WORKING SET SIZE REDUCTION BY RESTRUCTURING APL WORKSPACES

D. Kropp, H. Wrobel
IBM Scientific Center Heidelberg
Tiergartenstrasse 15
6900 Heidelberg, Germany

ABSTRACT

APL Workspaces usually have a random distribution of the defined APL functions. The physical location of a defined APL function within an APL workspace depends on its history of creation, modification, or copying. Furthermore in APL a good programming style results in a large number of small functions. The execution of an APL program in a virtual storage environment usually leads to comparatively large working set sizes because the APL functions are scattered over the workspace. However, the interrelation between the functions can be analysed and exploited to reorganize a workspace and thus reduce the working set sizes. Methods of restructuring an APL workspace are proposed. Results of an investigation on a large APL/CMS workspace using these methods are presented.

INTRODUCTION

Virtual storage computing systems have come into wide use in recent years. Most APL installations on system /370 machines today run under virtual storage operating systems, giving more freedom to the APL programmer because of the availability of much larger workspace sizes. Consequently large APL programs are being created. Running in a virtual storage environment they can have bad performance characteristics, as has been observed in other large programs. Under high machine load conditions a program will run most efficiently, if it has a high locality of reference corresponding with small working

sets. Experimental techniques have been developed, and methods proposed to improve locality in a program by rearranging relocatable sections of code (1-4) and by following certain guidelines in program design (5).

Certainly one intention of virtual storage is to relieve the application programmer from having to deal with physical storage limitations. On the other hand, large programs in a virtual storage environment tend to have performance problems, unless attention is paid to locality of reference. For frequently used programs, the application programmer needs to be concerned with the efficient execution of his program.

APL systems more than any other programming language shield the programmer from the internals of his workspace. On the surface there seems to be no easy way to influence the structure of the internal representation of an APL program. But the fact that APL programs usually consist of many rather small Defined Functions suggests in principle the possibility of locality improvements of APL workspaces.

This paper presents a feasible procedure to get a controlled internal structure of an APL/CMS workspace. It describes how functional interrelations are used for restructuring a workspace. Some results from experiments with an APL workspace with 511 Defined Functions are presented.

REARRANGING AN APL WORKSPACE

Control of workspace structure

The internal structure of an APL workspace depends on the history of performing copy operations, erasing objects, and defining or modifying Functions. The internal structure can be controlled by using the)COPY,)PCOPY, and)GROUP commands. Investigating the effect of the)COPY and)PCOPY commands in APL/CMS shows that they can easily be used to control the internal relative location of Defined Functions and/or Global Variables. These commands, when executed for single objects or defined Groups of objects, append the copied objects to the content of the target workspace maintaining the sequence as specified in the Group. For economic reasons one will use Groups to rearrange a

large number of objects. The normal APL capabilities to manipulate Group definitions are somewhat limited. Fortunately, the Stack Processor provides a means to define a Group from a large list of names. In APL/CMS, the)COPY command places the portion of the symbol table for the Group of objects copied adjacent to the Group, thus scattering the symbol table over the workspace. If one wants to avoid this one can copy a Group representing the whole wanted structure.

The restructuring method

The intention of restructuring is increasing the degree of locality of the program to be executed. Locality means "keeping a program's address-space references confined to as few pages as possible for as long as possible" (5). In terms of the working set good locality corresponds with small working set sizes. Locality and working set size of a program vary with time as a program proceeds through different execution phases.

Improvements in performance by restructuring can in particular be expected, when the relocatable sections of a program are smaller than the page size of the paging system. In an APL workspace the relocatable sections are the Defined Functions and the Global Variables. In the example discussed below the internal representation of an average of 8 Functions fills one page of 4 k bytes. This figure of about 8 Functions per page seems quite normal for common APL coding.

Two kinds of restructuring are usually distinguished: static and dynamic. Restructuring based on an analysis of the program prior to execution is called static. Quite elaborate static analysis of interrelations and cyclic structures can be performed and used for restructuring algorithms (e. g. 6).

If one wants to make use of the dynamic behaviour of the program one has to analyse run time data like instruction traces. By using dynamic restructuring substantial improvements of the paging behaviour are possible (1,3,4).

A dynamic approach like the one described in (1) applied to an APL workspace would be difficult and time consuming. So far we have

confined ourselves to a static restructuring procedure working on the set of Defined Functions. The essential locality improvements may be achieved by an algorithm that separates code of disjunctive program phases into separate clusters and puts Functions as close as possible to the calling Functions. Obviously compromises have to be made, since many Functions are called from different parts in the program.

The algorithm to create the restructuring list was coded in APL. It works as follows: First the name of the main Function is put into the list. Next the names of all the Functions used by it are added to the list. For each Function in the list the names of the Functions it references are added to the list immediately after its name. If a name has already appeared above its possible position, it is not added. Duplicate names are removed starting from the bottom of the list. No attempt to detect loops is made. The example in Fig. 1 illustrates this process: Function 1 refers line by line to Functions 2,3,4 und 5. After name 1 the names 2,3,4,5 are appended to the list. Next Function 2 is analysed providing the names 6,10,7, and 4. 6,10, and 7 are inserted into the list between 2 and 3. Then Function 6 is analysed and so forth. The created list of this example is shown in Fig. 1.

Starting from a main Function, to which all other Functions are connected directly or indirectly, the process of creating the structured sequence of names can be done automatically. The program which does the restructuring also accepts a list of Function names the "subtrees" of which are to be excluded from the process. Thus existing knowledge of the overall structure of the program can be included in its restructuring.

Names of Functions which are called from several places and do not call other Functions are extracted afterwards into a separate list of names of "isolated" Functions. The objects of this list are comparable with objects one would have put into a root segment of an overlay structure.

From the lists of names Groups are defined and transferred to the workspace to be restructured. With the)PCOPY command then a restructured workspace can be built up in a Clear Workspace. Names contained in more than one Group make no problems. The)PCOPY command will copy a Function only the first time its name appears. From this follows that groupings considered to be most important (e.g. the

"isolated" Functions) should be copied first.

EXPERIMENTAL RESULTS

The investigation was made with an APL workspace of 420 k bytes. This workspace consists of 511 Defined Functions covering about 245 k bytes and 17 permanent Global Variables needing 45 k bytes. The size of the available work area is 130 k bytes. The Gobal Variables were not considered to be put into a prefered place or sequence. The workspace contains an interactive program having a number of distinct phases.

Obviously the area accessible to improvements is that with 245 k bytes or about 60 pages of 4 k bytes. The experiment was run on a system /370 model 145 with the APL/CMS microcode assist feature. The paging technique used by VM/370 is demand paging with an LRU (Least Recently Used) algorithm. The real storage available to the program was controlled by use of the VM/370 LOCK command for a different user. Runs of 4 minutes virtual CPU time were measured for each defined real storage size. The VM/370 MONITOR facility was used to get the presented data.

The runs took place in a "locked out" virtual machine, saying it had to page against itself. A comparison is made of runs of the original workspace with runs of the restructured workspace. In Fig.2 are the "parachor" curves (5) shown derived from these runs, and in Fig.3 the performance improvements as paging ratios are given. Considering the numbers of page reads the greatest improvement is seen in the left part where paging starts to be excessive. This corresponds with the results in (1). More interesting with respect to "what is saved in a multi user situation" may be to compare pages of real storage for equal paging rates of the two workspaces. These savings are larger in the right part of the curves. In fact, the more desirable load situation for the programs is, when they can work right of the knee of the parachor curves. The real storage savings in numbers of pages lie between 8 and 11 pages which is a 14-17% improvement. Taking into account that essential parts of the working set are formed by the APL interpreter, the APL variables, and some CP/CMS activity the improvement is more significant.

More detailed information can be seen from Fig. 4-5. They show the frequency distributions (in %) of "projected working-set sizes" as defined in CP. The comparison is made at 4 paging rates for interactive phases (Fig. 4a-d) and CPU-intense phases (Fig. 5a-d) which are distinguished by CP. The curves reflect a common structure of frequency of 2 prefered "projected working set sizes", and the curves of the restructured workspace are relatively displaced to the left. This can be interpreted as: smaller working set sizes are more frequent all through the different program phases.

These results show that static restructuring of an APL workspace has a reasonable effect. The algorithm used so far is a somewhat "ad hoc" solution, to get a first impression, what might be possible in restructuring APL workspaces. Clustering algorithms better exploiting the static overall interrelation between Defined Functions in some analogy to the "nearness matrix" of the dynamic approach of (1) seem possible. On the other hand, a program so intensively used like the APL interpreter itself might gain performance in a virtual storage environment by restructuring it either dynamically or with respect to frequency of usage of APL Primitive Functions. One also might think of some permanent feature in an APL system that does a static restructuring when a workspace is copied into a Clear Workspace.

In virtual storage environments so far the aspect of locality mostly is neglected by the user, probably, since restructuring methods usually are not easy to handle, and the effect, though actually present, becomes largely hidden in the multiprogramming situation. But in APL as well as in other programming languages with an automatic mechanism, available to the user as an option, large programs can be structured for better performance without special knowledge.

REFERENCES

(1) Hatfield, D.J. and Gerald,J.:Program Restructuring for Virtual Memory. IBM Systems Journal, 10, 3, 168-192, 1971.

(2) Hatfield, D. J.: Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance. IBM Journal of Research and Development 16, 1, 58-66, 1972.

(3) Ferrari,D.: Improved Locality by Critical Working Sets. Communications of the ACM, 17, 11, 614-620, 1974.

(4) Baier,J.L. and Sager, G.R.: Dynamic Improvement of Locality in Virtual Memory Systems. IEEE Transactions on Software Engineering, SE-2, 1, 54-62, 1976.

(5) Morrison, J.E.: User Program Performance in Virtual Storage Systems. IBM Systems Journal, 12, 3, 216-237, 1973.

(6) Baer,J.L. and Caughey,R.: Segmentation and Optimization of Programs from Cyclic Structure Analysis. Proc. AFIPS 1972, Spring Joint Computer Conference, 40, 23-36.

FIGURES

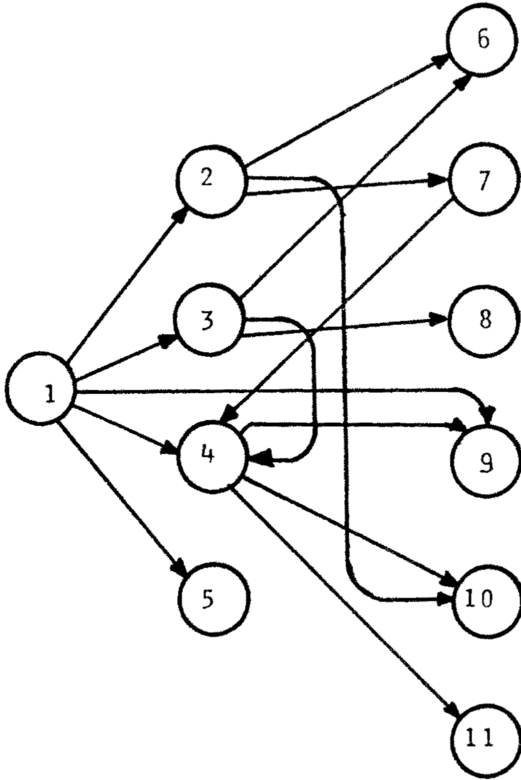
Fig. 1: Graph of "Function Call" interrelations and restructured Function names list.

Fig. 2: Page Reads ("parachor" curves) and Page Writes plotted against available real storage. The thick lines are the curves of the restructured workspace.

Fig. 3: Relative performance as ratio of (page reads of original workspace) to (page reads of restructured workspace).

Fig. 4a-d: Frequency distributions of "projected working set sizes" at interactive phases of paging rates A,B,C,D (Fig. 2).

Fig. 5a-d: Frequency distributions of "projected working set sizes" at CPU-intense phases of paging rates A,B,C,D (Fig.2).



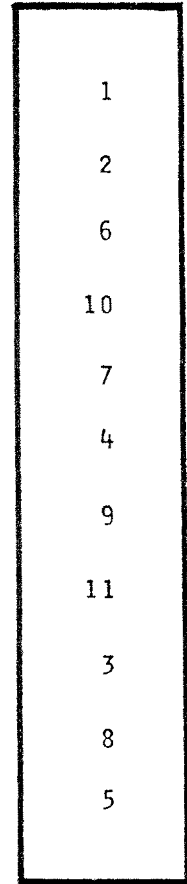


Fig. 1

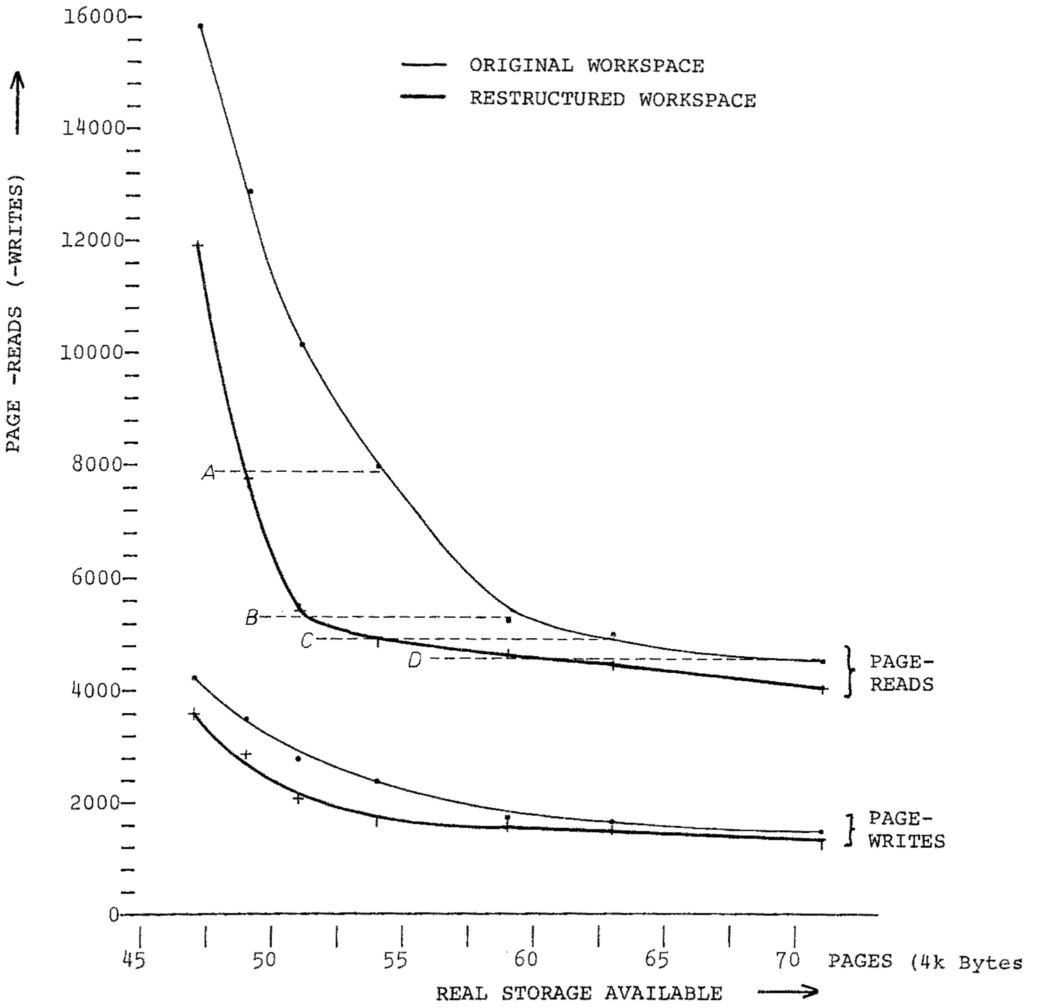


Fig. 2

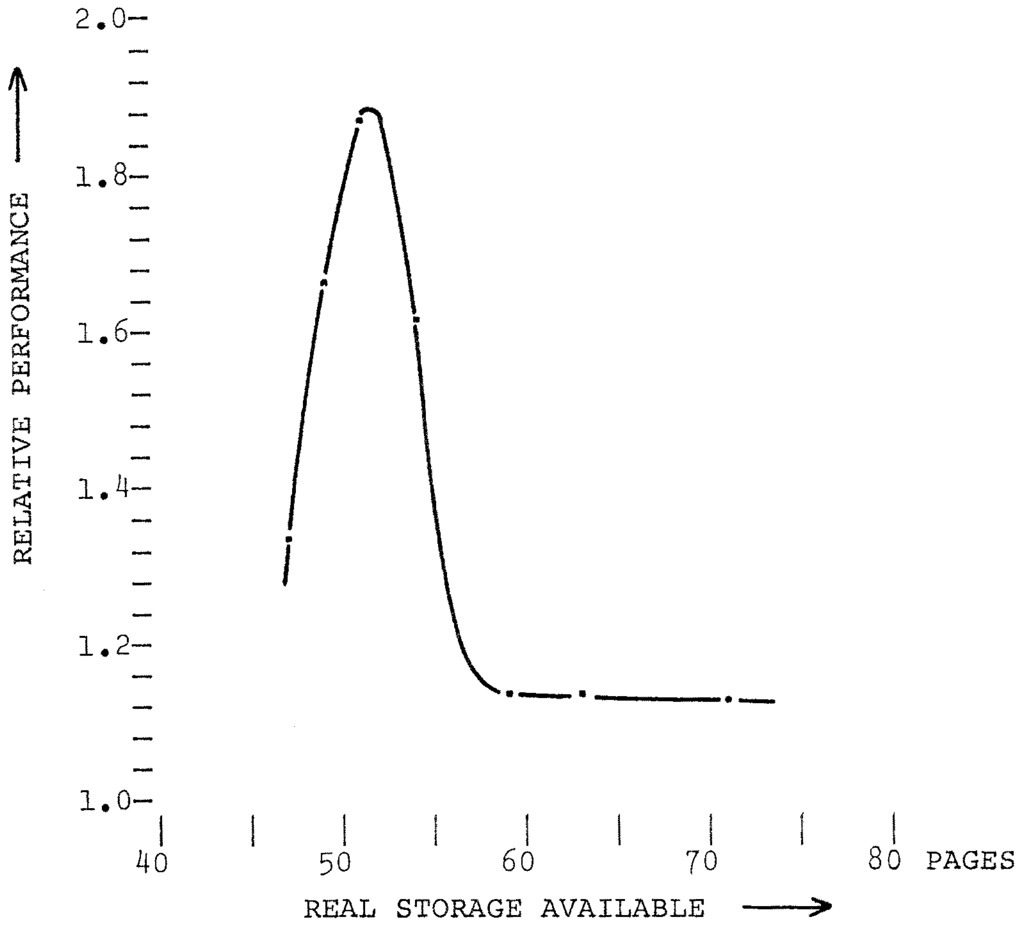


Fig. 3

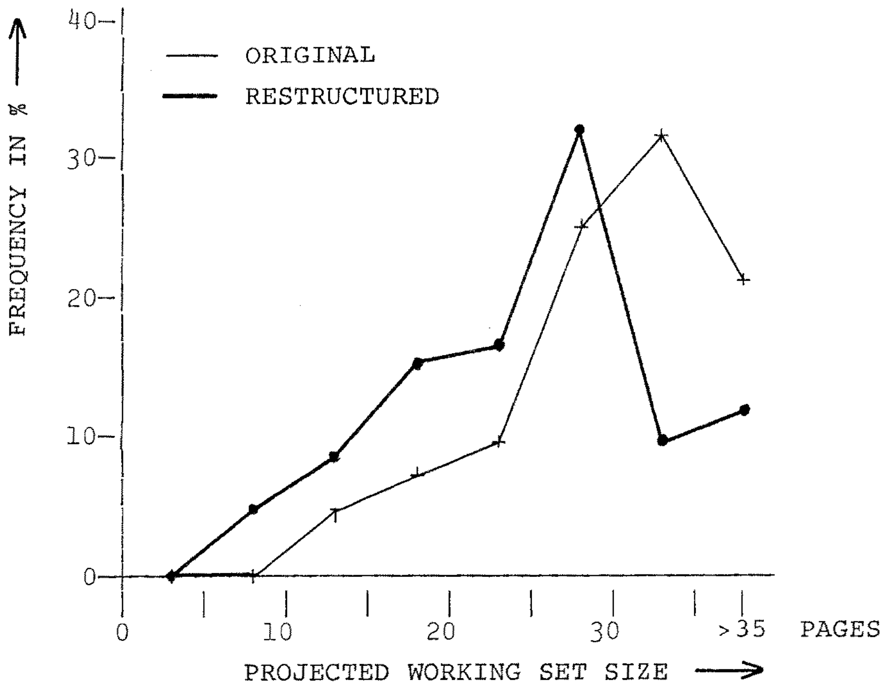


Fig. 4a

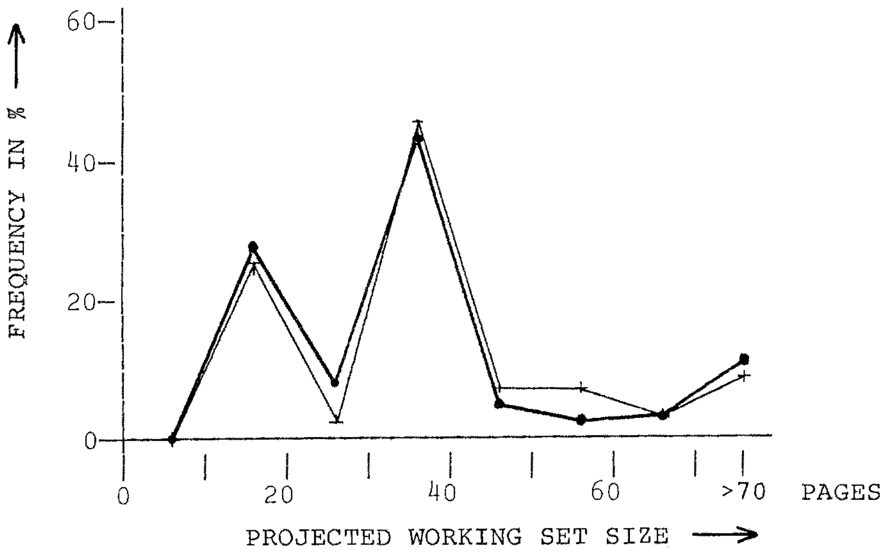


Fig. 5a

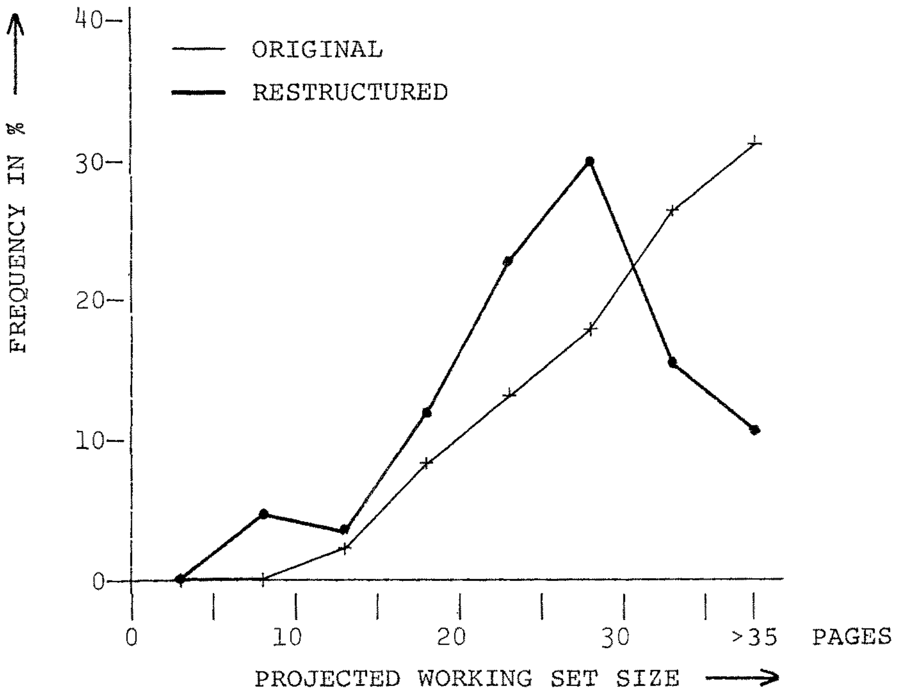


Fig. 4b

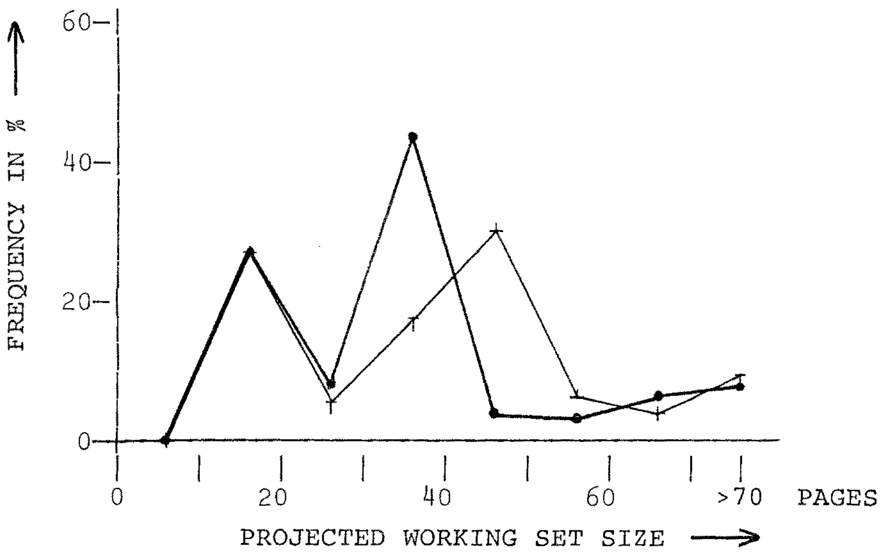


Fig. 5b

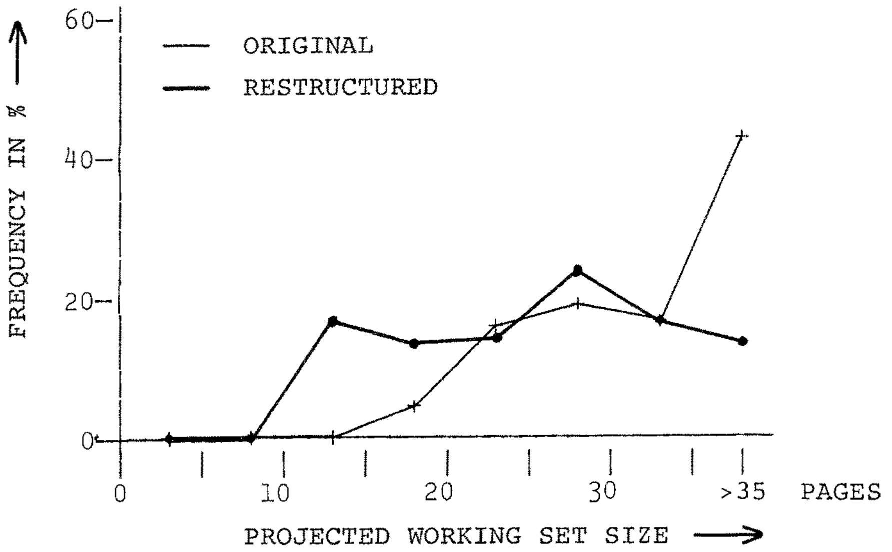


Fig. 4c

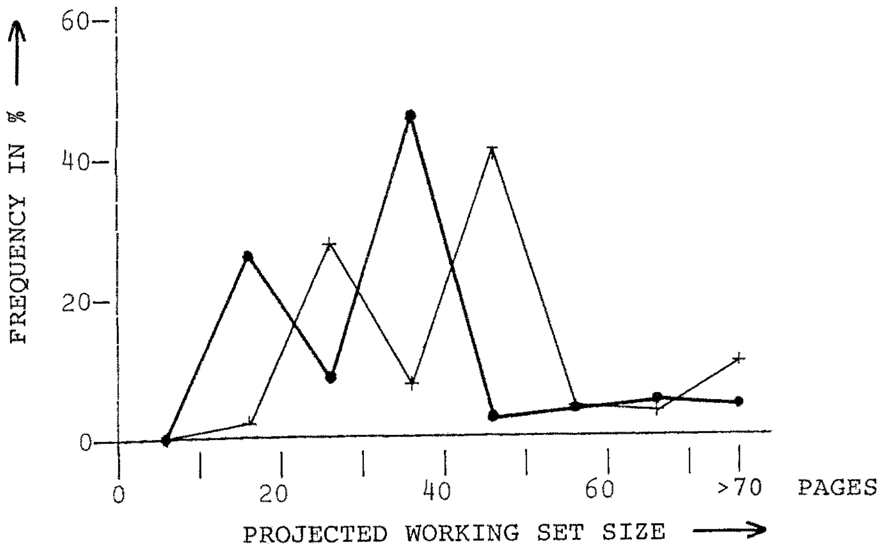


Fig. 5c

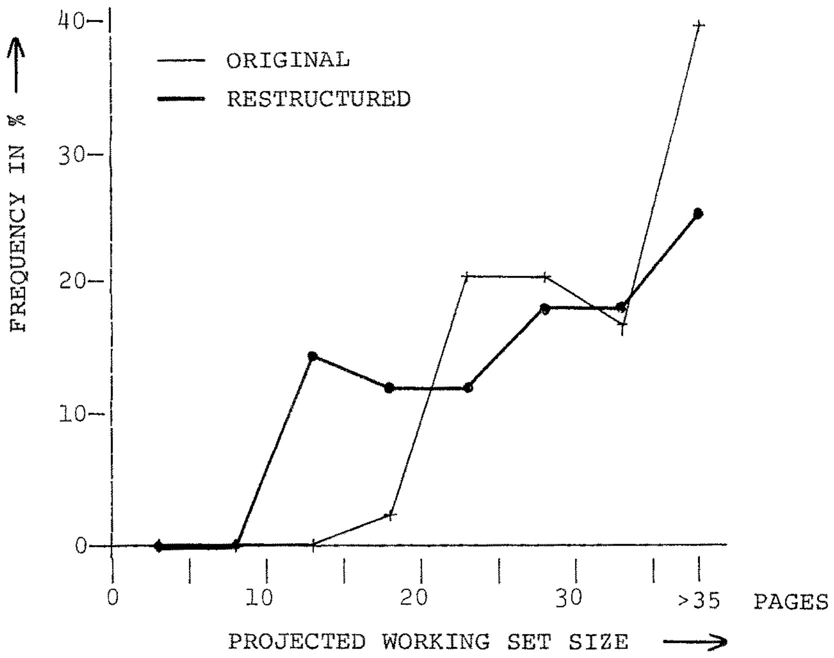


Fig. 4d

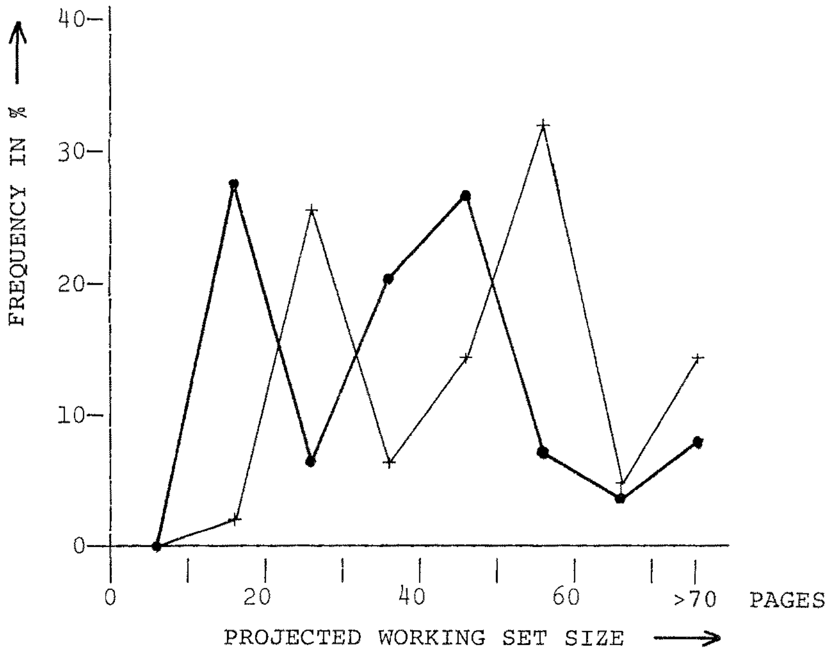


Fig. 5d