 data was used to investigate various clustering techniques.

In most of the clustering procedures, a search is first made for the two chunks which have the greatest similarity to each other. These are grouped together, and a new similarity is

defined between this group, treated as a single entity, and all the other chunks. This process is then repeated: find the greatest link (including the newly defined ones), combine the appropriate (groups of) chunks, and work out, in some defined way, new similarity coefficients. We thus obtain clusters of chunks which, as the procedure continues, grow by merging with unclustered chunks or with each other. So far this has a close resemblance to some algorithms of numerical taxonomy, but a departure is necessitated by the requirement that the ultimate groups of chunks be less than a page in size. In the algorithm finally adopted, clusters were simply not allowed to grow greater than a page: if two clusters had a combined size more than this, their similarity coefficient was set to zero, thus ensuring that they would not merge. Some earlier attempts were made to avoid this rather unsatisfactory 'discontinuous' effect of page size. In one case, clusters were allowed to grow indefinitely, pages being peeled off as they formed; in others attempts were made to reflect cluster size in the similarity coefficient. However the results of these more complex algorithms were generally less good, and they were abandoned.

The above method demands a definition of similarity between two groups of chunks. The choice found to be most successful, and finally adopted, was the arithmetic mean of all the inter-group similarities of the constituent chunks. This is not particularly logical (a choice more consistent with the adopted similarity coefficient would be the greatest link), but it gave the best results of the fairly simple definitions tested.

The other area of experiment in clustering made use of the Theory of Clumps (refs.32,33). This rigorously defines the

notion of a cluster, by specifying that it should have some precise property. Various types of clump are defined; an example is the R-Clump (ref. 32) which is a set S such that for every element of S, the sum of the connectivities (similarities) to the remaining members exceeds the sum of the connectors to all elements not in S. Methods are given in the references of discovering clumps (of which there may be a very large number) from a given similarity array.

Unfortunately, no way was found of relating clumps to the fixed size groupings required, and generally little success was had in applying clumps to this problem - unless non-overlapping clumps a little under a page in size happened to exist. Even careful hand clustering after examining clumps produced did not give as good results as the automatic procedure described previously. However it was felt that clumps could make natural units of program, and in certain circumstances might be useful in the construction of suitable variable size segments, the intersection of clumps perhaps giving an indication of which chunks could be usefully duplicated.

The following small example illustrates the clustering method finally adopted. We assume the chunks (denoted A to F) are of unit size, and the page size is three units. Suppose the interval reference record were as shown in the array below.

| ints. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| chunks | | | | | | | | | | | | | |
| A | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| B | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| C | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| D | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| E | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| F | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| G | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |

The similarity matrix (after setting its diagonal elements to zero) is:

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| A | 0 | 4 | 5 | 1 | 2 | 2 | 3 |
| B | | 0 | 2 | 4 | 2 | 0 | 7 |
| C | | | 0 | 0 | 4 | 4 | 1 |
| D | | | | 0 | 2 | 0 | 4 |
| E | | sym. | | | 0 | 3 | 2 |
| F | | | | | | 0 | 0 |
| G | | | | | | | 0 |

(B,G) is the maximum element; chunks B and G therefore combine to give BG, and we define a new array:

| | A | BG | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 3.5 | 5 | 1 | 2 | 2 |
| BG | | 0 | 1.5 | 4 | 2 | 0 |
| C | | | 0 | 0 | 4 | 4 |
| D | | | | 0 | 2 | 0 |
| E | | | | | 0 | 3 |
| F | | | | | | 0 |

E.g. $(BG,C) = \frac{1}{2}((B,C)+(G,C)) = \frac{1}{2}(2+1) = 1.5$

A and C now combine to give:

| | AC | BG | D | E | F |
|---|---|---|---|---|---|
| AC | 0 | 0 | 0.5 | 3 | 3 |
| BG | | 0 | 4 | 2 | 0 |
| D | | | 0 | 2 | 0 |
| E | | | | 0 | 3 |
| F | | | | | 0 |

Note that the size of the chunks AC+BG has exceeded the page size, so (AC,BG)=0. If this had not been so, the similarity would have been $((A,B)+(A,G)+(B,C)+(C,G))/4$, i.e. the average of the links between the <u>original</u> constituents.

BG and D combine to give:

| | AC | BGD | E | F |
|---|---|---|---|---|
| AC | 0 | 0 | 3 | 3 |
| BGD | | 0 | 0 | 0 |
| E | | | 0 | 3 |
| F | | | | 0 |

An arbitrary choice is made from the remaining equal non-zero elements, say (AC,E). This gives as the final packing:

ACE BGD F

In the original thirteen intervals, this would have given a total page reference of 23. (If in the final array, (E,F) had been chosen as the greatest element, the result would have been 24 with AC BGD EF)

## 9. COMMENTS AND CONCLUSIONS

A page boundary cannot be envisaged like a segment boundary, dividing two logically separate parts of a program. The intelligent choice of page layout, although profoundly affecting program behaviour, is essentially a low level decision, being dependent on the lengths of various bits of code and data, and the fairly short period reference pattern between them.

Section 6 showed that the problem of reduction of average working set size can be formulated quite precisely in terms of observed reference behaviour, but in seeking a practical solution we must abandon precision and work upon ideas of a 'dynamic distance' between one chunk and another, with respect to the chosen time-scale. It is this time-scale which is important, and makes structural program models which concentrate on direct links between one chunk and another, of not much use when applied to paging problems.

In general there is no way of telling how close to the optimum are the results from the clustering algorithm; this is particularly true with the large scale problems arising from big programs (and it is after all just these whose optimisation is most important). Extrapolation from the results of small problems like the example of the last section, is hardly valid. However a large number of restructurings (many 'by hand' after careful examination of the

results) was tried with the KDF9 data, and there was never a significant difference between the best result and that from the adopted clustering algorithm. There is thus good reason to believe that the algorithm achieves by far the greater part of the possible improvement.

In practice, it is the degree of improvement which these methods attain, and the feasibility of using them, which are the important matters. These are the concern of the rest of this thesis.

# IV PRACTICAL ASPECTS OF THE RESTRUCTURING TECHNIQUE

## 1. INTRODUCTION

This chapter examines the practical decisions which must be made in the accomplishment of program restructuring based on the ideas of III.5 etc. For an optimisation process to be generally worthwhile, it should not only produce a significant improvement in paging behaviour, but itself should be as rapid as possible. The ideal is that this process should be as transparent to the user as perhaps paging itself should be: the program during its normal course of running would be monitored (without apparent decrease in efficiency) and restructure itself when necessary, without user intervention. It is impossible to achieve quite this, in normal circumstances, so the results obtained from restructuring can only be practically judged in the light of the effort in achieving them.

In order to test the practicality of the methods, and investigate the degree of improvement obtainable, a restructuring scheme (henceforth referred to as RS) was implemented for programs written in the IMP language (a development of Atlas Autocode) for the ICL 4-75 computer. This scheme is described in full in chapter V and appendix B. However this chapter, in discussing general practical principles, makes reference to the decisions made in the implementation (although some of these were necessarily determined by the IMP language or the compiler on which RS was based).

## 2. CHUNKS

### Chunk size

Two militating considerations affect the average size of chunks. It should be clear that the smaller they are, the better are the potential results of restructuring. There is less chance that the same chunk will contain parts of a program dynamically far removed from each other, there is more flexibility in repacking, and scarcely used sections (failure paths, etc.) are more likely to be isolated and able to be removed from the main paths. On the other hand the average size of chunks affects their total number; the similarity array storing reference data for a big program becomes impracticably large if the chunks are too small. Not only the amount of computation by the clustering algorithm increases (roughly) as the square of the number of chunks, but - more crucially - constant reference to a very large array during both data collection and restructuring phases will make the whole process very inefficient in a paging environment. To a certain extent, this can be overcome in a manner described later in section 4; similarity data is only collected on the more commonly used chunks, some merging of chunks takes place on the basis of this, and the process is repeated. The disadvantage of a very large array is replaced by the necessity for a number of data collection phases. However it seems to be the case that in large programs, many chunks are scarcely used; providing that the array contains information on, say, 1/4 of the number of chunks, very few collection phases are necessary. The restructuring of programs with several hundred chunks has been quite satisfactory using arrays with a side of length less than one hundred.

This means that although a feasible choice for a chunk would be the entire unit of compilation (module), we can deal with much smaller chunk sizes. We therefore consider a fine chunking within the unit of compilation: this is in any case essential if the structure of the language, or the customary use made of it, leads to modules very large compared to a page size (as is currently true with IMP).

Chunk boundaries in instructions

A chunk boundary is likely to be superfluous unless it is in one of the following positions (which we shall denote branch points). The two types will often coincide.

1) Immediately following an instruction which can transfer control elsewhere.

2) Immediately preceding an instruction to which control may be transferred from elsewhere.

This second case is very important - initialisation code often 'drops through' into a main loop, and the former 'once only' code will normally be better removed to another page.

Example

```
                                comment initialisation;
                                        ...
                                        ...
branch point type 2_ _ _ _ _ _ L: comment end of initialisation;
                                        ...
                                        ...
branch point type 1_ _ _ _ _ _    goto L;
```

Note that goto L being a non-conditional branch, it must also be followed by a branch point of type 2 or by redundant code.

A chunk boundary at any other than branch points would separate the code into two parts practically certain to be both entered or

both not entered in the same time-slices. (Although the boundary would not be quite pointless: the smaller chunks might give more flexibility - perhaps in filling up available spaces in two pages which were bound to be called in the same time-slices anyway. Also the two parts are not necessarily dynamically identical, as the end of a time-interval will sometimes occur while control is within one of them). The finest worthwhile partitioning is thus obtained by taking boundaries at all branchpoints. In practice, however, some subset of these would have to be chosen to avoid the very large number of chunks which would arise. An obvious first approximation would be the starts and ends of subroutines, since these are almost certain to be dynamically separate from their contextual surroundings. It is the finer subdivision upon which it is far more difficult to decide.

A programmer choosing boundaries himself would be able to make intelligent choices based on a prediction of his program's dynamic flow. One example in which such knowledge is useful, is the branchpoint following a conditional branch.

Example

```
(1)      if A then goto L
(2)   M:  ...
```

A boundary after statement (1) is valuable if control normally passes to L, and (2) is a large chunk of rarely used code - perhaps a failure path. Similarly it is needed if (2) is often entered from elsewhere (via label M) again making (1) and(2) dynamically removed from each other. On the other hand, the boundary may be superfluous if the commonest path passes through (1) and (2), A normally being false.

If chunking is being performed automatically by a compiler,

some method must be devised of choosing a suitable subset of the branchpoints. These are easily recognisable by a high-level language compiler (not necessarily true in an assembly language with branchpoints of type 2), and the language statements can give useful clues as to which points are most likely to be suitable for chunk boundaries. V.4 describes and discusses the way RS makes its choice.

If a chunk does not end in a non-conditional branch (not including procedure calls) there is a chance of 'dropping through' to the next chunk. Obviously if the two chunks are not adjacent after restructuring, a branch instruction will have to be generated after the first. For this reason, and to facilitate monitoring (see next section), it is convenient to generate at compilation time a nonconditional branch at the end of every chunk that does not already terminate in one. (In RS the nature of the choice of chunk boundaries ensure that many already do). Thus the above example would be compiled as if it had been:

```
                    (1)     if A then goto L
chunk boundary- - - - - -   goto M
                    (2)  M: ...
```

Little need be lost, for if, after restructuring, chunks (1) and (2) are still in the same page, they can be made adjacent, and the goto M removed (as it could be from any chunk which originally ended in this instruction and is adjacent to (2) in the final arrangement).

Data chunking

IMP is an Algol-like language where much of the data will be dynamically created on a run-time stack; for this to be chunked in the manner defined previously is not feasible. Own (static)

variables could be included, the own variables of each subroutine probably being conveniently regarded as a chunk. On many timesharing systems, the requirement of shareable, and therefore invariant, code would preclude the packing of data with the code which refers to it. In any case, only a little thought is required to see that such a procedure, often advocated, could save at most a single page of working set size - over the policy of simply ensuring that as code sections are packed together, so are the corresponding data sections.

In general, the choice of effective chunk boundaries on data will be fairly clear; there are however two practical considerations against data-chunking.

1) Collection of reference data will probably require interpretation.

2) Repacking data chunks and keeping references correct can be very complicated.

RS confines itself to code (and some constants) only; it would therefore be of little use with programs whose paging problems arise from large scale data reference: matrix inversion, list-processing etc. It is in any case felt that such problems are not amenable to formal restructuring methods, and are better attacked at the design stage (refs.10,19).

## 3. COLLECTION OF REFERENCE DATA

Unless some sophisticated hardware monitoring device is available, the methods of collecting the necessary chunk reference information are:

1) interpretation of the program,

2) monitoring by means of additional instructions planted within the program.

Interpretation is almost essential if data chunking is to be included, the generation of instructions to trap every data reference being hardly practicable. However the slowness of interpretive methods makes them most unsuitable for gathering the amount of representative data required, especially for a large piece of data-dependent software where many runs might be necessary before a valid restructuring can be performed. Assuming code restructuring only, we therefore look at just where instructions have to be incorporated for a program to monitor its own behaviour in order to yield the required information.

We note that the data in the similarity array can be collected by knowing only the processing instants of chunk changes; behaviour within a chunk, or in a routine external to the area of restructuring is not relevant. Now once the 'dropping through' case has been eliminated (see last section), a chunk can only be entered by a branch instruction. Thus all necessary information can be obtained by monitoring the instructions which transfer control, there being no need to include any which are known to lead outside the restructuring area (although entries into the area may have to be trapped), or which are known to leave control within the same chunk. This can be achieved by replacing all such branches by instructions which pass the original target address to monitoring routines. These can make a record of the target chunk, update the similarity array if necessary and then return directly to the target address in the program. The only additional data required is a means of finding the containing chunk from the target address.

In RS, the compiler generates an internal label (an integer) for every branchpoint (excluding those following procedure calls). A monitored branch instruction consists of a load of the target internal label into a register, followed by a jump into monitor. Arrays which were formed at compile time can translate internal labels both to addresses and containing-chunk numbers.

The efficiency of the self-monitoring program thus depends on the number of instructions which have to be replaced, and on how efficient the monitoring routines can be made. Most computation and additional store reference is performed when the similarity array has to be updated at the end of each (simulated) working set interval; the length of this then also has an effect.

## 4. PARTIAL CLUSTERING

A clustering algorithm will take the similarity array and chunk sizes as input, and produce a list of new chunk groupings by the method of III.8. Here we describe the 'partial clustering' procedure mentioned earlier.

Suppose there are m chunks, but the maximum size of similarity array we are prepared to accept has space for only n (less than m). During the first set of monitoring runs, similarity data is only stored for n of the chunks - denote this subset (the <u>similarity subset</u>) by N. N can be chosen arbitrarily, but given no other information, the first n chunks referenced during the runs can be used. An arbitrary subset might be wasteful in that some of its chunks may not be referenced at all. Ideally, the similarity subset would consist of the n most commonly used chunks, but it is not known in advance which these will be. Apart from the

similarity data on N, a count is kept of the frequency of use (i.e. number of intervals during which access is made) for each of all the chunks.

When it is decided to perform the first clustering, a search is made for the highest frequency of use, say f, of all the chunks not in N. The clustering procedure outlined in III.8 is only continued up to the point where no inter-group similarities remain which are greater than f. It cannot be taken further because chunks not in N may have had linkages as great as f (if a full size similarity array had been formed) with chunks in N. The program is now regarded as consisting of a new set of chunks: the original m-n not in N, any groups which formed during the clustering process, and those chunks in N which did not go into groups. Unless no groups at all were formed, there will be less chunks than before.

The process of data gathering and clustering is now repeated, but with one important improvement. The subset N need no longer be random, since we now have data from the previous set of runs. Taking the frequency of use of a group of chunks as that of its most used member, we take as our subset N, the n most commonly used 'new chunks'. In this way although little clustering may take place during the first restructuring, the second and subsequent ones are far more successful, and the number of chunks decreases rapidly.

A complete clustering can obviously take place when there are less than n chunks remaining. Alternatively, some accepted degree of use can be decided upon, say 0.1% of the total intervals, and the process completed when none of the chunks outside N are used

more than this, it being assumed that the contribution of such chunks to the mean working-set size is insignificant.

The smaller is the similarity array, the more times the cycle of data gathering and restructuring must be repeated. However the data in the early stages need not be representative in the sense of III.6, since only a subset of the chunks are really involved. This is particularly true of the initial monitoring phase when little clustering will probably occur; it primarily serves to obtain the approximate relative use frequencies of the more common chunks. It is not easy for RS to decide when sufficient data has been obtained; the scheme makes no attempt to do this and leaves it to the discretion of the user when to restructure.

## 5. REPACKING CHUNKS

The clustering procedure calculates how chunks should be grouped into pages, but actually achieving this packing can give practical difficulties. Again we have the situation that the finer the level of chunking, the more problems are created; if we had adopted the complete unit of compilation as a chunk, repacking would be simply achieved by presenting the modules to loader in the appropriate order.

Repacking involves changing all references to (relative) instruction addresses within the restructuring area. However it is just such references that have already been intercepted because of the requirements of monitoring; relocation of code thus presents fewer additional difficulties. In RS, all the relevant branchpoints have their addresses in an array used by monitor

(see section 3); if repacking is followed by updating this array, we have a rearranged but still monitoring version of the program without altering the code in each chunk at all. The partial clustering process can thus be physically carried out quite easily.

The final replacement of monitoring instructions with the original jumps is complicated if the former take up more space. (In RS, the implementation of the compiler on which it was based was such that the monitoring instructions took up no more room than those they replaced, and no problem arose - see Appendix B). One could overwrite the superfluous code with dummy instructions, but this means that the extra size of the code may counteract the gain achieved by restructuring. Both this problem, and the untidyness and heavy machine dependence of the repacking routines (in RS) can be alleviated by operating on an intermediate code which is machine independent, with symbolic code addresses. Repacking would be performed on this, the only machine dependent data required being the chunk sizes (when non-monitoring code is produced), and the page size; a translation of the intermediate code would produce object code, monitoring or otherwise.

In some cases it might be possible to restructure at the source language level itself. However normally the structure of the language (scope of labels and names, do-loops, etc.) apart from the structure of individual statements (e.g. see cycle statement in chapter V), makes it impossible to perform the restructuring of small instruction chunks at this level. In IMP all that could be achieved would be the repacking of subroutines within their containing block, quite simply performed on the source code. It would be also necessary to have a directive such as align on page boundary within the language.

# V THE RESTRUCTURING SCHEME

## 1. INTRODUCTION

This chapter describes the structure and use of a restructuring scheme designed to improve paging behaviour of IMP programs written for the ICL 4-75. This is a high-speed 32 bit word machine (with an instruction set identical to that of the IBM 360/50). If operated in paged mode, a virtual store is provided to the extent of 24 bit addressing; the page size is 1024 words and the addressing mechanism regards the virtual store as divided into 16 page 'segments'. Addresses are specified in bytes which are 8-bit units; a word aligned address is thus one divisible by 4. Average instruction times are about 2$\mu$s.

An initial motivation for implementing the restructuring scheme was the proposed Edinburgh Multi-Access System (EMAS), a general purpose time-sharing system being written for the 4-75 by a joint team from ICL and the Edinburgh University Department of Computer Science. However the system is not yet fully operational (at the time of writing), and all results of program restructuring have been obtained by running under 7J, a batch system run on the 4-75 in non-paged mode. This was made possible by using the System Interface Module (SIM), written by the Edinburgh Regional Computing Centre. SIM provides an EMAS-type interface on top of 7J, allowing the loading, linking and running of object program files which have been produced to the EMAS specifications and conventions.

An evaluation of the results of restructuring could be made quite simply by means of the very monitoring instructions which the scheme plants in the programs under test. These were used, apart

from gathering chunk reference information, to investigate paging behaviour (over code only), so that working set graphs could be constructed before and after restructuring. There was thus little logical difference between obtaining such results on EMAS or 7J/SIM. However two areas of information are necessarily lacking: the effect of WS size on program performance under a typical time-sharing scheduler, and the performance of the restructuring scheme itself in a paged environment.

The next chapter details the results of experiments on fairly large programs. However the size of these makes them unsuitable for providing examples of output from the scheme in the compact form necessary here. This chapter, therefore, makes all its illustrative references to a single small program written especially for this purpose. The code of this program is only just over a page in size, so the actual results of restructuring it are of no significance; but it is hoped that it demonstrates the good and bad features of the scheme.

Low level aspects of design and implementation are left to Appendix B. Of EMAS itself, few features concern us here (see refs.34,35), those which do are dealt with very briefly.

## 2. PROGRAMS IN IMP

The restructuring scheme is designed for programs written in IMP, the language in which most of EMAS is written. IMP was developed from Atlas Autocode, an Algol-like language with block and procedure structure and a run-time data stack. Such features of the language as are relevant here should be made clear by the example and the discussion in section 4.

Units of compilation

Some of the environmental features of IMP have developed in a slightly ad hoc manner and the terminology to date is not quite standard; that adopted here is personal. The unit of compilation - the module - may consist of either of the following.

1) A main-program. This has its outer block delimited by begin and end of program.

2) One or more external routines. The module will have the form:

```
external routine p1(possible parameters)
        ...
end
external routine p2(        ...        )
        ...
end
        ...
```

terminated by end of file

Each external routine is similar to any other procedure in an IMP program, except that it may be entered, and parameters passed, from an independently compiled module. The word external, and the fact of not being contained in an outer block, is the only difference in form.

Object Program files

An object program file in EMAS (ref. 35) is divided into three distinct areas.

1) Code area: this contains instructions and constants, invariant during the running of the program, and thus capable of being shared if necessary by independent users.

2) General linkage area pattern (GLAP): this contains initialised data required by the program. There will also be space for the insertion of linkage information (i.e. absolute addresses not known until load time) enabling references to be made to other modules.

3) Linkage data area: contains information about the entry points in the code of this file for use by the loader in satisfying external references by other files.

Such a file is produced by compilation of an IMP module. A main-program will have a single entry point at the beginning, given a standard name by the compiler; a set of external routines will have one for each routine(named p1, p2, etc. in the example above).

To run a program, a set of object files must be specified to the loader - it is assumed that one of these files contains a main-program. The user is provided with his own copy of the GLAP (known as the GIA) for each file, necessary linkages are made and the main-program is entered. This entry, and all entries made to other external routines, follow conventions as to the contents of certain machine registers and parts of the run-time stack. The modules of a program need not all have been written in IMP, providing that the code produced, and the object file format follow the conventions.

If at least one module is in IMP, additional associated external routines are automatically linked in. Apart from standard I/O routines there is a module known as Perm (permanent) containing standard material required by the compiled code but not conveniently compiled in-line.

The unit of restructuring for our purposes is the module, the

unit of compilation. Thus if the required area of restructuring is a set of external routines, these must be compiled together. Reference data is collected by an enlarged Perm: this must be explicitly linked in at run-time since the chunking IMP compiler is not one of the standard system compilers. It is assumed that the code area of an object module begins on a page boundary; obviously nothing can be achieved if page boundary positions cannot be guaranteed.

## 3. SUMMARY OF RESTRUCTURING SCHEME

Fig.5-1 shows the course of the production of an object module with optimised structure. The production or use of files is shown by dotted lines.

The module is compiled to give code containing monitoring instructions, and additional tables in the GLAP. We shall say such an object file is in 'M-format' (Monitoring). Compilation also produces a chunk information file which contains various information (size of chunks, etc.) necessary to the restructuring routines, and stores reference data (notably the similarity array) between runs.

After each run of the object module, or to be precise a program containing it, the chunk information file is automatically updated with the latest chunk reference data. (The information file is not required for the program run, but only for the storing of the similarity array at the end; under EMAS, although the information file could only be associated with one process at any time, others could share the object file if they so wished.) When it is decided that sufficient data is collected (a user decision) the

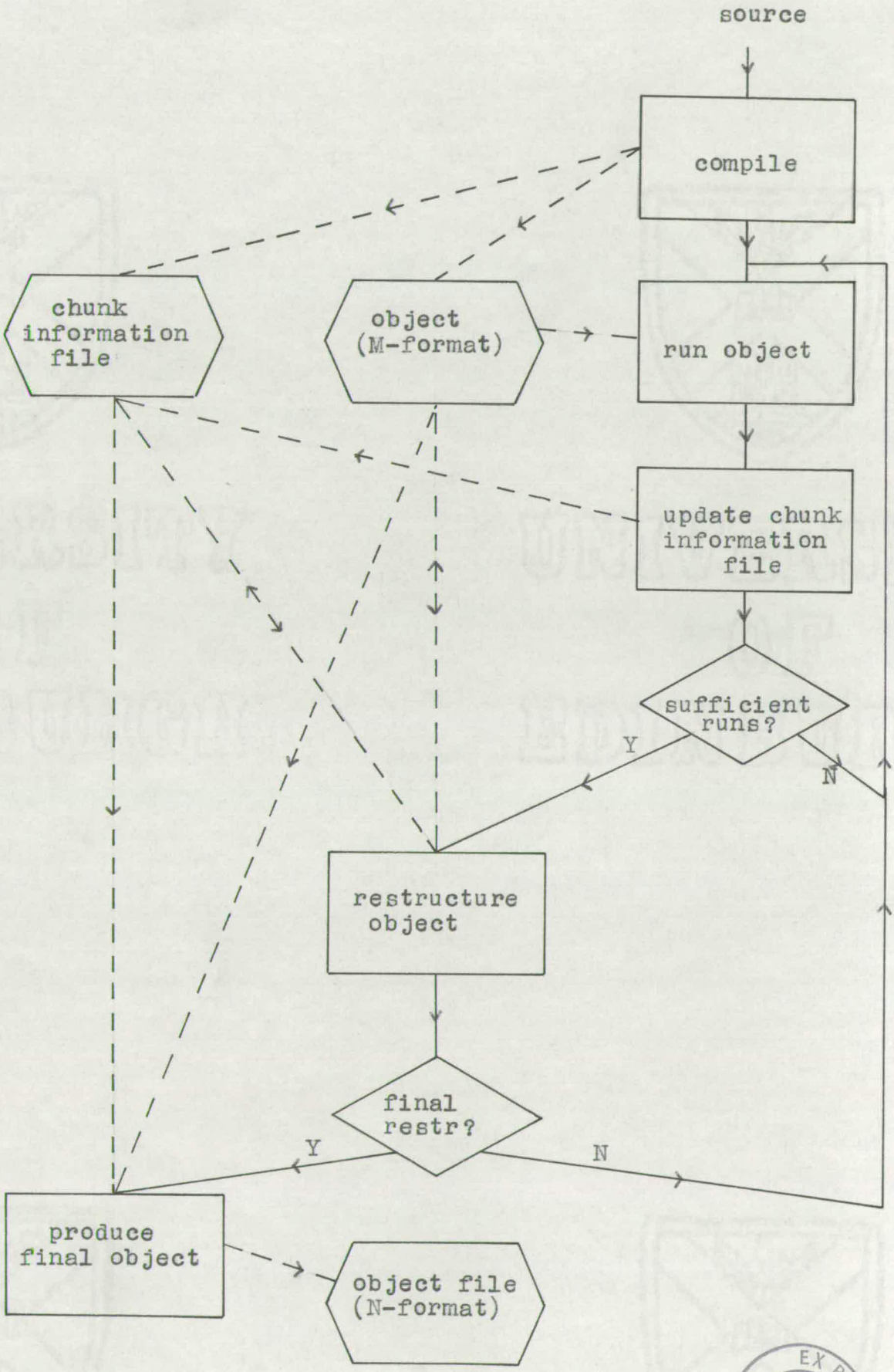



Fig 5-1 Restructuring scheme

restructuring program is used to produce a new object file (still in M-format) and chunk information file. For a large module, this process of running and restructuring may have to be repeated several times. After the final restructuring (a system decision), a further program converts the M-format object file to an ordinary N-format (Normal) file.

## 4. THE COMPILER

The compiler for the restructuring scheme was developed from an early IMP compiler written for the EMAS project by M. Falla, assisted by A.Freeman and T.Head. When implementation had reached an advanced state, this compiler was superseded by another, and effectively abandoned. Its availability, and a structure highly suitable for the necessary developments, made it an obvious choice as a basis for the restructuring scheme. A brief structural description, and the changes which were made, are given in Appendix B.

If required, the compiler will produce an N-format object module directly; its normal use however is to produce a chunked and self-monitoring object program, together with an associated chunk information file. The whereabouts of these files must be specified on appropriate job control cards. M-format code is exactly the same length as N-format, but the GLAP is much greater since it contains tables necessary for the running of the monitoring routines.

Figs. 5-2,5-3 show the compiler listing of a small demonstration program (this generates pseudo-random bridge hands, and prints them with an opening bid). The statements omitted at the

```
14          %BYTEINTEGERARRAY HAND(0:3,2:14)
15          %OWNBYTEINTEGERARRAY S(0:3)='S','H','D','C'
16          %INTEGER POINTS,BALANCE,LIMIT,CUT
17          %INTEGERARRAY DIST,PTS(0:3)
18          %OWNINTEGER HANDCOUNT=0
19          %ROUTINE INITIALISE; ! INITIALISES PACK
20              %INTEGER I,J,K
21              K=-1
22              %CYCLE I=0,16,48; ! SUIT
23                  %CYCLE J=2,1,14; ! VALUE
24                      DEAL(K+J)=I+J
25                      %REPEAT; K=K+13; %REPEAT
26              %END; ! OF INITIALISE

27          %ROUTINE SHUFFLE; ! SHUFFLES AND CUTS PACK
28              %BYTEINTEGERARRAY A(1:52)
29              %INTEGER I,J
30              %CYCLE I=1,1,26
31                  A(2*I-1)=DEAL(I); A(2*I)=DEAL(53-I)
32                  %REPEAT
33              CUT=CUT+11
34              %IF CUT>=52 %THEN CUT=CUT-52
35              %IF CUT=0 %THEN CUT=1
36              %CYCLE I=1,1,CUT
37                  DEAL(52-CUT+I)=A(I); %REPEAT
38              %CYCLE I=CUT+1,1,52
39                  DEAL(I-CUT)=A(I); %REPEAT
40              %END; ! OF SHUFFLE

41          %ROUTINE DEALPACK
42              %OWNBYTEINTEGERARRAY T(11:14)='J','Q','K','A
43              %INTEGER I,J,K,M
44              POINTS=0;BALANCE=0
45              %IF HANDCOUNT&7=0 %THENSTART
46                  NEWPAGE; %PRINTTEXT'BRIDGEHANDS'
47                  NEWLINES(3); %FINISH
48              %CYCLE I=0,1,3
49                  %CYCLE J=2,1,14
50                      HAND(I,J)=0; %REPEAT; %REPEAT
51              %CYCLE I=1,4,49
52                  HAND(DEAL(I)>>4,DEAL(I)&15)=1
53                  %REPEAT
54              %CYCLE I=0,1,3
55                  K=0; PRINTSYMBOL(S(I)); SPACES(3)
56                  M=0; ! COUNTS POINTS
57                  %CYCLE J=14,-1,11
58                      ->1 %IF HAND(I,J)=0
59                      SPACE;PRINTSYMBOL(T(J));K=K+1;M=M+J-10
60          1:          %REPEAT
61                  %IF HAND(I,10)#0 %THENSTART
62                      WRITE(10,2); K=K+1; %FINISH
63                  %CYCLE J=9,-1,2
64                      ->2 %IF HAND(I,J)=0
65                      K=K+1; WRITE(J,1)
66          2:          %REPEAT
67                  %IF K=0 %THEN %PRINTTEXT' --'
68                  %IF 1<=M<=3 %AND K=1 %THEN M=M-1
69                  PTS(I)=M; POINTS=POINTS+M; DIST(I)=K
70                  %IF K>5 %THEN BALANCE=BALANCE+K-5
71                          %IF K<2 %THEN BALANCE=BALANCE+2-
```

Fig 5-2 Compiler Listing

```
72                      NEWLINE; %REPEAT
73                  HANDCOUNT=HANDCOUNT+1
74                  %END; ! OF DEAL

75                  INITIALISE; READ(CUT); READ(LIMIT)
76                  ->2 %IF 0<=CUT<=52
77                  %PRINTTEXT 'INVALID CUT'; %STOP
78          2:      SHUFFLE; SHUFFLE; SHUFFLE; SHUFFLE
79          1:      SHUFFLE; DEALPACK
80                  %BEGIN; ! BIDDING BLOCK
81                      %INTEGER LS,NO,SUITPTS
82                              %SWITCH B(0:20)
83                      %INTEGER I,N
84                      %ROUTINE BIDSUIT(%INTEGER I)
85                          WRITE(I,1); SPACE; PRINTSYMBOL(S(LS))
86                          %END

87                       %ROUTINE NT(%INTEGER I)
88                          WRITE(I,1); %PRINTTEXT' NT'
89                          %END

90                      NO=3
91                      %CYCLE I=0,1,3
92                          %IF DIST(I)>NO %THENSTART
93                              LS=I; NO=DIST(I)
94                              %FINISH; %REPEAT
95                      %IF LS=0 %AND DIST(3)=NO %THEN LS=3
96                      SUITPTS=PTS(LS)+5*(NO-4); SPACES(18)
97                      N=BALANCE; %IF N>3 %THEN BALANCE=3
98                      ->B(BALANCE)
99              B(0): ->2 %IF POINTS>=12
100             NB:             %PRINTTEXT' NO BID'; ->1
101             2:      %IF POINTS<=15 %THENSTART
102                         ->3 %IF SUITPTS>10
103                         NT(1); ->1; %FINISH
104                     %IF POINTS<20 %THENSTART
105                 3:      BIDSUIT(1); ->1; %FINISH
106                     %IF POINTS>=23 %THEN ->TC
107                     %IF SUITPTS>=15 %THEN BIDSUIT(2) %ELSE NT(2
108                     ->1
109             B(1): %IF POINTS=11 %AND SUITPTS>10 %THEN ->3
110                     ->NB %IF POINTS<=11
111                     ->3 %IF POINTS<=18
112                     ->7 %IF POINTS<22
113             TC:     %PRINTTEXT' 2 C(ACOL)'; ->1
114             B(2): ->PR %IF POINTS<10
115                     ->6 %IF POINTS>=12
116                     ->3 %IF SUITPTS>10; ->NB
117             B(3): ->PR %IF POINTS<8
118             6:      ->3 %IF POINTS<=18-N
119                     ->TC %IF POINTS>=22-N
120             7:      %IF SUITPTS>15 %THEN I=2 %ELSE I=1
121                     ->8
122             PR:     SUITPTS=5*(NO-4)+POINTS
123                     ->NB %IF SUITPTS <=20; ->9 %IF LS<2
124             9:      %IF SUITPTS<=27 %THEN I=3 %ELSE I=4
125                     ->NB %IF SUITPTS <=22
126                     %IF SUITPTS >30 %THENSTART
127                         I=5; ->8; %FINISH
128             8:      BIDSUIT(I)
129             1:      %END; ! OF BIDDING BLOCK

130                         NEWLINES(3)
131                 ->1 %IF HANDCOUNT<LIMIT
132                 %ENDOFPROGRAM
```

Fig 5-3 Compiler Listing

```
CODE OCCUPIES      4246  BYTES


ROUTTABLE SIZE      164
CHUNKNUMBER          27

CHUNK          1      2      3      4      5      6      7      8      9     10
ADDRESS        0    224    460    822   1048   1406   1682   1988   2358   2506
LINE           1     19     27     36     41     51     57     63     75     78

CHUNK         11     12     13     14     15     16     17     18     19     20
ADDRESS     2668   2768   2928   3016   3308   3338   3360   3480   3630   3738
LINE          82     84     87     90     99    100    101    105    109    113

CHUNK         21     22     23     24     25     26     27
ADDRESS     3762   3836   3866   3926   3968   4042   4142
LINE         114    117    118    120    122    124    128
```

Fig 5-4 Chunk Positions

beginning are specifications of external I/O procedures. The source program is listed as it is read; since the compiler makes a syntax analysis of all the program statements prior to further processing (this was originally designed to improve paging behaviour of the compiler), the chunk boundaries are not worked out when the listing is made. These are given at the end of compilation (fig.5-4) and have been ruled on the listing for clarity. The following discussion of the choice of chunk boundaries is carried out with reference to this program. Keywords, for example %BEGIN, in the listing are written here as begin; the jump instruction '→ label', is denoted by 'goto label'.

Choice of chunk boundaries

A chunk boundary appears in the following positions.

1) Before and after procedure and function declarations. Thus one appears before line 19 (routine initialise), and after line 26; in this case there is no finer subdivision. Note there is no boundary after the end at line 129; this is simply the end of an inner block (starting at line 80), and program control will pass through it.

2) Before and after switch declarations. The statement 'switch B(0:20)' at line 82, compiles to a vector (over which program control jumps) of 21 words which contain the (relative) instruction addresses of left-hand labels B(0), B(1), etc. It happens here that only four of these labels are defined - the other addresses in the vector will be set 'unassigned'. Such a vector is always regarded as a single chunk. Here the final boundary is drawn after 'integer I,N' which is a declarator generating no code.

3) Before explicit left hand labels, but only if:

a) the preceding statement is a branch (conditional or otherwise), or b) the code length of the current chunk is greater than some limit (currently 256 bytes, i.e. 1/16 of a page).

Thus label 2 at line 78 causes a chunk boundary, being preceded by a stop instruction (which terminates the program run). Similarly there is a boundary at B(0) (line 99), preceded by a goto, and TC (line 113), by a conditional goto. On the other hand, label 1 (line 60) does not cause a boundary; it does not follow a branch, and the preceding chunk boundary is well within 256 bytes. Note that a procedure call is not counted as a branch in (a) above - thus label 1 at line 79 is not preceded by a boundary, despite the procedure call 'SHUFFLE'. There are no side exits from IMP procedures, so control must return from 'SHUFFLE' before reaching the label. The case for a boundary after a procedure call is thus not considered as strong as that for one after an ordinary branch.

4) Before cycle statements, but only if (b) holds. Cycle heralds a loop (similar to the Algol for statement), the end of which is marked by repeat. (The code generated at cycle consists of initialisation code, setting up the loop, followed by the implicit left-hand label, the top of the loop proper. The chunk boundary, if one is generated, goes before this label.) As examples, line 36 is preceded by a chunk boundary, line 38 being within 256 bytes of this, is not.

5) Before start (in conditions), but only if (b) holds. start and finish are statement brackets in IMP, used in the construction:

```
if A then start
    ...
finish else start
    ...
finish
```

The else clause is optional.

The condition of distance from the preceding boundary is never met in the illustrated program. Line 105 may appear to be an example, but the chunk boundary here arises from the label 3 following the implicit jump over the 'then' clause.

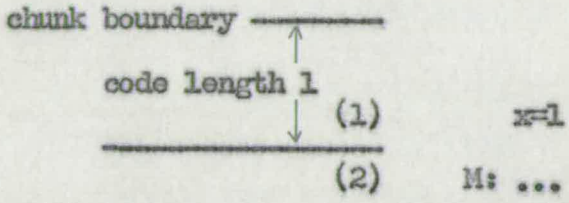
The above rules for the determination of chunk boundaries must be viewed with the realisation that the total number of chunks is not to be too large. An average of 25 chunks per page would give 500 chunks for a 20 page module. Using a similarity array of side 100, at least 5 restructurings would probably be necessary. This is acceptable for such a large program, but more chunks would lead to more restructurings or a larger similarity array: extending the time of optimisation and making it less worthwhile. Thus although it is easy to point to omissions in the choice of boundaries, it is less easy to find an improvement that does not lead to appreciably more chunks. The above selects those branchpoints which appear to have the strongest case (section IV.2).

Some discussion may explain the rules adopted. Consider the following situation.

```
(1)      if A then goto L
(2)   M:    ...
```

The statement (1) may be dynamically distant from (2) for two

reasons: either the path to L may be taken, or label M reached from somewhere else. If statement (1) is not a branch, there is less reason for a boundary between (1) and (2). Thus suppose we have:




Suppose in the above example, a boundary is not put before M, but in fact M is normally entered from elsewhere, the preceding code perhaps being concerned with initialisation. An amount of space l will then have been effectively wasted within the page; the greater is l, the more important this may be. This is the reason for the condition (b), about the length of the current chunk, being applied at certain times. An example in the illustrated program is at line 75, onwards. The initial entry to the program reaches here (after jumping over the procedure declarations). Up to label 1 (line 79) is initialisation code, the main loop of the program returning to label 1 as each bridge hand is generated. Thus although a boundary before this label would have been more intelligent than that before label 2, the only 'waste' is the comparatively short quantity of code generated by line 78. If label 2 had not been present, the initialisation code would have been long enough to force a boundary before label 1.

Boundaries are more likely to appear before explicit left hand labels than other branch-points -- it being considered that labels are likely to be entered from a number of statically distant points. The last third of the example program has a large number of chunks compared with the rest, the explicit labels producing

more boundaries than cycle or start statements. Whether in general this is a good decision depends on the style of programming in the module under test; it may be that the preponderance of explicit labels in IMP programs is still an influence of the Atlas Autocode from which it was developed (this did not contain the statement brackets start or finish). The chunking part of the compiler has been written so as to make changing the choice procedure for chunk boundaries an easy matter.

The idea of obtaining information from the programmer as to the choice of chunk boundaries was rejected as being an interference and intolerably arduous in a large program. Knowing the algorithm he can of course force a decision - for example, writing goto 1 before line 79, in the program shown. There is a case for the provision of a directive which forces a chunk boundary and which the programmer may add if he wishes having seen the boundaries which the compiler produces.

## 5. COLLECTION OF REFERENCE DATA

At run-time, the desired length of working set interval (the similarity interval) must be specified to job control, and a special Perm linked in; as computation proceeds, the similarity array is then built up in the GLA of the module concerned. Processing time was increased about five times in the experimental programs. This is rather more than hoped, but the monitoring code in Perm could have been rewritten in machine code to give some improvement. Little effort was made in this direction: the processing time was not great enough to be a problem in the tests made, and the performance in a time-sharing system would probably be a function

far more of paging behaviour than processing time. (As an example of the extra paging, consider a program with 300 chunks (say 12 pages of code) and a similarity array of side 100. The array is symmetrical, and its elements halfword, it would thus occupy 2525 words. With this number of chunks, the additional tables would occupy about as much space again, so there would be an extra five frequently accessed pages. The additional length of code in the monitoring Perm is quite small.)

If required, additional 'page monitoring' can be simultaneously included. In this case, apart from recording the chunk entries and processing times (making allowance for the extra time spent obeying the monitoring code itself), Perm enters a further routine which can note the current code page. In this way, any points on the average working set graph can be found (valid because the M-format module is the same size as the normal compiled one would have been). Use of 'page monitoring' increases processing time still more - about double in the tests made.

At the end of each run (when the instruction stop or endofprogram is reached), the similarity array and other reference data is written to the chunk information file (if this has not been specified, the results are simply lost), or merged with any information there from previous runs. If the similarity array embraces all the chunks, merging consists simply of adding on the current array; but if not (section IV.4), the similarity subset on the file may not be the same as that in the current run. In this case, a valid similarity array may only exist for the intersection of the chunk subsets (see Appendix B). The problems which then arise may be visible to the user through the increased time of the

file updating process.

Fig.5-5a shows the end of the printer O/P from a run of the small demonstration program. The similarity interval is 10ms. (shown as 1000, since the unit is 10 microseconds). The only output here from the monitoring Perm is the final line, giving the number of similarity intervals. The remainder, after the 'program ends' message, is output by the page-monitoring routines. These have recorded page references in intervals of lengths between one quarter and four times the similarity interval. The 'total faults' column gives the number of pages accessed in all processing intervals of each length; thus mean working set sizes can be calculated. For example, in this run the mean WS size for the similarity interval itself (1000) is 217/190 = 1.14. Normally the output from several runs will be used to evaluate the results. We also have the frequency (number of similarity intervals) with which each page is accessed. Fig. 5-5b shows similar output from a large program (program S - see next chapter).

If required, important parts of the chunk information file can be printed. An example after two runs of our test program is shown in figs. 5-6, 5-7. The quantity 'selectno' is the size of the similarity array if this has been restricted to less than the number of chunks. It was here deliberately made very small, at 18 small enough to ensure partial restructuring was necessary (for demonstration purposes). This maximum allowable length of side of the array is determined at compilation by an initialised variable in the compiler (normally 80, at present) - it could be made a data parameter to the scheme if necessary.

For each chunk, its length in bytes is given, for use by the

```
S     A 9 5
H     A Q 4
D     J 7
C     J 8 7 5 3
                    1 NT


S     J 10 8 4
H     K 10 9 2
D     K 9
C     A 10 9
                    NO BID


*****PROGRAM ENDS*****

INT. LENGTH    NO. INTS    TOTAL FAULTS
      250          728          751
      500          380          408
     1000          190          217
     2000           95          122
     4000           47           74

PAGE  ACCESS FREQUENCY (INTLENGTH=  1000)
  1          190
  2           27

INTERVALS  190
```

a)

```
INT. LENGTH    NO. INTS    TOTAL FAULTS
      625          877         2558
     1250          439         1707
     2500          219         1122
     5000          109          719
    10000           54          416

PAGE  ACCESS FREQUENCY (INTLENGTH=  2500)
  1           51
  2          171
  3           72
  4           24
  5           51
  6           77
  7           22
  8           23
  9           63
 10           57
 11           92
 12           93
 13           95
 14           88
 15          141
 16            2

INTERVALS  219
```

b)

Fig 5-5 Output from Monitoring Routines

DIT ASSOCIATED FILE

| | |
|---|---|
| UNS SO FAR | 2 |
| O OF INTERVALS | 404 |
| ELECTNO | 18 |
| O OF RECHUNKINGS | 0 |
| O OF CHUNKS | 27 |
| ILE LENGTH | 8 |

| HUNK | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| HUNKLENGTH | 224 | 228 | 362 | 218 | 358 | 276 | 306 | 370 | 148 | 160 | 92 | 152 | 88 | 292 | 30 | 22 | 120 | 150 | 108 | 24 |
| RANCHTO | 7 | 0 | 19 | 0 | 30 | 32 | 40 | 0 | 0 | 59 | 0 | 0 | 0 | 0 | 101 | 102 | 113 | 102 | 116 | 102 |
| RANSLATE | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | . | 13 | 14 | 15 | . | . | . | . |
| REQ OF USE | 2 | 2 | 115 | 116 | 97 | 209 | 204 | 226 | 2 | 182 | 59 | 11 | 8 | 63 | 31 | 40 | 10 | 12 | 18 | 0 |

| HUNK | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
|---|---|---|---|---|---|---|---|
| HUNKLENGTH | 74 | 30 | 60 | 42 | 74 | 100 | 104 |
| RANCHTO | 101 | 135 | 129 | 147 | 152 | 147 | 0 |
| RANSLATE | . | . | . | . | . | . | 16 |
| REQ OF USE | 7 | 3 | 5 | 2 | 5 | 0 | 58 |

Fig 5-6 Chunk Information File

LEADING PARTS OF CHUNKING ARRAY

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | | | 115 | 68 | 9 | 0 | 0 | 2 | 1 | 79 | 40 | 6 | 9 | 21 | 29 | 48 | 0 | 0 |
| 4 | | | | 116 | 57 | 19 | 0 | 0 | 0 | 80 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| 5 | | | | | 97 | 59 | 35 | 27 | 0 | 58 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | | | | | | 209 | 183 | 177 | 0 | 27 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 |
| 7 | | | | | | | 204 | 196 | 0 | 26 | 2 | 1 | 25 | 2 | 1 | 0 | 0 | 0 |
| 8 | | | | | | | | 226 | 0 | 53 | 18 | 3 | 53 | 10 | 13 | 9 | 0 | 0 |
| 9 | | | | | | | | | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | | | | | | | | | | 182 | 59 | 11 | 63 | 31 | 40 | 58 | 0 | 0 |
| 11 | | | | | | | | | | | 59 | 11 | 28 | 31 | 40 | 50 | 0 | 0 |
| 12 | | | | | | | | | | | | 11 | 5 | 2 | 0 | 9 | 0 | 0 |
| 13 | | | | | | | | | | | | | 63 | 16 | 17 | 19 | 0 | 0 |
| 14 | | | | | | | | | | | | | | 31 | 21 | 27 | 0 | 0 |
| 15 | | | | | | | | | | | | | | | 40 | 33 | 0 | 0 |
| 16 | | | | | | | | | | | | | | | | 58 | 0 | 0 |
| 17 | | | | | | | | | | | | | | | | | 0 | 0 |
| 18 | | | | | | | | | | | | | | | | | | 0 |

Fig 5-7 Similarity Array

restructuring process. The line 'branchto' will be explained in appendix B. 'Translate', only present if 'selectno' has restricted the array, selects the subset of chunks in the similarity array - thus the 13th. row and column of the latter refers to chunk 14. Although a similarity array of size 18 was allowed, the two subsets of these two runs were not the same, and data on only 16 chunks could be kept in the similarity array. The last line gives the total frequency of use (in similarity intervals) of each chunk in the runs to date. Fig.5-7 shows the complete similarity array. (Normally only the 20x20 submatrices centred on the principal diagonal are printed. Since here 'selectno' is less than 20, we obtain the whole array). As an example, note a common but unfortunate situation, from the restructuring point of view. There is a high similarity between chunks 3 and 4, and between 3 and 27 (translated to 16 in the array), but chunks 4 and 27 have only two intervals in common. Chunk 27 is always shortly followed by 3 and this leads to chunk 4, but the latter is nearly always too distant from 27 to be in the same 10ms. time-slice.

## 6. RESTRUCTURING

When a sequence of runs is judged sufficiently representative, the user calls the restructuring program. This produces a restructured object file - still in M-format - and a new chunk information file. Normally the total number of chunks will have decreased, some having merged during the clustering process (see IV.4). If no chunks outside the similarity subset were referenced at all in the last series of runs (or the array embraced all the chunks anyway), a final restructuring will be performed.

This means that the clustering process proceeds to completion, and each group of chunks is output from the start of a page boundary, instead of simply following one after the other, as when a partial clustering has taken place. Note that even a final clustering produces an object file in M-format; this is necessary at present in order that the page monitoring routines can be used to investigate final paging behaviour. A conversion program is used to overwrite the monitoring branches (tables in the chunk information file are required), to produce an N-format file.

With a large module, the normal course will be to repeat the process of run and restructure until the scheme produces a final restructuring. The paging behaviour will improve slightly each time - the grouping of chunks although not correct with respect to page boundaries, will have some effect. If a restructured version is required quickly, one can demand a 'final restructuring' any time, and produce an N-format file from this. (This could be used for temporary service purposes while continuing the optimisation of the original version). The effect of premature final restructuring is reported in the next chapter.

The restructuring program is in two distinct phases: (1) the clustering algorithm, which works out the new chunk groupings, and (2) the section which works out chunk orderings and generates the new files. (See appendix B for some of the problems which arise.) Fig.5-8 shows output from the first restructuring - from the array of fig.5-7 - of the bridge-hand program. Two groups of chunks have formed; there are only 17 chunks in the new object module (and the new chunk information file). Note that the actual chunk numbers are given under the 'old chunks' heading, not the translated

```
RECHUNKING ROUTINE

INTERMEDIATE RECHUNKING

NEWCHUNK   OLD CHUNKS
        1          1
        2          2
        3          3    4   10    5   11   27   16   15
        4          6    7    8   14
        5          9
        6         12
        7         13
        8         17
        9         18
       10         19
       11         20
       12         21
       13         22
       14         23
       15         24
       16         25
       17         26
```

```
****      0        1   224

****    224        2   228

****    452        3   354
****    806        4   218
****   1024        5   358
****   1382       10   160
****   1542       27   104
****   1648       11    92
****   1740       15    22
****   1762       16    22

****   1784       14   292
****   2078        6   268
****   2346        7   298
****   2644        8   370

****   3014        9   148

****   3164       12   152

****   3316       13    88

****   3404       17   120
```

Fig 5-8  Output from Restructuring Program

```
RECHUNKING ROUTINE

FINAL RECHUNKING

NEWCHUNK  OLD CHUNKS
       1          1   2   5   3   4   6   9  10  14  12   7
       2         11  17
```

```
****      0        1  224
****    224        2  228
****    452        3 1332
****   1784        4 1230
****   3014        5  148
****   3162       10  108
****   3270       12   74
****   3344        6  152
****   3496        9  150
****   3648        7   88
****   3736        8  120
****   3856       13   22
****   3878       14   52
****   3930       15   42
****   3972       16   74

****   4098       11   24
****   4122       17  100
```

a)

```
S     A K 10 7 2
H     J 6 3
D     10 2
C     7 6 2
                    NO BID


*****PROGRAM ENDS*****

INT. LENGTH    NO. INTS    TOTAL FAULTS
      250          1630          1631
      500           855           856
     1000           427           428
     2000           213           214
     4000           106           107

PAGE  ACCESS FREQUENCY (INTLENGTH=  1000)
  1         427
  2           1

INTERVALS  427
```

b)

Fig 5-9  Final Restructuring and Run

versions which specify similarity array positions. The three columns of figures with asterisks show the new addresses of the chunks - see appendix B. The new object module was run, and a final restructuring performed (necessarily, since the number of chunks is less than 'selectno'); fig.5-9a shows this (the first line of chunks is cut off at the right hand side).

We notice that the only achievement of restructuring this program was to remove two chunks not used at all during the test runs (originally chunks 20 and 26), the remainder of the program being packed into a single page. Fig.5-9b shows the end of a long run of the final M-format module; during this run, (original) chunk 20 happened to be entered once giving the result shown.

Obviously for service use, the conversion program is used to obtain a final M-format file.

# VI RESULTS AND CONCLUSIONS

## 1. EXPERIMENTAL DATA

This chapter describes the results of restructuring four IMP modules, with code sizes between 4 and 16 pages. The only changes made to the source modules as received from their authors were the trivial ones made necessary by the restrictions in the rather out-of-date IMP compiler on which the restructuring scheme is based. Some details of the modules (referred to as P,Q,R,S) are as follows.

| | description | code-size (words) 1 page = 1024 wds | no. of chunks |
|---|---|---|---|
| P (main-program) | Generator of syntax-tables for Q (below) from a phrase-structure grammar | 4150 | 59 |
| Q (main-program) | A syntax-analyser for IMP, incorporating a syntactic macro-scheme | 8300 | 268 |
| R (main-program) | Part of an interpreter for a (simulated) on-line symbol manipulation language. | 13050 | 292 |
| S (external rtn.) | Phase 2 of the chunking IMP compiler. | 15500 | 334 |

The number of chunks is determined not only by the length of the program but by other factors: numbers of jumps, programming style (see V.4). Thus although the choice procedure yields an average of about 25 chunks per page, this can vary greatly between programs. In particular P has far less chunks than would be expected, Q far more.

For each module, the maximum allowed size (determined at compilation time) of similarity array was 80. Thus only for

program P could a complete array be generated at run-time, giving the necessity for only one restructuring. For program S the number of chunks would have made a larger similarity array advisable in normal circumstances.

## 2. EXPERIMENTS

The main experiments consisted of performing the complete restructuring process on each program, aimed at minimising the mean working-set size (code only) over 25 ms. periods. Points on the working-set curves were found before and after restructuring; these were for the multiples $\frac{1}{4}, \frac{1}{2}, 1, 2$ and 4 of the similarity interval. Figs. 6-1 to 6-4 show the two graphs obtained for each module. (Note the difference in the scales of the vertical axes in the four figures.)

The table below gives the reductions (as a fraction of the original) in the average WS size, for the similarity interval, and also for $6\frac{1}{4}$ and 100 ms. intervals.

| | P | Q | R | S |
|---|---|---|---|---|
| 25ms. | 0.60 | 0.40 | 0.48 | 0.42 |
| $6\frac{1}{4}$ms. | 0.53 | 0.25 | 0.36 | 0.32 |
| 100ms. | 0.58 | 0.44 | 0.51 | 0.44 |
| no. restructurings | 1 | 4 | 4 | 6 |
| no. similarity ints. | 1100 | 700 | 470 | 640 |

For 25ms. intervals, there is a saving in mean WS size of at least 40% in all the programs. For longer intervals the variation is fairly small but the reduction worsens for very short time-intervals. This is because the WS sizes are very small and the

page size becomes dominant; no restructuring can cause less than one page to be referenced in an interval.

Below each graph in figs. 6-1 to 6-4 is shown the number of similarity intervals in which each page is accessed before an after restructuring (some were not referenced at all afterwards - see below). Note that in programs P and R the most used page after restructuring is accessed less times than the most used before.

Data

All the programs were highly data-dependent, and considerable effort was required to establish data which was considered fairly representative of the presumed use of each program. Program P had a single set of data (giving a long run), programs Q and S each had two different sets, while R had three. All the data was used in each monitoring phase of the restructuring process. The number of similarity intervals in the table above refers to the total number with all the sets of data. The variation in mean WS sizes in the same program for different sets of data were generally fairly small; the greatest difference was between the two sets of data of program Q (one contained a large number of macros and the other not). The results with the individual runs (initially, and after restructuring) are shown by the dotted lines in fig. 6-2.

One statistic which appeared not to be reflected in the results was the number of chunks not referenced at all during the runs of each program. This varied from the least in program P, almost wholly used, to the greatest in program Q, in which about one third of the program chunks were not referenced at all (this was despite the data for Q being considered representative - much of the unreferenced material dealt with the analysis of assembly-code

statements which may be incorporated in IMP programs, although would be rare). This contrasts with the fact that P gave the greatest reduction in WS size after restructuring, and Q the least of the four programs. The compression of programs simply by the removal of non-used chunks is not such a significant factor in the WS size reduction as might have been thought. This had accidental support from an early test of program Q when an error existed in the file-updating routine which led to a slightly invalid clustering (not detected by examination of the results), although all the unused chunks were still removed. The reduction in mean WS size was then only 15%.

Additional Tests

Figs. 6-5a and b show the effect of restructuring with different similarity intervals. Apart from the 25ms. tests, program Q was restructured using a 10ms. interval, and program S using a 200ms interval; the working set graphs of the results are shown as dotted lines. In each case the graph differs little from the former 25ms. restructuring (continuous lines), but the effect discussed in III.5 and shown in fig.3-4, is quite visible.

Fig. 6-6 demonstrates premature final restructuring with program R, the worst behaved, although not the largest, program. The results were obtained with a single set of test data. After each run, apart from partial restructuring and continuing with the test, a final restructuring (see V.6) was produced and a WS curve obtained from the result. Note the surprising fact that the greatest improvement is obtained at the last restructuring, implying that the positions of lesser used chunks in the program are quite significant.

## 3. CONCLUSIONS

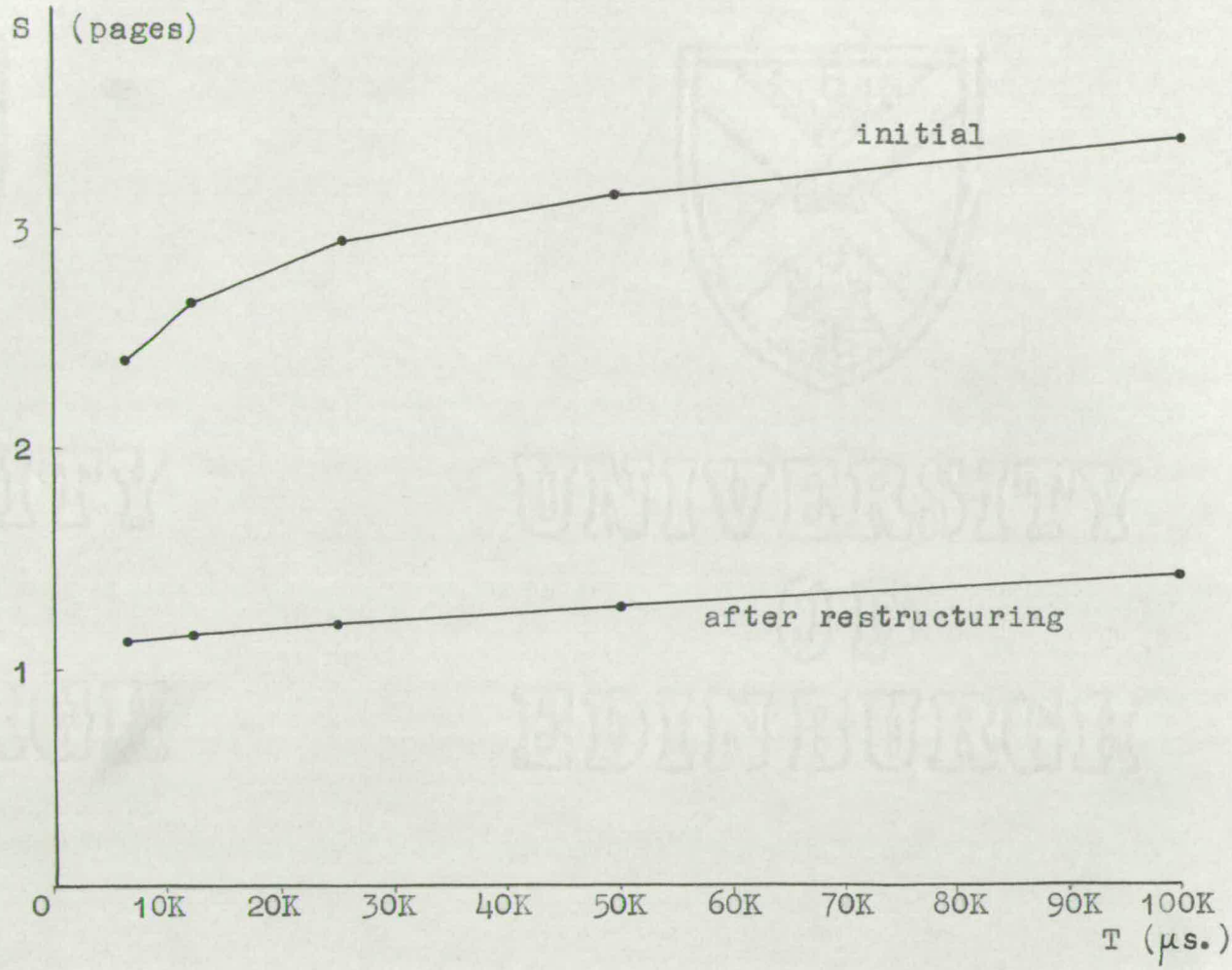
Reorganising parts of a program to improve its paging behaviour is essentially an engineering problem; the result may bear little relation to the conceptual logical structure necessary for good design, construction and understanding of the program. Thus nothing can be achieved by attempting to examine the chunk groupings produced in such large programs as those just considered. This is an argument for restructuring being an automatic process with which the programmer would not wish to be concerned.

A very large degree of improvement is obtainable, although it must be borne in mind that our results are reductions of mean WS size over code only, and in the programs tested data reference might have been expected to contribute as much or more to the WS size as the code. The effect on page-faulting in a genuine operating system would depend on the scheduling algorithm; one would expect that normally the reduction would be at least as good as that of the WS size, but it could be much more. Thus we have not attempted to simulate behaviour under, say, a restricted store scheduler (I.4) and claimed (with judicious choice of store size) a 95% reduction in page-faults; such experiments would be misleading and a waste of time.

The severest problems are practical ones: the large quantity of data, the cost of obtaining reference information from the program, etc. Unless these problems can be solved, discussion of theoretical techniques for restructuring programs becomes quite academic. It is hoped that the methods described in this thesis are sufficiently practical and produce sufficiently large improvements to be worthy of consideration in the improvement of the behaviour of large and frequently used programs. The

clustering techniques at the basis of the methods appear to be very effective in this field and may have more general application in the area of program structuring: organising, say, the components of a system for efficient movement within a storage hierarchy.

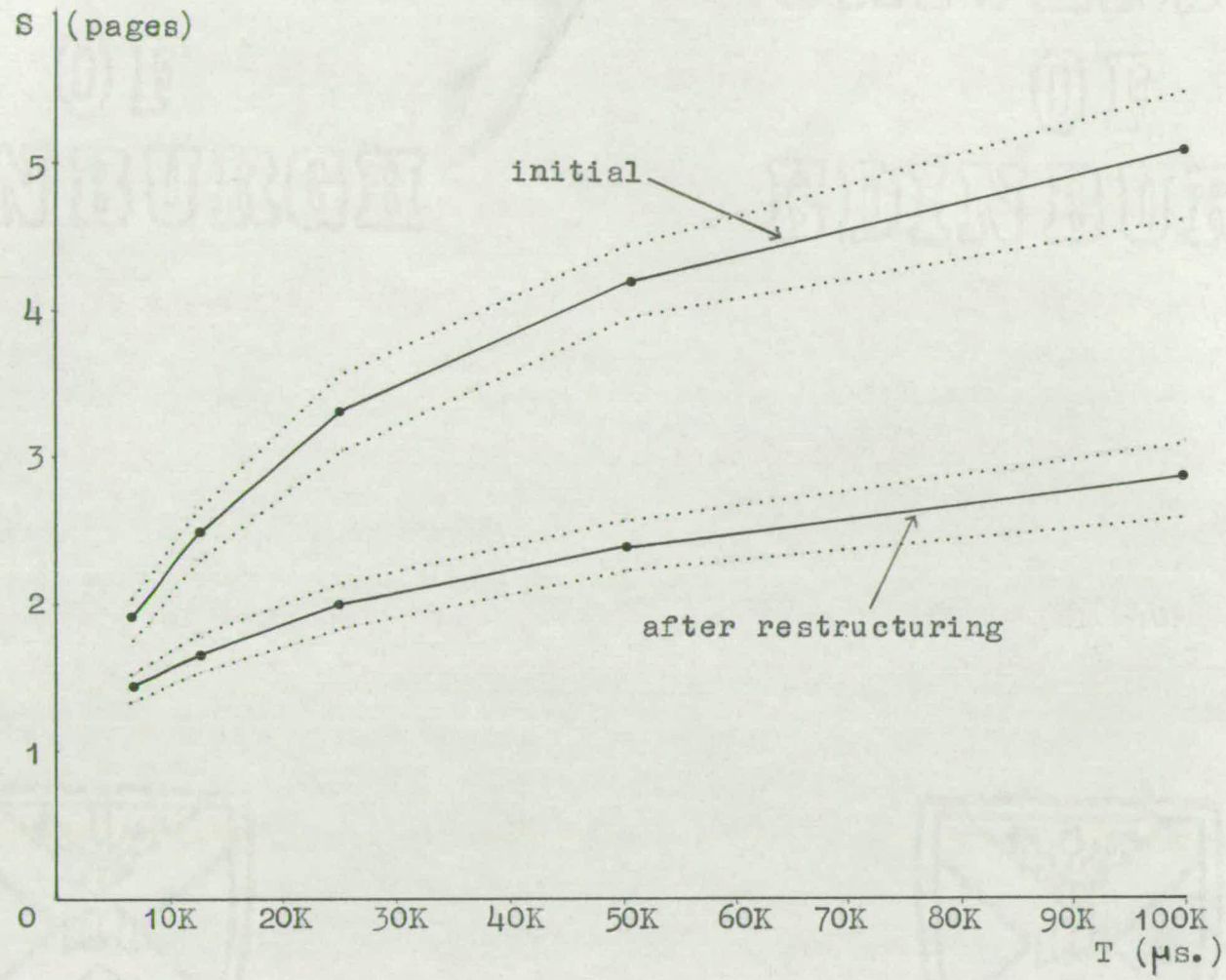
Randell and Kuehner (ref.2) have written: 'In the present state-of-the-art, any but the most minor attempts at re-packing are probably best regarded as last-ditch efforts at recovering from inadequate hardware, operating system strategies, and/or programming style.' It is hoped that this thesis has advanced the 'state-of-the-art', if only in small measure enough to cause some modification of the above statement.




Spread of (25ms. interval) accesses amongst pages

| Before restructuring | After restructuring |
|---|---|
| 703 | 21 |
| 598 | 583 |
| 581 | 602 |
| 810 | 108 |
| 543 | 4 |

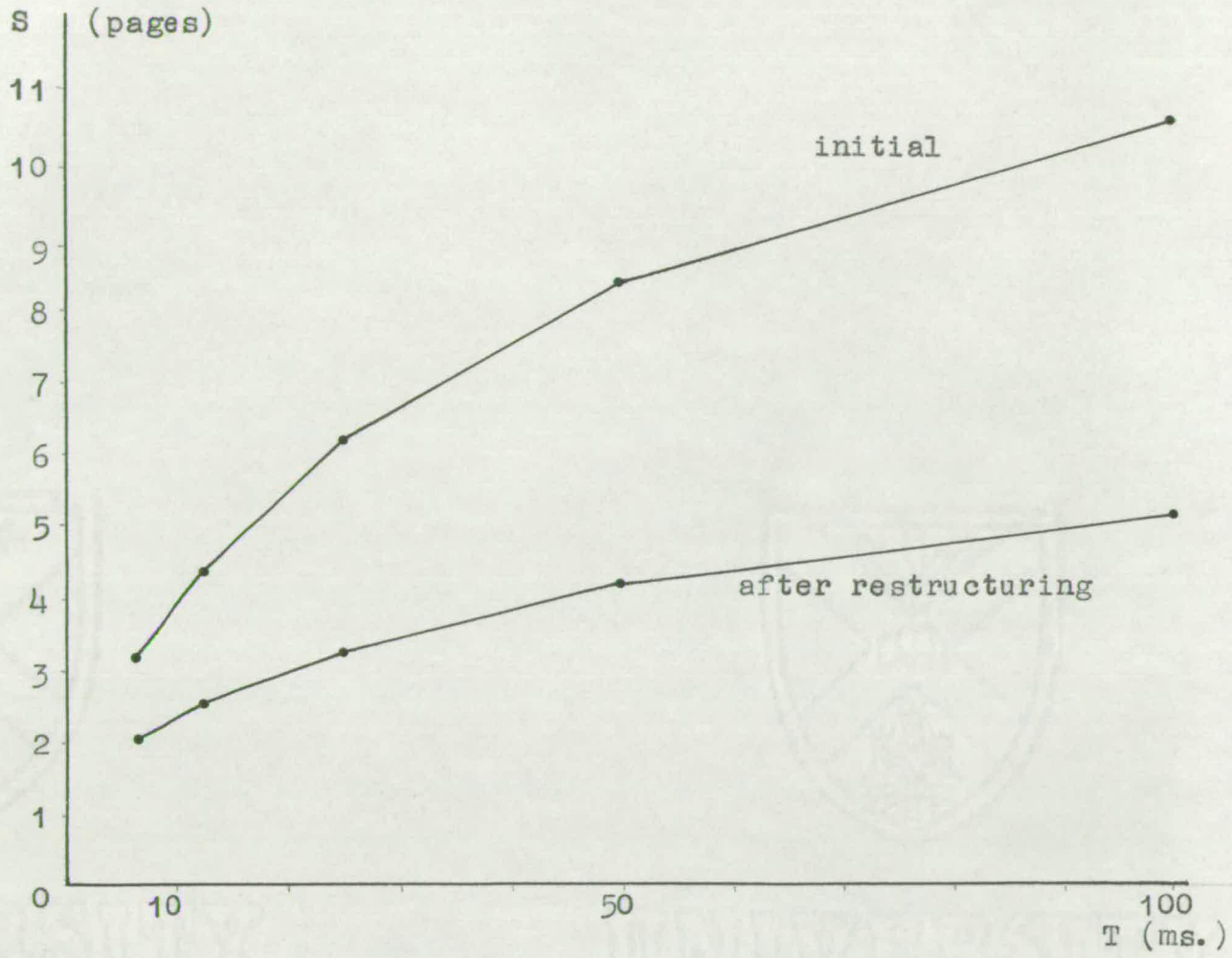
Fig 6-1 Results of restructuring program P




Spread of (25ms. intervals) accesses amongst pages

| Before restructuring | After restructuring |
|---|---|
| 587 | 420 |
| 623 | 662 |
| 379 | 175 |
| 74 | 76 |
| 105 | 51 |
| 250 | 28 |
| 12 | 0 |
| 299 | 0 |
| 45 | 0 |

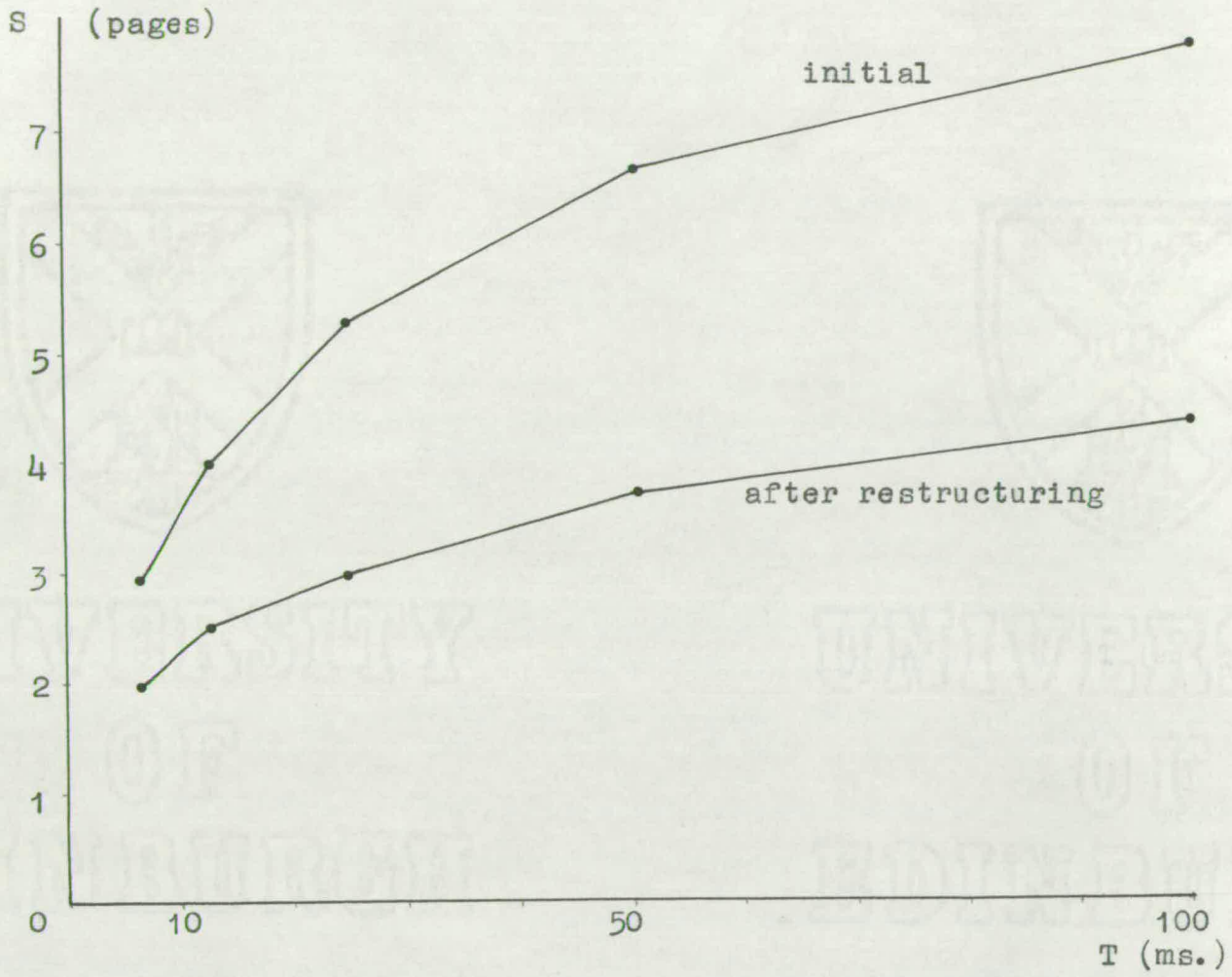
Fig 6-2 Results of restructuring program Q




Spread of (25ms. intervals) accesses amongst pages

| Before restructuring | After restructuring |
|---|---|
| 164 | 359 |
| 422 | 101 |
| 330 | 411 |
| 175 | 161 |
| 201 | 320 |
| 239 | 76 |
| 227 | 48 |
| 101 | 70 |
| 117 | 63 |
| 211 | 7 |
| 234 | 0 |
| 211 | 0 |
| 176 | 0 |

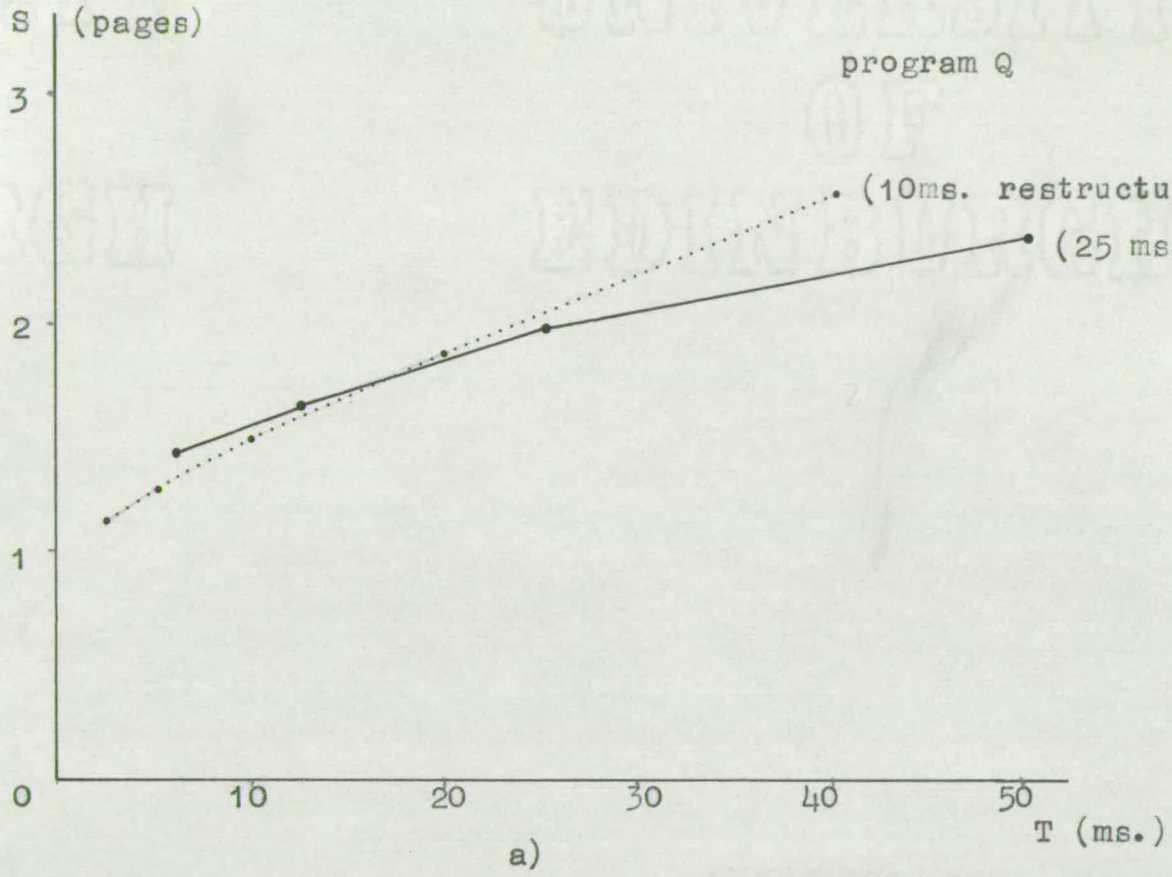
Fig 6-3 Results of restructuring program R




Spread of (25ms. intervals) accesses amongst pages

| Before restructuring | After restructuring |
|---|---|
| 138 | 216 |
| 484 | 65 |
| 241 | 314 |
| 65 | 478 |
| 181 | 115 |
| 247 | 297 |
| 68 | 223 |
| 62 | 90 |
| 206 | 31 |
| 191 | 31 |
| 272 | 21 |
| 259 | 37 |
| 267 | 5 |
| 263 | 0 |
| 419 | 0 |
| 4 | 0 |

Fig 6-4 Results of restructuring program S



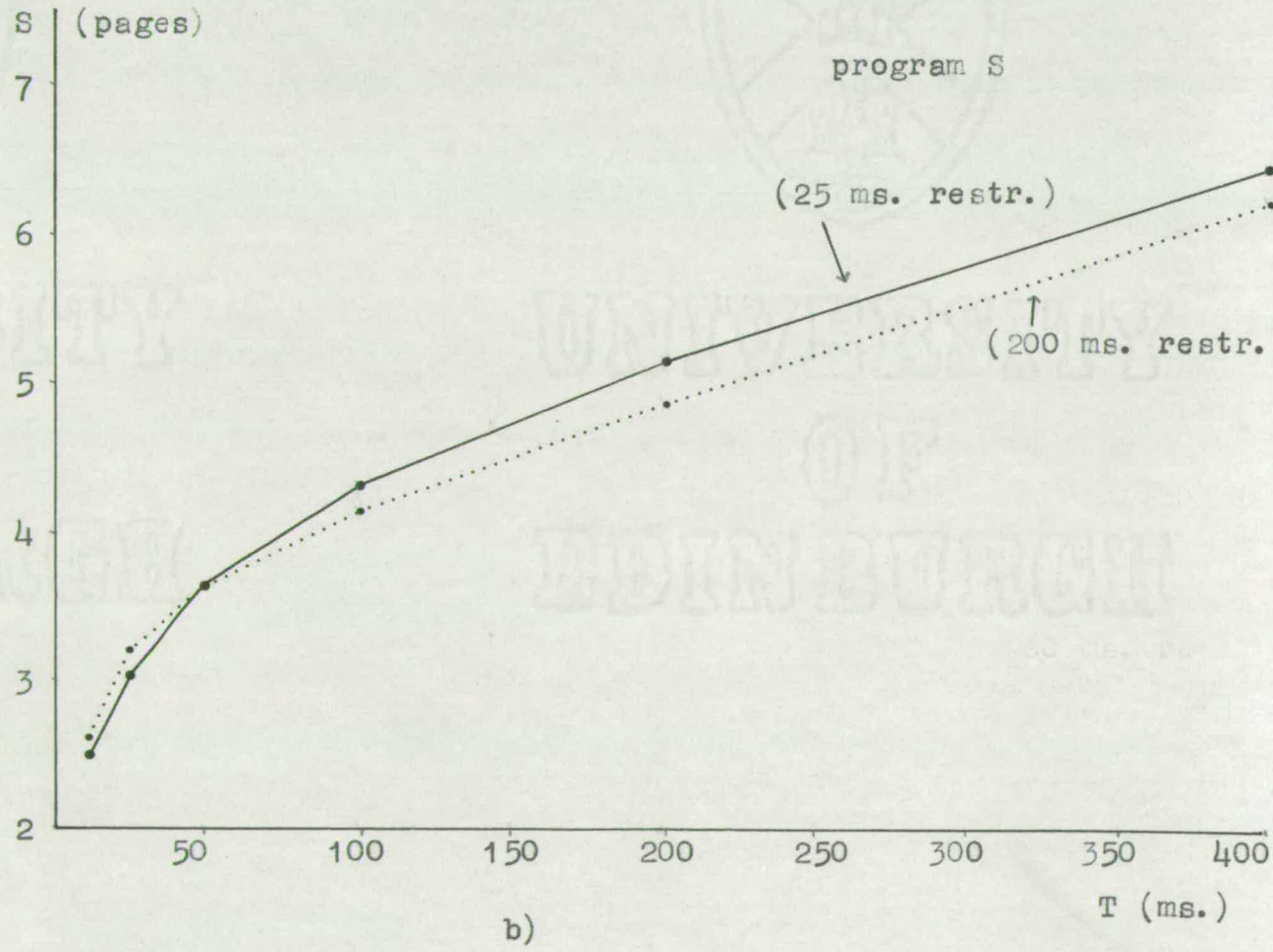




Fig 6-5 Restructuring to different lengths of WS interval

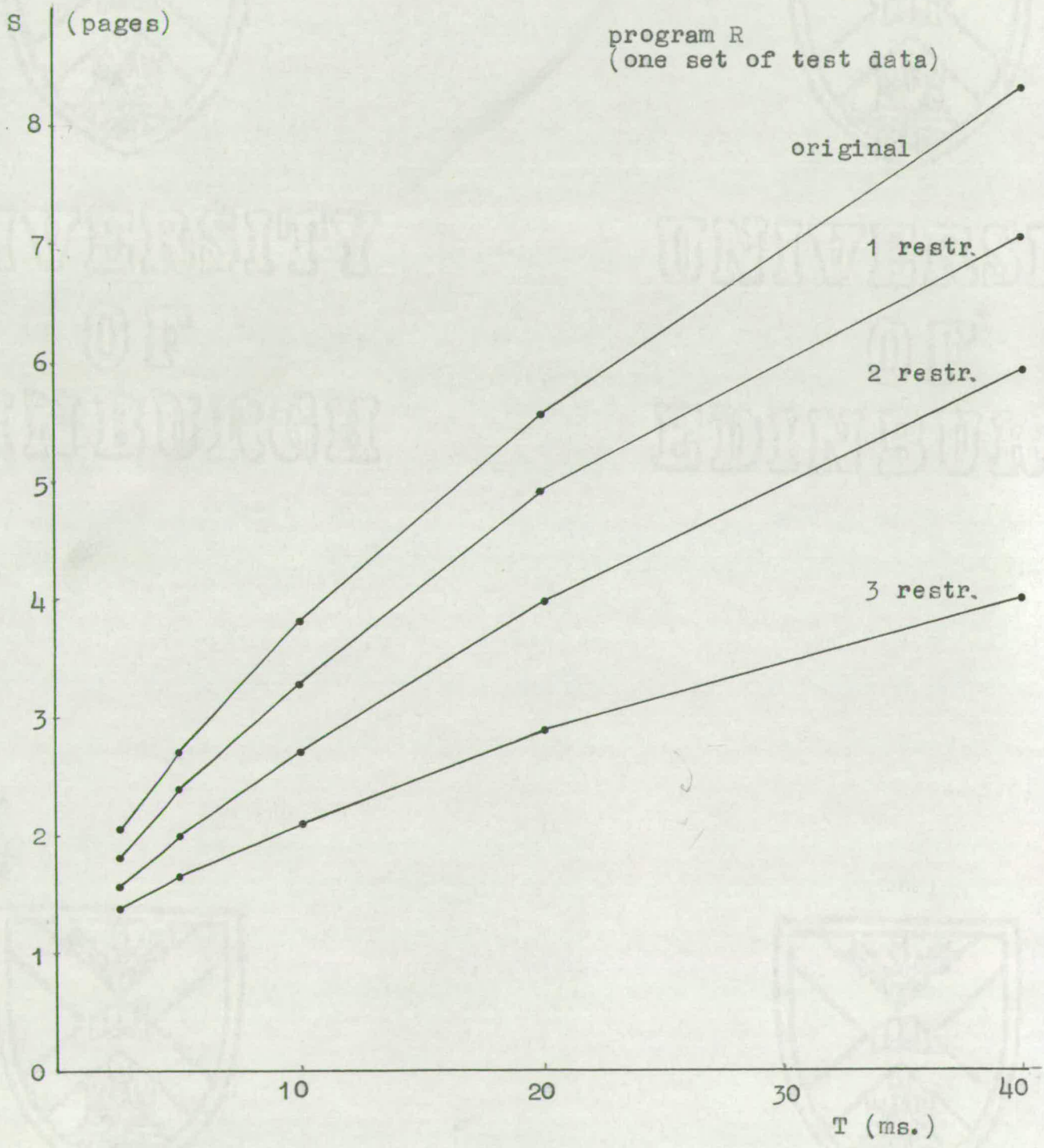



Fig 6-6 Effect of premature final restructuring

REFERENCES

In order of citation.


1. DENNIS, J.B. Segmentation and the design of multiprogrammed computer systems. J. ACM 12,4 (Oct. 65), 589-602.
2. RANDELL, B. AND KUEHNER, C.J. Dynamic storage allocation systems. Comm. ACM 11,5 (May 68), 297-305.
3. KILBURN, T., EDWARDS, D.B.G., LANIGAN, M.J. AND SUMNER, F.H. One-level storage system. IRE Trans. EC 11,2 (Apr. 62), 223-235.
4. SHEMER, J.E. AND SHIPPEY, G.A. Statistical analysis of paged and segmented computer systems. IEEE Trans. EC 15,6 (Dec.66), 855-863.
5. WEIZER, N. AND OPPENHEIMER, G. Virtual memory management in a paging environment. AFIPS Proc. SJCC, Vol. 34 (1969), 249-256.
6. DENNING, P.J. The working set model for paging behaviour. Comm. ACM 11,5 (May 68), 323-333.
7. DENNING, P.J. Thrashing: its causes and prevention. AFIPS Proc. FJCC, Vol.33 (1968), 915-922.
8. GAVER, D.P. Probability models for multiprogramming computer systems. J. ACM 14,3 (July 67), 423-438.
9. FINE, G.H., JACKSON, C.W. AND MCISAAC, P.V. Dynamic program behaviour under paging. Proc. ACM 21st National Meeting 1966.
10. BRAWN, BARBARA S., AND GUSTAVSON, FRANCES G. Program behaviour in a paging environment. AFIPS Proc. FJCC, Vol. 33 (1968), 1019-1032.

11. COFFMAN, E.G. AND VARIAN L.C. Further experimental data on the behaviour of programs in a paging environment. Comm. ACM 11,7 (July 68), 471-474.

12. FREIBERGS, I.F. The dynamic behaviour of programs. AFIPS Proc. FJCC, Vol. 33 (1968), 1163-1167.

13. O'NEILL, R.W. Experience using a time-shared multi-programming system with dynamic address relocation hardware. AFIPS Proc. SJCC, Vol. 30 (1967), 611-621.

14. BELADY, L.A., NELSON, R.A AND SHEDLER, G.S. An anomaly in space-time characteristics of certain programs running in a paging machine. Comm. ACM 12,6 (June 69), 349-353.

15. BELADY, L.A. A study of replacement algorithms for a virtual storage computer. IBM Systems J. 5,2 (1966), 78-101.

16. DENNING, P.J. Resource allocation in multi-process computer systems. (Ph.D. thesis) Massachusetts Institute of Technology, May 1968.

17. BELADY, L.A. AND KUEHNER, C.J. Dynamic space-sharing in computer systems. Comm. ACM 12,5 (May 69), 282-288.

18. KUEHNER, C.J. AND RANDELL, B. Demand paging in perspective. AFIPS Proc. FJCC, Vol. 33 (1968), 1011-1017.

19. McKELLAR, A.C. AND COFFMAN, E.G. The organization of matrices and matrix operations in a paged multiprogramming environment. Comm. ACM 12,3 (Mar. 68), 153-164.

20. COMEAU, L.W. A study of the effect of user program optimisation in a paging system. ACM Symposium on operating system principles, Gatlinburgh, Tennessee, 1967.

21. RAMAMOORTHY, C.V. The analytic design of a dynamic look ahead and program segmenting system for multiprogrammed computers. Proc. ACM 21st National Meeting 1966.

22. MARIMONT, ROSALIND B. Application of graphs and Boolean matrices to computer programming. Siam Review 2,4 (Oct. 60), 259-268.

23. KRAL, J. One way of estimating frequencies of jumps in a program. Comm. ACM 11,7 (July 68), 475-480.

24. LOWE, T.C. Analysis of Boolean program models for time-shared paged environments. Comm. ACM 12,4, (Apr. 69), 199-205.

25. LOWE, T.C. Automatic segmentation of cyclic program structures based on connectivity and processor timing. Comm. ACM 13,1 (Jan. 70), 3-6.

26. FOLEY, J.D. A Markovian model of the University of Michigan executive system. Comm. ACM 10.9 (Sept. 67), 584-588.

27. FELLER, W. Introduction to probability theory and its applications, Vol. 1. John Wiley & Sons (New York), 1950.

28. GOMORY, R.E. The travelling salesman problem. Proc. IBM Scientific Computing Symposium on Combinatorial Problems (1964), 93-117.

29. SOKAL, R.R. AND SNEATH, P.H.A. Principles of numerical taxonomy. W.H. Freeman & Co. (San Francisco and London), 1963.

30. BONNER, R.E. On some clustering techniques. IBM J. Res. and Dev. 8,1 (Jan. 64), 22-32.

31. MATTSON, R.L. AND DAMMAN, J.E. A technique for determining and coding subclasses in pattern recognition problems. IBM J. Res. and Dev. 9,4 (July 65), 294-302.

32. NEEDHAM, R.M. AND PARKER-RHODES, A.F. The theory of clumps. Cambridge Language Research Unit Report M.L.126 (1960).

33. NEEDHAM, R.M. The theory of clumps II. Cambridge Language Research Unit Report M.L.139 (1961).

34. WALKER, J.G. AND WHITFIELD H. (ED.) EMAS System Reference Manual. Edinburgh University Department of Computer Science and ICL (1970).

35. WALKER, J.G. AND WHITFIELD H. (ED.) EMAS Primary Subsystem Reference Manual. Edinburgh University Department of Computer Science and ICL (1970).

# APPENDIX A. A BRIEF STUDY OF PROGRAM BEHAVIOUR ON THE KDF9

## 1. INTRODUCTION

In order to obtain data for preliminary development and testing of effective restructuring algorithms based on the ideas of chapter III, an initial study was made of the reference behaviour of some programs written in Atlas Autocode (A.A) for an English Electric KDF9 computer. This was a single-address machine with a memory of 16K 48-bit words, only the first 8K being available for instructions. It was non paged, and the programs tested had thus in no way been organised to run in a paging environment.

## 2. COLLECTION OF REFERENCE DATA

Most of the data was obtained using a modification of an already available KDF9 interpreter which had been previously written by Mr. T. Head. This was itself written in A.A., and when compiled with another A.A. program caused the latter to be obeyed interpretively - the flow of address references could thus be easily traced. To simulate chunking (see chapter III), grids were imposed separately on the code and data, dividing them into equal size blocks. The interpreter was made to produce (on magnetic tape) a stream of reference statistics; each consisting of the number of instructions obeyed so far (representing time) followed by a block number. Successive accesses to the same block were not recorded. On the code a block size of 50 words was normally imposed. This was rather large and also the arbitrary divisions imposed by the grid would normally in no way coincide with natural structure; in fact a division might easily pass through a single instruction. However it was supposed that although this would

substantially reduce the degree of improvement obtainable by restructuring, it would not invalidate the testing and development of the restructuring methods.

Immediate problems which arose if a reasonable length of run of the program was to be monitored were:

a) an immense quantity of data was produced,

b) the interpreter was very time-consuming, partly due to a).

It was necessary to reduce the amount of data in a way which would not affect the results of such experiments as would be performed on it. These would consist mainly of forming similarity arrays and investigating working set sizes (see chapters I,III), on the basis of intervals of various lengths starting from about 1000 instructions. This being large compared with the time between different block references, the following reductions in the quantity of data had a negligible effect on results.

a) As far as repeated accesses to the same block were concerned, data and code were treated separately. Thus if two successive references to the same code (say) block were separated by a data chunk reference, the code chunk would not appear the second time.

b) The 'time' was only output every 128 instructions. To identify this it was preceded by zero; instruction chunks were given positive numbers, data, negative.

c) Within a 128 instruction period, any data sequence of the form $d_1\ d_2\ d_1\ d_2 \ldots d_1\ d_2$ (ignoring intervening instruction chunks) resulted simply in the output $d_1\ d_2$. Such sequences were very common in the programs examined, owing to the common situation of references to parts of a large array being interspersed with access

to scalars at the base of the data stack.

d) Since rearrangement was to be performed only on the code, the imposition of 50 word chunks was confined to this; both the data and 'Perm' (the permanent or slave routines which the compiler linked to every A.A. program) being blocked into the minimum page size which would be simulated (250 words). This meant that page reference data could still be obtained for the program as a whole.

The interpretation speed was then processor dependent and was about 50000 instructions per minute; this number of instructions producing on average about 4000 words of data. Even with the fairly limited machine time available, it now became quite feasible to interpret a million instructions during a run (although even this only represents a few seconds of the program's actual run-time.)

In one case (program D below), data was gathered not by the interpreter but by instructions hand written into the A.A. program - these monitored the flow of reference only within the code. Chunk boundaries could then be chosen at natural dividing points. In practice most of the chunks were taken simply as the subroutines of the program: the insertion of monitoring instructions to trap any transfers of control between chunks was then a straightforward procedure. The chunk numbers and corresponding CPU time (in place of instruction count) were written up to magnetic tape in the same format as that produced by the interpreter.

Once a data tape was produced, all investigations were made on this, no further reference to the program being made.

P1 and P2 (simulate program run)

P1 simultaneously simulated runs of the program under each of a set of given time-slices with a single given store allocation. Every chunk had an associated page number: part of the data to P1. Thus each chunk on the data stream gives rise to a page reference, and the paging behaviour (under some simulated page size) of the program can be studied either in its original form or with any redistribution of chunks into pages.

The simulated store associated with each length of time-slice is represented by a vector, with an entry for each page to indicate whether it is in or out of store; if in, a note of the time of the most recent reference is kept. Within a time-slice, pages were loaded only when referenced. Once the allocated store was full, subsequent references external to this store would result in a page being unloaded according to the 'least recently used' strategy. At the end of a time-slice, the whole store was cleared. For each length of time-slice, a record was kept of the following.

a) The total number of page-faults (references to out-of-store pages).

b) If required, the proportion of the total number of intervals in which exactly i faults occurred, for all i from 1 to some given n.

c) The number of intervals in which more distinct pages were referenced than the amount of store allocated (i.e. unloading had to take place).

By taking a very large store allocation, it can be ensured that

no unloading can occur during any time-slice. The average number of page-faults per interval then gives an approximation to the average working set size (section I.5) for the corresponding time-slice length.

If a large time-slice is taken, with a smaller store allocation, the restricted store behaviour can be studied, within each interval. (Only for very small store sizes will the result be significantly affected by the measures to reduce data described above.)

P2 was a variant of P1 written to give a more explicit examination of restricted store behaviour. No time-slicing was performed, but several store allocations could be considered simultaneously. The time when the store was first filled was recorded: the average fault rate after this time thus gave a measure of genuine restricted store behaviour. If required, P2 would also print out the number of periods between page-faults which were less than any specified quantity.

Timing: This depended largely on the amount of information required from one pass through the data stream. The elapsed times of the simulations of a million instructions all lay between 5 and 12 minutes.

Figs. A-1, A-2 give examples of printer O/P from P1 (the length of the time-slice is indicated by 'residence period'). Fig. A-3 gives an example from P2. In this last figure the fact that time is only recorded every 128 instructions results in the 'store filled' time being the same for store sizes 12,13 and 14; in fact they were of course slightly different. This arose from program B (see section 4).

P3 (produce similarity array)

This generated a similarity array (section III.7) for each of a set of interval lengths read in as data. The results were written up to magnetic tape. The average amount of store (in terms of the chunk unit) referenced per interval was recorded - representing the average working set size if a very small page size was taken, and obviously very much a lower bound to the possible average size obtainable after restructuring.

Later versions of this program (and P4 below) reduced the number of chunks by not including any which were not referenced at all during the run. A translation table was used to renumber chunks which were accessed. Before this was done, the largest program examined gave a similarity array too great for the available KDF9 store.

Timing: for the largest arrays an average of 4 minutes elapsed time.

The diagonal elements of the arrays were printed out in order: these represent the frequency of chunk use. Fig.A-4 shows an example of the printer O/P (cut off at the right hand side). The 'sections' message indicates where the array is written on magnetic tape.

P4 (restructure into pages)

P4 clustered the chunks into pages on the basis of a similarity array output by P3, the page size being data to P4. Many methods of clustering were tried, the reduction in the average working set size being the criterion of judgement of results. Chapter III gives some details and describes the algorithm finally adopted. The final chunk positions were both printed and written to magnetic

```
RESIDENCE PERIOD =   2500
SPACE AVAILABLE =  21

NO INTERVALS   187
PAGE FAULTS/INTERVAL   10.81


PROPN OF INTERVALS IN WHICH TOP FAULTS OCCUR  0.000

RESIDENCE PERIOD =   5000
SPACE AVAILABLE =  21

NO INTERVALS    94
PAGE FAULTS/INTERVAL   11.35


PROPN OF INTERVALS IN WHICH TOP FAULTS OCCUR  0.000

RESIDENCE PERIOD =  10000
SPACE AVAILABLE =  21

NO INTERVALS    47
PAGE FAULTS/INTERVAL   12.40


PROPN OF INTERVALS IN WHICH TOP FAULTS OCCUR  0.000

RESIDENCE PERIOD =  15000
SPACE AVAILABLE =  21

NO INTERVALS    32
PAGE FAULTS/INTERVAL   13.37


PROPN OF INTERVALS IN WHICH TOP FAULTS OCCUR  0.000

RESIDENCE PERIOD =  20000
SPACE AVAILABLE =  21

NO INTERVALS    24
PAGE FAULTS/INTERVAL   13.58


PROPN OF INTERVALS IN WHICH TOP FAULTS OCCUR  0.000

RESIDENCE PERIOD =  30000
SPACE AVAILABLE =  21

NO INTERVALS    16
PAGE FAULTS/INTERVAL   13.75


PROPN OF INTERVALS IN WHICH TOP FAULTS OCCUR  0.000
```

Fig A-1 Example of output from P1

```
RESIDENCE PERIOD =   5000
SPACE AVAILABLE =  25

NO INTERVALS   200
PAGE FAULTS/INTERVAL   11.74

DISTN OF PAGE FAULTS/INTERVAL
  1        0.000
  2        0.005
  3        0.010
  4        0.000
  5        0.000
  6        0.000
  7        0.050
  8        0.030
  9        0.150
 10        0.150
 11        0.115
 12        0.075
 13        0.115
 14        0.070
 15        0.105
 16        0.085
 17        0.035
 18        0.000
 19        0.005
 20        0.000

PROPN OF INTERVALS IN WHICH TOP FAULTS OCCUR  0.000

RESIDENCE PERIOD =  10000
SPACE AVAILABLE =  25

NO INTERVALS   100
PAGE FAULTS/INTERVAL   13.85

DISTN OF PAGE FAULTS/INTERVAL
  1        0.000
  2        0.000
  3        0.010
  4        0.000
  5        0.000
  6        0.000
  7        0.010
  8        0.000
  9        0.040
 10        0.090
 11        0.040
 12        0.070
 13        0.120
 14        0.140
 15        0.190
 16        0.150
 17        0.110
 18        0.010
 19        0.010
 20        0.010

PROPN OF INTERVALS IN WHICH TOP FAULTS OCCUR  0.000
```

Fig A-2  Example of output from P1

```
TESTTIME=  600000

STORESIZE=   15
STORE FILLED AT 191104
NO PAGE FAULTS  171
LAST FAULT AT  459776
NO INTS LESS THAN      5000      149
                       1000      144


STORESIZE=   14
STORE FILLED AT 190976
NO PAGE FAULTS  209
LAST FAULT AT  459776
NO INTS LESS THAN      5000      185
                       1000      172


STORESIZE=   13
STORE FILLED AT 190976
NO PAGE FAULTS  276
LAST FAULT AT  459776
NO INTS LESS THAN      5000      251
                       1000      235


STORESIZE=   12
STORE FILLED AT 190976
NO PAGE FAULTS  360
LAST FAULT AT  459776
NO INTS LESS THAN      5000      336
                       1000      318
```

Fig A-3  Example of output from P2

```
NO INTERVALS        46
SIZE/INTERVAL    6054.0
SECTIONS    366 -  373

USE FREQUENCY
    19    46    31    46    45    19    18    18    18    19     5     1
    22    20    19    19     4    20    17    16    12    12     2    11
    14     6     5     4     4     3     6     4     2     2     2     2


NO INTERVALS        23
SIZE/INTERVAL    6800.0
SECTIONS    374 -  381

USE FREQUENCY
    10    23    19    23    23    10     9     9     9    10     4     1
    14    13    13    13     4    13    13    11     9     9     2     9
     9     6     5     4     4     3     5     3     2     1     1     1
```

Fig A-4 Example of part of output from P3

```
PAGE     CHUNKS
   1        1  83  84  85  86   2   3   4  32  33        LENGTH
   2        5   6   8  34  35  76  80  81  82   7        LENGTH
   3        9  48  49  56  50  52  51  53  79        LENGTH   450
   4       10  54  58  62  57  66  61  55  29  63        LENGTH
   5       11  12  13  14  23  24  37  38  39  40        LENGTH
   6       15  16  47  60  59  74  75  91  90  36        LENGTH
   7       17  18  19  22  20  21  25  26  27  28        LENGTH
   8       30  31          LENGTH   100
   9       41  72  73  87  88  42  89  43  77  78        LENGTH
  10       44  45  46          LENGTH   150
  11       64  65          LENGTH   100
  12       67  68  69  70  71          LENGTH   250
```

Fig A-5 Example of part of output from P4

tape in a suitable form for input to P1 or P2.

Timing: processing time roughly proportional to the size of the array - about one minute for 40 chunks.

Fig.A-5 gives an example of printer O/P from P4.

## 4. INVESTIGATIONS AND RESULTS

The reference data from four A.A. programs was used to develop the restructuring methods. The way that data is presented to P1 permits a page size of any multiple of the largest chunk size to be simulated; however the results presented here assume a 500 (48-bit) word page size. Working set curves, each covering intervals between 2K and 50K instructions, were obtained before and after restructuring; some restricted store behaviour was also examined. The table gives details of the programs, assuming a 500 word page size. 'Perm' was about 1500 words, i.e. 3 pages, and was not considered part of the code for restructuring purposes.

| | no.code pages | no.(used) data pages | total size | instructions interpreted |
|---|---|---|---|---|
| A (numerical analysis) | 5 | 2 | 10 | ¼ million |
| B (simulated on-line interpreter) | 7 | 12 | 22 | ½ m. |
| C (2nd phase A.A. compiler) | 11 | 11 | 25 | 1 m. |
| D (1st phase A.A. compiler)[*] | 6 | - | - | (20 sec. CPU time) |

The results with the program in its initial form, and after restructuring using the clustering algorithm finally adopted, are shown in figs.A-6 to A-9. The restricted store behaviour where given refers to the average fault-rate once the allocated store is

[*] Data obtained by instructions added by hand - see section 2

first filled, and not over the whole program run.

Prog. A

Starting at any time, this uses nearly all its pages very rapidly, a sudden flattening of the average WS curve occurring between W=8 and W=9. This is reflected in the restricted store behaviour, the average fault rate changing from excessive to virtually zero with store allocations of 8 and 9. The effect of restructuring (to a time-slice of 10K) reduces the WS sizes by about a page and a half. This small reduction makes an enormous difference to the paging rate that results when the program is restricted to run in 7 or 8 pages. This is an indication of how unsatisfactory is restricted store behaviour as a summary of program reference pattern.

Prog. B

This program consisted of two clear phases, the first of which was quite compact, no more than 10 pages being referenced in any interval. The code of the phases was quite separate in the initial version, and restructuring could make little improvement in this respect. The reduction in mean WS size at T=10K is about 10%.

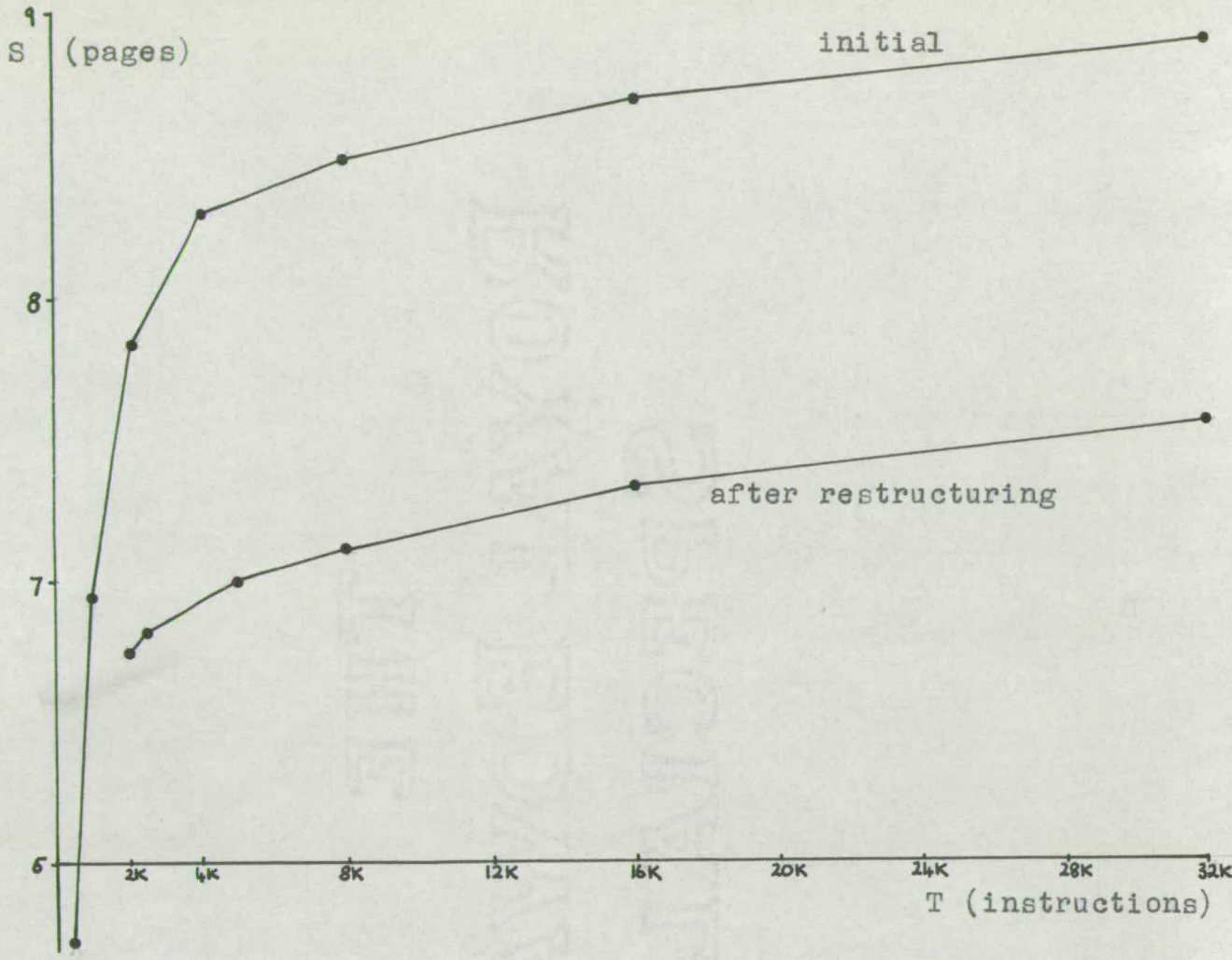
The difference in size between the two phases meant that there was a high variation in the WS size about its mean - there is thus no connection between the restricted store behaviour and the average WS curve (see section I.5). Fig.A-3 is the P2 output for program B, and it shows that a store of 12 pages was not filled until some 200K instructions: the restricted store behaviour is effectively that for the (much larger) second phase alone.

Prog. C

The slope of the WS curve for this program falls off quite slowly - even at T=50K it is still quite steep. Program C would thus be badly behaved (in the sense of section I.5). The mean WS size for 10K intervals is reduced by about 13%. With well chosen chunk boundaries this large and loosely connected program has probably considerable potential for improvement. Fig 1-3 is the restricted store curve for prog. C originally.

Prog. D

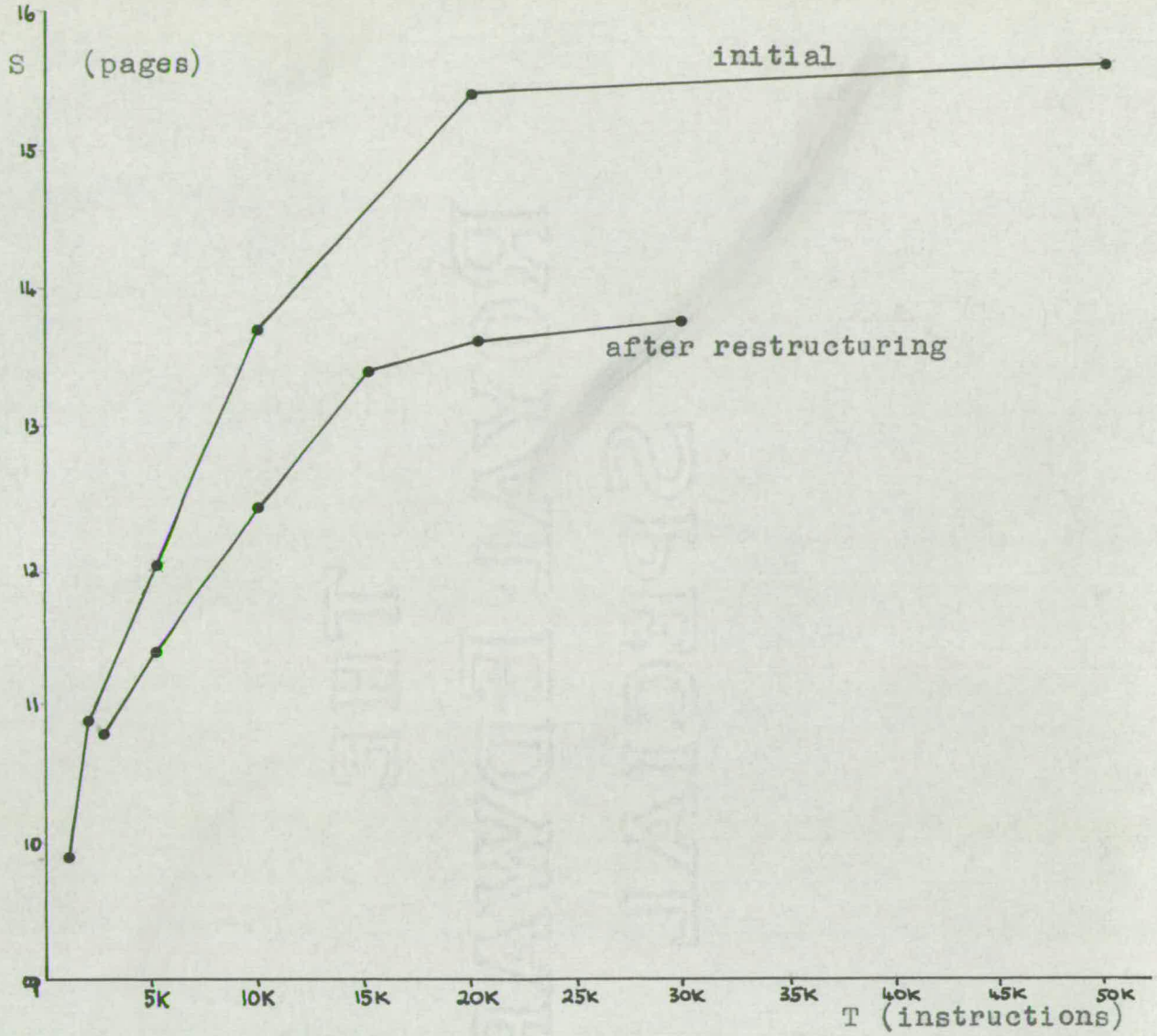
This cannot be compared directly with the others as the working sets refer only to the code. Because of the intelligent choice of chunk boundaries program D would be expected to yield results similar to those of the restructuring scheme (see chapter VI); the main difference is that the chunks here are larger and the potential for improvement presumably less. There is approximately a 35% reduction in the average code WS size for 20K intervals after restructuring.




restricted store behaviour

| store allocated | mean page-fault rate/10K insts. original | after restr. |
|---|---|---|
| 9 | 0.04 | - |
| 8 | 5.3 | 0.22 |
| 7 | 24 | 1.6 |
| 6 | - | 23 |

Fig A-6 WS curves and restricted store behaviour, program A




restricted store behaviour

| store allocated | mean page-fault rate/10K insts. original | after restr. |
|---|---|---|
| 15 | 4.2 | 2.6 |
| 14 | 5.1 | 3.7 |
| 13 | 6.8 | 5.3 |
| 12 | 8.8 | 7.9 |

Fig A-7 WS curves and restricted store behaviour, program B

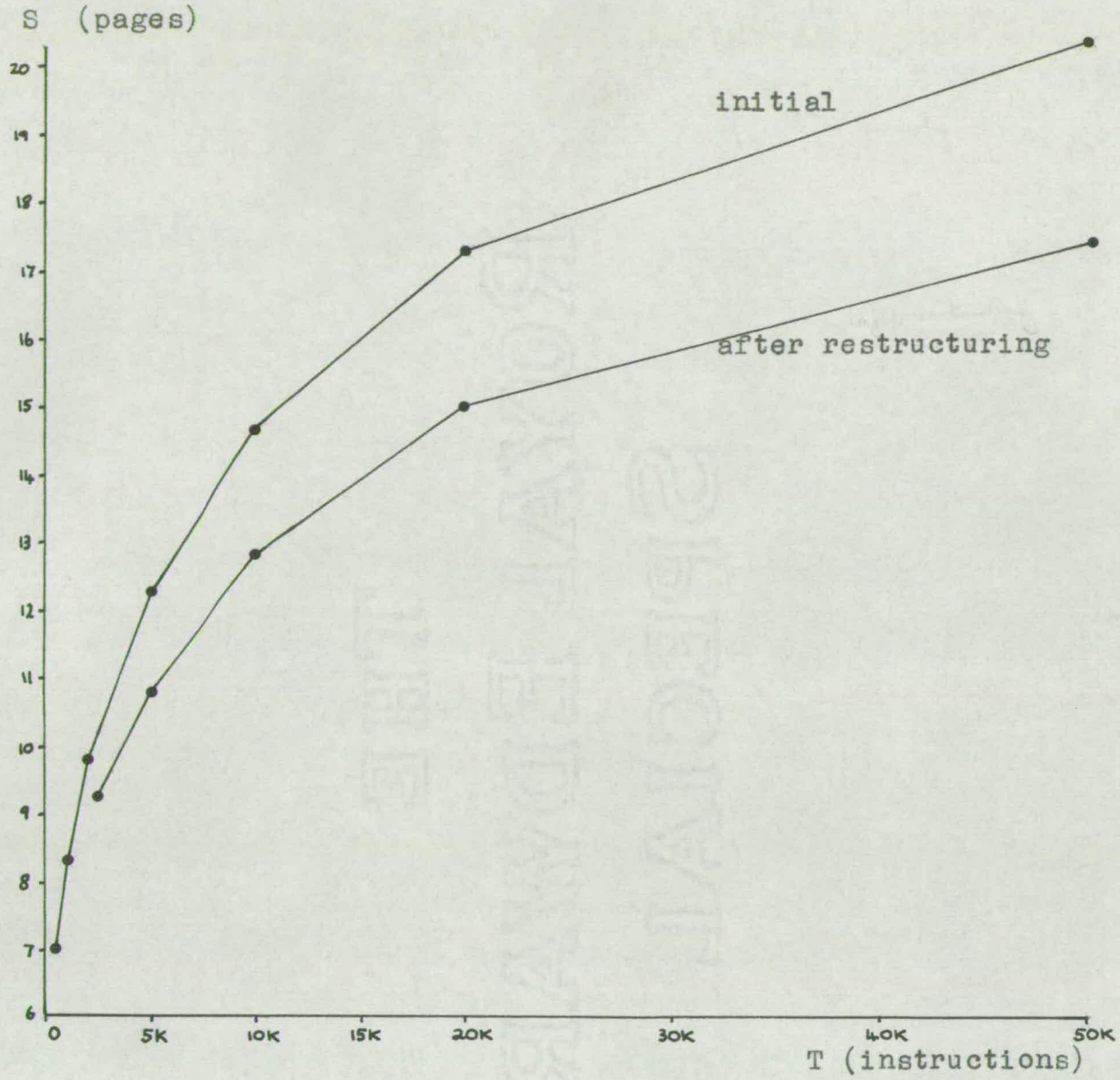



Fig A-8 WS curves, program C

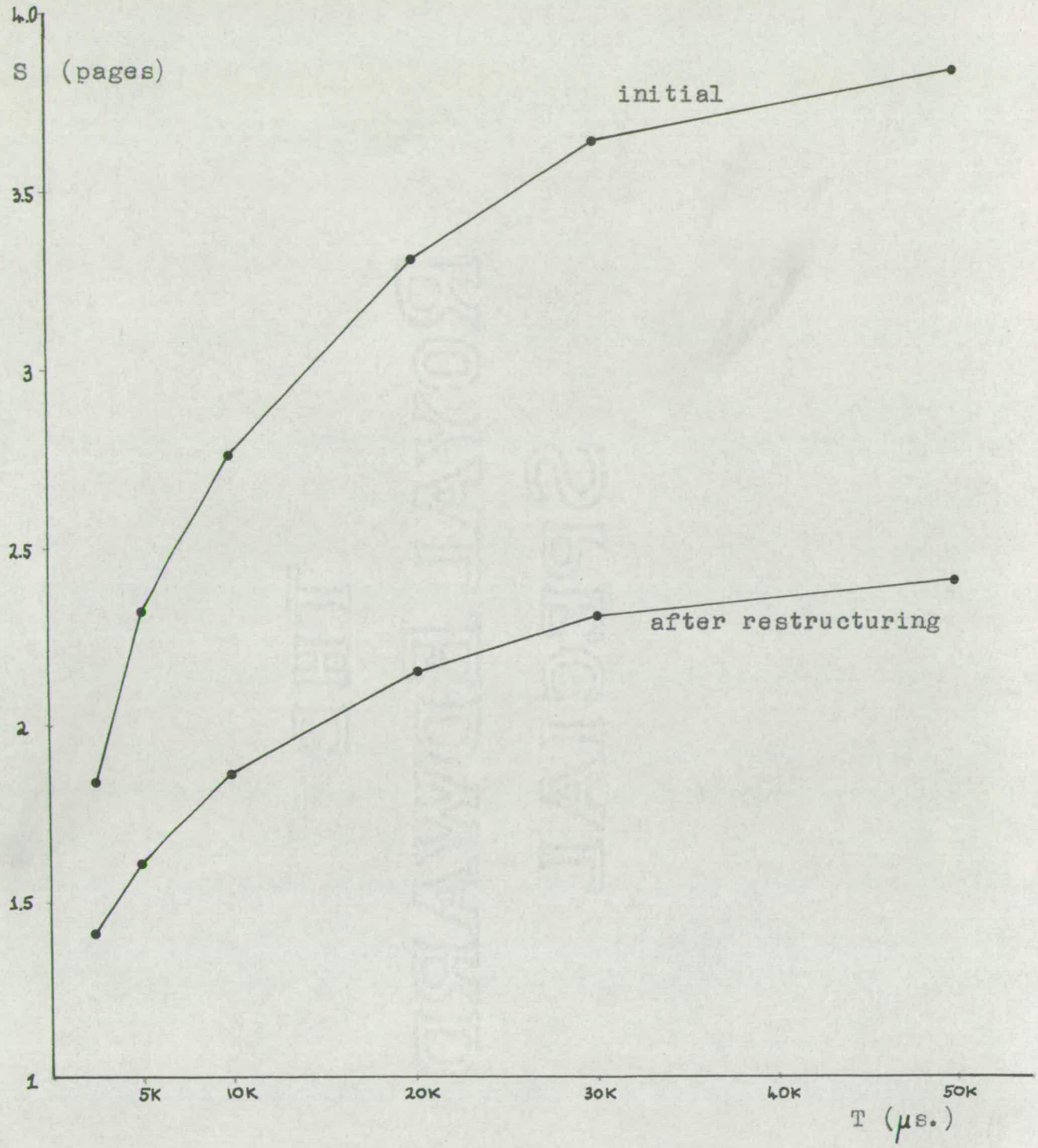



Fig A-9 WS curves (code only), program D

## APPENDIX B. SOME IMPLEMENTATION DETAILS OF RESTRUCTURING SCHEME

### 1. THE COMPILER AND CODE PRODUCED

The compiler, both as originally received and in the developed version, consists of three main phases.

Phase 1: this performs a syntax analysis of each source statement and produces a sequential file of 'analysis records'. There is also some degree of global statement checking (e.g. correspondence of start and finish).

Phase 2: this is the main compilation phase. The analysis records are processed, and a sequential file of so called 'code units' produced.

Phase 3: object code is generated from the phase 2 O/P.

Apart from the data file, a large quantity of global data links the phases. In principle, phases 1 and 2 could be merged, the analysis record of each statement being passed to phase 2 as it is generated; they had been separated in the initial design in order to try and improve program behaviour in the paging environment for which the compiler was originally planned.

Prior to developments directly connected with the restructuring scheme, the following are the main changes made to the compiler as initially received.

(1) External routine compilation was implemented.

(2) 'Perm' (see V.2), which originally had to be compiled with (i.e. the source text placed immediately in front of) any program, was made into an external routine. Linked in at run-time, this could be shared by any modules produced by this compiler.

(3) The compiler itself, initially a single 'main-program'

(section V.2) was divided into external routines - the three phases above, additional global routines, and a controlling main-program. All global variables were passed between external routines through parameters. This caused a slight degree of inefficiency, but the immense gain in convenience of development (which was normally concentrating on one phase at a time) strongly outweighed this.

Henceforth, references to the 'original' compiler assume the above changes made. Only those features of the compiler and code generated, which affect the restructuring scheme are described in any detail.

Branching instructions

Production of a self-monitoring program involved changes to the instructions that transfer control; the implementation of these in the original compiler is therefore described. Reference is made to figs. B-1a, B-2a, which show parts of the M-format code generated from the small demonstration program discussed in chapter V. The line numbers, printed out for each statement, correspond to those in figs. 5-2, 5-3. With this line number is given the current code address (repeated).

There are 16 32-bit registers on the 4-75. Any may be used in addressing, only the lower 24 bits being significant in this case. For addressing purposes, register 0 is always taken as containing zero. In the following assembly code representation of instructions, b (base), i (index), and l (link) refer to registers (i.e. are integers between 0 and 15), d (displacement) is an integer between 0 and 4095, and c is an integer between 0 and 15 which yields a truth value when compared with the current value of a hardware condition code (in particular, true is always given if

c=15). Addresses are evaluated in bytes.

The only machine-code branching instructions which the compiler generates are:

| | | |
|---|---|---|
| 1) | BCR c,b | (Branch on Condition Register) If c yields true, branch to address contained in register b |
| 2) | BC c,d(b,i) | (Branch on Condition) If c yields true, branch to the address obtained by summing d, the contents of b and the contents of i. |
| 3) | BAL 1,d(b,i) | (Branch And Link) Evaluate address as above, and branch there, leaving the link (address of instruction following the BAL) in register 1 |
| 4) | BALR 1,b | (Branch And Link Register) Branch to address contained in register b, leaving link in register 1. |

During the running of code produced by the compiler, the contents of two registers remain constant, and are used to evaluate addresses within the code area. Register 12 (set up according to a system standard by the calling module) points to the beginning of the code of the current module. Register 9 points to the beginning of Perm (see V.2). This is set up on entry to the module, the address of Perm (an external routine) being found from the linkage information in the GLA.

Perm entries

The various entries into Perm required by the compiled code are all in the first 4096 bytes (1 page-length). This means they can be accessed by a single branching instruction to an appropriate displacement on the contents of register 9.

Thus an array access (e.g. line 96, first statement) causes

```
.    ST     11, 0( 0, 13)
LINE    88  CA=      2958 2958
.    L      11, 0( 0, 13)
.    STM    4, 14, 16( 11)
.    LA     15, 72( 0, 11)
.    ST     15, 0( 0, 13)
.    MVC    64( 4, 11), 64( 7)
.    MVC    68( 4, 11), 420( 13)
.    LM     12, 14, 316( 13)
.    BALR   15, 14
.    ST     11, 0( 0, 13)
LINE    88  CA=      2996 2996
.    BAL    15, 40( 0, 9)
.    DC     X'0003'
 NT

LINE    89  CA=      3006 3006
.    ST     7, 0( 0, 13)
.    LM     4, 15, 16( 7)
.    BCR    15, 15
LINE    90  CA=      3016 3016
.    MVC    192( 4, 8), 432( 13)
LINE    91  CA=      3022 3022
.    LA     7, 200( 0, 8)
.    ST     [illegible]
.    L      [illegible]
.    BC     7, 3060( 10, 12)
LINE    95  CA=      3158 3158
.    BALR   10, 0
.    L      0, 428( 0, 13)
.    C      0, 188( 0, 8)
.    BC     7, 42( 0, 10)
.    LA     2, 3( 0, 0)
.    LA     14, 152( 0, 8)
.    BAL    15, 444( 0, 9)
.    L      0, 192( 0, 8)
.    C      0, 0( 0, 14)
.    BC     7, 42( 0, 10)
.    MVC    188( 4, 8), 432( 13)
LINE    96  CA=      3202 3202
.    L      2, 188( 0, 8)
.    LA     14, 168( 0, 8)
.    BAL    15, 444( 0, 9)
.    L      0, 192( 0, 8)
.    S      0, 512( 0, 13)
.    L      3, 528( 0, 13)
.    MR     2, 0
.    A      3, 0( 0, 14)
.    ST     3, 196( 0, 8)
LINE    96  CA=      3236 3236
.    L      11, 0( 0, 13)
.    STM    4, 14, 16( 11)
.    LA     15, 72( 0, 11)
.    ST     15, 0( 0, 13)
.    MVC    64( 4, 11), 532( 13)
.    LM     12, 14, 372( 13)
.    BALR   15, 14
.    ST     11, 0( 0, 13)
LINE    97  CA=      3268 3268
.    MVC    204( 4, 8), 140( 8)
LINE    97  CA=      3274 3274
```

a) N-format code

```
.    ST     11, 0( 0, 13)
LINE    88  CA=      2958 2958
.    L      11, 0( 0, 13)
.    STM    4, 14, 16( 11)
.    LA     15, 72( 0, 11)
.    ST     15, 0( 0, 13)
.    MVC    64( 4, 11), 64( 7)
.    MVC    68( 4, 11), 420( 13)
.    LM     12, 14, 316( 13)
.    BALR   15, 14
.    ST     11, 0( 0, 13)
LINE    88  CA=      2996 2996
.    BAL    15, 40( 0, 9)
.    DC     X'0003'
 NT

LINE    89  CA=      3006 3006
.    ST     7, 0( 0, 13)
.    LR     14, 7
.    BC     15, 1708( 0, 9)
LINE    90  CA=      3016 3016
.    MVC    192( 4, 8), 432( 13)
LINE    91  CA=      3022 3022
.    LA     7, 200( 0, 8)
.    ST     [illegible]
.    LA     10, 88( 0, 0)
.    BC     7, 1576( 0, 9)
LINE    95  CA=      3158 3158
.    BALR   10, 0
.    L      0, 428( 0, 13)
.    C      0, 188( 0, 8)
.    BC     7, 42( 0, 10)
.    LA     2, 3( 0, 0)
.    LA     14, 152( 0, 8)
.    BAL    15, 444( 0, 9)
.    L      0, 192( 0, 8)
.    C      0, 0( 0, 14)
.    BC     7, 42( 0, 10)
.    MVC    188( 4, 8), 432( 13)
LINE    96  CA=      3202 3202
.    L      2, 188( 0, 8)
.    LA     14, 168( 0, 8)
.    BAL    15, 444( 0, 9)
.    L      0, 192( 0, 8)
.    S      0, 512( 0, 13)
.    L      3, 528( 0, 13)
.    MR     2, 0
.    A      3, 0( 0, 14)
.    ST     3, 196( 0, 8)
LINE    96  CA=      3236 3236
.    L      11, 0( 0, 13)
.    STM    4, 14, 16( 11)
.    LA     15, 72( 0, 11)
.    ST     15, 0( 0, 13)
.    MVC    64( 4, 11), 532( 13)
.    LM     12, 14, 372( 13)
.    BALR   15, 14
.    ST     11, 0( 0, 13)
LINE    97  CA=      3268 3268
.    MVC    204( 4, 8), 140( 8)
LINE    97  CA=      3274 3274
```

b) M-format code

Fig B-1

```
.   MVC   140( 4, 8), 432( 13)
LINE   98  CA=      3294 3294
.   L     2, 140( 0, 8)
.   BCR   0, 0
.   BAL   15, 1396( 0, 9)
.   DC    X'00000A6C'
LINE   99  CA=      3308 3308
LINE   99  CA=      3308 3308
.   BALR  10, 0
.   L     0, 536( 0, 13)
.   C     0, 136( 0, 8)
.   BC    3, 20( 0, 10)
.   L     10, 332( 0, 9)
.   BC    15, 3360( 10, 12)
LINE  100  CA=      3330 3330
.   L     10, 332( 0, 9)
.   BC    15, 3338( 10, 12)
LINE  100  CA=      3338 3338
.   BAL   15, 40( 0, 9)
.   DC    X'0007'
 NO BID

LINE  100  CA=      3352 3352
.   L     10, 336( 0, 9)
.   BC    15, 76( 10, 12)
LINE  101  CA=      3360 3360
LINE  101  CA=      3360 3360
.   BALR  10, 0
.   L     0, 520( 0, 13)
.   C     0, 136( 0, 8)
.   BC    5, 20( 0, 10)
.   L     10, 332( 0, 9)
.   BC    15, 3390( 10, 12)
.   L     10, 332( 0, 9)
.   BC    15, 3450( 10, 12)
LINE  102  CA=      3390 3390
.   BALR  10, 0
.   L     0, 524( 0, 13)
.   C     0, 196( 0, 8)
.   BC    11, 20( 0, 10)
.   L     10, 332( 0, 9)
.   BC    15, 3480( 10, 12)
LINE  103  CA=      3412 3412
.   L     11, 0( 0, 13)
.   STM   4, 14, 16( 11)
.   LA    15, 72( 0, 11)
.   ST    15, 0( 0, 13)
.   MVC   64( 4, 11), 420( 13)
.   L     14, 332( 0, 9)
.   BAL   15, 2928( 14, 12)
LINE  103  CA=      3442 3442
.   L     10, 336( 0, 9)
.   BC    15, 76( 10, 12)
LINE  103  CA=      3450 3450
LINE  104  CA=      3450 3450
.   BALR  10, 0
.   L     0, 540( 0, 13)
.   C     0, 136( 0, 8)
```

a) N-format code

```
.   MVC   140( 4, 8), 432( 13)
LINE   98  CA=      3294 3294
.   L     2, 140( 0, 8)
.   BCR   0, 0
.   BAL   15, 1396( 0, 9)
.   DC    X'0000003C'
LINE   99  CA=      3308 3308
LINE   99  CA=      3308 3308
.   BALR  10, 0
.   L     0, 536( 0, 13)
.   C     0, 136( 0, 8)
.   BC    3, 20( 0, 10)
.   LA    10, 100( 0, 0)
.   BC    15, 1576( 0, 9)
LINE  100  CA=      3330 3330
.   LA    10, 101( 0, 0)
.   BC    15, 1576( 0, 9)
LINE  100  CA=      3338 3338
.   BAL   15, 40( 0, 9)
.   DC    X'0007'
 NO BID

LINE  100  CA=      3352 3352
.   LA    10, 102( 0, 0)
.   BC    15, 1576( 0, 9)
LINE  101  CA=      3360 3360
LINE  101  CA=      3360 3360
.   BALR  10, 0
.   L     0, 520( 0, 13)
.   C     0, 136( 0, 8)
.   BC    5, 20( 0, 10)
.   LA    10, 105( 0, 0)
.   BC    15, 1576( 0, 9)
.   LA    10, 106( 0, 0)
.   BC    15, 1576( 0, 9)
LINE  102  CA=      3390 3390
.   BALR  10, 0
.   L     0, 524( 0, 13)
.   C     0, 196( 0, 8)
.   BC    11, 20( 0, 10)
.   LA    10, 109( 0, 0)
.   BC    15, 1576( 0, 9)
LINE  103  CA=      3412 3412
.   L     11, 0( 0, 13)
.   STM   4, 14, 16( 11)
.   LA    15, 72( 0, 11)
.   ST    15, 0( 0, 13)
.   MVC   64( 4, 11), 420( 13)
.   LA    14, 85( 0, 0)
.   BAL   15, 1640( 0, 9)
LINE  103  CA=      3442 3442
.   LA    10, 102( 0, 0)
.   BC    15, 1576( 0, 9)
LINE  103  CA=      3450 3450
LINE  104  CA=      3450 3450
.   BALR  10, 0
.   L     0, 540( 0, 13)
.   C     0, 176( 0, 8)
```

b) M-format code

Fig B-2

the instruction 'BAL 15,444(0,9)' to be generated. The displacement addresses of the various entries to Perm are of course known to the compiler.

Internal jumps

A general address within the code of the compiled module is not immediately accessible, and two instructions are required to transfer control. Consider a jump to address 'a' relative to the beginning of the module. Choose the nearest multiple of 4096 below, or equal to, a, i.e. find p so that (a-4096p) is between 0 and 4095 inclusive (say, d). p is the number of the page (relative to the code start) in which the target address lies, and 4096p the (relative) page address.

Now Perm contains an immediately accessible table of multiples of 4096; the appropriate multiple is loaded into register 10:

L 10,disp(0,9) load register 10 with 4096p

and the transfer of control to address a can be written:

BC c,d(10,12) branch (on condition) to address a.

An example is goto 1, the last statement of line 100. The (relative) address of label 1 is 4172, and 336 is the address in Perm of the integer 4096.

Note that jumps into the first 4096 bytes of the module are treated in the same way, zero (the zero multiple of 4096) being loaded into register 10. This is despite the fact that such addresses are accessible in one instruction; this inefficiency in the original compiler was convenient for the restructuring scheme.

Internal procedure calls

These are very similar, the page address being loaded into register 14 instead of 10. An example is the first statement of

line 103, the last two instructions. The previous instructions here evaluate the procedure parameter and store it on the stack, together with current values of the registers.

Conditionals

The evaluation of conditions leads to short-distance forward jumps which can be treated differently from above. The absolute address of the beginning of the conditional statement is loaded into register 10 (achieved by BALR 10,0 which loads the link without branching), and subsequent jumps are made relative to the address in this register. An example is at line 95.

Switches

A jump to a switch label involves access to the switch vector of relative code-addresses, the vector itself being stored in the code area. An entry in Perm deals with this; the in-line code simply evaluates the switch index and links into Perm, these instructions being followed by the relative address of the switch vector. An example appears at line 98; the dummy instruction 'BCR 0,0' has been generated to align the switch address on a word boundary. This address is given in hexadecimal form after DC (Define Constant). Perm can locate the vector, pick up the appropriate address inside it, add the code base (register 12) and branch to the resulting address.

Procedure returns

The absolute return address has been stored on the stack at call time; this is loaded to register 15 (along with the restoration of other registers with their call-time values), and a direct return made using the BCR instruction (e.g. line 89).

External procedure calls

These are calls of an external routine in another module, and the calling sequence must follow a system standard. The GLA (the base of which is indexed by register 13) contains the absolute address of the external routine entry point and other necessary information. This is loaded into registers; in particular, register 14 will contain the entry point address, and exit is made to the new module via BALR 15,14 (see line 96, second statement).

Compilation

As compilation proceeds, an array 'rttable' is constructed - this contains the address of every point within the module to which explicit reference can be made, e.g. labels, procedure entries, switch vectors, etc. The phase 2 output makes every code address reference by an index to rttable. For example, consider the sequence:

(1) L: goto M

(2) M: goto L

On compilation of (1), space would be reserved in rttable for L and M (assuming this is the first reference to them). The code address of L would be entered in rttable, but that for M is not yet known. The phase 2 output for goto M, would refer to the index of the rttable entry for M. Compilation of (2) would find space for L and M reserved, and the address of M can now be entered. The code unit output by phase 2 for goto L still refers to the rttable index, despite the fact that the address of L is now known; only during phase 3 are the contents of rttable required.

Monitoring instructions

In an M-format object module, code address references are still made through rttable, which thus becomes a run-time array. There are four additional entries to Perm, the monitoring entries concerned with (i) jumps, (ii) procedure calls, (iii) switch jumps, and (iv) procedure returns. Ordinary Perm entries and external procedure calls do not have to be monitored since they lead outside the restructuring area and control must return inside the same chunk. The code for these is therefore unchanged. Figs. B-1b, B-2b show the M-format code corresponding to B-1a, etc. The details of the monitoring entries are as follows.

(i) A branch to the address in rttable(i) would appear as:

| | |
|---|---|
| LA 10,i(0,0) | (Load Address)<br>Set i in register 10 |
| BC c,1576(0,9) | Branch (on condition) to<br>appropriate Perm entry. |

As before this is two four-byte instructions - an example is the final statement of line 100. The index i can only be loaded in a single instruction if it is less than 4096. This would not be a problem with any but very large modules (probably greater than 30 code pages); rttable had less than 2000 entries with the largest program tested. The remedy is to replace i by i-4096 and branch to a different entry in Perm, but this trivial extension has not been implemented.

(ii) An internal procedure call is exactly similar (line 103). The Perm entry address is seen to be 1640.

(iii) The (relative) code addresses in the switch vector are replaced by the indices of their rttable entries. Also, following the switch jump, the address of the switch vector itself is

replaced again by the rttable index. The example of the latter at line 98 shows that the address of the switch monitoring entry to the special Perm is the same as the switch entry address in the ordinary Perm.

(iv) At a procedure return, the pointer to the stack area where registers were preserved at call time is loaded into register 14, and a branch into Perm performed. As can be seen from the example at line 89, the total code-length is unaltered.

The branches inside evaluation of conditions remain unaltered; these need not be monitored as they cannot lead outside the containing chunk.

Chunking, phase 2

Phase 2 determines chunk boundaries as described in V.4. The chunks are numbered from 1 upwards, and at this stage three arrays preserve all the necessary information about each chunk. These arrays are:

1) chunklength - contains length in bytes of each chunk,

2) chunkrn - indexes the rttable entry which contains the address of the beginning of each chunk,

3) branchto - if a chunk ends in a branch to an explicit target address (i.e. not a procedure return), this contains the rttable index of the target.

A non-conditional jump is generated at the end of any chunk which does not already terminate in one; for example at the end of line 99 (chunk 15). These are also generated in the N-format code, making this very slightly longer (normally less than 1%) than in the original compiler. Apart from this, and the three arrays above, the output from phase 2 is the same as it was before the

restructuring developments.

Note that by the rules for the determination of chunk boundaries, some chunks will consist simply of a non-conditional branch. The case:

```
chunk boundary------
                    L: goto M
                   ------
                    N: ...
```

is obvious. More common is the case of branches which occur over procedures or switch vectors. The former, especially, occurs frequently; the procedures in IMP are normally placed together at the beginning or end of the containing block. Between each procedure, this compiler generates a branch to skip over it, giving a trivial chunk.

Such chunks are ignored in the chunking arrays and the rttable entry is changed to eliminate the chunk from the dynamic flow, i.e. the entry for L in the above example would be changed to the address of M. As an example from the small test program, we see (fig.5-4) that chunk 12 begins at relative address 2768 and chunk 13 at 2928, but the length of chunk 12 (fig.5-6) as appears in 'chunklength' is only 152 bytes. The odd chunk will disappear entirely after the first restructuring.

Phase 3

As phase 3 encounters code units which would generate branches of the types (i)-(iv) discussed previously, the monitoring versions are output if an 'M-format' flag has been set. These being the same length as the instructions they have replaced, nothing else need be altered in the production of the code area of the object

file.

Phase 3 must also output the positions of all the monitoring instructions; this is so they may be eventually overwritten after a final restructuring is performed. The positions are output as displacements from the containing chunk boundary, each being characterised by one of the four types.

One further array is constructed, known as 'chunkno'. This is the same length as rttable, and gives the number of the chunk which contains the address in each rttable entry. The three arrays chunkno, chunkrn, and rttable are output in the data area of the compiled program. All other tables are not needed during the program run and are written to the chunk information file. These consist of 'chunklength', 'branchto', and the tables indicating positions and types of monitoring instructions within the code.

We have seen how certain features of the compiler could be used to provide the symbolic code addressing convenient for restructuring; but the question may arise as to why the phase 2 output was not itself taken as the level of restructuring, a policy to which the brief discussion at the end of IV.5 may have pointed. Although possible, there were two disadvantages against using this intermediate code.

1) It was very space-consuming (about three times the length of the final object file); this together with the very large quantity of global data required would have meant that a vast quantity of information had to be kept throughout the restructuring of a large program.

2) In some areas considerable redesign would have been necessary to make the code units relocateable within the intermediate

code file.

It was felt that these overcame the (undoubted) disadvantages of working at machine code level.

## 2. MONITORING REFERENCE BEHAVIOUR

In general three arrays collect run-time chunk reference information.

1) The similarity array s. We refer to the (i,j)th. element as s(i,j).

2) The chunk reference vector. This indicates which chunks have been referenced during the current working-set interval.

3) The translate table, tr. This is only present if the number of chunks is greater than 'selectno', the allowed size of the similarity array. For each chunk i there is an entry tr(i). If tr(i) is negative or zero, it contains the negated frequency of use (i.e. the number of similarity intervals in which reference has occurred) of chunk i so far. If positive, chunk i is in the similarity subset, and tr(i) gives the row/column of this chunk in the similarity array. The frequency of use will then of course be contained in the diagonal element of that array.

### Timing

To monitor the behaviour in similarity intervals, it is obviously necessary to know the program processing time as it would be if it were not being monitored (its N-time), i.e. all the time spent inside monitoring routines must be subtracted from the processing time as obtained from supervisor. The current value of this extra time is maintained (see below) in a variable 'timeerror', and the N-time can thus be obtained. At each monitoring entry to

Perm, if the current similarity interval has not expired, the chunk reference vector is marked with the current chunk. For each of the four types of monitoring entries, a standard path with a standard amount of processing is followed, and this amount is added to 'time-error'. However if the M-time shows that a similarity interval is over, the similarity array must be updated (described below), involving an unspecified amount of computation; to maintain the value of 'timeerror' correctly, the processing time is obtained before and after the updating procedure, and the difference added in.

Updating similarity array

If a full similarity array is being used, one is added to every element s(i,j) if chunks i and j are both marked in the interval reference vector.

However the situation is more complex if a partial array is being used. The translate table is first updated. For each chunk i marked in the chunk reference vector:

(a) if tr(i) > 0, no action,

(b) if tr(i) < 0, set tr(i)=tr(i)-1 (frequency of use),

(c) if tr(i) = 0: if the similarity subset is as large as allowed, perform (b); otherwise set tr(i) to the next unassigned row/column of the similarity array (i enters the similarity subset).

Then for all i,j marked in the chunk reference vector such that tr(i) and tr(j) are positive, add one to s(tr(i),tr(j)).

Although the translation process involves a good deal of computation, note that it is only performed at the end of similarity intervals; most of the monitoring entries into Perm simply involve updating the chunk reference vector.

Monitoring entries

Any entry to a new chunk will be trapped by one of the four types of monitoring instructions (i) to (iv) (see section I); updating the reference information as described above will be termed 'metering' the chunk.

Entries (i) and (ii): the rttable index of the target address enables the latter and the containing chunk number (from 'chunkno') to be found. The chunk is metered and exit made to the target address. In the case of entry (ii), the standard procedure call mechanism has preserved the values of the registers on the stack, to be restored at procedure return. One of these locations, whose contents are not required at return, is replaced with the current (calling) chunk number.

Entry (iii): the address and chunk-number of the switch vector is evaluated from the rttable index supplied; this chunk is then metered. The appropriate entry in the switch vector is then accessed, this gives the rttable index of the target address, whose chunk is metered. Control is finally transferred to the target address.

Entry (iv): at a procedure return, Perm is entered with a pointer to the work area in the stack where the values of registers at call time were preserved - these give the return address and the chunk number (entry (ii)). A chunk cannot end with a procedure call instruction, so the chunk after return will be the same as that at call-time, the chunk can therefore be metered before returning to the target address. Note that to find the containing chunk simply from the return address would have required a search through all the chunk addresses.

Page monitoring

If 'page monitoring' is switched on, target addresses are passed, together with the N-time, to an external routine which records paging behaviour (processing time readings are made before and after each entry, to maintain the correct value of timeerror). This routine can examine behaviour in the same way as the 'simulate running' programs in the KDF9 study (appendix A), remembering that only code references are being trapped here. Mean working-set sizes were calculated for the multiples $\frac{1}{4}, \frac{1}{2}, 1, 2$ and 4 of the similarity interval. This would of course have been considerably more complex if the size of the M-format code were not the same as that of N-format; a mapping onto N-format addresses would have been necessary to examine the correct paging behaviour of the original program.

A very slight error may be introduced; if a chunk lies over a page boundary (which cannot occur after the final restructuring) and is entered in the first page, a drop-through into the second page may not be recorded if the next branch obeyed exits from it. This could make paging behaviour initially appear slightly better than it really is, and the effect of restructuring slightly less good, but the effect is probably insignificant.

Updating chunk information file

Reference information is written to the chunk information file at the entry to Perm arising from stop or endofprogram. If this is the first run of this particular structuring of the module, the similarity array and the translate table (if present) are simply copied on to the file. If translate tables are not present, the similarity arrays from subsequent runs are simply added element by

element to that on the file. If partial arrays are present however there is the problem that the similarity subset of a run may not be the same as that already on the file; a new similarity array has to be constructed from two others which contain information on two different subsets of the chunks.

The first task is the construction of the new similarity subset for the new array. Any chunk with a negative entry in either translate table is debarred as some information on its similarity with other chunks is not available. Also there is no point in including any chunk not referenced so far at all (a zero in both translate tables). The number of chunks eligible may be greater than the similarity array allows, in which case some will have to be discarded; it may be less - as was the case with the bridge-hand program (chapter V) after two runs. Given the new similarity subset, all the information is available to produce a new translate table and similarity array; the process is straightforward, if long.

These complexities could be avoided by using a standard similarity subset - this could either be chosen at compile-time or initialised at run-time from that on the chunk information file. The former course might be wasteful (see IV.4), the latter would mean that the program run was not independent of the chunk information file; it was thus decided to implement the process as described above. However it is not clear that it was worthwhile; with large programs, little clustering occurs during the first restructuring, and after this, translate tables are written back into the GIAP in any case.

## 3. CLUSTERING

The clustering routine, the first phase of the restructuring program, contains the algorithm briefly described in III.8. Using the similarity array and chunk lengths on the information file, it forms groups of chunks, either proceeding to completion or stopping when the greatest linkage between clusters falls below a specified quantity.

Suppose there are m chunks, and the similarity subset (N) is of size n. Thus a translate table exists only if n is less than m.

Initially, the m chunk numbers are stored as m single-element lists which will be linked together appropriately as the chunks combine into clusters. A table p is set up consisting of m pointers to these lists; as clusters form, each non-zero element of p points to the head of the list of chunks in a cluster (the order of chunks at this stage is arbitrary). Elements of p not pointing to lists are set to zero. Initially p(1) to p(n) correspond to the similarity array rows; thus if a translate table is present, list i is indexed by p(tr(i)), otherwise by p(i). In the former case, the remaining p(n+1) to p(m) point arbitrarily to the rest of the lists corresponding to chunks not in N.

Two further arrays have, in a similar manner, entries corresponding initially to each chunk and later to each cluster. These are:

num(i): contains the number of chunks in each cluster (initialised to 1)

size(i): contains the size of each cluster in bytes (initially the size of each chunk)

The variable 'limit' is set to the maximum frequency of use of all chunks outside the similarity subset, or to zero if a final

restructuring is taking place. The diagonal elements of the similarity array are set to zero. Remembering its symmetry, s(i,j) is regarded as the same element as s(j,i) in the following sketch of the clustering algorithm.

1) Find the greatest element of array, say s(k,l). Exit if this is not greater than 'limit'.

2) Link list p(l) on the end of list p(k). Set p(l) to zero.

3) For each i≠k or l, such that p(i)≠0:

$$s(i,k) := \frac{num(k).s(i,k) + num(l).s(i,l)}{num(k) + num(l)}$$

s(i,l) := 0

4) num(k) := num(k) + num(l)
size(k) := size(k) + size(l)

5) For each i≠k such that p(i)≠0 and size (i)+size(k)>pagesize, set s(k,i)=0.

6) Go to step 1.

The formula in step 3 for the average similarity between the original constituent chunks of two clusters is easily verified to be correct at any stage of the clustering procedure. To aid the search for the maximum element in step 1, a vector is maintained to contain the maximum element of each row of the array; merging of clusters normally affects few of the maxima, and the total time of searching is reduced.

If a final restructuring is being performed, it remains to pack all the chunks as tightly as possible (including those originally not in N) within page-size groupings.

At termination, the non-zero elements of p point to the list-heads of the new chunk groupings; these of course contain actual chunk numbers, the complication of translation having been removed at the start. This set of pointers and the lists of chunks are

passed to the second phase of the restructuring program.

## 4. GENERATION OF NEW FILES

### Code area

The code area of the restructured object file is generated a chunk at a time. If a final restructuring is being performed, each cluster of chunks (which forms a 'new chunk') begins on a page boundary, otherwise the groups are output immediately following each other. Since all explicit code-address references are made through rttable, the chunks can be reordered without any change to them being necessary. The only difficulty in generation of the code area is the determination of the order of the chunks within each group (or page, in a final restructuring). This is affected by two considerations.

1) Alignment: in certain circumstances (e.g. a jump to a switch label), full-word constants appear in the code and these must be aligned on a full-word boundary. After restructuring, then, the containing chunk must be aligned in the same way as before. Although it can cause slight inefficiency, it is convenient not to make special cases, and to align all chunks on the same type of boundary (with respect to full-word) as previously.

2) A chunk ending in a branch to the beginning of another chunk in the same cluster can be placed immediately before it, and the branch removed.

Normally as each output is made there is a search for a suitably aligned chunk (otherwise a halfword of dummy instructions is necessary). This will be overridden by case (2) which saves two words of the branch instruction.

An example of removal of the final branch is chunk 15 (figs. 5-4,5-6). At compilation, this effectively had to have a 'goto NB' added at the end, the total length then being 30 bytes. Fig 5-8 shows chunks 15 and 16 are in the same cluster in the first restructuring. The three columns below show the new code address, chunk number and length of each old chunk as it is generated; it is seen that chunk 15 is before 16, and its length is now only 22.

Fig 5-9a, the final restructuring shows an example of alignment; the chunks 11 and 17 (renumberings after first restructuring) being neither full-word aligned have caused the second cluster to begin at address 4098, instead of the page address 4096.

Other changes

A new GLAP has to be written; not only the rttable entries are changed, but chunkrn and chunkno must refer to the new 'chunks'. Also, if necessary, a translate table is written back into the GLAP to select the similarity subset for the next series of runs. This will consist of the most frequently used chunks (regarding the frequency of a cluster as being that of its most used constituent chunk) of the last series of runs. Also changed are the addresses of the entry points to the module; a new linkage data area of the object file is therefore necessary.

Obviously, a new chunk information file has also to be written. At the end of this process, the information and object files have exactly the format as might have been produced by the compiler, except that a similarity subset may be written into the GLAP of the object file, and the number of restructurings so far is recorded in the information file.

The final restructuring produces files in the same format; the

new chunks are now of course pages. To convert the M-format file to N-format, the monitoring instructions, whose locations are obtainable from the tables in the information file, are overwritten with the original branching instructions (using addresses obtained from rttable). The only other alteration is the reduction of the length of the GLAP by removing the various chunking arrays.

## APPENDIX C. PROPERTIES OF THE MEAN WORKING SET SIZE

We give here proofs of the mean working-set properties stated in I.5 (this is an alternative approach to that of Denning, ref.16).

We take I as a large processing interval (0,i), perhaps extending over many runs of the program. The lower end-point is taken as zero for convenience; we make no assumption that pages are not referenced outside the period I, i.e. before time zero or after time i. The working set W(t,T) and its size s(t,T) refer to pages referenced in the process time-interval (t-T,t) which we shall define as half-open, i.e. include the lower end-point, but not the upper.

Then:

$$S^I(T) \quad = \quad \frac{1}{i}\int_0^i s(t,T)dt$$

Where I is understood, we shall write this as S(T).

1) $S^I(T)$ is continuous and right and left differentiable; and its slope is the mean rate at which pages outside W(t,T) are referenced.

Proof

When a reference is made to a page not in W(t,T), the page is said to enter the working set. For a given T, denote the times of all such entries by $0=t_0, t_1, \ldots t_N=i$. (This set is of course a function of T. For convenience we have taken $t_0=0$, $t_N=i$; in fact it is not necessary for the end-points to be themselves working-set entry points). Denote the lengths of the intervals between entry points by $i_{n+1} = t_{n+1} - t_n$. Each $t_n$ marks a page fault if the contents of the main memory were maintained strictly

at the working set of T.

Take $e > 0$ but less than the store cycle time, i.e. less than the interval between any two successive page references. This is certainly smaller than any $i_n$.

Suppose t is in the interval $(t_n, t_{n+1})$ (see fig. C-1).

Consider the function $f(t) = s(t+e, T+e) - s(t, T)$. This represents the number of distinct pages referenced in the interval $B(t, t+e)$ which were not referenced in $A(t-T, t)$.

For $t_n \leqslant t < t_{n+1} - e$, $f(t)=0$. For otherwise there is a first extra page referenced at some $d < t_{n+1}$ and not referenced in $(t,d)$ or in A. It thus enters the working set $W(d,T)$, which would make d an additional entry point between $t_n$ and $t_{n+1}$.

For $t_{n+1}-e < t \leqslant t_{n+1}$, $f(t)=1$. The entry page at $t_{n+1}$ is the extra page. Since e is less than all $i_n$, $t_{n+2}$ cannot be reached and a similar argument to that above shows that f(t) is no greater than 1.

(Note that the above does not involve pages leaving working sets, since the two intervals concerned in f have a common starting point.)

Then: $$\int_{t_n}^{t_{n+1}} f(t)dt = 0.(t_{n+1} - e - t_n) + 1.(t_{n+1} - t_{n+1} + e)$$
$$= e$$

Summing all such integrals gives:

$$\int_{t_0=0}^{t_{N-1}} s(t+e, T+e)dt - \int_0^{i} s(t,T)dt = Ne$$

Transforming t+e to t in the first integral and substituting S for its definition gives eventually:

$$\frac{S(T+e)-S(T)}{e} = \frac{N}{i} + \frac{1}{ie}\int_0^{e} s(t,T+e)dt - \frac{1}{ie}\int_i^{i+e} s(t,T+e)dt$$

For sufficiently small e, both integrands take a constant value. (Consider for instance the first. The integrand represents the pages referenced in (t-T-e,t). Pages are accessed at discrete intervals: if the first access before time -T is at $-T-e_1$, and the first after time 0 is at $e_2$, take e less than $e_1$ and $e_2$. Then for $0 < t \leq e$, $s(t,T+e)=s(0,T)+1$, zero being a working-set entry point.) We thus have:

$$\frac{S(T+e)-S(T)}{e} = \frac{N}{i} + \frac{E}{i} \qquad \text{where } E = s(0,T)-s(i,T)$$

S(T) is therefore right-differentiable (and is similarly shown to be left differentiable, but the two slopes are not necessarily equal. The S curve is continuous but consists of very small straight line segments). For large i, the term E/i is negligible, and the slope of S(T) is thus N/i. N is the total number of page-faults and i is the total time: the result follows by definition.

2) S(0)=0 and S(T) is a non-decreasing function of T. These properties are obvious.

3) If for $T > t$, $s(t,T) = s(t,t)$ for a range of values of T, then S(T) is linear or concave downwards at all points in the range. The condition must be made since we have not disallowed the possibility of page references before t=0, which would appear in the working-sets of instants within I. Without the condition, the theorem is not necessarily true.

Proof

Consider instants t,t+e, where $0 < e < T$ (see fig.C-2).

$s_1$ = $s(t+e,T+e)-s(t,T)$ represents the number of distinct

pages referenced in the interval A(t,t+e) which were <u>not</u> referenced in the interval B(t-T,t).

$s_2$ = s(t+e,T+2e) -s(t,T+e) represents the number of distinct pages referenced in A and <u>not</u> in C(t-T-e,t).

But C contains B, so the pages referenced in B are a subset of those in C. It follows that:

$$s_1 \geqslant s_2$$

Integrating over t in I, we have

$$\int_0^i s(t+e,T+e)dt - \int_0^i s(t,T)dt \geqslant \int_0^i s(t+e,T+2e)dt - \int_0^i s(t,T+e)dt$$

Dividing by i, transforming the t+e to t in two of the integrals, and substituting S for its definition gives:

$$2S(T+e)-S(T)-S(T+2e) \geqslant$$

$$\frac{1}{i}\int_i^{i+e} (s(t,T+2e)-s(t,T+e))dt + \frac{1}{i}\int_0^e (s(t,T+e)-s(t,T+2e))dt$$

On the RHS: First integral is greater than zero, since so is the integrand; second integral is zero, since by the initial assumption, s(t,T+e)=s(t,T+2e)+s(t,t) for $0 \leqslant t \leqslant e$.

Thus:

$$S(T+2e) - S(T+e) \leqslant S(T+e) - S(T)$$

This is a sufficient condition for concavity, and the result is proved.

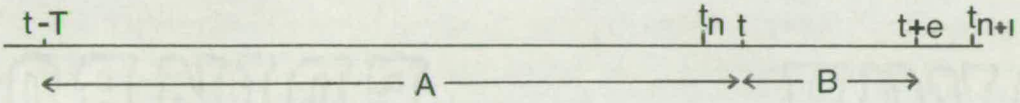



Fig C-1

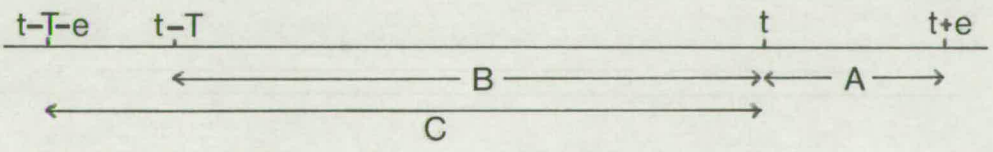



Fig C-2