COSC 460
RESEARCH PROJECT
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF CANTERBURY

A VIRTUAL MEMORY TECHNIQUE FOR A BCPL COMPILER

Alan Mayo
October 1980

TABLE OF CONTENTS

1. INTRODUCTION

The topic of this honours project is "A virtual memory technique for a BCPL compiler". BCPL is a high level language suitable for writing system software such as editors and compilers. It is available on the Computer Science Department's Data General Eclipse S/130 computer. The compiler is written in BCPL.

During a compilation the BCPL compiler creates a syntax tree and a symbol table in a work-space of fixed size. This places a limit on the size of a compilable BCPL program, which would be removed if a virtual memory technique for the storage of the syntax tree and the symbol table was to be implemented.

The aim of this project was to investigate this possibility. The first step was to gain some knowledge of the working of the compiler by studying the code and to obtain some statistics by inserting probes into the compiler. Using the knowledge gained and by simulating various virtual memory techniques a paging algorithm was designed and implemented.

This report begins with a brief overview of BCPL and a detailed statement of the problem. The studies, design, implementation, and results are then presented followed by a conclusion.

The guidance of Dr M.A. Maclean, Project Supervisor, is gratefully acknowledged.

2. BCPL LANGUAGE OVERVIEW

BCPL stands for Basic Combined Programming Language. A language CPL (Combined Programming Language), developed jointly at Cambridge and London Universities, was simplified by Martin Richards of Cambridge University to BCPL.

The language has most commands offered by other high-level languages, with some additions, but supports only simple data structures. It has advanced looping constructs, call by value (but not name), recursion, Pascal-like local variables and vectors (one dimensional arrays), a global vector (similar to Fortran's Common), and very flexible syntax rules.

The basic unit of data is a cell which is one word of memory. A cell may contain an integer, a bit pattern (e.g. an Ascii code), a pointer to another cell, or a pointer to program code. How the content of a cell is interpreted, is determined by the context in which it is used. This gives the language flexibility, but means that checking for such errors as 'invalid index' cannot be done.

The Global vector is used to implement global variables and global procedures. The cells of the vector, which can be referenced from any part of a program, may contain values or pointers to program code.

3. STATEMENT OF THE PROBLEM

3.1 Present Operation of the BCPL Compiler.

3.1.1 O-code

Source programs are translated into a language, between high level and assembly language, called O-code, which is then translated into assembly code. O-code commands perform basic operations on a hypothetical stack machine. The use of O-code increases the machine independency and portability of the compiler and simplifies the writing of new code generators (O-code to assembly code) for the BCPL compiler.

3.1.2 Phases

A BCPL compilation is comprised of 4 main phases:

i) Lexical and syntactical analysis of the source program to produce a complete syntax tree of the program and a symbol table.

ii) Translation of the syntax tree into O-code.

iii) Generation of Eclipse assembly code from the O-code.

iv) Assembly to produce relocatable binary code.

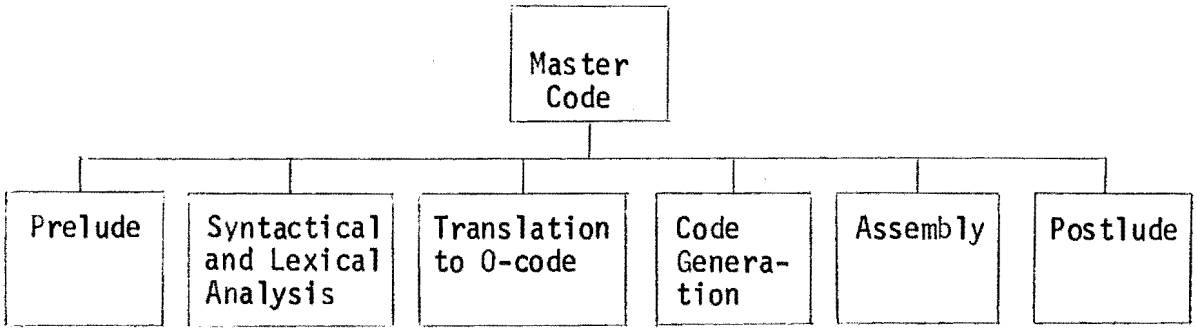
3.1.3 Intermediate Code

The syntax tree and symbol table produced by lexical and syntactical analysis are stored in a vector declared in the compiler. The O-code produced is written to a disc file.

The O-code is translated to assembly code which is then assembled to relocatable binary code. Both the assembly code and the relocatable binary code are stored on disc.

3.1.4 Overlay Structure

Each of the main phases, and also a prelude and postlude, constitute an overlay segment. There is one controlling segment of resident code.

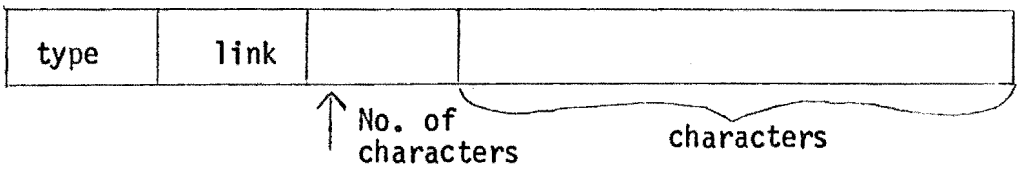


3.1.5 Syntax Tree and Symbol Table

The syntax tree and the symbol table are stored in one vector. Space for the vector is allocated in 'nodes' of varying sizes. The function which allocates and creates nodes returns the actual storage address of the node and these addresses are used to link the structure together.

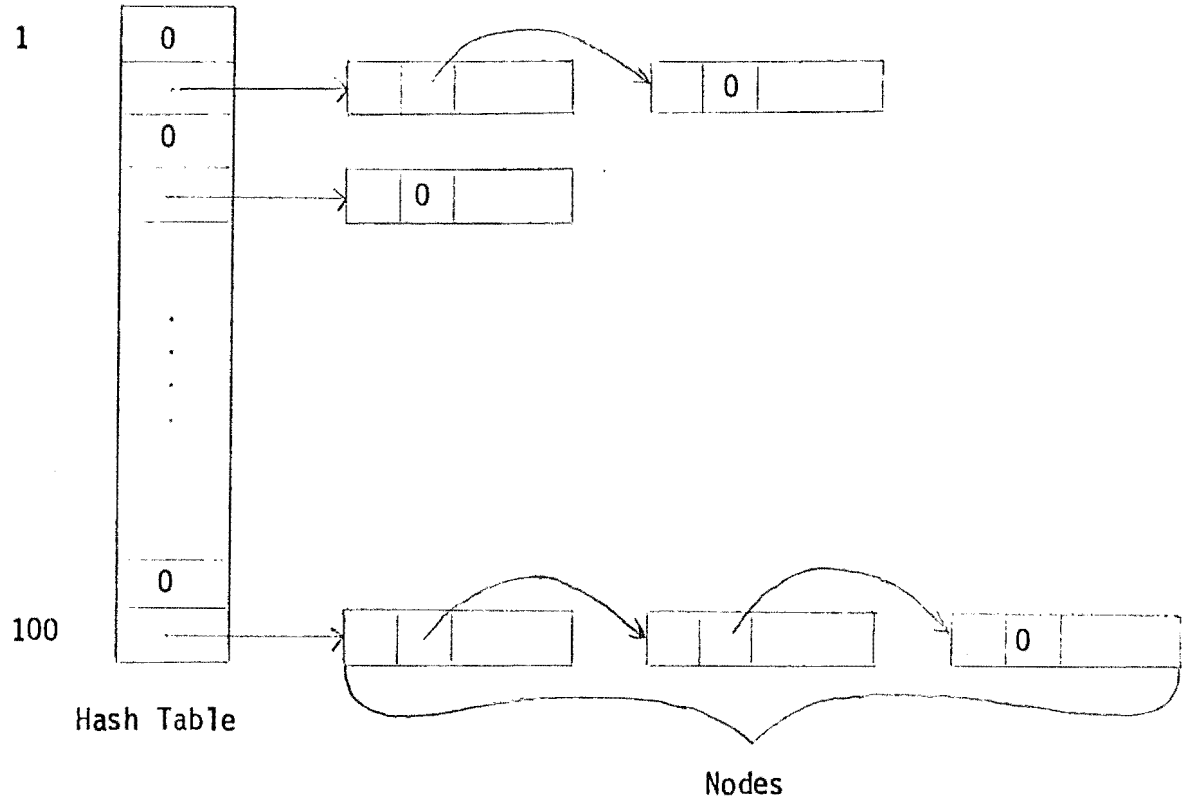
The syntax tree is a single root tree with a variable number of branches off each node. It also has links to the symbol table.

The symbol table is referenced through a hash table whose entries point to linked lists of nodes containing identifiers. The symbol table contains program-declared identifiers and compiler-declared identifiers (i.e. system words). The format of a node is



The type field contains a compiler-declared constant indicating the type of the identifier. Program-declared identifiers are of type S.NAME. Compiler-declared identifiers indicate the relevant system word. Examples are S.GOTO, S.WHILE, and S.TRUE.

The symbol table organisation is drawn below:



3.2 Deficiencies of the Compiler.

i) Since the syntax tree and the symbol table are stored in a vector of fixed length, a program's size is limited to approximately 500 lines of uncommented code. Because of this, when changing a program, a user may inadvertently exceed the maximum program size. To split up a section of code or to move code to another section can require a complex rearrangement of functions and a revision of the global declarations.

ii) The size of the compiler is increased by requiring a large vector in which to store the syntax tree and symbol table. 24K words of memory are needed to run the compiler, which is a limitation when the Eclipse runs in a 2-user mode.

3.3 Objectives of the Project.

The objectives were to investigate the possibility of, and if possible to design and implement, a virtual memory system for the compiler's work space. This would remove the arbitrary limit on the size

of a program and should decrease the size of the compiler. The data structures considered for paging were the symbol table, the syntax tree, and the hash table.

A proviso to the above was that the compilation time of a program should not increase significantly. Lexical and syntactical analysis of the source program and translation of the tree to O-code are the only phases affected by the paging. These phases only take up approximately 20% of the total compilation time, so that a proportionally large increase in their time would not have the same proportional effect on total compilation time.

4. STUDIES OF COMPILER OPERATION

4.1 The Hash table.

4.1.1 Hash Table Size

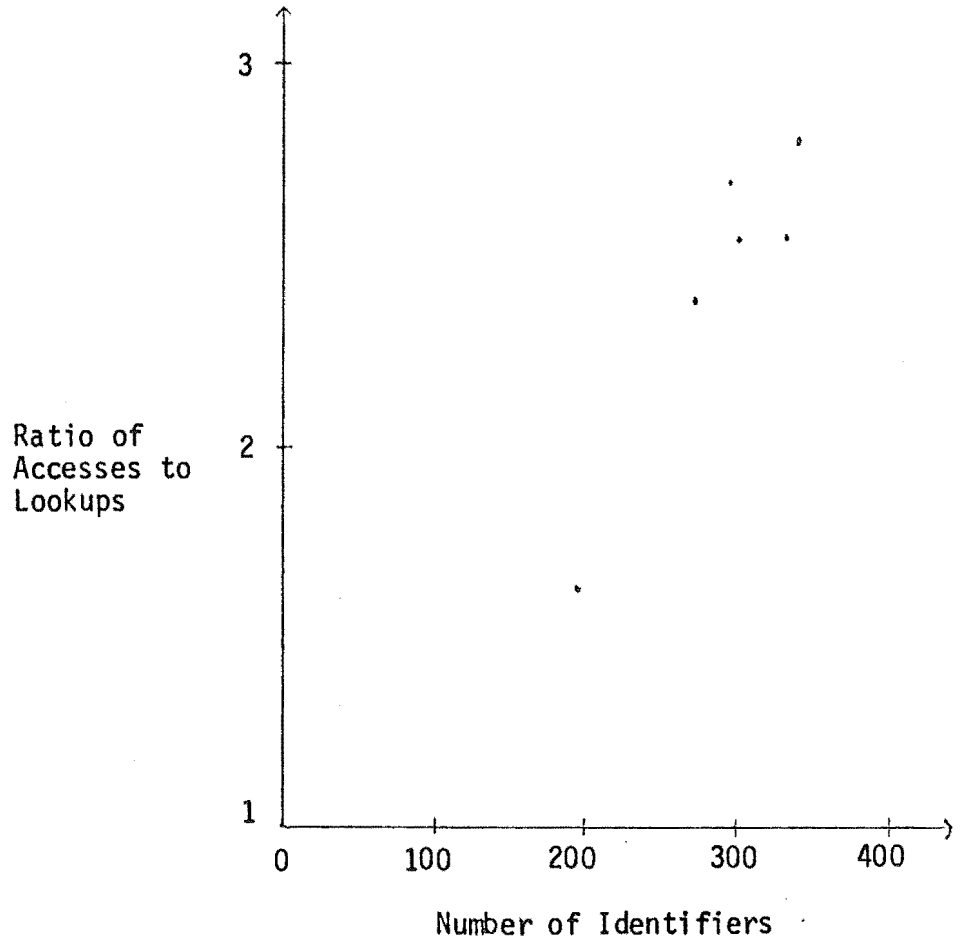
The original compiler's hash table size was 100. This meant that as the total number of identifiers increased beyond two or three hundred, the linked lists would become long, and thus the average number of accesses to locate an identifier would increase.

To further investigate this, code was added to the compiler to accumulate both the number of 'lookups' of identifiers in the symbol table, and the number of accesses to symbol table nodes. The ratio of accesses to lookups could then be compared with the theoretical minimum of 1.

To analyse the operation of the compiler in this and other areas, six sample programs were used. They are sections of a compiler and so should be representative of the type of software the compiler can be expected to compile. The programs and the number of words in their respective syntax trees and symbol tables are:

Name	Size
MST	2663
AET	3786
TRA	6240
LEX	6756
TRB	7205
SYN	7370

The results from compiling these six programs are shown on the next page.



The graph indicates that approximately 2.6 accesses of symbol table nodes per lookup can be expected for a compilation of a program with 300 identifiers. Such programs are common and were found to have greater than 1000 lookups.

If the symbol table was to be paged, the number of page faults, in the worst possible case of a page fault per access, would be greater than 2,600. Since accesses to the symbol table are approximately random, the expected number of page faults should be of the same order of magnitude as the worst case. As this order of magnitude is unacceptably high, further investigation was undertaken as discussed below.

4.1.2 Experiments.

The original compiler's hash table size is determined by a constant in one of the program sections. To evaluate various hash table sizes by changing this constant requires the editing and compiling of the relevant program section, and the reloading of the compiler.

To enable hash table sizes to be evaluated without this rebuilding of the compiler each time, the hash table vector was declared to be the maximum hash table size to be tested (as BCPL does not allow dynamic sizing of vectors). When a compilation started the hash table size to be evaluated was read from a 'command file'. This mechanism has been used throughout the design to allow different values of various parameters to be tested without rebuilding the compiler.

Experiment 1

Hash table sizes of 100 to 1300 in steps of 200 were evaluated. As explained by Knuth[1] the exact hash table size should be chosen with care to ensure a minimum number of collisions. The graphs at the end of the section and especially the anomaly which they show at a hash table size of 900, confirm the value of this advice.

Experiment 2

Knuth[1] gives a method for finding hash table sizes as follows:

Choose H, the hash table size, such that

i) H is prime

ii) $0 \ll (r^k \text{ Modulo } H) \ll H$

where k is a small integer and r is the radix of the alphabet.

The alphabet for the compiler is the 8 bit Ascii character set which has a radix of 256. Prime numbers were evaluated around the values 100 to 1300 with k having the values 1 to 4.

For example the prime numbers 101 and 109 give the following results.

(256^k Modulo 101) yields 54, 88, 5, 68

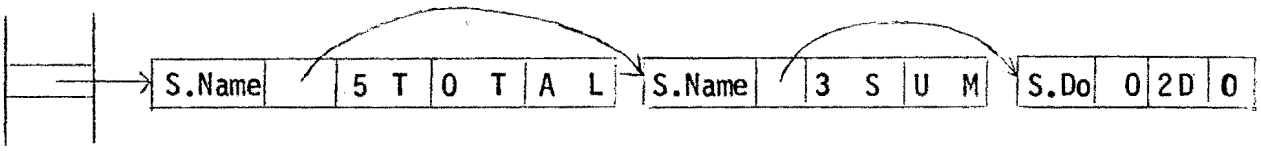
(256^k Modulo 109) yields 38, 27, 45, 75

The hash table size 109 meets the criteria but 101 does not.

Some of the sizes that meet the criteria gave poor results when tested by the compilation of a small program. This may be because the range of k was too small. Other sizes were tested until the following acceptable hash table sizes were found: 109, 307, 521, 701, 881, 1123, and 1303. These sizes were evaluated using all the sample programs and gave good results as shown by the graphs at the end of this section.

Experiment 3

During compilation, identifier nodes are added onto the front end of symbol table linked lists. Therefore program-declared identifiers will be in front of system-declared identifiers, as all system words are declared by the compiler at the beginning of a compilation. For example if SUM, TOTAL, and DO have the same hash value and SUM is found before TOTAL the logical organisation will be

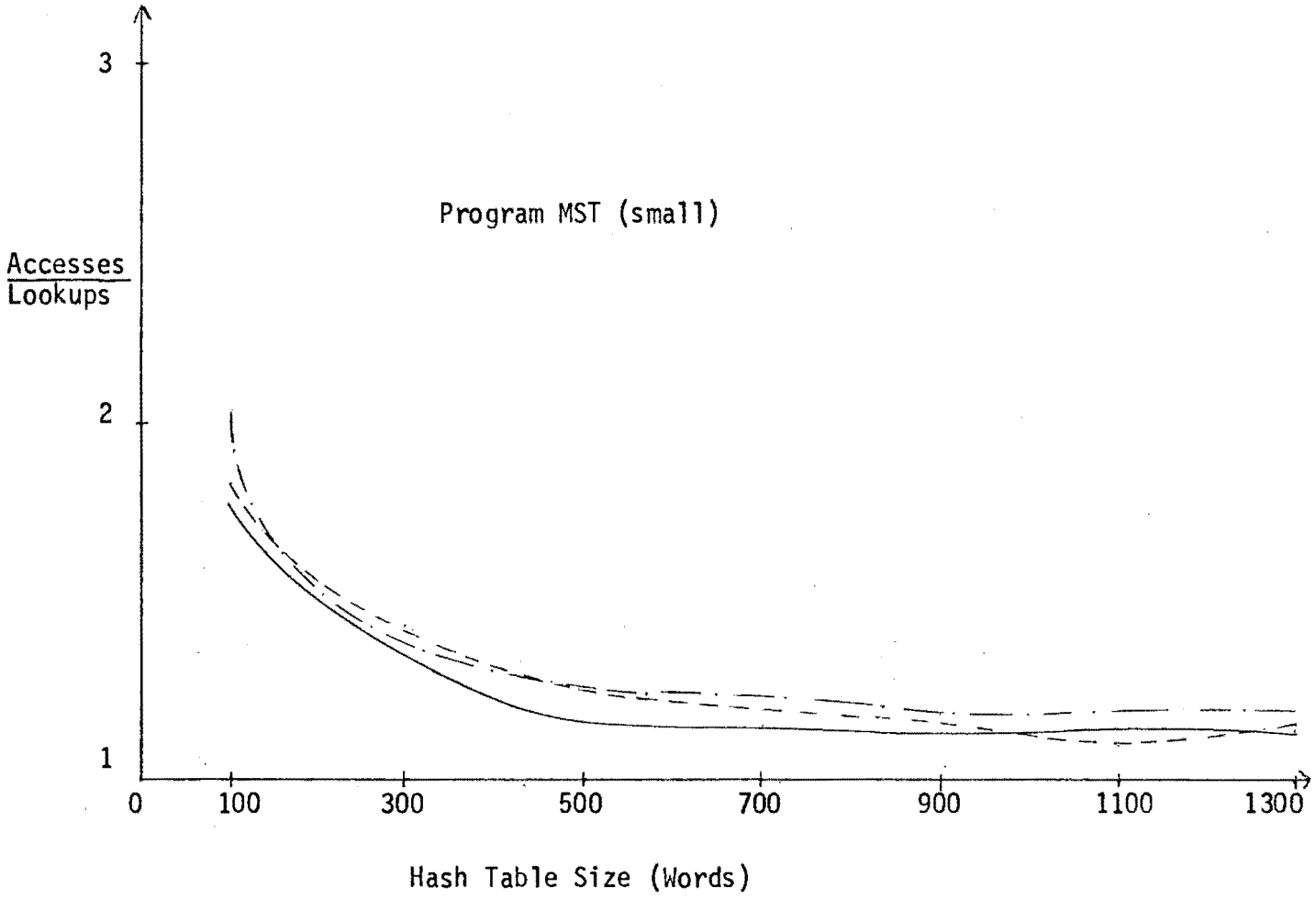


hash table

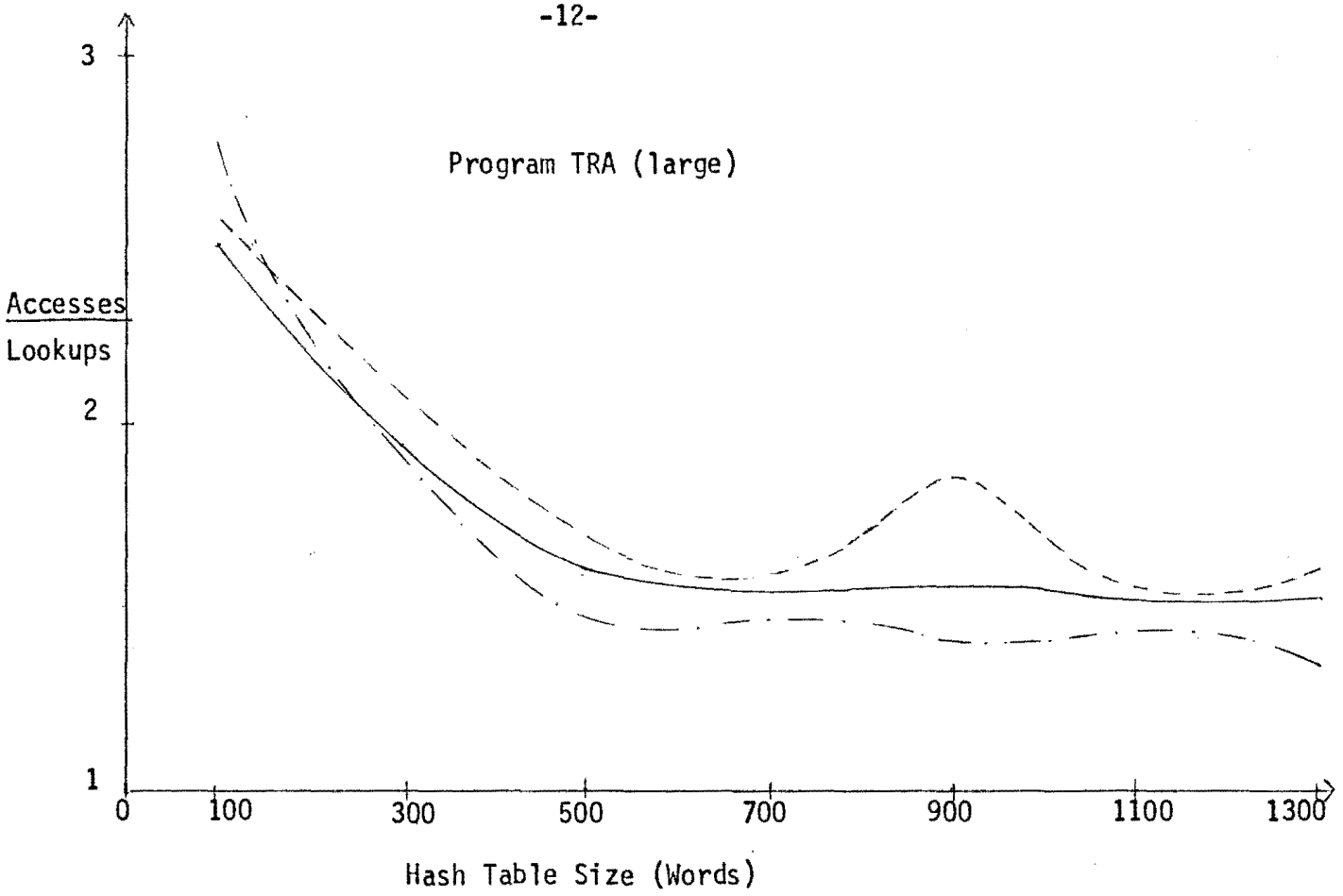
Whenever DO is searched for, SUM and TOTAL must be tested first. If SUM and TOTAL are rarely referenced this will be inefficient.

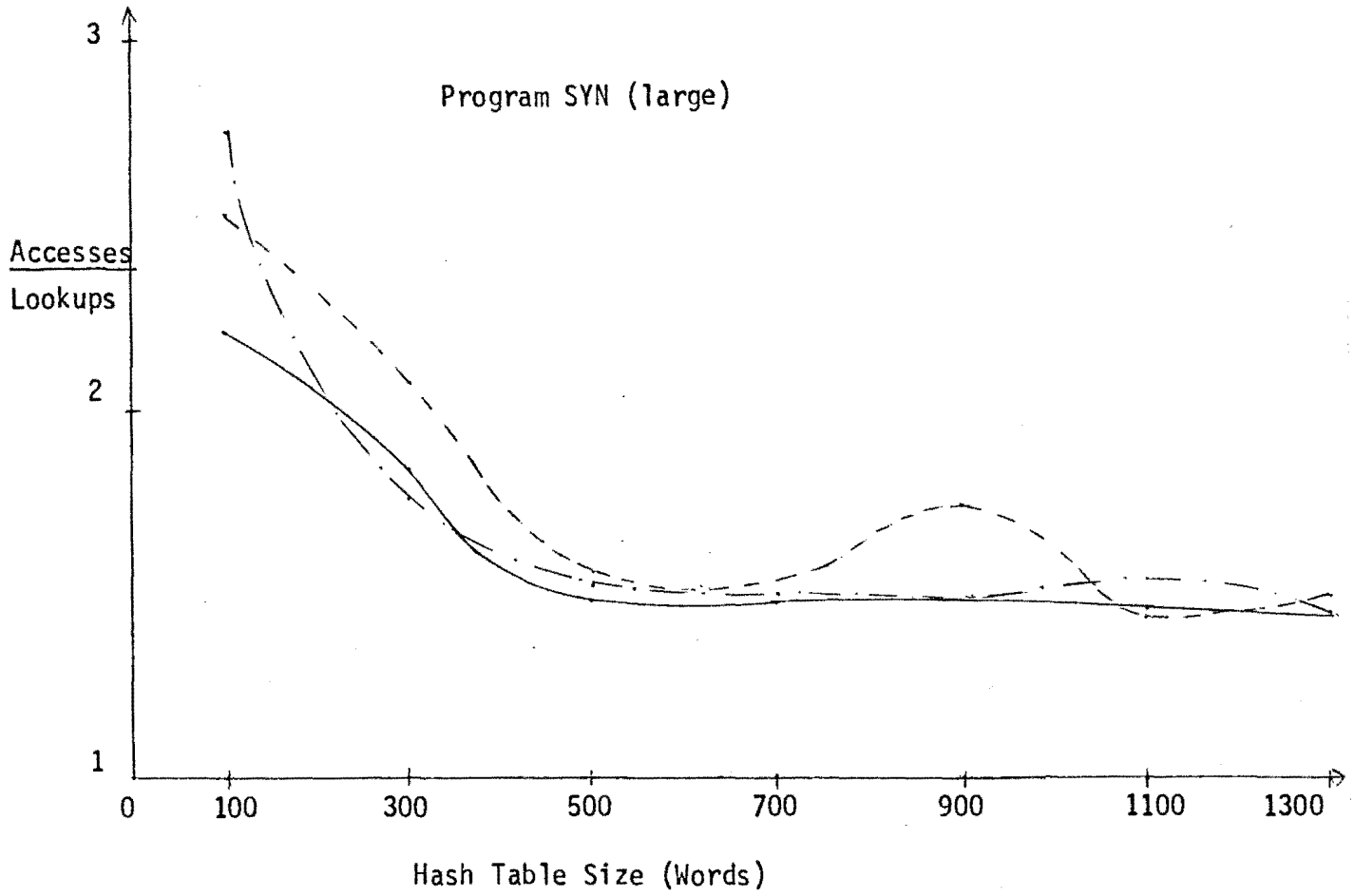
The compiler was modified to add new identifier nodes onto the ends of the linked lists.

The graphs below show that for small programs the initial ordering is better than the new ordering. For some large programs the new ordering is better, but for others the two orderings give similar results.



Ratio of Accesses to Lookups for different Hash Table Sizes and Symbol Table Organisations


Program TRA (large)
Accesses
Lookups
3
2
1
0
100
300
500
700
900
1100
1300
Hash Table Size (Words)


Program SYN (large)
Accesses
Lookups
3
2
1
0
100
300
500
700
900
1100
1300
Hash Table Size (Words)

4.2 Syntax Tree and Symbol Table.

4.2.1 Storage Method

The symbol table and the syntax tree are stored in a vector of fixed size with nodes of each structure intermingled throughout the vector.

During translation, the order in which syntax tree nodes are accessed depends on their positions in the 'logical' syntax tree. Any correlation between nodes logical positions and their physical positions in the vector may be able to be exploited by a paging system to reduce page faults.

By examining the compiler code it was apparent that during syntax analysis a mixture of recursive and iterative techniques were used. The recursion is often embedded within procedures which create nodes. This results in the sub-branches of the tree being created first. Interation has the opposite effect; the roots are created first.

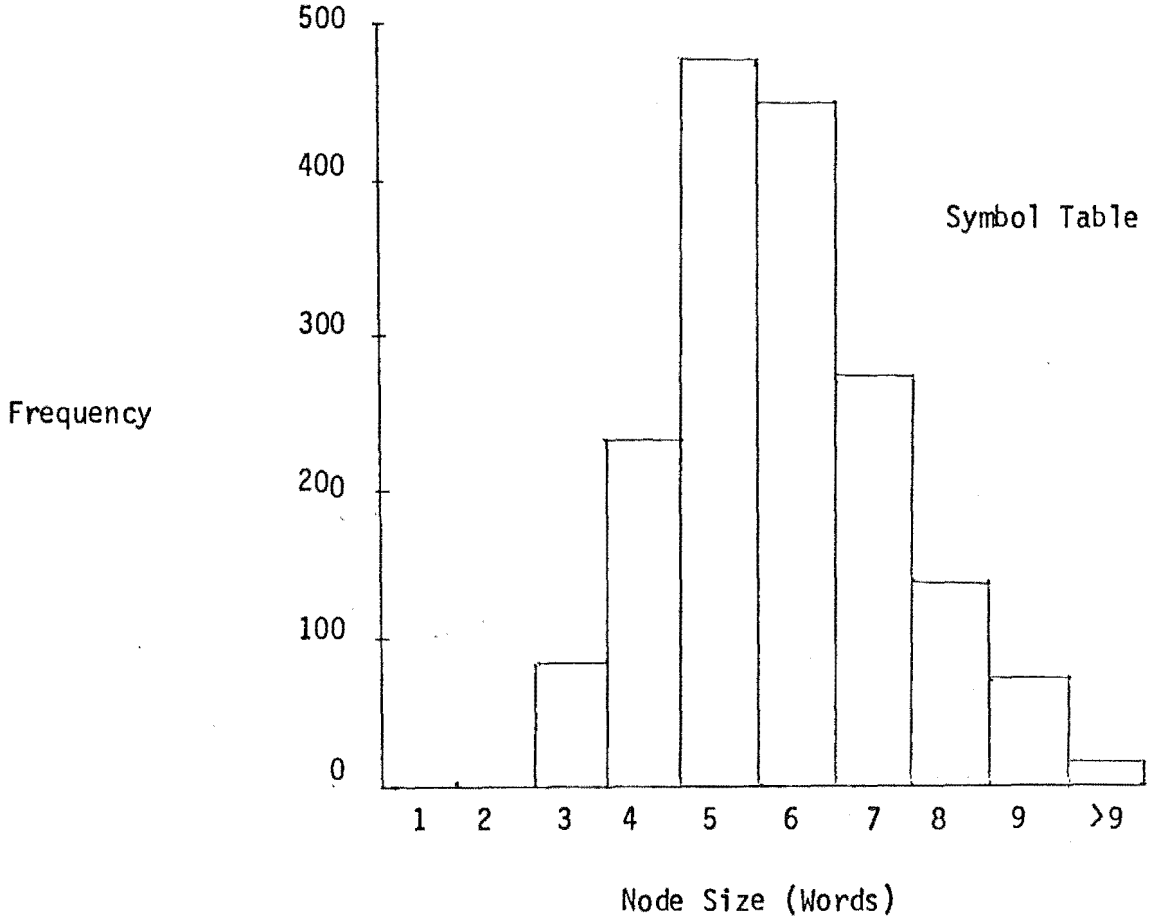
The structure is further complicated by the symbol table nodes. The node for a 'new' identifier is created when the identifier is first found by the lexical analysis procedure, which is called from many different syntactical analysis procedures. This means that when a specific construct is analysed, the positions of the resulting syntax tree nodes will have little correlation to the positions of the resulting symbol table nodes.

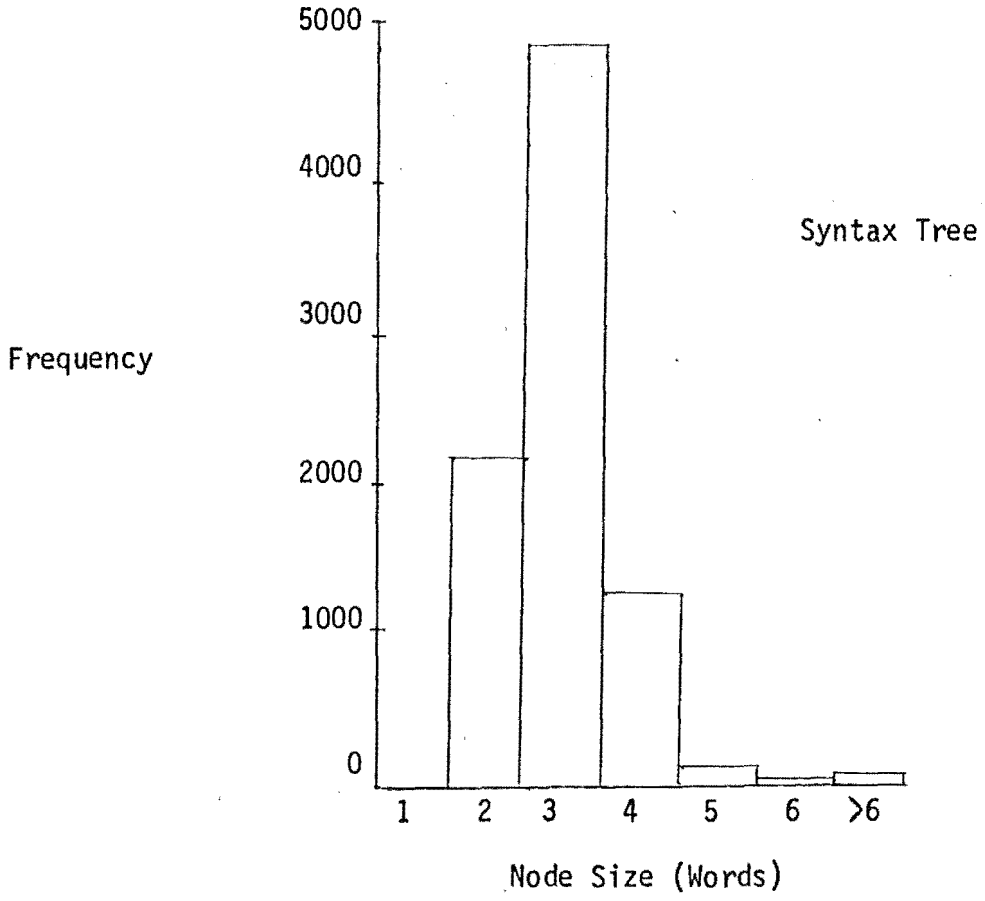
A desk check of the syntactical and lexical analysis of a small program confirmed the above. There was some relationship between the logical and physical positions of syntax tree nodes, but the use of two techniques during syntactical analysis destroys any overall structure. Therefore a linear pass through the whole vector during translation is unlikely, but linear passes through smaller sections of the vector are likely.

4.2.2 Node Size Distribution

Statements were inserted into the compiler to accumulate and print out the number of nodes of each size for the syntax tree and the symbol table.

The figures were totalled over the six program sections to yield the following histograms.





Distribution of Node Sizes

The distributions show that the majority of nodes are syntax tree nodes of 2, 3, or 4 words (note the different frequency scales).

4.2.3 Relative sizes

The figures totalled for the six programs were:

Syntax Tree	25,030 words
Symbol Table	9,960 words

The ratio of tree words to table words is approximately 5/2. This ratio is used later in determining how the syntax tree and symbol table are stored on disc.

5. DESIGN OF A PAGING SCHEME

5.1 Criteria used.

In the following design, whenever practicable, simulation has been used to evaluate different paging strategies. The inputs to these simulations were the compilation records of the 6 sample programs used in the previous section.

When making design decisions the following criteria were used:

i) When there has been a conflict between maximising the compiler's performance for compilations of large programs and small programs, more emphasis has been placed on the compilations of large programs. This is because a proportionally large increase in the compilation time of a small program is preferable to the same increase in a large program (most BCPL programs are 'systems software', and thus tend to be large programs).

ii) A proportionally large increase in the time taken by the lexical and syntactical analysis phase plus the translation phase is allowable, as this time constitutes only approximately 20% of the total compilation time.

iii) To ensure a decrease in the compiler's size, any data structures and code inserted must occupy less space than the compiler workspace that will be removed.

5.2 Choice of a Hash Table Size.

By examining the graphs of section 4.1 it can be seen that a significant improvement is gained by increasing the hash table size beyond 300 but no significant improvement is gained from increasing the size beyond 701. To determine which size to use (521 or 701), and which ordering of the linked lists to use, the weighted averages of the ratio of lookups to accesses over all the sample programs were computed. The weights used were the number of lookups. The weighted averages were:

Order	Size (words)	
	521	701
Initial Order	1.56	1.52
Reverse Order	1.51	1.50

The improvement gained by the reverse order is not enough to justify the added complexity that is involved in the code, and the improvement gained by having a size of 721 is not enough to justify the extra 180 words of memory. Therefore the initial order and a hash table size of 521 were decided upon.

5.3 The Replacement Strategy.

The replacement strategy chosen was 'Least Recently Used' (LRU). LRU is generally accepted to be the best replacement strategy, but is not often used by virtual-memory operating systems due to its high cost in terms of hardware and time. Normally an approximation of LRU is used.

Since the paging of the syntax tree and symbol table is to be performed solely by software, the cost of implementing LRU is not high. As pointed out by Shaw[2], the page size and the number of pages are more critical factors than the replacement strategy.

For the above reasons LRU was chosen and no other replacement strategies have been considered.

5.4 What to Page.

The three data structures which can be paged are the hash table, the syntax tree, and the symbol table.

The compilation of a typical large program generates approximately 1000 calls to 'lookup' identifiers, which results in 1000 accesses of the hash table. If the hash table is paged, the number of page faults caused by each memory word saved can be calculated as follows:

$$\text{Number of faults} = \text{Total accesses} \times \text{Probability of a fault}$$
$$= 1000 \times \frac{1}{521}$$
$$\doteq 2$$

A saving of one word of memory does not warrant the cost of 2 page faults so the hash table was not paged.

Paging only one of the syntax tree or the symbol table would not remove the arbitrary limit placed on a program due to the fixed size of the other data structure. Therefore it was decided to page both data structures.

5.5 To Split or not to Split.

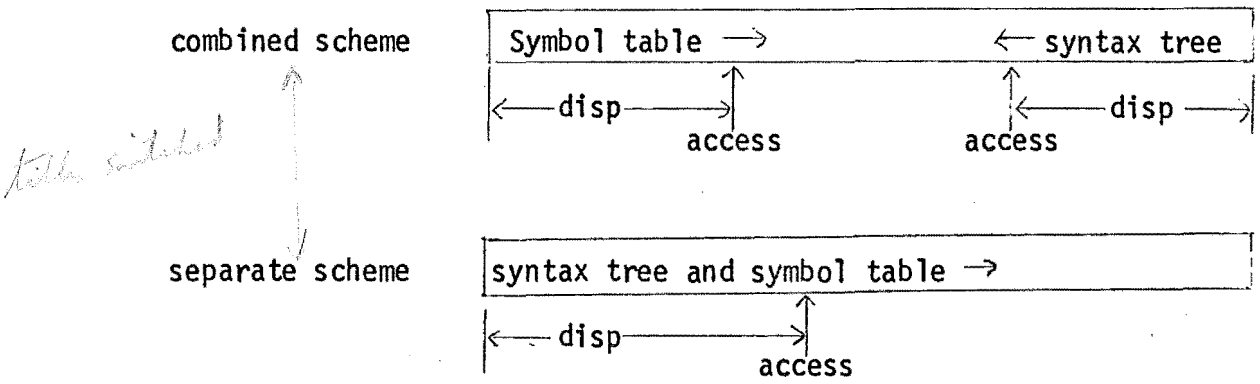
The original compiler created the syntax tree and symbol table in the same area of memory. The paging could either be done using this organisation, or the two data structures could be paged separately.

If the combined scheme was to be implemented, accesses to nodes would be made by two relatively independent processes. Accesses to symbol table nodes would be approximately random, but accesses to the syntax tree might have some order, which could be utilised by the paging system to produce better results. Therefore combining the two sequences of accesses into one might nullify any improvement gained from the order of syntax tree accesses.

To compare the two methods, the compiler was altered to produce a file of node access numbers, and a simulation program was written as follows:

i) Code was inserted in the compiler to allow the data structures to grow separately from opposite ends of the vector or together from one end of the vector. A parameter in the command file controlled which scheme was to be used during a compilation.

ii) All references to the syntax tree and symbol table were replaced by procedure calls to enable these references to be trapped. These procedures performed the appropriate referencing, and also calculated and wrote to disc the displacement of the referenced address from the end of the vector from which the data structure was growing. This displacement is shown as 'disp' in the diagrams below.



The displacement represents the virtual address of the word accessed in a paged memory system.

Separate files of numbers were produced for the lexical and syntactical phase and for the translation phase. Thus for the combined scheme two files were produced and for the separate scheme four files were produced.

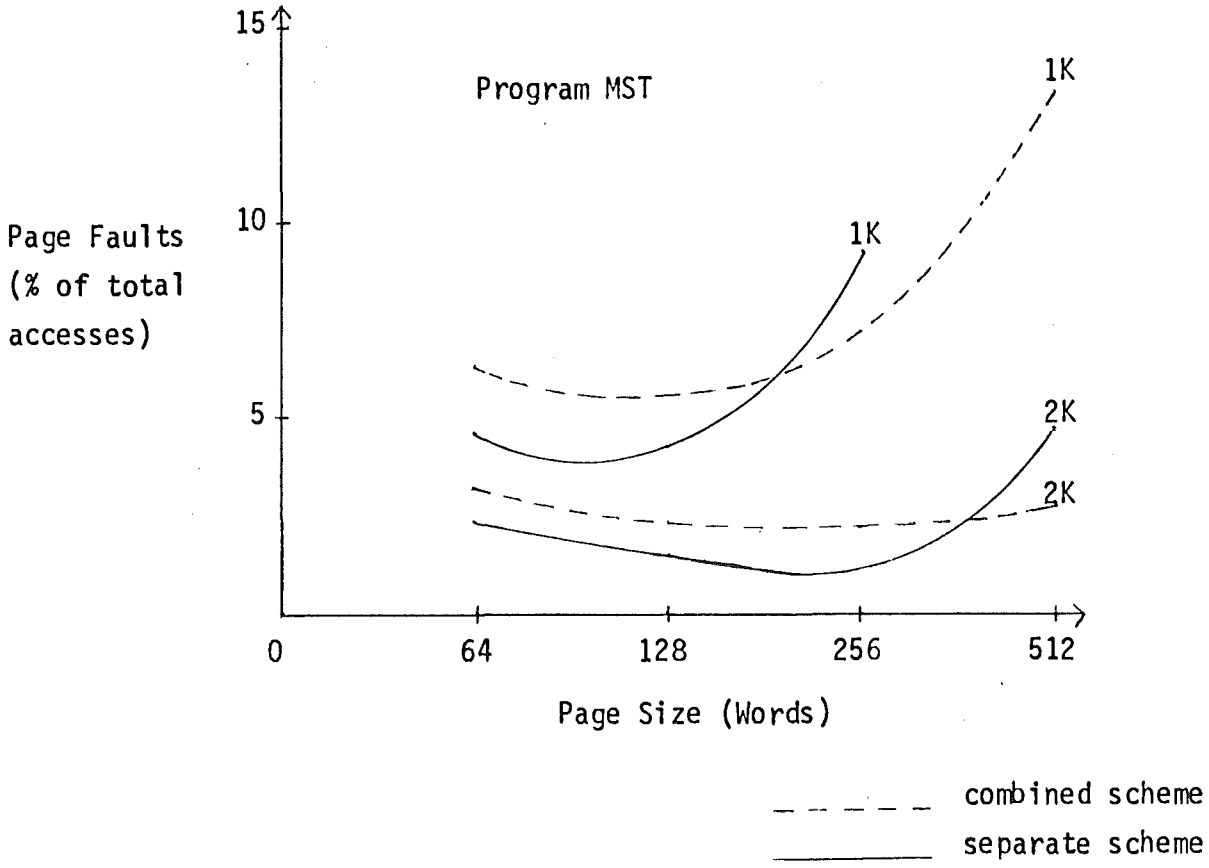
iii) Parts of the compiler had to be rewritten due to some variables being able to contain program addresses or syntax tree addresses, which complicated the insertion of paging procedure calls.

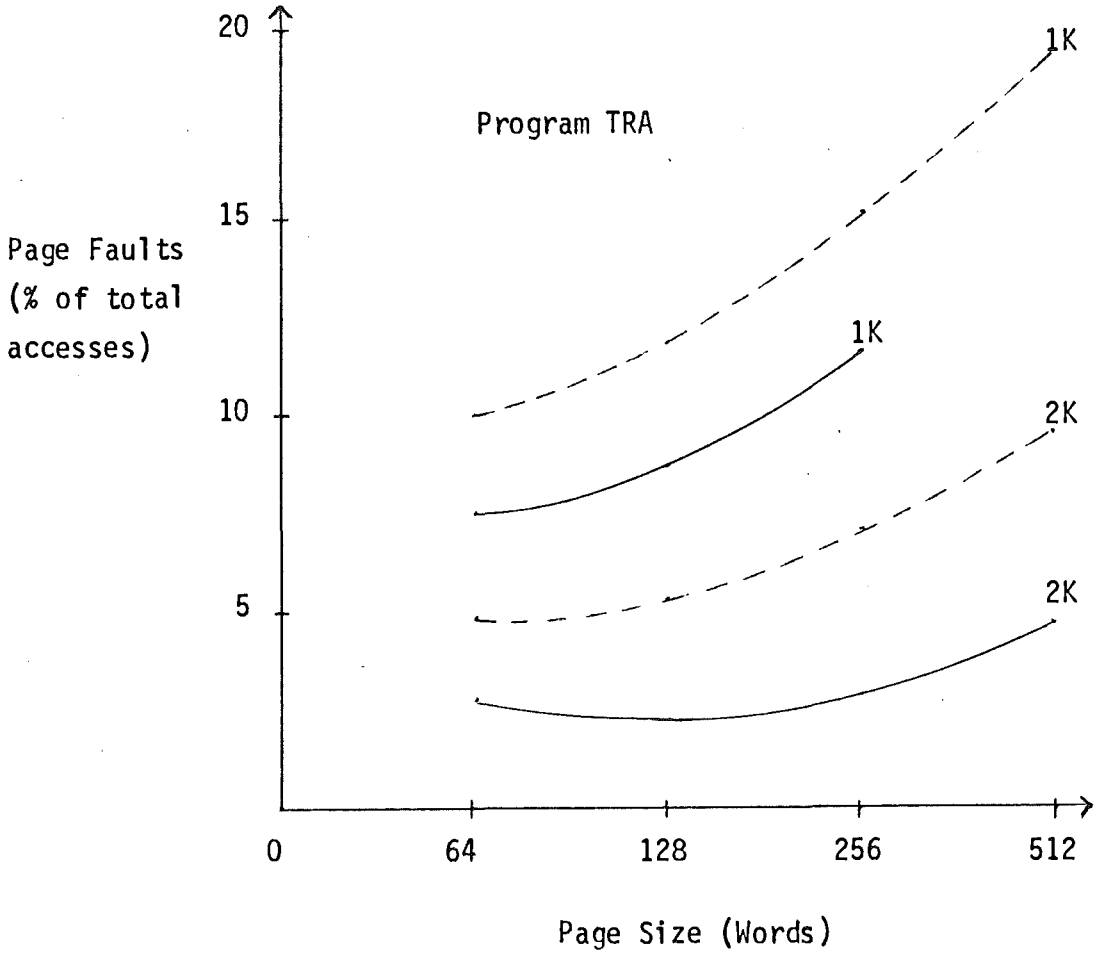
iv) A simulation program was written to simulate a paging scheme with a least recently used replacement algorithm. Inputs to the program were: a page size, a number of pages and a file. The program processed the file and produced the number of page faults.

Files of accesses were obtained from the compilation of sample programs MST and TRA. The files were processed by the simulation program with total memory sizes of 1K and 2K words, and for page sizes of 64, 128, 256 and 512 words.

For the separate scheme the total number of pages had to be divided between the syntax tree and the symbol table. A ratio of three pages of symbol table to one page of syntax tree was found to be superior to other simple ratios.

The totals of numbers of page faults for each page size, and total amount of memory, were used to produce the graphs below.





Performance of Separate and Combined Schemes with 1K and 2K words of Memory

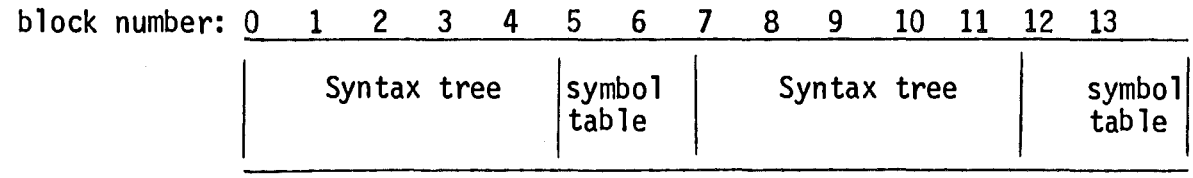
For the large program (TRA) the graph shows that the separate scheme is superior for both memory sizes and all page sizes. For the smaller program (MST) the best results obtained were for the separate scheme.

It was therefore decided to implement the separate scheme, but to do it in such a way that with only minor changes the performance of the combined scheme could also be evaluated.

5.6 How to store Two Data Structures on Disc.

One storage possibility was for the syntax tree and symbol table to be paged to two different disc files. However, to avoid the complications of having to handle two files, it was decided to store both data structures in one file as done by the CHEF editor on the Eclipse S/130.

To achieve this, it was decided to calculate different disc block numbers for each data structure; this was done in such a way that the two structures would be interleaved in the file. Since the ratio of syntax tree words to symbol table words is approximately 5/2, the disc file blocks were allocated in this way, resulting in the following organisation.



Since the ratio 5 to 2 is an average ratio, there may be gaps in the file, but as it is a temporary file this is unimportant.

The disc block numbers of the two data structures are computed from the virtual block numbers as follows:-

Let DB be the disc block number and VB be the virtual block number. For the syntax tree:

DB = (VB DIV 5) * 7 + (VB REM 5).

For the symbol table:

DB = (VB DIV 2) * 7 + (VB REM 2) + 5.

5.7 Increasing the Maximum Program Size.

The restriction on source program size is imposed by the size of the vector which stores the syntax tree and symbol table. The vector's size is 8K words which requires only 13 out of the 16 bits available for addressing purposes. In a virtual memory system, the size of the virtual memory space is determined by the addressing range of the word used, and the size of the 'addressable unit'.

The addressing range of a 16-bit word is 64K, which gives a theoretical 8-fold increase in the memory available for the syntax tree and symbol table. Due to links in the syntax tree, which may point to syntax tree or symbol table nodes, one bit of the addressing word must be used to indicate the type of link. To achieve this, it was decided to use the positive numbers for syntax tree virtual addresses, and the negative numbers for symbol table virtual addresses. As the ratio of syntax tree words to symbol table words is 5 to 2, only 70% or 44.8K words of the virtual memory space is likely to be utilised, but this still increases the maximum program size to 5.6 times the original size.

This increase is quite sufficient, but if more space was required the addressing range could be split at -14,000 instead of 0 to give no wastage.

If still more space is required, or if some of the bits of the addressing word are required for other purposes, the addressable unit size could be increased to 3 words. Using the figures from section 4.2.3, this would result in 20% wastage due to unused words in 3 word units, but would still increase the virtual memory space 2.4 times.

To get the maximum possible usage of the virtual memory space, nodes could be allowed to overlap page boundaries. Accessing overlapping symbol table nodes would produce a page fault which might not have occurred if overlapping nodes were not allowed.

For this reason, and because not allowing overlapping nodes results in little wastage of virtual memory space (since the majority of nodes are 3, 4, or 5 words long), it was decided not to allow overlapping nodes.

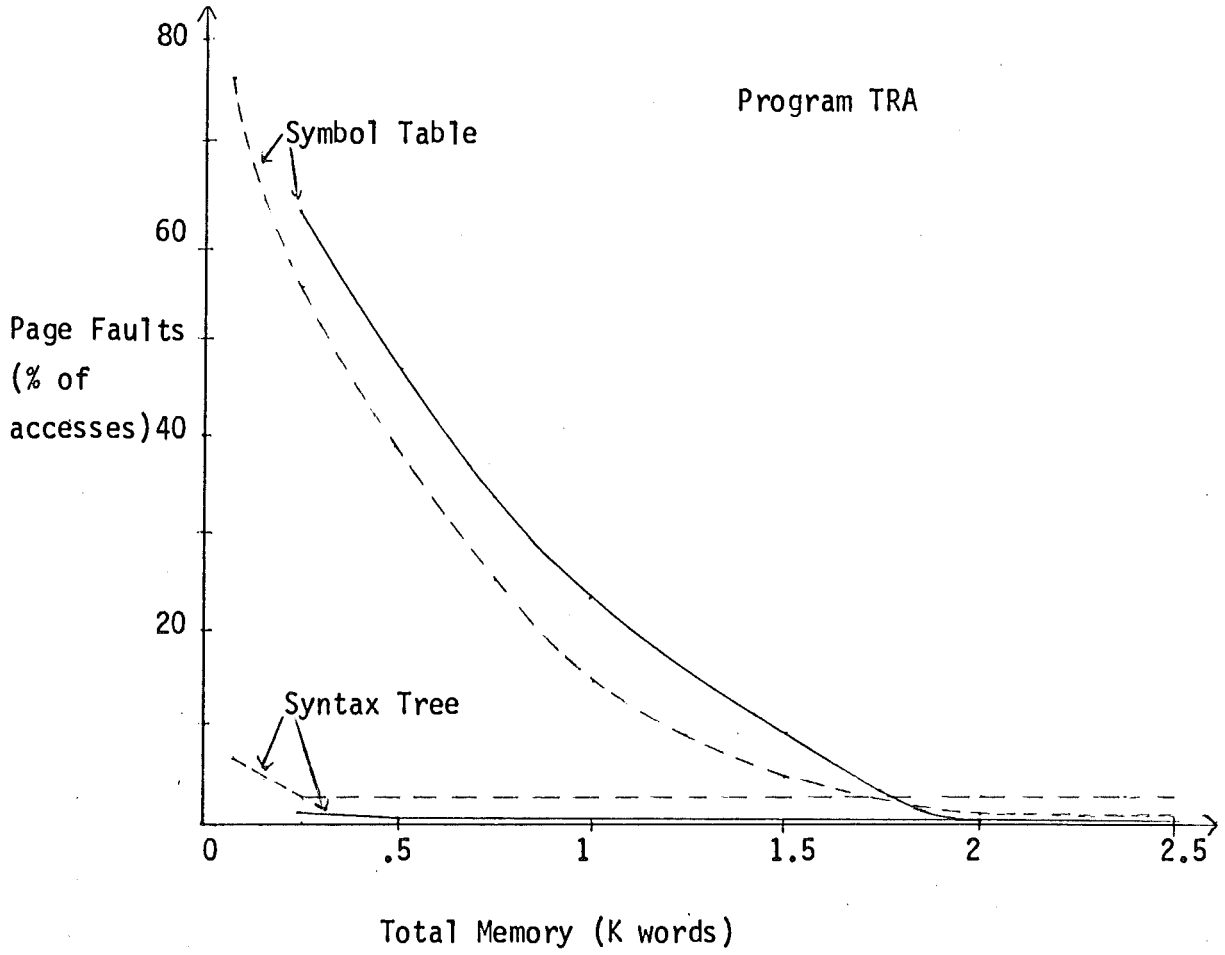
5.8 Page Size and Number of Pages.

Since the cost of evaluating page sizes and numbers of pages would be high, if performed once the virtual memory system was implemented, it was decided to do some preliminary investigation. It was not expected that this would give the final page size and number of pages but that the effect that the page size and number of pages have on the number of page faults would be shown. To obtain a detailed appraisal of the system, the lexical and syntactical analysis phase and the translation phase were considered separately for the syntax tree and the symbol table.

Page sizes of 64 and 256 words and a total memory space of up to 2K words for each data structure were tested. One page size is small and the other is relatively large, and both sizes can be read or written via Eclipse S/130 system directives. No special system directives exist for other page sizes.

The software additions to the compiler described in section 5.5 were used to obtain files of access numbers for four sample programs. Simulations were performed, using these files as input, for varying numbers of pages and the two page sizes.

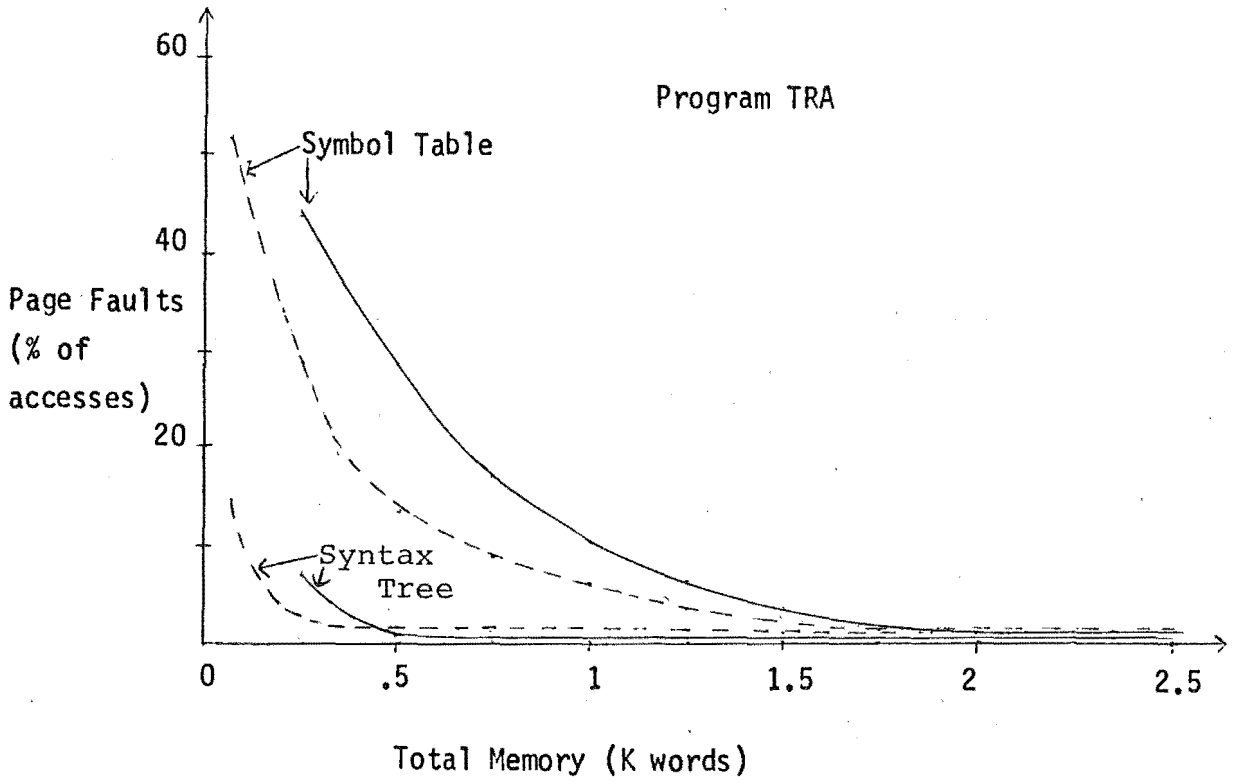
The results obtained from the simulations were similar for all the programs, therefore the results for only TRA are presented.



— — — — — Page Size = 64 words

——————— Page Size = 256 words

Syntax Tree and Symbol Table Page Faults produced by the Syntactical and Lexical Analysis Phase



Syntax Tree and Symbol Table Page Faults Produced by the Translation Phase

The graphs show that very little memory is required for the syntax tree, and that the page size is not critical as the number of page faults is small.

As the amount of time a page fault uses is not known, it is impossible to put an upper limit on the number of permissible page faults. If the ratio of page faults to accesses is to be kept below 2% for the symbol table, many pages are needed and the page size is not critical.

If more than a 2% ratio is acceptable, the 64 word page size is better. The symbol table lines indicate a nearly linear relationship between page faults and size of memory. Therefore, up to a certain limit, each extra page used gives a constant decrease in the number of page faults. The upper limit is reached when the whole symbol table fits into memory.

6. IMPLEMENTATION

6.1 Code.

The design of the previous section was implemented. The calls to procedures required for the paging scheme had been inserted previously to enable the collection of access numbers, and so only the paging procedures had to be written. These were grouped together in a new compiler section (see Appendix A) that was made memory-resident, since two overlays syntactical and lexical analysis, and translation, contain calls to the procedures.

Procedures were written to return the value stored at a virtual address, and to return the actual storage address corresponding to a virtual address. These procedures make calls, where appropriate, to procedures which pick the least recently used page, write a page to disc, and read a page from disc.

A record of whether a page has been written on while in memory, is kept to avoid writing unaltered pages to disc. The time of the last access to each page in memory is kept to enable the least recently used page to be determined.

The procedures which read and write pages, call BCPL library procedures which use system directives to perform the reading and writing.

The debugging of the software proved difficult and it was only by stepping through a compilation of a small program that all the bugs were removed. To gauge the performance of the compiler, counters were inserted to count accesses and faults and to time the compiler, and were removed after the page size and number of pages to use were determined. A sample of the output produced by these counters is presented in Appendix B.

6.2 Page Size and Number of Pages.

The first organisation tried was a 256 word page, 4 pages of syntax tree, and 10 pages of symbol table. The times for compiling TRA with the

original non-paged compiler and the new paged compiler were:

	Non-paged	Paged
Lexical and Syntactical, and Translation Phases	21	26
Other Phases	115	115
Total	136	141

As this organisation gave a very small increase in time, others were tried as shown below. The times are the total time of the syntactical and lexical analysis phase and the translation phase, in seconds.

Memory (K words)	Page Size 64 words	Page Size 256 words	Ratio of Tree Pages to Symbol Table Pages
3.5	28	26	5:2
3	28	27	3:1
2.5	29	31	7:3
2.0	30	47	3:1
1.5	32	69	5:1
1.25	37	84	4:1
1	42	100	3:1

Since a decrease in the compiler's size was desirable a page size of 64 was chosen. A page size of 256 words was slightly superior for 3.5K and 3K words of memory, but for the more important smaller memory sizes it was very inferior.

A total amount of memory of 1.5K words was chosen, as increasing the amount of memory above 1.5K gives only a slight decrease in time (1 or 2 seconds per .5K words), and decreasing the amount of memory gives a significant increase in time (10 seconds per .5K words).

The 1.5K words of memory in the table above were split into 4 pages of syntax tree and 20 pages of symbol table for the 64 word page. This split was obtained by trial and error and is approximately optimal for the compilation of the sample program TRA, but different programs will have different optimal splits. A total memory size of 3.5K words split into 16 pages of syntax tree and 40 pages of symbol table, which gives each data structure at least as many pages as likely in any split of 1.5K words, decreased the compilation time by a maximum of 4 seconds when tested on 4 programs. Therefore it is safe to assume that TRA's optimal split is approximately optimal for all programs.

In summary the following values were chosen

Page size	=	64 words
Total memory	=	1.5K words
Syntax Tree pages	=	4
Symbol Table pages	=	20

7. RESULTS

7.1 Compilation Time.

The two most widely used operating systems on the Eclipse are NEWSYS and CUSYS, both which support Foreground/Background operation. With NEWSYS there is 42K of memory available for user programs, but as CUSYS has smaller buffers there is 50K available.

The following results from timing complete compilations were obtained with NEWSYS.

Program	Non-paged Compiler	Paged Compiler	% Increase
MST	59	62	5.1
AET	90	98	8.9
TRA	138	157	13.8
LEX	148	164	12.2
TRB	149	170	14.1
SYN	152	170	13.8

The compiler was tested with CUSYS to determine the effects of the system buffers. The time taken for the total of the syntactical and lexical analysis phase and the translation phase is shown below for two sample programs.

Program	NEWSYS	CUSYS
MST	13	17
TRA	30	48

The results show that the size of the operating system buffers has a substantial effect on the time taken to service page faults.

The compiler was altered to allow the two data structures to be combined into one as in the original compiler. The total time of the syntactical and lexical phase and the translation phase is shown below.

Memory (K words)	Page Size: 64 words		256 words	
	Separate	Combined	Separate	Combined
2	30	47	47	79
1.5	32	48	69	99
1.25	37	51	84	114
1.1	42	55	100	136

7.2 Compiler Size.

The changes to the compiler size due to the removal of the work-space vector, the addition of page buffers and tables, a larger hash table, and the addition of extra code are summarised below.

Increases - Code	1.26K
- Hash table	.41K
- Page buffers and tables	1.57K
	3.24K
Decreases - removal of vector	7.75K
Net decrease in size	4.51K
Original Compiler Size	23.61K
Less Net decrease	4.51K
New Compiler Size	19.10K

7.3 Page Wastage.

The maximum wastage of virtual memory space due to nodes not being allowed to overlap page boundaries was 180 words, which is only 2% of the total space used.

7.4 Confirmation of the Design.

The results obtained from testing various page sizes and numbers of pages confirmed that:

1. a 64 word page produces fewer page faults than a 256 word page;
2. the best results are obtained by having more syntax tree pages than tree pages;
3. the projected numbers of faults produced by the simulation program are accurate;
4. separating the syntax tree and the symbol table is worthwhile as it reduces compilation time;
5. page wastage due to not allowing nodes to overlap page boundaries is insignificant.

8. CONCLUSIONS

The implementation of the paging system was successful because:

i) the limit on source program size was increased 5.6 times to approximately 3,000 lines of uncommented code,

ii) the increase in compilation time averaged 11.3% or 14 seconds which is quite acceptable,

iii) the compiler's size was decreased from 23.61K words to 19.10K words.

The hash table size and the page size have a significant effect on the compiler's performance.

The splitting up of the syntax tree and the symbol table improves the compiler's performance but is less significant.

REFERENCES

1. Knuth, D.E. (1973) 'The Art of Computer Programming, Vol. 3.' Addison-Wesley, p. 508-509.
2. Shaw, A.C. (1976) 'The Logical Design of Operating Systems', Prentice-Hall, p. 143.

BIBLIOGRAPHY

1. Habermann, A.N. (1976), 'Introduction to Operating Systems', Science Research Associates.
2. Knuth, D.E. (1973), 'The Art of Computer Programming, Vol. 3.'
3. Madnick, S.E. and Donovan, J.J. (1974), 'Operating Systems'
4. Richards, M. 'BCPL: A Tool for Compiler Writing and Systems Programming', AFIPS Conference Proceedings, Vol. 34 (1969), p. 557-566.
5. Richards, M. and Whitby-Strevens, C. (1979), 'BCPL the Language and it's Compiler', Cambridge University Press.
6. Shaw, A.C. (1976), 'The Logical Design of Operating Systems', Prentice-Hall.

APPENDIX A: Program Listing of Paging Software

```
                                    - 1-                                AET
                                                                        ---

////////////////////////////////////////////////////////////////////////////////
//
//
//    THIS COMPILER SECTION CONTAINS ALL THE PAGING FUNCTIONS AND ROUTINES.
//
//
//      THE FUNCTIONS IN THIS SECTION CALLED FROM THE OUTSIDE WORLD ARE:
//
//                      BCPL_TO_MCODE
//                      AETVA1
//                      AETVA2
//                      AETAD1
//                      AETAD2
//
//
PARAMETER. "*"AET*"G187.187"
GET. "PGHDR"
//
//------------------------------------------------------------------------------
//
//                        CONSTANTS
//                        ---------
//
MANIFEST    $(
    PAGESIZE      =  64        //  PAGE SIZE IN WORDS.
    SYMB_PAGES    =  20        //  # OF SYMBOL TABLE PAGES IN MEMORY.
    TREE_PAGES    =   4        //  # OF SYNTAX TREE PAGES IN MEMORY.
    PM_SYMB       =   2        //  # OF SYMB PAGES IN THE 'FILE MIX'.
    PM_TREE       =   5        //  # OF TREE PAGES IN THE 'FILE MIX'.
    TREE          =   0        //  USED AS AN INDICATOR.
    SYMB          =   1        //  USED AS AN INDICATOR.

    TOTAL_PAGES   =   SYMB_PAGES + TREE_PAGES
    PM_TOTAL      =   PM_SYMB + PM_TREE
          $)
//
//------------------------------------------------------------------------------
//
//               SEMI-PERMANENT  VARIABLES
//               -------------------------
//
//
STATIC      $(
    TREEP         =   1        // TREE POINTER.
    SYMBP         =   1        // SYMBOL TABLE POINTER.
    TREE_PG_MAX   =  -1        // MAX TREE PAGE # USED .
    SYMB_PG_MAX   =  -1        // MAX SYMBOL TABLE PAGE # USED .
    CLOCK         =   0        // A SORT OF CLOCK : 1 TICK PER ACCESS !
    BASE_ADD      =   0        // BASE ADDRESS OF PAGES IN MEMORY.
    PAGING_STREAM =   0        // I/O STREAM FOR DATA.

     //  THE FOLLOWING ARE ARRAY NAMES (ALLOCATED SPACE LATER).
    PAGE_N        =   0        // VIRTUAL PAGE #'S IN MEMORY.
    WRITTEN       =   0        // TRUE IF ABOVE PAGES WRITTEN ON.
    TIMER         =   0        // TIME OF THE LAST ACCESS TO ABOVE PAGES.
          $)
//
//
//
```

```
//
//
//-------------------------------------------------------------------------------
//
//              BCPL_TO_MCODE  ALLOCATES SPACE FOR THE ARRAYS AND CALLS
//                 BCPL_TO_MCODE2 WHICH DOES THE ACTUAL CONVERSION
//                 OF SOURCE PROGRAM TO MCODE.(MCODE = OCODE).
//
//
LET BCPL_TO_MCODE() = VALOF
    $(1 LET V = VEC TOTAL_PAGES ; PAGE_N := V
                                FOR I = 1 TO TOTAL_PAGES DO PAGE_N!I := -1
     $( LET V = VEC TOTAL_PAGES ; WRITTEN := V
     $( LET V = VEC TOTAL_PAGES ; TIMER   := V
     $( LET V = VEC PAGESIZE*TOTAL_PAGES ; BASE_ADD := V-PAGESIZE
     PAGING_STREAM := CREATEOUTPUT( PAGING_FILE )
     IF NOT BCPL_TO_MCODE2() THEN
        RESULTIS FALSE
     SELECTOUTPUT( CONSOLE_OUT_STREAM )
     WRITEF("TREE SIZE              = %I5*N*
            *SYMBOL TABLE SIZE      = %I5*N*N",TREEP,SYMBP )
     RESULTIS TRUE
   $)1
//
//
//-------------------------------------------------------------------------------
//
//             AETVA1 AND AETVA2 ACCEPT A VIRTUAL ADDRESS,CALL CELL_AD TO FIND
//                 THE CORRESPONDING STORAGE ADDRESS,AND RETURN THE VALUE
//                 STORED AT THAT ADDRESS.
//
//
AND AETVA1(N) = !CELL_AD( N,FALSE )

AND AETVA2(N,M) = VALOF
    $( TEST N<0 THEN
          N := N - M
       ELSE
          N := N + M
       RESULTIS AETVA1( N )      $)
//
//
//-------------------------------------------------------------------------------
//
//           AETAD1 AND AETAD2 ACCEPT A VIRTUAL ADDRESS AND RETURN THE
//              CORRESPONDING STORAGE ADDRESS.
//
//
AND AETAD1(N) = CELL_AD( N,TRUE )

AND AETAD2(N,M) = VALOF
    $( TEST N<0 THEN
          N := N - M
       ELSE
          N := N + M
       RESULTIS AETAD1(N)   $)
//
//
//
```

```
//-------------------------------------------------------------------------
//
//       NEWVEC AND STVEC ALLOCATE STORAGE IN THE SYNTAX TREE AND SYMBOL TABLE
//          VIRTUAL ADDRESS SPACES.THEY CHECK FOR OVERLAPPING NODES.THE VALUE
//          RETURNED IS THE VIRTUAL ADDRESS OF THE FIRST WORD ALLOCATED.
//
AND NEWVEC(N) = VALOF
    $( IF (TREEP REM PAGESIZE) + N > PAGESIZE THEN
          TREEP := (TREEP / PAGESIZE + 1) * PAGESIZE
       TREEP := TREEP + N
       RESULTIS ( TREEP-N )
    $)

AND STVEC(N) = VALOF
    $( IF (SYMBP REM PAGESIZE) + N > PAGESIZE THEN
          SYMBP := (SYMBP / PAGESIZE + 1) * PAGESIZE
       SYMBP := SYMBP + N
       RESULTIS ( N-SYMBP )
    $)
//
//-------------------------------------------------------------------------
//
//        CELL_AD ACCEPTS EITHER VIRTUAL TREE ADRESSES (+VE) OR VIRTUAL SYMBOL
//           TABLE ADDRESS (-VE).IT CALLS LOAD TO ENSURE THE RELEVANT PAGE IS
//           IN MEMORY,UPDATES THE PAGES TIMER,AND RETURNS THE STORAGE ADDRESS
//           OF THE VIRTUAL ADDRESS PASSED TO IT.
//
AND CELL_AD( N,STORE ) = VALOF
    $( LET BUFFER,TYPE,PAGE,OFFSET = 0,TREE,0,0
       IF N < 0 THEN
          N,TYPE := -N,SYMB
       PAGE := N / PAGESIZE
       OFFSET := N REM PAGESIZE
       BUFFER := LOAD( PAGE,TYPE )
       CLOCK := CLOCK + 1
       IF CLOCK = 20000 THEN
          FOR I = 1 TO TOTAL_PAGES DO TIMER!I := TIMER!I - 20000
       TIMER!BUFFER := CLOCK
       IF STORE THEN
          WRITTEN!BUFFER := TRUE
       RESULTIS (BASE_ADD + BUFFER*PAGESIZE + OFFSET )
     $)
//
//-------------------------------------------------------------------------
//
//      LOAD CHECKS THE PAGE NUMBER.IF NOT A NEW PAGE IT TRIES TO FIND IT IN
//         MEMORY.IF FOUND THE BUFFER NUMBER WHICH IT IS IN,IS RETURNED.
//         OTHERWISE PICKPAGE IS CALLED TO FIND AN EMPTY BUFFER OR THE LEAST
//         RECENTLY USED PAGE AND SWAP IS CALLED TO GET THE PAGE INTO MEMORY.
//
AND LOAD( PAGE,TYPE ) = VALOF
    $( LET BUFFER = 0
       IF PAGE <= (TYPE -> SYMB_PG_MAX,TREE_PG_MAX) THEN
          $( BUFFER := FIND_PAGE( PAGE,TYPE )
             IF BUFFER ~= -1 THEN RESULTIS BUFFER $)
       BUFFER := PICK_PAGE( TYPE )
       SWAP( PAGE,BUFFER,TYPE )
       RESULTIS BUFFER
    $)
```

```
//-------------------------------------------------------------------------------
//
//    FIND_PAGE SEARCHES FOR A PAGE IN MEMORY.IF FOUND IT RETURNS THE BUFFER
//         NUMBER.IF NOT FOUND IT RETURNS -1.
//
AND FIND_PAGE( PAGE,TYPE ) = VALOF
    $( FOR I =      (TYPE=SYMB -> 1             ,SYMB_PAGES + 1)
             TO (TYPE=SYMB -> SYMB_PAGES   ,SYMB_PAGES + TREE_PAGES) DO
          IF PAGE_N!I = PAGE THEN
             RESULTIS I
       RESULTIS -1
    $)
//
//-------------------------------------------------------------------------------
//
//      PICK_PAGE LOOKS FOR AN EMPTY BUFFER.IF FOUND THEN THE BUFFER NUMBER IS
//        RETURNED.OTHERWISE THE LEAST RECENTLY USED PAGE IN MEMORY IS
//        DETERMINED AND NUMBER OF THE BUFFER CONTAINING IT IS RETURNED.
//
//
AND PICK_PAGE( TYPE ) = VALOF
    $( LET IT,MIN = 0,32676
       FOR I =      (TYPE=SYMB -> 1             ,SYMB_PAGES + 1)
            TO   (TYPE=SYMB -> SYMB_PAGES   ,TOTAL_PAGES) DO
          $( IF PAGE_N!I = -1 THEN
                RESULTIS I
             IF TIMER!I < MIN THEN
                IT,MIN := I,TIMER!I
          $)
       RESULTIS IT
    $)
//
//-------------------------------------------------------------------------------
//
//        SWAP_PAGE TESTS THE PAGE TO BE SWAPPED OUT OF MEMORY.IF WRITTEN ON
//             THEN THE PAGE IS SAVED (WRITTEN TO DISC).IT THEN CALLS RESTORE
//             TO GET THE PAGE INTO MEMORY.
//
//
AND SWAP( PAGE,BUFFER,TYPE ) BE
   $( IF WRITTEN!BUFFER LOGAND (PAGE_N!BUFFER NEQV -1) THEN
         SAVE( BUFFER,TYPE )
      TEST TYPE = SYMB THEN
         SYMB_PG_MAX := RESTORE( PAGE,BUFFER,SYMB )
      ELSE
         TREE_PG_MAX := RESTORE( PAGE,BUFFER,TREE )
      PAGE_N!BUFFER := PAGE
      WRITTEN!BUFFER := FALSE
   $)
//
//-------------------------------------------------------------------------------
//
//     SAVE_PAGE SAVES A PAGE BY WRITING IT TO DISC.
//
AND SAVE( BUFFER,TYPE ) BE
    $( LET PAGE = (TYPE = SYMB -> REAL_SYMB_PG( PAGE_N!BUFFER ),
                                  REAL_TREE_PG( PAGE_N!BUFFER ))
       WRITEREC( PAGING_STREAM,BASE_ADD+BUFFER*PAGESIZE,PAGE )
    $)
```

```
//
//-------------------------------------------------------------------------------
//
//       RESTORE CHECKS FOR A NEW PAGE.IF A NEW PAGE THEN THE VALUE OF THE
//           MAXIMUM PAGE NUMBER USED SO FAR IS UPDATED AND RETURNED.IF NOT
//           THE OLD PAGE IS READ OFF DISC.
//
//
AND RESTORE( PAGE,BUFFER,TYPE ) = VALOF
    $( LET REAL_PAGE = 0
       IF PAGE > (TYPE = SYMB -> SYMB_PG_MAX,TREE_PG_MAX) THEN
          RESULTIS PAGE
       REAL_PAGE := (TYPE = SYMB -> REAL_SYMB_PG( PAGE ),
                                    REAL_TREE_PG( PAGE ))
       READREC( PAGING_STREAM,BASE_ADD+BUFFER*PAGESIZE,REAL_PAGE )
       RESULTIS (TYPE = SYMB -> SYMB_PG_MAX,TREE_PG_MAX)
    $)
//
//-------------------------------------------------------------------------------
//
//         REAL_SYMB_PAGE AND REAL_TREE_PAGE ACCEPT A VIRTUAL PAGE NUMBER AND
//           RETURN THE CORRESPONDING DISC PAGE NUMBER.
//
//
AND REAL_SYMB_PG( PAGE ) =
       (PAGE / PM_SYMB) * PM_TOTAL    +  (PAGE REM PM_SYMB) + PM_TREE

AND REAL_TREE_PG( PAGE ) =
       (PAGE / PM_TREE) * PM_TOTAL    +  (PAGE REM PM_TREE)
//
//-------------------------------------------------------------------------------
//
//    LIST1 TO LIST6 CREATE AND INITIALISE SYNTAX TREE NODES.
//
//
AND LIST1(X) = VALOF
     $( LET VIRTUAL = NEWVEC(1)
        LET REAL    = AETAD1(VIRTUAL)
       REAL!0 := X
       RESULTIS VIRTUAL  $)

AND LIST2(X, Y) = VALOF
     $( LET VIRTUAL = NEWVEC(2)
        LET REAL    = AETAD1(VIRTUAL)
        REAL!0, REAL!1 := X, Y
        RESULTIS VIRTUAL   $)

AND LIST3(X, Y, Z) = VALOF
     $( LET VIRTUAL = NEWVEC(3)
        LET REAL    = AETAD1(VIRTUAL)
        REAL!0, REAL!1, REAL!2 := X, Y, Z
        RESULTIS VIRTUAL     $)

AND LIST4(X, Y, Z, T) = VALOF
     $( LET VIRTUAL = NEWVEC(4)
        LET REAL    = AETAD1(VIRTUAL)
        REAL!0, REAL!1, REAL!2, REAL!3 := X, Y, Z, T
        RESULTIS VIRTUAL   $)
```

```
AND LIST5(X, Y, Z, T, U) = VALOF
     $( LET VIRTUAL = NEWVEC(5)
        LET REAL    = AETAD1(VIRTUAL)
        REAL!0, REAL!1, REAL!2, REAL!3, REAL!4 := X, Y, Z, T, U
        RESULTIS VIRTUAL   $)

AND LIST6(X, Y, Z, T, U, V) = VALOF
     $( LET VIRTUAL = NEWVEC(6)
        LET REAL    = AETAD1(VIRTUAL)
        REAL!0, REAL!1, REAL!2, REAL!3, REAL!4, REAL!5 := X, Y, Z, T, U, V
        RESULTIS VIRTUAL  $)
//
//
////////////////////////////////////////////////////////////////////////////////
```

APPENDIX B: Sample Output

```
UBC - BCPL/RDOS VERSION 2 (76-08-08)
NMAX = 063476, HMA = 073777
SOURCE FILE    = TRA
SYNTAX ONLY    = TRUE
LISTING        = FALSE
SAVE ASSEMBLY = FALSE
SAVE MCODE     = FALSE
SECTION G392
 SETTINGS:
   PAGESIZE             =   256
   TREE PAGES           =     1
   SYMBOL TABLE PAGES =     5

TREE SIZE               =  4895
SYMBOL TABLE SIZE       =  1847

TREE WASTAGE            =    16
SYMBOL TABLE WASTAGE    =    15

ACCESSES:
    TREE                =  9637
    SYMBOL TABLE        =  5660

FAULTS:
  TREE          READS   =   459
                WRITES  =    44
  SYMBOL TABLE  READS   =   516
                WRITES  =   438
TRANSLATION TO ASSEMBLER SUPPRESSED
TIMING RESULTS:
 START TIME              23:44:15
 BCPL_MCODE FINISHED     23:45:24
 MCODE_ASM  FINISHED
 ASM_RB     FINISHED

0 BCPL ERRORS DETECTED
JELP & GFS
```

```
UBC - BCPL/RDOS VERSION 2 (76-08-08)
NMAX = 063476, HMA = 073777
SOURCE FILE    = MST
SYNTAX ONLY    = TRUE
LISTING        = FALSE
SAVE ASSEMBLY  = FALSE
SAVE MCODE     = FALSE
SECTION MST
 SETTINGS:
   PAGESIZE             =    64
   TREE PAGES           =     4
   SYMBOL TABLE PAGES   =    20

TREE SIZE               =  1868
SYMBOL TABLE SIZE       =  1140

TREE WASTAGE            =   125
SYMBOL TABLE WASTAGE    =    34

ACCESSES:
    TREE                =  3130
    SYMBOL TABLE        =  1171

FAULTS:
  TREE          READS   =    55
                WRITES  =    44
  SYMBOL TABLE  READS   =     0
                WRITES  =     0
TRANSLATION TO ASSEMBLER SUPPRESSED
TIMING RESULTS:
 START TIME             23:39:55
 BCPL_MCODE FINISHED    23:40:08
 MCODE_ASM  FINISHED
 ASM_RB     FINISHED

0 BCPL ERRORS DETECTED
JELP & GFS
```

```
UBC - BCPL/RDOS VERSION 2 (76-08-08)
NMAX = 063476, HMA = 073777
SOURCE FILE    = LEX
SYNTAX ONLY    = FALSE
LISTING        = FALSE
SAVE ASSEMBLY = FALSE
SAVE MCODE     = FALSE
 SETTINGS:
   PAGESIZE             =   256
   TREE PAGES           =     4
   SYMBOL TABLE PAGES =    10

TREE SIZE               =  5091
SYMBOL TABLE SIZE       =  1667

TREE WASTAGE            =    22
SYMBOL TABLE WASTAGE    =    15

ACCESSES:
    TREE                =  9910
    SYMBOL TABLE        =  4238

FAULTS:
  TREE          READS   =    36
                WRITES  =    32
  SYMBOL TABLE  READS   =     0
                WRITES  =     0
ASSEMBLER CODE= 3114(006052) LINES
MAC LEX.HD LEX.CD;POP

                    .TITL LEX
TIMING RESULTS:
 START TIME             20:59:56
 BCPL_MCODE FINISHED    21:00:22
 MCODE_ASM  FINISHED    21:00:41
 ASM_RB     FINISHED    21:02:23

0 BCPL ERRORS DETECTED
JELP & GFS
```