University of Canterbury

Department of Computer Science

Honours Project Report

Code optimisation for the NZTAB Pascal compiler.

Ian M. Douthwaite

Supervisor: Dr. B.J. McKenzie

October 1983.

Table of Contents

Introduction

The aim of this project was optimisation of the code produced by the NZTAB Pascal compiler. This compiler is used for almost all the code in the TAB's nationwide betting system. The desire to improve the performance of the code was prompted by two reasons:

* Certain heavily-used parts of the code are presently written in assembler for efficiency. For maintainability, it would be better to have all code written in Pascal.

* The performance of the betting system is significantly affected by certain system control functions.

The TAB wished to pursue the adding of an optimisation stage to the compiler so that performance might be improved in that way.

The task of this project, then, was to consider the problem of adding optimisation to the TAB compiler, assessing possible optimisation techniques, and implementing the best of these.

There are several aspects to this project and the structure of this report reflects that. This does not mean, however, that each aspect represents a chronological phase. The first section deals with the problem and with the constraints placed on possible optimisation methods. Section two briefly presents various kinds of optimisations that are often done and reviews the main methods available for doing them. In section three there is an analysis of the nature of the code currently produced by the TAB compiler and the implications of this for optimisation. Fourthly, various optimisation strategies that were considered are presented with

their problems and advantages. Section five deals with two strategies for which some implementation was done, with the problems that arise in their implementation, and with their potential benefits. Finally, in section six, some questions are posed that are relevant to if and how the project should proceed and some recommendations are given.

Naturally, the ideal result of such a project would have been the completion of an optimiser that performed sufficiently well for the earlier goals to be realised: this has not been achieved. One should remember though that this project can viewed in two ways. In one way it can be seen as being required to add some form of optimisation to a Pascal compiler. At the other level, however, it can be seen as needing to provide optimisation that is sufficient for the goals given at the start of this introduction, and that is obtained in a way that is acceptable to the TAB environment. This latter goal is far more difficult but it is towards this goal that the efforts of this project have been directed.

ONE

The Nature of the Problem

In this section, I wish to consider the relevant background to the problem since this has had a marked effect on the direction the project has taken. The effect of the background culminates in the list of constraints at the end of this section. The establishing of these constraints represents a not insignificant effort and an awareness of them is essential to assessing the suitability of various solutions. Their direct effect on the feasibility of solutions will be made clearer when the various solutions are discussed.

The TAB system is essentially a transaction processing system. The system runs on a distributed network of Perkin-Elmer 3240 processors and controls a large terminal network. The Perkin-Elmer 3240s are 32-bit "megamini" computers with an architecture resembling the IBM 360/370 series. The two most significant features of the Perkin-Elmer equipment as far as this project is concerned is their rather complex instruction set and the fact that they have no virtual memory capability. The former makes the compiler's (or optimiser's) task difficult when code generation is required. The second means that the space requirements of the compiled code and of the compiler itself must be taken into account by any optimisation process.

The software runs under the standard Perkin-Elmer OS/32 operating system. The Pascal software, however, runs directly under the control of a virtual operating system (written by the TAB) which handles Pascal's run-time needs.

The Compiler

The compiler itself is a recursive descent compiler written in the dialect of Pascal it defines. A major significant feature of the compiler is that it is single-pass; that is, it produces object code directly as a result of parsing the source code: there is no intermediate step. Also, it effectively compiles a procedure at a time in that it generates and outputs object code on a procedure-by-procedure basis. A further significant feature of the compiler is that it has been substantially modified and patched during its history and thus has become extremely complex to follow. It originated at the University of Tokyo, was converted for IBM equipment by the Australian Atomic Energy Commission, and finally converted for Perkin-Elmer machines and substantially modified by the TAB.

The most distinctive features of TAB Pascal are the extensions to the language. These range from minor syntactic matters such as allowing anonymous array declarations in formal parameter declarations to more important changes such as the built-in type changing functions and new control flow structures. Further discussion of these extensions is omitted since the real relevance of the constructs is that they make TAB Pascal substantially different to standard Pascal.

The most significant feature of the TAB implementation as far as this project is concerned is that of default "tight packing" of data items. Under this scheme, data items are allocated the minimum number of bytes they require. Thus, a normal integer is stored in a fullword (four bytes) but an integer subrange 0..1000 uses only two bytes. Although minimal storage is allocated most objects must be aligned on half or full word boundaries depending on their size. Thus, the advantage of the tight packing is lost if data items are declared in a haphazard order. In the TAB

software, pains have been taken to maximise the packing of data used by programs.

The Environment

The final major influence on this project has been its setting. The fact that the problem consists of improving, by optimisation, the performance of a specific system has made the task both easier and more difficult. The optimisation should be easier since a specific set of software is the target thus removing the problem of dealing with "all" programs. In the event though this advantage failed to materialise since it proved impossible, for logistical reasons, to perform an analysis of the dynamic behaviour of the live system and thus pinpoint trouble spots.

Apart from this "advantage", the other significant environmental aspect of the problem was that any modifications to the compiler etc. had to be totally transparent. Since the correctness of the applications software depends heavily on certain low-level features of the implementation, such as the tight packing, fulfilling this transparency criterion was always going to be a difficult task. Ideally, with a high-level language it should be possible to alter the implementation strategy quite dramatically without compromising the integrity of the software: this is definitely not the case for the TAB.

Constraints

To summarise, the following list of constraints indicates the demands of the problem and the restrictions on the possible solutions.

* The compiled code should execute faster. No criterion level has been set for the improvement but if compiled Pascal code is to

replace heavily used assembler, as is proposed in this case, then it seems reasonable to assume that the improvement should be substantial.

* The compiled code should not take up any more space. Many optimisations should, as a matter of course, reduce the size of the code. Optimisations that trade space for time, however, are ruled out by this criterion.

* Any modification must be transparent at both the source and object code levels. The first is obvious; the second is a result of the dependence of the software on specific implementation details. Similarly all the interfaces with the virtual operating system etc. must be maintained.

* Because of the critical role of the compiler in the system, it is vital that the correctness of the compiler be maintained. This tends to rule out much in the way of changes to the compiler since there is a high risk that such changes may introduce compiler bugs due to the complexity of the compiler.

* The optimisation done has to be general in nature in spite of the fact that a specific system is to be improved. This is a direct result of the fact that a dynamic analysis of the system could only be carried out in Wellington and would require a lot of time and resources.

* This project was carried out in parallel with the installation and operation of the live TAB system at the TAB's Christchurch Regional Site. Thus, the amount and convenience of access to the equipment was somewhat limited.

TWO

Optimisation

Optimisation is a process where an attempt is made to impart to compiler-generated code some of the efficiency that could be achieved if a program were written in assembler directly. This is a somewhat misleading definition since a good optimising compiler can produce better code than a programmer where large programs are involved because of its ability to keep track of variables etc. Nevertheless, the definition certainly conveys the spirit of optimisation. There are some important observations that should be made before discussing forms and methods of optimisation.

The first observation to make is that the term "optimisation" is a misnomer: there is no such thing as the optimum program. It is frequently possible to further optimise a program and so the question of how to optimise and to what degree is a question of trade-offs between the improvement that may be gained and the cost of obtaining it. The gain is usually in the space or time requirements of the compiled and optimised code. The cost is in terms of the compilation time and space required, and the time needed to implement the optimisation.

Secondly, there is the need to maintain the correctness of the compiler. This may seem an obvious requirement but optimising compilers are notorious for their failure to satisfy it. The complexity of the TAB's compiler makes this requirement a particularly difficult one to satisfy. The critical role of the compiler makes it a particularly vital requirement.

Of all the aspects of designing and building compilers,

optimisation has perhaps received the least attention. To a certain extent this reflects its non-essential nature and, to a greater extent, its lack of importance in the presence of cheap computer power and virtual memory. The upshot of this is that there is not a great deal of literature about optimisation and what there is is of somewhat limited usefulness. Much of the work has been strongly theoretical and is not very helpful at all as far as implementation issues are concerned. In addition, the techniques that are well established were developed for programming languages and styles that have changed substantially. The major value of the literature on optimisation is that its more general instances do document the range of possible optimisations that may be done.

A final point to make about optimisation before discussing optimisations and techniques for achieving them is that the applicability and success of optimisation depends in no small part on the features of the language being compiled, the programming style, and on the representation of the program that the optimising process has to work on. One thing to emerge from this project has been that perhaps optimisation techniques developed for optimising number-crunching programs written by FORTRAN programmers in the early seventies may not be very applicable to the non-numerical, structured software written in procedural languages of recent years.

Some Optimisations

The following is a list of the common optimisations as given in Aho & Ullman (1977). It is important not to confuse optimisations with the techniques by which they may be achieved. A discussion of the major techniques follows the list of optimisations.

* **Constant propagation and folding.**
  This simply consists of doing run-time arithmetic at compile time where possible. Hence,

```
i := 2 + 3
```

  becomes

```
i := 5
```

  Folding may operate only at this simple level where the constants are manifest or it may try to handle constant values of variables. Thus,

```
i := 2
...
i := i + 3
```

  becomes

```
i := 5
```

* **Strength reduction**
  Here, an expensive operation is replaced by a cheaper one. So,

```
i := i * 2
```

  might become

```
i := i + i
```

  depending on the relative space/time costs of addition and multiplication on the target machine.

* **Common sub-expression elimination.**
An attempt is made to avoid calculating the same expression twice unnecessarily. Thus,

```
a[i+1] := a[i+1] + 1
```

becomes

```
t := i + 1
a[t] := a[t] + 1
```

This is an important optimisation as this sort of usage is quite common and also because it can help improve code generated by the compiler over which the programmer has no control, especially in the case of calculating array indices (in this case a[t]).

* **Code motion.**
This optimisation involves moving constant code out of loops in the program. Hence,

```
for i := 1 to 1000 do
   begin
   t := 0 ;
   ... { t unaltered }
   end ;
```

would be replaced by

```
t := 0 ;
for i := 1 to 1000 do
   ...
```

* **Redundant code elimination**

  There are two common things that can be done here. One is getting rid of redundant load or store instructions. This is usually done by the process of register management or common sub-expression elimination. The other saving is by spotting unreachable code which may be eliminated altogether. This might be generated, for example, in an if statement controlled by a constant, e.g.

```
const
   debug = false ;
...
if debug then
   { unreachable code }
```

Finally, it should be noted that, when used together, these optimisations tend to interact and thus increase the amount of optimisation that can be done up to a point. For instance, constant folding may reveal code that can be moved out of a loop.

Optimisation Techniques

There are three major ways in which optimisation may be achieved and these are now discussed. Central to all of them, however, is the concept of a **basic block.** A basic block is a sequence of code with no branches into it, except at the beginning, and none out, except at the end. Hence, within a basic block, one can predict what will happen to the values of objects if one knows their initial values: the same cannot be said across block boundaries.

The concept of a basic block is essential because of the fundamental difference between the sequence of code that the

compiler or optimiser sees and the sequence that is executed when the program runs. The basic block boundaries indicate places where two consecutive instructions may not always be executed consecutively.

Global Flow Analysis

The most developed optimisation technique is that of global flow analysis. This was originated by Allen (1969) and has been refined theoretically by many others. It consists of two main components: **structural analysis and data flow analysis.**

Structural analysis combines the basic blocks of a program into a **flowgraph** by joining them together by edges which represent the ways in which control may flow between blocks. Hence, the paths in the flowgraph represent all the possible execution sequences that can occur. This structure can then be analysed to detect sets of basic blocks that together form loops in the program and which should, therefore, be concentrated on for optimisation. This information is essential for the optimisation of moving constant code out of loops.

The structural information of a flowgraph is also indispensable for the data flow analysis. The aim of data flow analysis is to find out the values of data items at various points. This is done by noting that if the value of an object is known at the start of a block or is explicitly defined within a block then one can calculate its value at the end of the block. The values of items leaving a block can then be propagated through the flowgraph structure to form the initial values for other blocks: algorithms for doing this are discussed in Aho & Ullman (op. cit.). The aim is to specify precise values for as many objects at as many points as possible so that constant folding etc. can be done as effectively as possible.

Quite a few of the optimisations given above can be performed without the need for data flow analysis. However, the information such an analysis provides can greatly enhance the amount of optimisation that can be done since many are based on knowing that the value of an object is a constant at some point. Other optimisations (code motion) do rely on the structural information also.

Better code generation

The second method of producing better code is that of improving the code generation phase of the compiler. The major concerns here are that good code is generated and that efficient use is made of the registers. This means that the compiler should keep track of what is currently in the registers and should try and keep frequently used items in the registers. Good register management is often described as one of the richest sources of optimisation.

Peephole Optimisation

The final optimisation method is peephole optimisation, which is generally performed on object code although a good description of the technique by Tanenbaum (1982) is in the context of intermediate code.

Peephole techniques concentrate on a small window of the code and look for patterns of instructions that may be replaced by more efficient sequences. The elimination of redundant loads and stores is a common optimisation done in this way. For correctness, it is still necessary to account for basic block boundaries as the optimiser must not replace sequences that span a boundary since they do not necessarily represent a true

sequence. Failure to do this was a common problem in early optimising compilers which would always delete a load of a variable that immediately followed a store of the same variable, even if the load could be branched to from somewhere else in the program.

THREE

Analysis of the Code

In order to decide what sort of optimisation is likely to be the most profitable, it is necessary to get an idea of what the code currently generated by the compiler is like. This was done in several ways at various stages in the project.

Static Analysis

The first check on the nature of the compiled code was by simply examining the code. Carefully selected test program tended to highlight weaknesses in the code generation. The classic example of this is the sequence

```
i := 0 ;
j := 0 ;
```

which generated the following code,

```
LIS 10,0        { load 0 into reg. 10 }
ST  10,i            { i is an address }
LIS 10,0
ST  10,j
```

Such obviously poor code was one of the major reasons for the TAB beginning to explore the possibility of optimisation. Other programs yield similar results with respect to reloading and recalculating quantities unnecessarily. The general conclusion to be made is that the compiler is not at all good at remembering what is in the registers. Apart from this, though, the code shows no glaring inefficiencies.

FREQUENCY HISTOGRAM OF PROCEDURE CALLS

```
ASSIGNME  ***    (   10)
BLOCK     **     (    6)
CALL      *************************    (   74)
CALLNONS  ******************    (   58)
CASESTAT  *    (    3)
FACTOR    ************************************************************************    (  203)
IFSTATEM  ***********    (   34)
PROFILE       (    1)
SIMPLEEX  ***************************************************************    (  194)
TERM      ****************************************************************    (  195)
WITHSTAT  *    (    3)
```

Notes:

CALL-CALLNONS = number of standard procedure calls
CALLNONS = number of calls to user procedures
SIMPLEX-EXPRESSI = number of relational operators
FACTOR-TERM = number of multiply-type operators (includes AND)
TERM-SIMPLEEX = number of addition-type operators (includes OR)
BLOCK = number of blocks defined in the module

Fig 3.1

STATIC PROFILE OF WCOLLADJ

```
BALR    (     0)
BTCR    (     0)
BFCR ***  (     5)
NR      (     0)
CLR     (     0)
OR    *  (     1)
XR      (     0)
LR    ********************  (   .39)
CR      (     0)
AR    *  (     2)
SR      (     0)
SRLS    (     0)
SLLS  *  (     1)
MR      (     0)
DR      (     0)
BTBS    (     0)
BTFS  ************************************************  (    99)
BFBS    (     0)
BFFS  ************  (    26)
LIS   ***************  (    29)
LCS     (     0)
AIS     (     0)
SIS     (     0)
STH   *******  (    13)
BAL   ************************************************************************************************************  (   185)
BTC   *******  (    14)
BFC   ********************************  (    63)
NH      (     0)
CLH     (     0)
OH      (     0)
LH    ****  (     7)
CH      (     0)
AH      (     0)
SH      (     0)
ST    ************************************  (    74)
AM      (     0)
N       (     0)
CL      (     0)
O       (     0)
X       (     0)
L     ***************************  (    53)
C       (     0)
A       (     0)
S       (     0)
M     **  (     4)
D       (     0)
AHM     (     0)
ATL     (     0)
ABL     (     0)
RTL     (     0)
RBL     (     0)
LHL     (     0)
TBT     (     0)
```

```
 SBT     (     0)
 RBT     (     0)
?RXRX ***  (     6)
 THI     (     0)
 NHI     (     0)
 CLHI ***  (     5)
 OHI     (     0)
 XHI  ***  (     6)
 LHI  ****  (     7)
 CHI  **********  (    19)
 AHI  *  (     2)
 SHI  *  (     1)
 STM  ***  (     5)
 LM   ***  (     5)
 STB  ************  (    25)
 LB   ******************  (    36)
 SVC  ************************************  (    75)
 LA   ********************************************  (    92)
 RRL  *  (     1)
 RLL     (     0)
 SRL     (     0)
 SLL  ***  (     5)
 SRA     (     0)
 SLA     (     0)
 TI      (     0)
 NI      (     0)
 CLI  **********************  (    46)
 OI      (     0)
 XI      (     0)
 LI   *  (     2)
 CI   *  (     1)
 AI   ***  (     5)
 SI      (     0)
```

Notes:

BTCR, BFCR, BTBS, BTFS,
BFBS, BFFS, BTC, BFC are all branch instructions

BAL, BALR are subroutine calls

L, LIS, LCS,LHL are load instructions
LH, LA

STH, ST, STB are store instructions

Next, static profiles of both source and object code were produced. This was done for a sample of modules that are part of the TAB system itself. A typical example of the results obtained for such profiles is given in Figs 3.1 & 3.2. The analysis of the source code failed to reveal anything remarkable. Expressions are common but only a few actually involve an operator. There are also a reasonable number of calls to user-defined procedures. Very few blocks are defined within the same module which implies that many of the procedure calls are to externally declared procedures and that the bodies of loops and if statements are only simple statements as a rule.

The profile of the generated code, naturally, reflects that of the source code with an abundance of load and store operations and subroutine calls. The notable feature, though, is the number of branch instructions present, the importance of which will become apparent. These tend to occur in such numbers because short branches are frequently generated by the compiler to skip one or two instructions in a 'fragment' of code. A common source of such fragments is the code generated for range checking etc.

Procedure and Block Analysis

At a later stage it became possible to analyse the structure of modules, procedures and the basic blocks that make up the procedures. The results of this analysis are summarised in Figs 3.3 to 3.6. The major point of interest here is the distribution of basic block sizes and numbers with the huge bias to lots of little blocks - the result of all the branch instructions noted above. The central role of the basic block in the optimisation processes described in the previous section

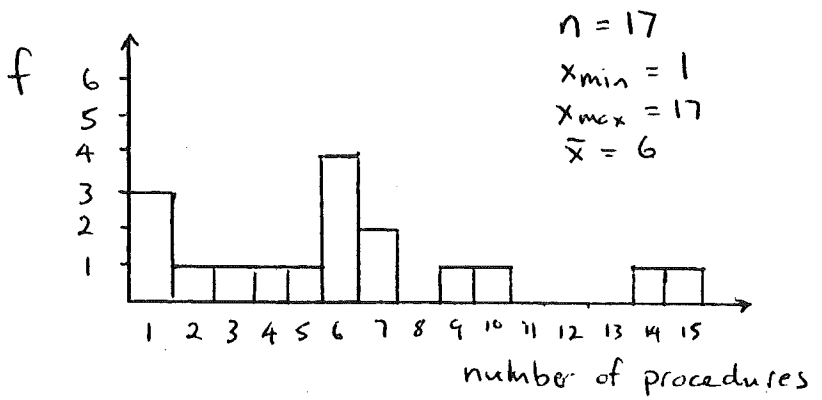


Fig 3.3 Distribution of procedures in modules

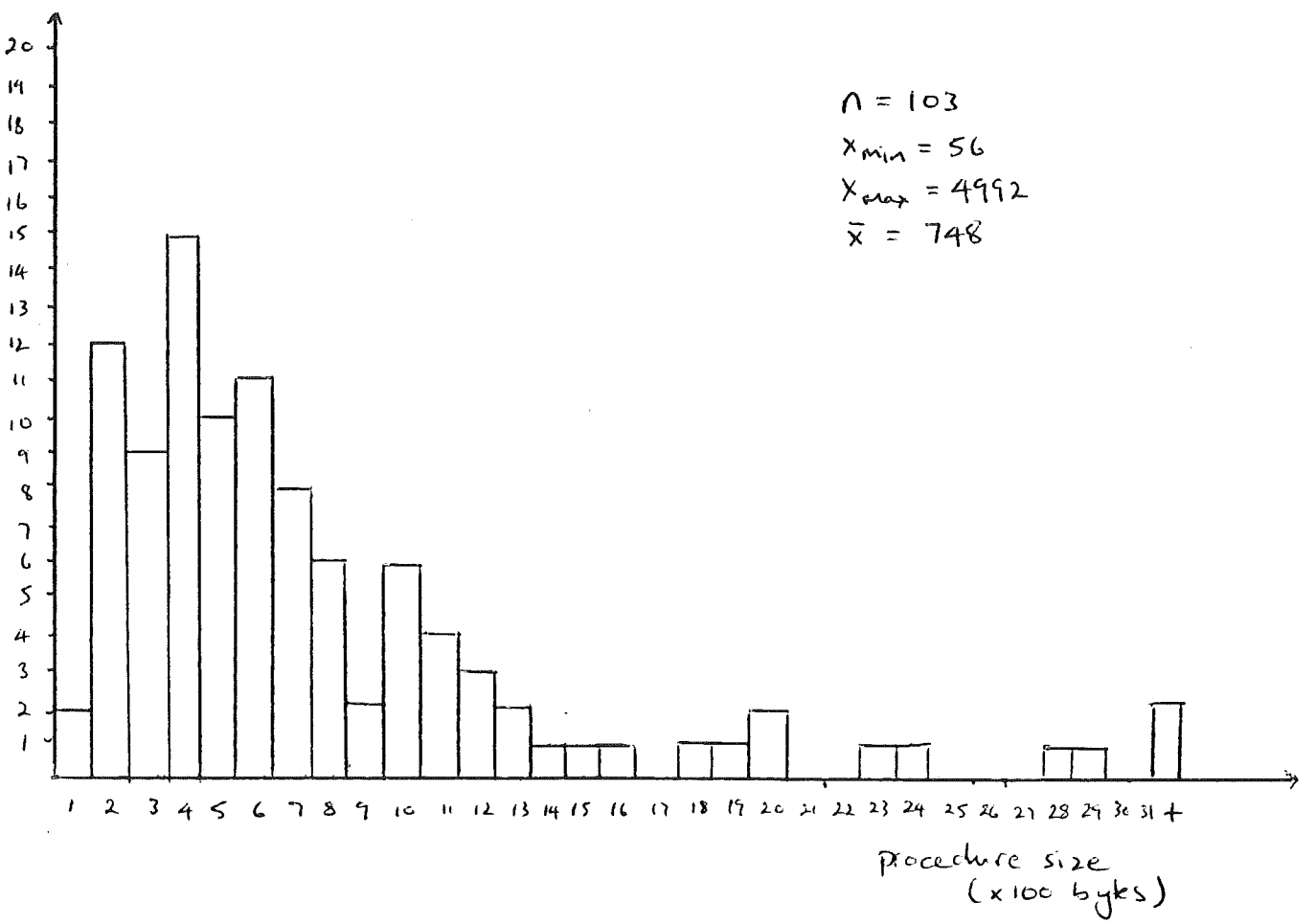


Fig 3.4 Distribution of procedure sizes

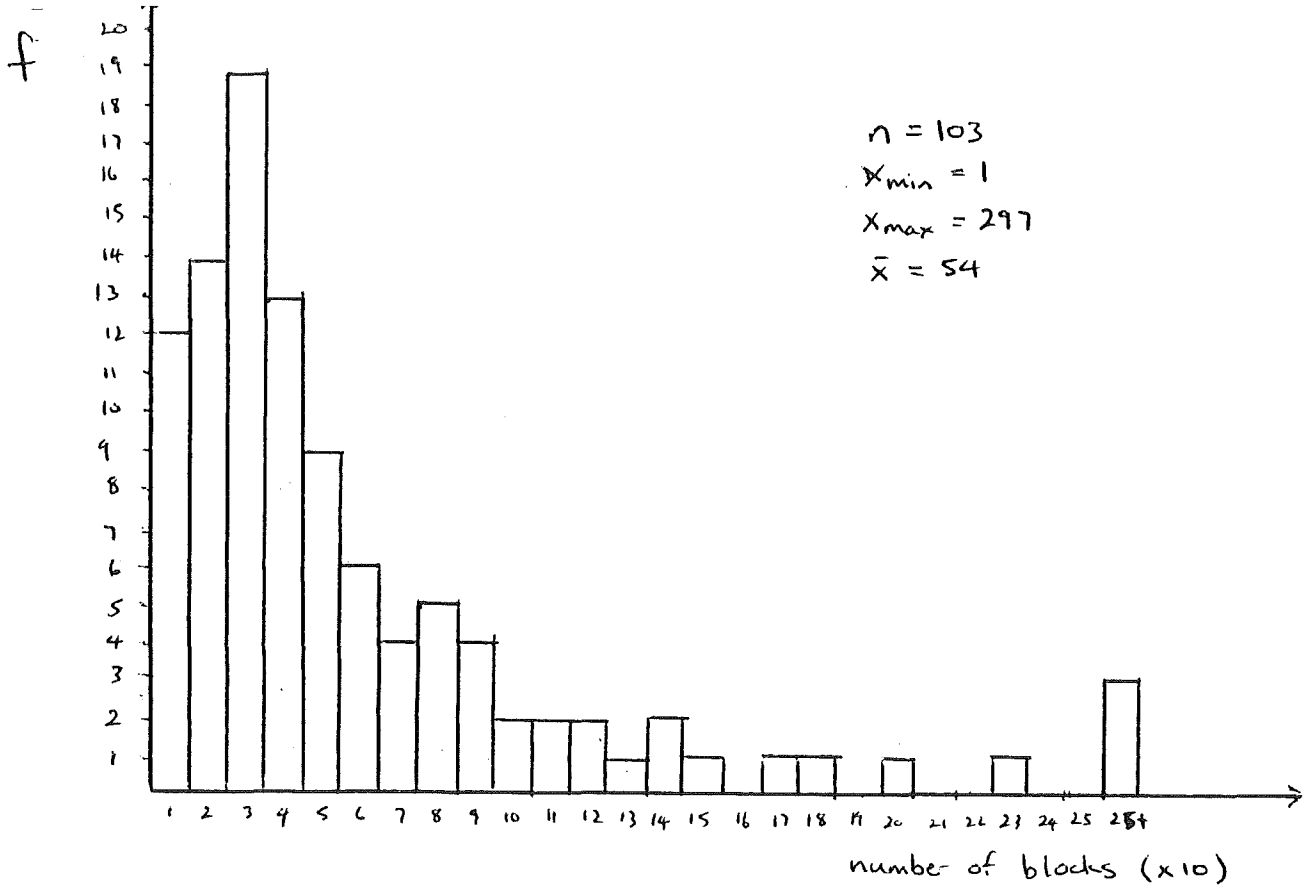


Fig 3.5 Distribution of blocks/procedure

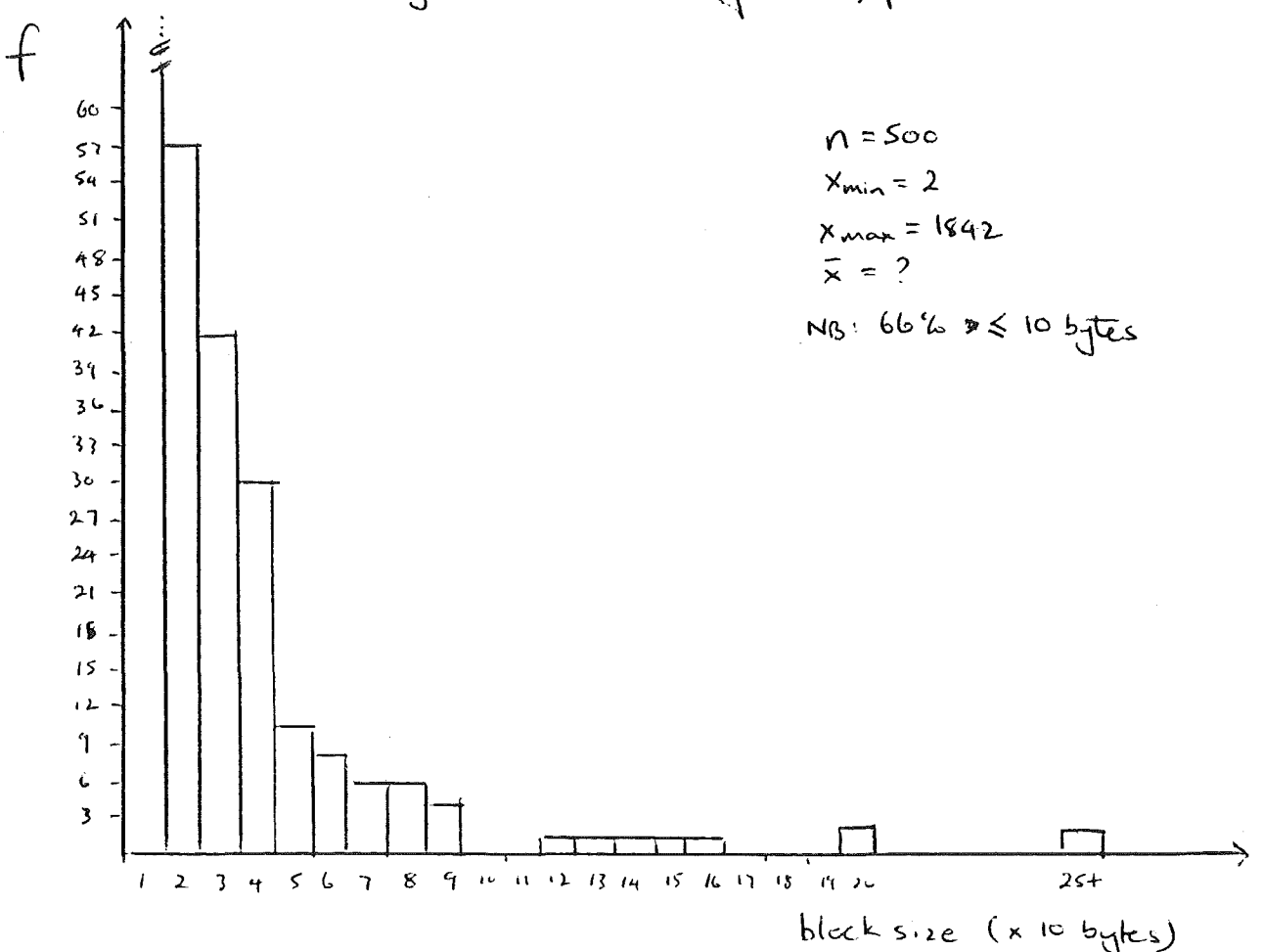


Fig 3.6 Distribution of block sizes

means that this result has a lot of significance for the effectiveness of such optimisation methods.

FOUR

Approaches Considered

The information from the analyses described did not provide much guidance for the evaluation and selection of a suitable optimisation method. In deciding how to approach the problem, therefore, the following criteria were used:

* Since a major objective was the replacement of heavily used assembler code by Pascal, the problem requires that as much optimisation be done as possible.

* There is no particular weak spot in the compiler other than its failure to keep track of the registers.

With these points in mind the following possibilities were explored.

Peephole Optimisation

In some ways, this would be the easiest method to use since it is oriented towards object code, which, in this case, is easily accessible. There are, however, two good reasons for not selecting the peephole approach. The first of these is that the nature of the peephole technique is to replace patterns of instructions with better patterns. The analysis of the TAB code, however, suggests that this is not really a problem for this compiler. The other reason is that the peephole method still has to work within block boundaries to ensure correctness and so the vast number of very small blocks would not offer any scope for a reasonable window size.

Intermediate Code

An important point about the global flow analysis techniques is that, without exception, they assume that they are working with an intermediate level representation of the program to be optimised. Although the TAB compiler does not contain an intermediate code step, the possibility of introducing such a step was keenly pursued. This was done because of the substantial advantages that intermediate code was seen to offer.

* The optimisation problem would then resemble the case for which there were established techniques.

* Use of an established intermediate code would save a lot of work and possibly make available existing optimisers.

* A further step towards greater speed may be taken by having the intermediate code interpreted by microcode. (The TAB were interested in exploring the possibility of utilising the microprogramming capability of the P-E 3240)

To implement such a step required modification of the compiler so that it would first produce intermediate code and then translate the intermediate code to object code. Unfortunately, it quickly became apparent that such a modification would entail almost completely rewriting the compiler since, as a single pass compiler, it was doing the work of a parser, code generator, and assembler all at once and was thus dealing with the program at the highest and lowest levels simultaneously. A major rewrite of the compiler was desirable for neither the TAB nor a project of limited time. Before abandoning the idea of compilation and optimisation via intermediate code, however, the possibility of a new compiler was considered.

New Compiler

Obviously, a new compiler would be just as undesirable as a rewritten old one from the TAB's point of view. However, modifying a compiler that already produced intermediate code so that it supported the TAB extensions to Pascal would be far easier than rewriting the TAB compiler. Also, the advantages of having the intermediate code would probably outweigh the problem of switching to a new compiler. Support for this approach was boosted by the ease with which an intermediate code producing compiler (the VU Pascal compiler) was ported to the TAB system. Although adding most of the TAB's extensions would be reasonably straightforward, the tight packing scheme would have required major changes to the VU compiler and so the idea was abandoned.

At a later stage in the project, the question of a new compiler was again considered with respect to the possibility of using the Perkin-Elmer Pascal compiler. This compiler is reputed to be very sophisticated with regard to optimisation, having inherited much of the optimisation of the earlier FORTRAN compilers. Use of this compiler, however, was not really practicable since the sources of the compiler would be very expensive, if not unobtainable, and modification for the TAB's needs would be a major task.

Ad hoc techniques

Another potential approach was to make ad hoc patches to the compiler to perform optimisations such as constant folding, common sub-expression elimination, register management etc. Initially, this approach was not considered very satisfactory since it would be very unsystematic and, due to the complexity

of the compiler, deciding exactly where patches had to be made would be quite a difficult task. The natural outcome of this difficulty would be that there would be a strong likelihood of introducing esoteric bugs into the compiler. Work later done in this way (described in the next section) showed that this, in fact, tended to be the case.

Flow Analysis of Object Code

The consideration of the possibility of doing global flow analysis on the object code was partly prompted by the elimination of the alternatives. It had the substantial disadvantage that, as far as we were aware, it had not been done before. Further difficulties would become apparent as an attempt at implementation progressed (see next section). An attempt was, nevertheless, made because this approach was seen as having the following advantages that the other approaches lacked.

* Access to the object code required virtually no change to the compiler thus guaranteeing that no errors would be introduced in the compiler itself.

* It ought to be possible to achieve almost all of the optimisation that could be achieved by the same methods working on an intermediate code form, even if more effort was required to do it.

* The 'intermediate code' upon which these techniques have been used in the past is fairly crude and not too far removed from object code itself.

FIVE

Approaches Adopted

Flow Analysis of Object Code

The implementation of this method consists of getting the object code for a procedure just before it is output to the object file, optimising it, and returning it so that the process is effectively transparent to the compiler. The first task of the optimiser is to break the code into basic blocks and to build the flowgraph. The method of implementing the flowgraph is described in Appendix A. It is a fairly simple structure constructed using pointers and can handle procedures of up to 6000 bytes or 300 basic blocks. When the flowgraph has been built the analysis, described earlier, of procedures for block size and numbers can be done.

Once the flowgraph has been constructed, there are two aspects to performing optimisation on it. The major aspect is that of optimising each block of the graph independently. This process can be done directly but the amount of optimisation which it accomplishes will be greatly increased if it is preceded by a data flow analysis as described earlier.

Intrablock Optimisation

The implementation of this was attempted first since it should be possible to do simple constant folding etc. without the need for further analysis. The method chosen to optimise the blocks was that described in Aho & Ullman (op. cit.) which is to represent the block as a **directed acyclic graph (DAG)**. This structure represents a version of the block from which the

shortest code can be generated (or re-generated in this case). Its most important feature is that as a by-product of its construction it automatically detects common sub-expressions.

A DAG is built as follows. Each instruction in the block is considered as being of the form:

A := B op C

Note: A := B and A := op C are also possible but the discussion that follows should make clear how they are dealt with.

For each of the operands, B & C, a check is made to see if the identifier is already attached to a node in the DAG. If not, a new node is created which is labelled as the initial value of that identifier (i.e. the value of that object on entry to the block). Once a DAG node is known for each operand then a check is made to see if these nodes share a common parent labelled 'op'. If not then such a parent is created. The identifier A is attached to the 'op' node (whether it was found or created) and removed from the node it was previously attached to, if any. Fig 5.1 shows an example of the DAG building process. The process is also described in Aho & Ullman.

The result is a structure that may be viewed logically as a tree representing an expression (Fig 5.2) or physically as a graph (Fig 5.3). The leaves represent either constants or initial values and the interior nodes represent operations and their results. The sub-tree to whose root an identifier is currently attached represents the value of the object at this point in time. The logical tree represents the expression in full, whereas the physical representation shares common

$$k := (i+1) * j + (i+1)$$

(a) $T_1 := i + 1$
(b) $T_2 := T_1 * j$
(c) $T_3 := i + 1$
(d) $T_4 := T_2 + T_3$
(e) $k := T_4$

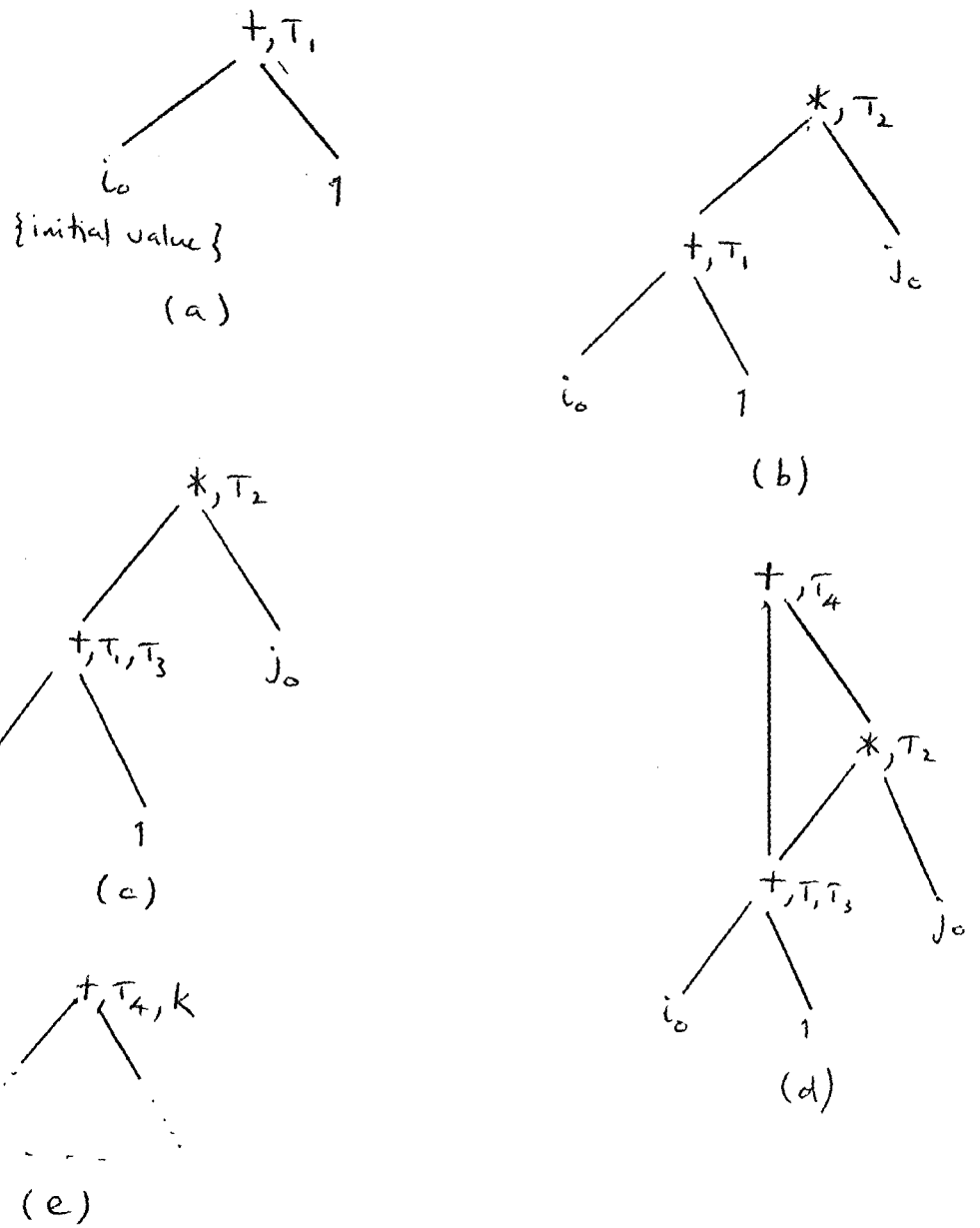


Fig 5.1 DAG building

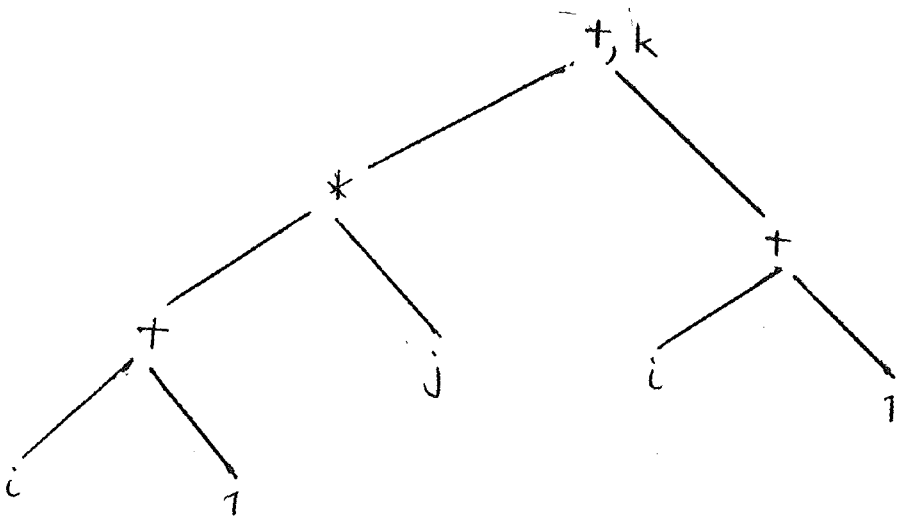


Fig 5.2 Logical DAG

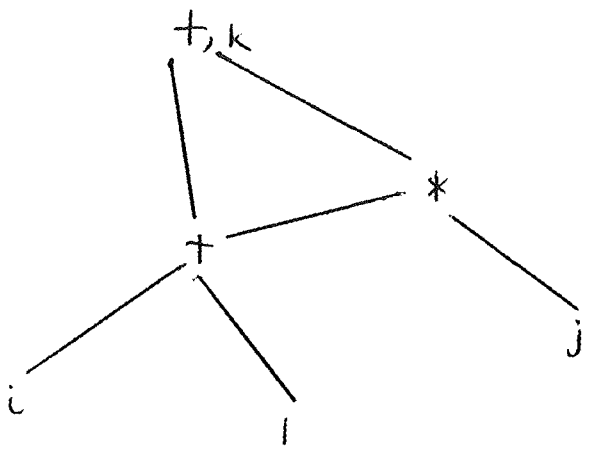


Fig 5.3 Physical DAG

sub-expressions within the tree and thus automatically takes care of using common sub-expressions when code is generated from the DAG.

The implementation of the DAG building process is no trivial task. It involves the construction of a complicated pointer structure whose nodes must be accessed, at various times, directly via addresses of data items, via a key based on two nodes and an operation, and via parent nodes in the DAG. The nature of the structure is such that this access must be done by use of a direct method such as hashing. This technique is further complicated by the fact that the mapping of objects to nodes is many to one and the fact that the mapping for a particular object changes during the DAG building process.

Once the DAG has been built, then it may be optimised for things such as constant folding, re-ordering of expressions etc. by taking a logical tree-like view of the DAG. Ideas on the sort of optimisations that may be performed on such a structure are discussed by Johnson (1979). Finally, code is generated from the DAG so as to yield the shortest instruction sequence, using the physical structure of the DAG to detect common sub-expressions.

Implementation of the DAG construction and optimisation process has reached the stage where DAGs can be built for each block in the flowgraph. Further details of this are discussed in Appendix A.

The important difference between the DAG building process just described and that actually implemented is that the former works on intermediate code whereas the latter uses object code. Thus for a statement such as

```
a := b + c
```

the actual code that would have to be dealt with will be of the form (using similar notation):

```
R10 := b
R10 := R10 + c
a   := R10
```

(where R10 is a register and a, b, c are addresses)

Furthermore, the "+" can be any one of a number of add instructions depending on the size of the operands whereas this would not be a relevant detail in intermediate code. Hence, the amount of work to do the same amount of optimisation becomes correspondingly greater because of the extra detail that exists at the object code level but which is irrelevant at the intermediate level. The actual implementation of the DAG building process has also been slowed by the fact that although a relatively small number of instructions are actually important in building the DAG, all the instructions have to be catered for.

There are several problems that arise with respect to DAGs, most of which only materialise when one attempts to implement the method. These problems are due to either a deficiency in the method or to the use of the object code or a combination of the two.

The first point is that a basic block does not always result in a tree structure since it may not represent the calculation of a single expression. This is easily handled by linking the separate roots together.

Secondly, there are little problems that arise from the

architecture of the machine. For example, there is the confusion that arises from the need for and use of register pairs for multiplication and division thus producing instructions where the result does not appear to reside in either of the instruction's operands. There is also the problem of ensuring that the condition codes are still set correctly at the end of a block so that correct branches are taken.

A far more serious problem is posed by the use of array references and their representation within a DAG. The first is discussed by Aho & Ullman and arises from the fact that the sequence

```
x := a[i]
a[j] := y
z := a[i]
```

would be optimised to produce code equivalent to

```
x := a[i]
z := x
a[j] := y
```

which is incorrect if i = j and y <> a[i]. In this case, a[i] has been falsely recognised as a common sub-expression. Their solution is to forget all such sub-expressions when a store into an array is done. Inevitably, this sacrifices some optimisation for the sake of correctness.

The second problem caused by array references is far more difficult and represents a deficiency in the DAG method. Array references on the right hand side of an assignment operator may be conveniently represented in the DAG as an indexing operation with the base address of the array and the expression that

represents the subscript as its operands (see Fig 5.4). However, since assignment is represented by attaching extra identifiers to the sub-tree that is built for the right hand side, application of this technique results in a messy structure (Fig 5.5). This is because the method described above has no facility for dealing with expressions on the left-hand side which the use of array elements on the left-hand side requires. The obvious solution to this is to represent assignment in the DAG as a dyadic operator like any other. Incorporating this modification will require a careful reconsideration of other aspects of the DAG method such as its way of detecting common sub-expressions and the generation of code from the DAG.

A further difficulty is caused by the problem of aliasing: that is, the same object may be referred to in more than one way. This means that using different guises, an object could acquire two different current values in the DAG thus causing errors to occur. A simple example of this, using the TAB Pascal extension of being able to have pointers to declared variables, is presented in Fig 5.6. The suggested method of dealing with this problem is to 'forget' all the values in the DAG when an indirect store is done. This is a fairly draconian measure and, though it should be effective in preventing errors, could decimate the amount of optimisation that was able to be done.

A final problem for this approach is again that of the structure of the code it will get to work on. More optimisation will be gained in a DAG if it contains more instructions from the original code. The results of the analysis in section three indicate that so many of the blocks are so small that the chances of extracting any optimisation from them is slight. Furthermore, as the building of a DAG has proved to be a costly exercise in terms of time and space, it is hard to justify the

q := a[i+1] + 1

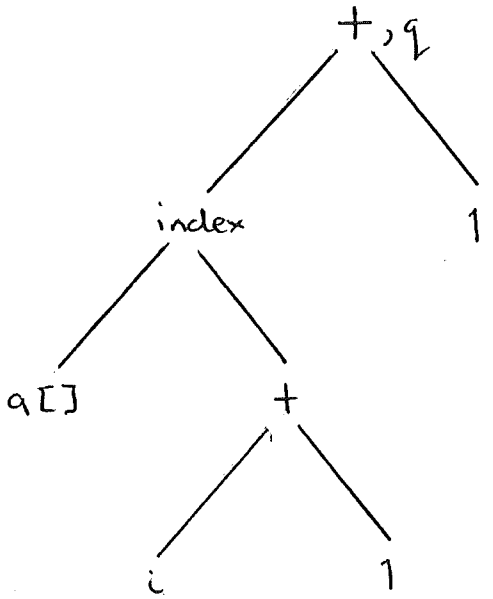


Fig 5.4 Array reference on RHS

a[i+1] := a[i+1] + 1

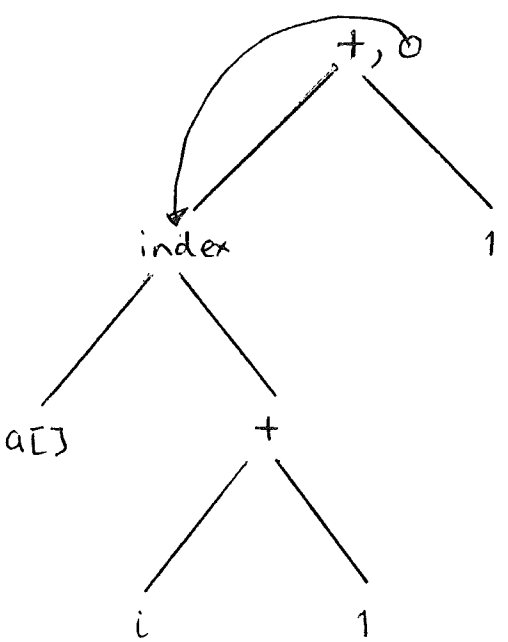
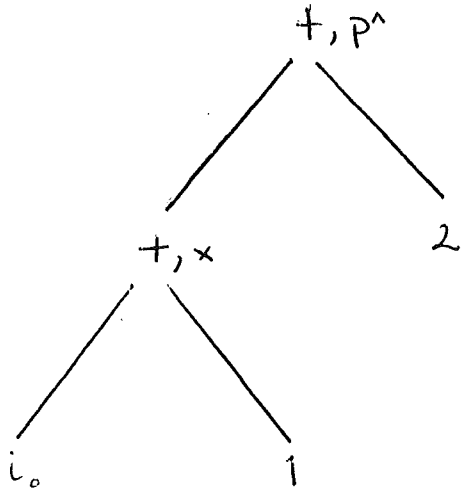


Fig 5.5 Array reference on LHS

```
p := ^x
...
x := i + 1
p^ := x + 2
```



N.B. x and p^ are the same object

Fig 5.6 Aliasing

effort for small blocks, thus eliminating nearly half the code from consideration for optimisation.

Data Flow Analysis

This step is required to increase the power of the intrablock optimisation by providing more information about what values are constant and where. Algorithms for doing this are discussed by Aho & Ullman and will not be reviewed here. It is sufficient to note that the algorithms, although relatively simple, require large amounts of processing time and storage capacity. The exact amount will depend on the number of blocks in the flowgraph and so, once again, the fragmented nature of the code will serve to increase the amount of work that has to be done. It should be noted, though, that some of the problems of resources that these algorithms bring about can be avoided by a more practical approach as suggested by Lowry and Medlock (1969) in their description of how these techniques were applied in a practical situation to the IBM FORTRAN H compiler.

Ad hoc Techniques

Some modifications were made to the compiler in an attempt to perform some optimisation during the compilation process. The results of this exercise were the same as was predicted when the approach was earlier considered and which are described in section four. Further details of the patches, listings of modified parts of the compiler and a sample unoptimised and optimised program are included in Appendix B.

The first patches made were to perform simple constant folding on expressions where both operands were declared or literal, boolean or integer constants. This patch was successful in that it was easy to do and required the modification of well-defined

parts of the compiler thus meaning that the continued correctness of the compiler was virtually guaranteed. As one would predict, however, the optimisations did not have a great impact as such constant expressions are not used very often. Recompiling the compiler with these optimisations in operation produced a space saving of 200 bytes out of 160,000 bytes and no appreciable increase in speed.

Another apparently simple optimisation that it seemed could be done was the elimination of code that was unreachable because of the value of an expression in an if statement being a constant. Although it is fairly obvious what patches would have to be made to do this, a problem arises in that there is no single clean interface between the parsing and code generation processes and so it is not possible to simply turn off the code generation for the unreachable code.

Since proper management of the registers is a particular weakness in the current compiler, it was felt that it would be potentially very fruitful to attempt some modifications so that the compiler 'remembered' the contents of the registers and thus did not do so many redundant loads and stores. For simplicity, this was attempted initially for fullword constants and simple variables only. This required the following steps:

i) before loading a constant or variable, check to see if it is already in a register.

ii) when loading a constant or variable, note that it now occupies the register

iii) if a new register is needed and none are free then reuse a register which contains a constant or variable

iv) if storing into a variable then destroy any register with an old value of that variable in it.

v) destroy all registers when a block boundary is crossed.

Although this technique works for simple programs, it is unreliable for many real programs since it tends to cause optimisations that which change the action of the program. This happens because, in some places, the compiler generates code that indicates a block boundary but uses knowledge about the context to assume the unaltered state of the registers across such boundaries. For safety, the optimisation technique above kills the registers across all such boundaries and incorporating the exceptions to this will make the method increasingly more complex as more exceptions to this rule are found.

It is possible that the problem described above could be overcome or that their are few enough exceptions to make special handling of them feasible. However, there is a further problem when patches for register management are extended to handle the case of expressions in registers (i.e. common sub-expression elimination). Not only does the task of checking to see if a register contains an expression become more difficult but also the problem arises that by the time a common sub-expression is detected the code for its evaluation has already been generated!

As suggested in the previous section, ad hoc patches face certain problems due to the complex nature of the compiler that affect the ease of their design and their effect on the compiler's correctness.

SIX

Conclusions

At the end of a report such as this, the question inevitably arises of what has been achieved, followed closely by that of where do we go from here? For a project for which the motivational factors are to be found in a very concrete problem these questions take on special relevance.

In the introduction to this report it was suggested that the aim of this project could be viewed on two levels. On the academic level, its goal was to implement optimisation for a Pascal compiler. On the practical level, it had to do this and fit into a tightly defined environment. This latter criterion makes the task considerably harder and yet it is the goal at this level which has been adopted as the guiding one for this project. This project has not been totally successful in achieving that goal but in as far as it has come, it has achieved several sub-goals that are invaluable in reassessing the original goal and evaluating possible methods by which that goal might be achieved. Specifically, the following things have been achieved:

* A fuller description of the problem and the other constraints on the methods for its solution.

* An appreciation of the aspects of the TAB compiler that make an optimisation exercise difficult (and which would also have serious implications for any other substantial modification or conversion).

* The identification of the major weakness of the TAB Pascal compiler, namely its failure to manage the registers during code generation.

* An assessment of most of the possible optimisation techniques, their problems and their potential, in light of the constraints.

* Beginnings of implementation work on two methods which, more than any others, satisfy the constraints and direct their efforts directly at the compiler's weaknesses.

To the question of where to go from this point, the answer must depend on a reappraisal of the original aims of the project and on the answers to certain vital questions:

* Why optimise? Are the reasons of a year ago still valid? Are there new reasons?

* How much improvement is necessary to achieve the goals that have motivated this project? Is this a realistic goal for optimisation to achieve?

* What is the nature of the code most in need of optimisation? Especially, what sort of Pascal code would be used to replace the current assembler components of the system?

* Is the trauma of conversion to another compiler so great as to rule out the possibility completely?

The answers to these questions should serve to indicate the direction in which this project should be continued, if at all.

realistically or optimistically expect from optimisation techniques (say, more than 15%) then other performance enhancement measures need to be considered. If, on the other hand, optimisation can offer the required improvement then careful consideration needs to be given to how it may best be achieved bearing in mind the discussion of the possible options in this report, namely ad hoc patches, flow analysis of object code, or even the more daring, but possibly more farsighted, switch to a new compiler.

It is difficult to suggest what is the best direction for this project to take without answers to the above questions. However, it should be said that the author's impression is that of the optimisation techniques considered, those that seemed the soundest to pursue were eliminated because of the TAB's constraints, namely the approaches which involved intermediate code. Although the full implementation of the flow analysis of object code is possible, it will be an extremely time-consuming process and one of dubious benefit unless the problems of the fragmentation of the object code can be overcome. If the need for optimisation is only desirable, and not critical, then the approach of trying to patch the compiler may be safest way to proceed but one would not like to guarantee the safety of such a course of action as far as compiler integrity is concerned.

Whatever the decisions taken may be, this report should hopefully form a foundation of knowledge about this problem, its possible solutions and how the problem and solutions interact, upon which those decisions may be based.

References

Aho, A.V. & Ullman, J.D. Principles of compiler design. Massachusetts: Addison-Wesley, 1977.

Allen, F.E. Program optimization. Annual Review in Automatic Programming. 1969, 5, 239-307.

Dayan, M. The planning and development of an integrated on-course and off-course betting system for New Zealand. New Zealand TAB, 1982.

Hynd, J.R. PASCAL.OVR: Pascal processor overview. On-line document. NZTAB, 1980.

Hynd, J.R. PASSTDAD.TXT: Pascal standard accompanying document. On-line document. NZTAB.

Hynd, J.R. PASCMPLR.TXT: Pascal compiler documentation. On-line document. NZTAB, 1979.

Hynd, J.R. PASCMPMD.TXT: Pascal compiler modifications. On-line document. NZTAB, 1979.

Johnson, S.C. A tour through the portable C compiler. Unix Programmers Manual, Volume 2B. New Jersey: Bell Telephone Laboratories Inc., 1979.

Lowry, E.S. & Medlock, C.W. Object code optimization. Communications of the ACM. 1969, 12, 13-22.

Tanenbaum, A.S., van Staveren, H., & Stevenson, J.W. Using peephole optimization on intermediate code. ACM Transactions on Programming Languages and Systems. 1982, 4, 21-36.

Appendix A

Implementation of flowgraph and DAGs

This appendix presents a brief description of the implementation of flowgraphs and DAGs. The following notes should serve as an accompaniment to the code for the implementation which is also included. The fact that the code consists of some 1300 lines of Pascal should serve as an indication of the complexity of the task it represents. Indeed, the code for building DAGs so far only handles fairly simple DAGs. None of the problems described in section five have had solutions implemented and doing so will certainly greatly increase the size of the code required.

The compiler/optimiser interface

When the compiler has finished compiling a procedure, all the code for the procedure resides in a set of buffers and is output to the object file. The optimiser is called as an external procedure just before this is done and takes its input from these buffers. Ultimately, the optimiser will restore the optimised code to these buffers before returning to the compiler, thus being totally transparent to the rest of the compiler.

Data structures for flowgraphs

The main data structure for the flowgraph is simply constructed out of a set of interlinked block nodes. The links between blocks represent the flow of control between blocks in a program. Each block node contains information about the block and several pointers. One pointer is to the head of a linked list of the instructions the block contains. Two other pointers show to

which blocks control will pass on exit from the current block. The first pointer is to the path taken if the branch at the end of the current block (if there is one) is taken; the other represents the default path. The structure thus built is shown in Fig A.1.

Since all the branches in the code work by relative addressing it is necessary, while building the flowgraph, to be able to identify instructions in the flowgraph by addresses. For this reason, an array is kept which may be indexed by address to yield a pointer to the instruction at that address. Indices to the array are byte addresses and so are divided by two since instructions may only start on even addresses. Naturally, all instructions are not two bytes long and so some entries in the array will not be used. The logical structure this array represents is shown in Fig A.2.

Building the flowgraph

Instructions are fetched from the compiler's code buffers and decoded one at a time. Decoding consists of deciding of the format of the instruction and its operands. A new instruction node is generated for each node which contains the opcode, the operands and a pointer to the block node to which it belongs. The new instruction node is added to the list of instructions for the current block.

Most of the work in building the flowgraph results from dealing with special instructions and building the block structure. Multiply and divide instructions are checked since if one operand is a constant then it will be stored following the code and the actual instruction will reference a memory location. The optimiser picks up the constant and stores this in the instruction instead. Another special case occurs with load

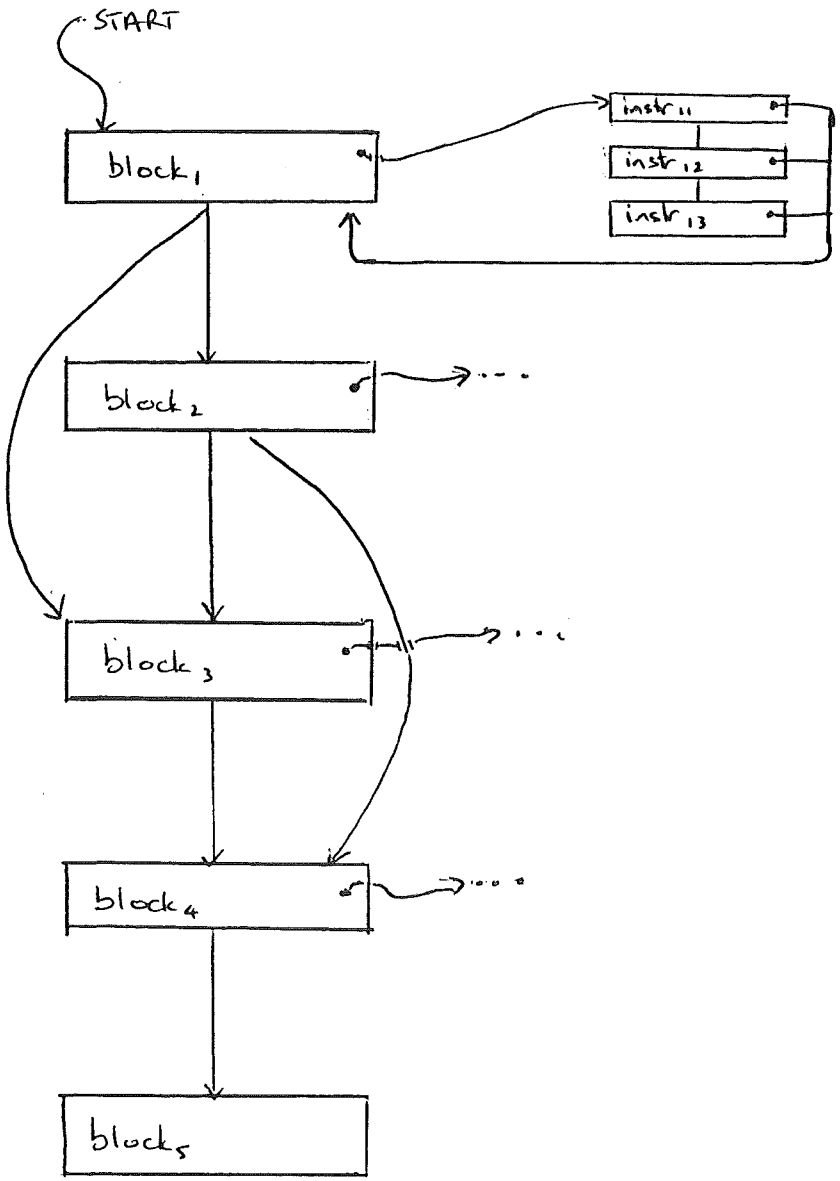


Fig A-1 Flowgraph

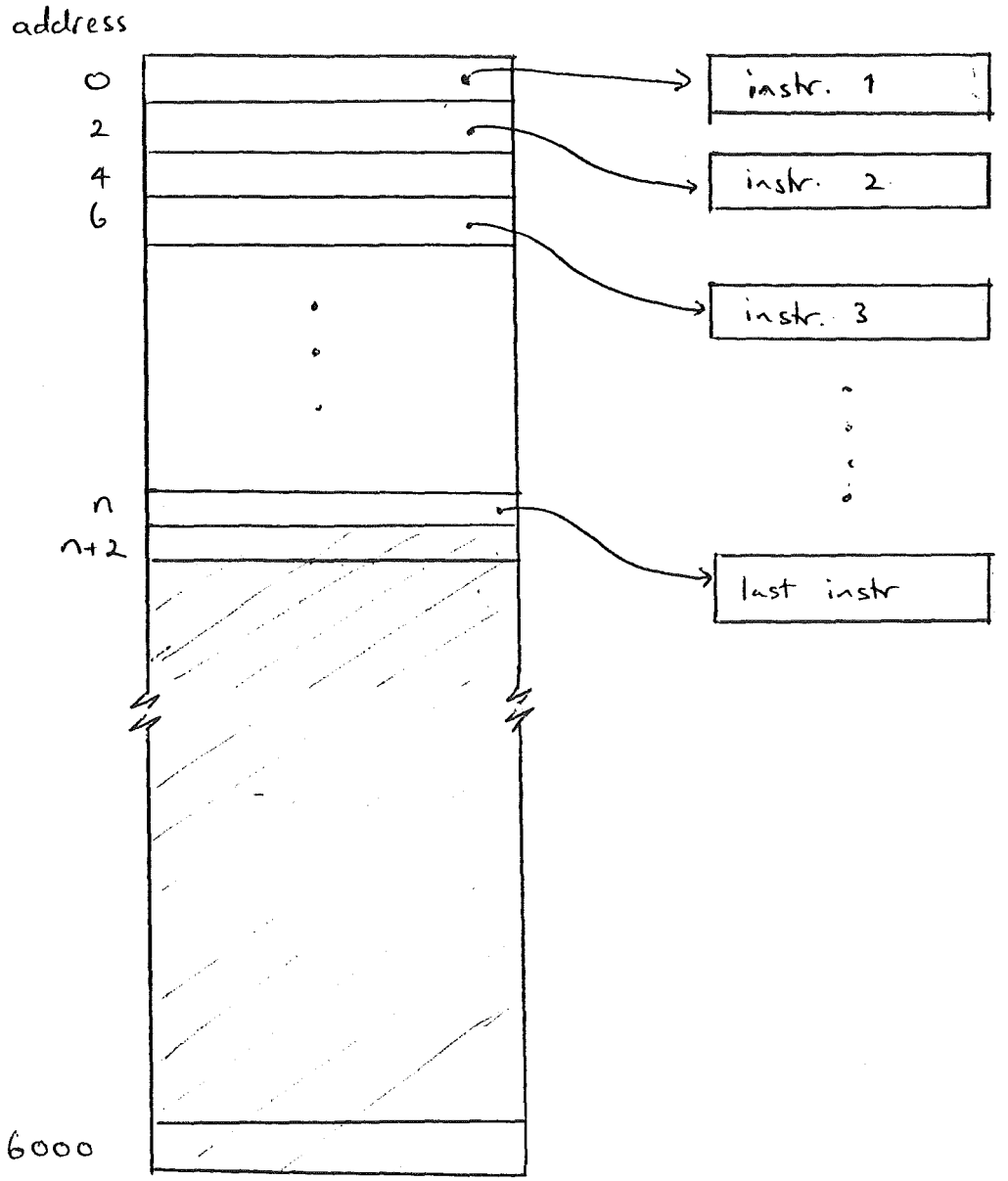


Fig A.2 Access to instructions via addresses

address and subroutine call instructions. These instructions are involved in the calling of procedures and runtime routines, all of which are treated as external references. For the loader's benefit, instructions which reference a certain external object, rather than having the address of the object as an operand (since this is not known) have the address within the current procedure of the previous reference to that object. These chains of addresses have to be replaced by chains of pointers in the flowgraph: this is a further use for the array which indexes instructions via addresses.

A similar task to the address chains is involved for branching which, once again, involve translating an operand which specifies an address to a pointer to an instruction. Dealing with branches, however, is complicated by the fact that a branch indicates that some modification to the block structure of the flowgraph may be needed. Forward branches are handled simply by leaving a pointer to the branch instruction at the place in the address array that represents the destination. Before processing an instruction, the entry in the array is checked. If not nil then a new block is set up. The array entry indicates the branch instruction which needs to be modified to point to the new block. Backward branches are, in some ways, trickier to deal with. The array can be used to find the instruction branched to and thus the block to which that instruction belongs. If the destination instruction is any other than the first instruction in a block then that block must be split into two at that point. This is a fairly tricky operation since it requires quite a lot of updating of pointers in the affected block nodes and in the instructions that belong to those blocks.

Data structures for DAGs.

Since a DAG is a graph, it may be represented by a pointer structure in Pascal. Each node in the graph is labelled with the operation it represents and contains pointers to nodes representing its operands. Attached to each node is a linked list of references to objects which currently contain the value represented by the sub-tree of which the node is the root. Since the graph is largely tree-like, many of the nodes will actually be leaves with only a value, not an operation. These leaves are treated exactly like other nodes but are labelled with the dummy operation LEAF and the values are attached in the usual fashion. It should also be noted that references to variables can be attached to two nodes in the DAG. In this case, one must be attached to a leaf and thus denotes the initial value of the variable. The structure of a DAG is illustrated in Fig A.3.

During the process of building a DAG it is necessary to be able to find, for a given object, the node of the DAG to which it is currently attached. This is done by means of a hashing function which provides a unique object index for the object. This index is then used to access an array which defines the current mapping of objects onto nodes of the DAG. A similar method is needed to access a node of the DAG which has two particular nodes as its descendants: that is, hashing is used to index an array which in turn provides a pointer to the node required.

A final note about the DAG structure is that a DAG may consist, either during or after its construction, of several physically separate **components**. Therefore, instead of being able to think of the DAG as a singly-rooted tree, it is necessary to actually implement it by a linked list of components, each of which represents a sub-tree (possibly of only one node).

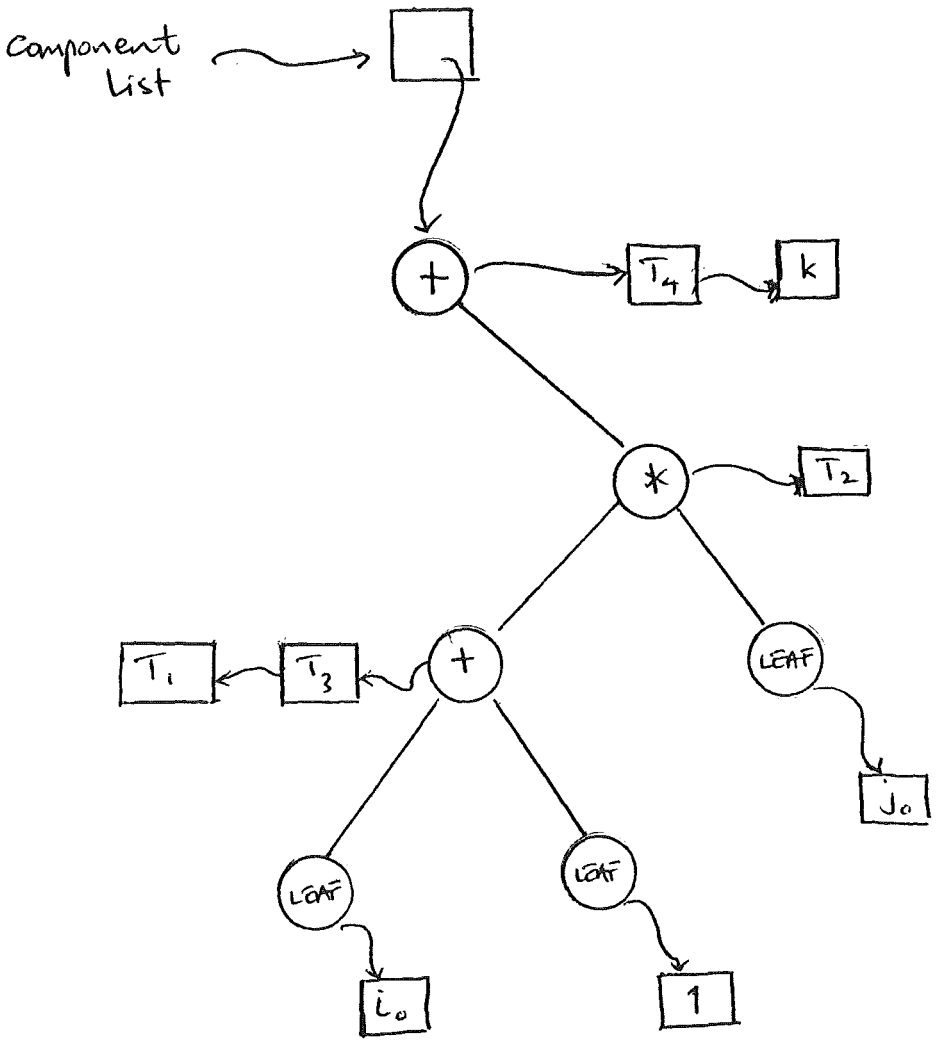


Fig A.3 A DAG
[from Fig 5.1 (e)]

Building a DAG

An instruction from a basic block is added to a DAG in the following manner. Firstly, the nodes in the DAG for the operands must be found, and if not found must be created. Secondly, a check is made to see if there is already a node in the DAG labelled with the operation that this instruction represents and with the two nodes just found for descendants. Such a node, if found, represents a common sub-expression. If such a node is not found, then one must be created with the appropriate label and with the two nodes previously found as descendants. Except for the multiply and divide instructions, the result of the operation will end up in the first operand. This is indicated by attaching the first operand to the node just created for the operation. Obviously, it must also be removed from the node to which it was previously attached and the array that defines the mapping of objects onto nodes must be updated. The final act in updating the DAG is to replace, in the list of components of the DAG, the components representing the two operands by the single component representing the new sub-tree.

Multiply and divide instructions have to be treated differently because the use of register pairs means that the result does not necessarily end up in the first operand of the instruction. Load and store operations also have to be treated separately since they involve no operation as such but entail only the moving of objects from one node to another in the DAG.

```
 1 {$E+}
 2 {$P+}
 3 program optimiser(input,output,pasobj,pasincl,$pasmsgs,
 4                   opfile,optout) ;
 5
 6 {$C+}
 7
 8 INCLUDE GLOBALS.PAS;
 9
10 procedure optimise( procname : alfa )  ;
11
12 const
13 INCLUDE OPS.CST;
14    maxaddr = 3000 ;
15    stats = false ;
16    list_code = false ;
17    list_blocks = true ;
18    maxblocks = 300 ;
19    maxn = 255 ; { maximum DAG node no. }
20    trace = false ;
21
22 type
23
24    dagptr = ^dagnodes ;
25
26    proc_name = packed array [1..20] of char ;
27    formats = (RR,SF,RX,RX1,RX2,RX3,RI1,RI2,RXRX,NULL) ;
28    opcode = 0..265 ;
29    opinfo = record
30       mnemonic : array [1..5] of char ;
31       format : formats ;
32       class : integer ;
33       end ;
34    oper_kinds = (register, constant, variable, indexed, location,
35                  result) ;
36    instrptr = ^instruction ;
37    blockptr = ^block ;
38    reg_range = 0..15 ;
39    pred_rec = record
40       this_pred : blockptr ;
41       next : ^pred_rec ;
42       end ;
43    operand = record
44       case kind : oper_kinds of
45          register : (number : reg_range) ;
46          constant : (value : integer ;
47                      reg   : reg_range ) ;
48          variable,
49          indexed  : (address : boolean ;
50                      base : reg_range ;
51                      index : reg_range ;
52                      offset : integer ) ;
53          location : (ptr : instrptr ) ;
54          result   : (result_kind : integer ;
55                      dagnode : dagptr ) ;
```

```
56          end ;
57    instruction = record
58       id : integer ;
59       op : opcode ;
60       format : formats ;
61       oper1,
62       oper2  : operand ;
63       next   : instrptr ;
64       owner  : blockptr ;
65       size   : integer ;
66       end ;
67    block = record
68       id : integer ;
69       instr_head,
70       instr_tail : instrptr ;
71       true_successor,
72       false_successor : blockptr ;
73       pred_list_head : ^pred_rec ;
74       size : integer ;
75       ninstrs : integer ;
76       case boolean of
77          true : (dummy : 0..255) ;
78          false: (flags : packed record
79                    visited,
80                    f2,f3,f4,f5,f6,f7,f8 : boolean ;
81                    end ) ;
82       end ;
83    stat_rec = record
84       n, min, max, avg : integer ;
85       end ;
86    string = packed array [1..12] of char ;
87
88       dagnodes = record { describing a DAG node }
89          killed : boolean ;
90          num : 0..maxn ;
91          op : opcode ;
92          opclass : integer ;
93          oper : array [1..2] of dagptr ;
94          first_item : ^item ;
95          end ;
96
97       item = record { describes an object attached to a node }
98          init_val : boolean ;
99          oper : operand ;
100         next : ^item ;
101         end ;
102
103
104 var
105    stores : integer ;
106    locs : array [0..maxaddr] of instrptr ;
107    dfn : array [1..256] of blockptr ;
108    pc : integer ;
109    id : integer ;
110    bid : integer ;
```

```
111     optable : array [0..265] of opinfo ;
112     finished : boolean ;
113     traced : boolean ;
114     hi, lo, flag, data, loc, num : integer ;
115     save, previous: instrptr ;
116     temp : integer ;
117     twoto : array [0..31] of integer ;
118     p, head : instrptr ;
119     b, blockhead : blockptr ;
120     pp : ^pred_rec ;
121     i, j, stat : integer ;
122     filen : packed array [1..16] of char ;
123     tail : packed array [1..8] of char ;
124     pname : packed array [1..8] of char ;
125     nprocs, nblocs,
126     totproc, totbloc,
127     minproc, maxproc, aveproc,
128     minbloc, maxbloc, avebloc : integer ;
129     newbloc, bloc : blockptr ;
130     instrcnt,
131     blocksize : stat_rec ;
132
133 procedure gettime( var tim : timerec ) ; cal ;
134
135 procedure update_stat( new_n : integer ;
136                        var stat : stat_rec ) ;
137
138     begin
139     with stat do
140        begin
141        n := n + 1 ;
142        if new_n < min then
143           min := new_n ;
144        if new_n > max then
145           max := new_n ;
146        avg := avg + new_n ;
147        end ;
148     end ;    { update_stat }
149
150 procedure report_stat( str : string ;
151                        stat : stat_rec ) ;
152
153     begin
154     with stat do
155        begin
156        write(optout,str,' n   = ',n:6) ;
157        write(optout,'  min = ',min:6) ;
158        write(optout,'  MAX  = ',max:6) ;
159        writeln(optout,'  avg  = ',(avg div n):6) ;
160        end ;
161     end ;    { report_stat }
162
163 procedure report_time( str : string ) ;
164
165     var
```

```
166       timenow : timerec ;
167
168    begin   { report_time }
169    gettime(timenow) ;
170    write(optout,str,' CPU = ',(timenow.usercpu - timehold.usercpu):6) ;
171    write(optout,'  SVC = ',(timenow.oscpu - timehold.oscpu):6) ;
172    write(optout,'  WAIT = ',timenow.wait - timehold.wait:6) ;
173    writeln(optout,'  ROLL = ',(timenow.rolled - timehold.rolled):6) ;
174    timehold := timenow ;
175    end ;   { report_time }
176
177    procedure opt_error( errno : integer ;
178                         where : proc_name ) ;
179
180       begin
181       if errno > 0 then
182          begin
183          write(optout,' *** Error ***  ') ;
184          case errno of
185             0:       write(' Dummy error.') ;
186             else:    write(' Error number ',errno:1) ;
187             end ;
188          end
189       else
190          begin
191          write(optout,' *** Warning ***  ') ;
192          case errno of
193             0:       write(' Dummy warning.') ;
194             else:    write(optout, ' Warning number ',errno:1) ;
195             end ;
196          end ;
197       writeln(optout,'  [',where,']') ;
198       end ;
199
200    procedure operout( oper : operand ) ;
201
202       begin
203       with oper do
204          case kind of
205             register :   write(optout,number:1) ;
206             constant :   begin
207                          write(optout,'=',value:1) ;
208                          if reg <> 0 then
209                             write(optout,'(',reg:1,')') ;
210                          end ;
211             variable :   write(optout,offset:1,'(',base:1,')') ;
212             indexed  :   write(optout,offset:1,'(',index:1,',',base:1,')') ;
213             location :   if ptr <> nil then
214                             write(optout,'*',ptr^.id:1)
215                          else
216                             write(optout,'#') ;
217             result:      write(optout,'@',dagnode^.num:1) ;
218 else:write(optout,'?') ;
219          end ;
220       end ;
```

```
221
222 procedure initialise ;
223
224    var
225       i : integer ;
226       val : integer ;
227
228    procedure init_stat( var stat : stat_rec ) ;
229
230       begin   { init_stat }
231       with stat do
232          begin
233          n := 0 ;
234          min := maxint - 1 ;
235          max := -1 ;
236          avg := 0 ;
237          end ;
238       end ;   { init_stat }
239
240    begin
241    writeln(optout,'Optimising ',procname,'  (',ic:1,' bytes)') ;
242    for i := 0 to maxaddr do
243       locs[i] := nil ;
244    val := 1 ;
245    for i := 0 to 31 do
246       begin
247       twoto[i] := val ;
248       if val <> 31 then val := val + val ;
249       end ;
250    stores := 0 ;
251    nblocs := 0 ;
252    nprocs := 0 ;
253    totbloc := 0 ;
254    totproc := 0 ;
255    minproc := maxint ;
256    minbloc := maxint ;
257    maxproc := -1 ;
258    maxbloc := -1 ;
259    tail := '.PRO,ERO' ;
260    init_stat(blocksize) ;
261    init_stat(instrcnt) ;
262    end ;
263
264
265 procedure initoptable ;
266
267    var
268       i, status : integer ;
269       j : 0..9 ;
270
271    begin
272
273    assignfl(opfile,'OPFILE,SRO',status) ;
274    if status = 0 then
275       begin
```

```
276       reset(opfile) ;
277       for i := 0 to 265 do
278          begin
279          for j := 1 to 5 do
280             read(opfile,optable[i].mnemonic[j]) ;
281          readln(opfile,j,optable[i].class) ;
282          optable[i].format := formats(j) ;
283          end ;
284       end ;
285
286    end ;
287
288
289 procedure splitbyte( byte : integer ;
290                      var hi, lo : integer ) ;
291
292    begin
293    hi := byte div 16 ;
294    lo := byte mod 16 ;
295    end ;
296
297 function fetch( i : integer ) : integer ;
298
299    var
300       locptr : codesptr ;
301
302    begin
303    locptr := codeptr[i div codeperseg] ;
304    fetch := ord( locptr^.bytes[i mod codeperseg] ) ;
305    end ;    { fetch }
306
307 function nextbyte : integer ;
308
309    begin    { nextbyte }
310    nextbyte := fetch(pc) ;
311    pc := pc + 1 ;
312    if (pc = 18) and pmd then
313       pc := pc + 24 ;
314    save^.size := save^.size + 1 ;
315    end ;    { nextbyte }
316
317 function signed( num : integer ;
318                  bits : integer ) : integer ;
319
320    begin
321    if num >= twoto[bits - 1] then
322       signed := num - twoto[bits]
323    else
324       signed := num ;
325    end ;
326
327 function value( size : integer ;
328                 sign : boolean ) : integer ;
329
330    var
```

```
331       magic : record
332          case boolean of
333             true :    (word : integer) ;
334             false:    (bytes: packed array [1..4] of 0..255) ;
335             end ;
336       i : integer ;
337
338    begin
339    magic.word := 0 ;
340    for i := (4 - size + 1) to 4 do
341       magic.bytes[i] := nextbyte ;
342    if sign and (size <> 4) then
343       value := signed(magic.word,size*8)
344    else
345       value := magic.word ;
346    end ;
347
348 procedure flowgraph ;
349
350 var
351    ip : instrptr ;
352    endoflist : boolean ;
353    rxrxflag : boolean ;
354
355    procedure add_predecessor( { to } bloc2,
356                         { preceded by } bloc1 : blockptr ) ;
357
358       var
359          predp : ^pred_rec ;
360
361       begin { add_predecessor }
362       new(predp) ;
363       predp^.this_pred := bloc1 ;
364       predp^.next := bloc2^.pred_list_head ;
365       bloc2^.pred_list_head := predp ;
366       end ;    { add_predecessor }
367
368    procedure swap_predecessors( bloc : blockptr ;
369                                 oldp, newp : blockptr ) ;
370
371       var
372          predp : ^pred_rec ;
373          swapped : boolean ;
374
375       begin   { swap_predecessors }
376       if bloc <> nil then
377       begin
378       swapped := false ;
379       predp := bloc^.pred_list_head ;
380       while not swapped and (predp <> nil) do
381          begin
382          if predp^.this_pred = oldp then
383             begin
384             predp^.this_pred := newp ;
385             swapped := true ;
```

```
386             end
387          else
388             predp := predp^.next ;
389          end ;
390       end ;
391       end ;    { swap_predecessors }
392
393    procedure newblock( var bp : blockptr ;
394                        bsize : integer ;
395                        ip : instrptr ;
396                        preb : blockptr ;
397                        trusucc, falsucc : boolean ;
398                        tsucb, fsucb : blockptr ) ;
399
400       begin
401       bid := bid + 1 ;
402       new(bp) ;
403       with bp^ do
404          begin
405          id := bid ;
406          dummy := 0 ;
407          size := bsize ;
408          ninstrs := 0 ;
409          instr_head := ip ;
410          true_successor := tsucb ;
411          false_successor := fsucb ;
412          pred_list_head := nil ;
413          end ;
414       if preb <> nil then
415          begin
416          add_predecessor(bp,preb) ;
417          if trusucc then
418             preb^.true_successor := bp ;
419          if falsucc then
420             preb^.false_successor := bp ;
421          end ;
422       end ;
423
424
425 begin
426 pc := 0 ;
427 id := 1 ;
428 bid := 0 ;
429 previous := nil ;
430 finished := false ;
431 rxrxflag := false ;
432 newblock(bloc,0,nil,nil,false,false,nil,nil) ;
433 while not finished do
434    begin
435    save := locs[pc div 2] ;
436    new(ip) ;
437    if (save <> nil) and (bloc^.instr_head <> nil) then
438       newblock(bloc,0,nil,bloc,true,true,nil,nil) ;
439    while save <> nil do
440       begin
```

```
441         save^.owner^.true_successor := bloc ;
442         add_predecessor(bloc,save^.owner) ;
443         p := save^.oper2.ptr ;
444         save^.oper2.ptr := ip ;
445         save := p ;
446         end ;
447      ip^.id := id ;
448      id := id + 1 ;
449      save := ip ;
450      ip^.size := 0 ;
451      ip^.next := nil ;
452      ip^.owner := bloc ;
453      bloc^.instr_tail := ip ;
454      if bloc^.instr_head = nil then
455         bloc^.instr_head := ip ;
456      locs[pc div 2] := ip ;
457      with locs[pc div 2]^ do
458         begin
459         op := nextbyte ;
460         format := optable[op].format ;
461         if rxrxflag or (format = RXRX) then
462            format := RX ;
463         case format of
464            RR  :    begin
465                     splitbyte(nextbyte,hi,lo) ;
466                     oper1.kind := register ;
467                     oper1.number := hi ;
468                     oper2.kind := register ;
469                     oper2.number := lo ;
470                     end ;
471            SF  :    begin
472                     splitbyte(nextbyte,hi,lo) ;
473                     oper1.kind := register ;
474                     oper1.number := hi ;
475                     oper2.kind := constant ;
476                     oper2.value := lo ;
477                     oper2.reg := 0 ;
478                     end ;
479            RX  :    begin
480                     splitbyte(nextbyte,hi,lo) ;
481                     oper1.kind := register ;
482                     oper1.number := hi ;
483                     temp := nextbyte ;
484                     splitbyte(temp,flag,data) ;
485                     if flag = 4 then { = 0100 }
486                        format := RX3
487                     else
488                        if flag >= 8 then { = 1xxx }
489                           format := RX2
490                        else
491                           format := RX1 ;
492                     case format of
493                        RX1:    begin
494                                oper2.kind := variable ;
495                                oper2.base := lo ;
```

```
496                         oper2.offset := temp * 256 + nextbyte ;
497                         end ;
498                      RX2:   begin
499                         oper2.kind := variable ;
500                         oper2.base := lo ;
501                         oper2.offset := signed(
502                            (temp - 128) * 256 + nextbyte, 15) ;
503                         end ;
504                      RX3:   begin
505                         oper2.kind := indexed ;
506                         oper2.base := lo ;
507                         oper2.index := data ;
508                         oper2.offset := nextbyte * 32768 +
509                           nextbyte * 256 + nextbyte ;
510                         end ;
511                   end ;
512                end ;
513          RI1,
514          RI2 :    begin
515                   splitbyte(nextbyte,hi,lo) ;
516                   oper1.kind := register ;
517                   oper1.number := hi ;
518                   oper2.kind := constant ;
519                   oper2.reg := lo ;
520                   if format = RI1 then
521                      oper2.value := value( 2, true )
522                   else
523                      oper2.value := value( 4, true ) ;
524                   end ;
525          NULL,
526          RXRX:    ;
527          end ;
528       bloc^.size := bloc^.size + size ;
529       bloc^.ninstrs := bloc^.ninstrs + 1 ;
530       if not rxrxflag then
531       case op of
532          M, MH,
533          D, DH:    if format = RX2 then
534                       if oper2.base = 0 then
535                          begin
536                          loc := pc + oper2.offset ;
537                          num := fetch(loc) * 256 + fetch(loc + 1) ;
538                          if (op = M) or (op = D) then
539                             num := num * 32768 + fetch(loc + 2) *
540                                    256 + fetch(loc + 3) ;
541                          oper2.kind := constant ;
542                          oper2.value := num ;
543                          oper2.reg := 0 ;
544                          end ;
545          ST,STH,STB :    stores := stores + 1 ;
546          BTBS..BFFS,
547          BTC,
548          BFC  :    begin
549                    if format = SF then
550                       if (op = BTFS) or (op = BFFS) then
```

```
                        loc := pc - 2 + oper2.offset * 2
                     else
                        loc := pc - 2 - oper2.offset * 2
                  else
                     if format = RX2 then
                        loc := pc + oper2.offset
                     else
                        loc := oper2.offset ;
                  oper2.kind := location ;
                  if loc < pc then
                     begin
                     oper2.ptr := locs[loc div 2] ;
                     b := oper2.ptr^.owner ;
                     if oper2.ptr <> b^.instr_head then
                        begin
                        newblock(newbloc,pc-loc,
                          oper2.ptr,b,true,true,b^.true_successor,
                          b^.false_successor) ;
                        swap_predecessors(newbloc^.true_successor,
                          b,newbloc) ;
                        swap_predecessors(newbloc^.false_successor,
                          b,newbloc) ;
                        p := newbloc^.instr_head ;
                        endoflist := false ;
                        while not endoflist do
                           begin
                           if p^.owner = b then
                              begin
                              newbloc^.ninstrs := newbloc^.ninstrs + 1 ;
                              p^.owner := newbloc
                              end
                           else
                              endoflist := true ;
                           p := p^.next ;
                           if p = nil then
                              endoflist := true ;
                           end ;
                        b^.ninstrs := b^.ninstrs - newbloc^.ninstrs ;
                        end ;
                     bloc^.true_successor := oper2.ptr^.owner ;
                     add_predecessor(oper2.ptr^.owner,bloc) ;
                     end
                  else
                     begin
                     oper2.ptr := locs[loc div 2] ;
                     locs[loc div 2] := save ;
                     end ;
                  newblock(bloc,0,nil,bloc,false,true,nil,nil) ;
                  end ;
         LA,
         SVCX,
         BAL   :  begin
                  if (format = RX3) and (op <> SVCX) then
                     if (oper2.base = 0) and (oper2.index = 0) then
                        begin   { relocatable item }
```

```
606                         oper2.kind := location ;
607                         if oper2.offset <> 0 then
608                            oper2.ptr := locs[(oper2.offset-2) div 2]
609                         else
610                            oper2.ptr := nil ;
611                         end ;
612                   if op <> LA then
613                      newblock(bloc,0,nil,bloc,true,true,nil,nil) ;
614                   end ;
615            BFCR:  if oper2.number = 15 then
616                      begin
617                      bloc^.true_successor := nil ;
618                      bloc^.false_successor := nil ;
619                      finished := true ;
620                      end ;
621            RXSQ:  rxrxflag := true ;
622            ELSE:  ;
623            end
624         else
625            begin
626            rxrxflag := false ;
627            splitbyte(op,hi,lo) ;
628            op := BLOP ;
629            if (hi mod 4) <> 0 then
630               previous^.op := lo + 260
631            else
632               previous^.op := lo + 256 ;
633            if hi in [4..7] then
634               begin
635               temp := oper1.number ;
636               oper1.kind := constant ;
637               oper1.value := temp ;
638               oper1.reg := 0 ;
639               end ;
640            if hi in [8..11] then
641               begin
642               temp := previous^.oper1.number ;
643               previous^.oper1.kind := constant ;
644               previous^.oper1.value := temp ;
645               previous^.oper1.reg := 0 ;
646               end ;
647            if hi in [12..15] then
648               begin
649               temp := previous^.oper1.number ;
650               previous^.oper1.kind := constant ;
651               previous^.oper1.value := temp ;
652               previous^.oper1.reg := 0 ;
653               temp := oper1.number ;
654               oper1.kind := constant ;
655               oper1.value := temp ;
656               oper1.reg := 0 ;
657               end ;
658            end ;
659         if previous <> nil then
660            previous^.next := save ;
```

```
661       previous := save ;
662       end ;
663    end ;
664 end ;
665
666 procedure intra_block_optimise ;
667
668    const
669       maxnodes = 256 ;    { maximum no. of nodes/DAG }
670       maxn = 255 ;
671
672    type
673
674       opnodes = record
675         occupied : boolean ;
676         opclass : integer ;
677         opn : array [1..2] of 0..maxn ;
678         dagnode : dagptr ;
679         end ;
680
681       comp_link = record
682          dagnode : dagptr ;
683          next : ^comp_link ;
684          end ;
685
686       objects = record {describes a data item }
687         occupied : boolean ;
688         oper : operand ;
689         dagnode : dagptr ;
690         end ;
691
692    var
693       object : array [0..maxn] of objects ;
694       nextn : 0..maxn ;
695       first_component, last_component : ^comp_link ;
696       base_of_heap : ^integer ;
697       tempoper : operand ;
698
699    procedure trip( procedure visit( dagptr, operand ) ;
700                    oper : operand ) ;
701
702    { 'visit's every node in the DAG by doing a post-order
703      traversal of each of the components: i.e. the trees
704      whose roots formed the comp_link list }
705
706       var
707          cp : ^comp_link ; { chains down list of roots }
708
709       procedure traverse( root : dagptr ;
710                           procedure visit( dagptr, operand ) ;
711                           oper :operand ) ;
712
713       { perform post-order traversal of tree belonging to
714         'root'. Each node is 'visit'ed }
715
```

```
716          begin   { traverse }
717          if root^.oper[1] <> nil then { traverse left subtree }
718             traverse(root^.oper[1],visit,oper) ;
719          if root^.oper[2] <> nil then { traverse right subtree }
720             traverse(root^.oper[2],visit,oper) ;
721          visit(root,oper) ; { process this node }
722          end ;   { traverse }
723
724       begin   { trip }
725       cp := first_component ;
726       while cp <> nil do { traverse tree for this component }
727          begin
728          traverse(cp^.dagnode,visit,oper) ;
729          cp := cp^.next ;
730          end ;
731       end ;   { trip }
732
733    procedure kill( dagnode : dagptr ;
734                    oper : operand ) ;
735
736    { kill the indicated node. oper is a dummy arg }
737
738       begin   { kill }
739       dagnode^.killed := true ;
740       end ;   { kill }
741
742    procedure arraykill( dagnode : dagptr ;
743                         oper : operand ) ;
744
745    { kill node if is an INDX node with base address 'oper' }
746
747       begin   { arraykill }
748       if dagnode^.op = INDX then
749          with dagnode^.oper[1]^.first_item^.oper do
750               if (base = oper.base) and (offset = oper.offset ) then
751                  dagnode^.killed := true ;
752       end ;   { arraykill }
753
754    procedure dump_node( dagnode : dagptr ;
755                         oper : operand ) ;
756
757       { Dump the contents of a DAG node. }
758
759       var
760          itemp : ^item ;
761
762       begin   { dump_node }
763       with dagnode^ do
764          begin
765          write(optout,'Node #',num:1,':  ',optable[op].mnemonic) ;
766          write(optout,' ') ;
767          if oper[1] <> nil then
768             write(optout,oper[1]^.num:1)
769          else
770             write(optout,'-') ;
```

```
771          write(optout,',') ;
772          if oper[2] <> nil then
773             write(optout,oper[2]^.num:1)
774          else
775             write(optout,'-') ;
776          writeln(optout) ;
777          write(optout,'  attached items: ') ;
778          itemp := first_item ;
779          while itemp <> nil do
780             begin
781             operout(itemp^.oper) ;
782             itemp := itemp^.next ;
783             if itemp <> nil then
784                write(optout,', ') ;
785             end ;
786          writeln(optout) ;
787          end ;
788       end ;   { dump_node }
789
790
791    function equal( oper1, oper2 : operand ) : boolean  ;
792
793       begin
794       equal := true ;
795       if oper1.kind = oper2.kind then
796          case oper1.kind of
797             register:   if oper1.number <> oper2.number then
798                            equal := false ;
799             constant:   begin
800                         if (oper1.value <> oper2.value) or
801                            (oper1.reg <> oper2.reg) then
802                            equal := false ;
803                         end ;
804             variable:   if (oper1.base <> oper2.base) or
805                            (oper1.offset <> oper2.offset) then
806                            equal := false ;
807             indexed:    if (oper1.base <> oper2.base) or
808                            (oper1.offset <> oper2.offset) or
809                            (oper1.index <> oper2.index) then
810                            equal := false ;
811             result:     if (oper1.result_kind <> oper2.result_kind)
812                         or (oper1.dagnode <> oper2.dagnode) then
813                            equal := false ;
814             else:       equal := false ;
815             end
816          else
817             equal := false ;
818       end ;
819
820    function object_ident( oper : operand ) : 0..maxn ;
821
822       var
823          id : 0..maxn ;
824
825       function hash_object( oper : operand ) : integer ;
```

```
826
827         begin   { hash_object }
828         with oper do
829            case kind of
830               register:  hash_object := number ;
831               constant:  hash_object := value mod (maxnodes - 16) + 16 ;
832               variable:  hash_object :=
833                 (base * ((maxnodes -16) div 6) + offset) mod
834                 (maxnodes - 16) + 16 ;
835               result:    hash_object :=
836                  dagnode^.num mod (maxnodes -16) + 16 ;
837               else:      hash_object := maxnodes - 1 ;
838               end ;
839          end ;   { hash_object }
840
841      function rehashop( n : integer ) : integer ;
842
843          begin
844          rehashop := (n + 7) mod maxnodes ;
845          end ;   { rehash }
846
847      begin   { object_ident }
848      id := hash_object(oper) ;
849      while not equal(oper,object[id].oper) and
850            (object[id].occupied) do
851         id := rehashop(id) ;
852      if not object[id].occupied then
853         begin
854         object[id].occupied := true ;
855         object[id].oper := oper ;
856         object[id].dagnode := nil ;
857         end ;
858      object_ident := id ;
859      end ;   { object_ident }
860
861   function newitem( oper : operand ) : ^item ;
862
863   { generate an item for oper }
864
865      var
866         itemp : ^item ;
867
868      begin   { newitem }
869      if trace then
870         begin
871         write(optout,'Generating new item for ') ;
872         operout(oper) ;
873         writeln(optout) ;
874         end ;
875      new(itemp) ;
876      itemp^.init_val := false ;
877      itemp^.oper := oper ;
878      itemp^.next := nil ;
879      newitem := itemp ;
880      end ;   { newitem }
```

```
881
882   procedure move_item( oper : operand ;
883                        from_node, to_node : dagptr ) ;
884
885     { Attach item for 'oper' to 'to_node' item list.
886       Delete from 'from_node' list if appropriate. }
887
888     var
889       itemp, lastitemp : ^item ;
890       found : boolean ;
891       id : 0..maxn ;
892
893     begin   { move_item }
894     if trace then
895        begin
896        write(optout,'Moving ') ;
897        operout(oper) ;
898        write(optout,' from ') ;
899        if from_node <> nil then
900           write(optout,from_node^.num:1)
901        else
902           write(optout,'-') ;
903        write(optout,' to ') ;
904        if to_node <> nil then
905           write(optout,to_node^.num:1)
906        else
907           write(optout,'-') ;
908        writeln(optout) ;
909        end ;
910     if from_node = nil then { new item needed }
911        itemp := newitem(oper)
912     else { find and remove current item }
913        begin
914        itemp := from_node^.first_item ;
915        lastitemp := nil ;
916        found := false ;
917        while not found and (itemp <> nil) do
918           if not equal(itemp^.oper,oper) then
919              begin
920              lastitemp := itemp ;
921              itemp := itemp^.next ;
922              end
923           else
924              found := true ;
925        if found then { remove from 'from_node' list }
926           if lastitemp = nil then { remove head of list }
927              from_node^.first_item := itemp^.next
928           else
929              lastitemp^.next := itemp^.next
930        else { *** error *** }
931           begin
932           opt_error(-1,'move_item           ') ;
933           itemp := newitem(oper) ;
934           end ;
935        itemp^.next := nil ;
```

```
936          end ;
937       itemp^.next := to_node^.first_item ;
938       to_node^.first_item := itemp ;
939       id := object_ident(oper) ;
940       object[id].dagnode := to_node ;
941       end ;    { move_item }
942
943    procedure replace( a, b, c : dagptr ) ;
944
945       { Replaces componets a and b by component c. This
946         currently only requires deletion of a and b. }
947
948       var
949          cp : ^comp_link ;
950          done : boolean ;
951
952       begin   { replace }
953       cp := first_component ;
954       last_component := nil ;
955       while cp <> nil do
956          begin
957          done := false ;
958          if (cp^.dagnode = a) or (cp^.dagnode = b) then
959             begin
960             if last_component <> nil then
961                last_component^.next := cp^.next
962             else
963                first_component := cp^.next ;
964             done := true ;
965             end ;
966          if not done then
967             last_component := cp ;
968          cp := cp^.next ;
969          end ;
970       end ;    { replace }
971
972    procedure build_dag( b : blockptr ) ;
973
974       var
975          opnode : array [0..maxn] of opnodes ;
976          ip : instrptr ;
977          tempoper : operand ;
978
979       function newnode( op : opcode ;
980                         opclass : integer ;
981                         son1, son2 : dagptr ) : dagptr ;
982
983          var
984             np : dagptr ;
985             cp : ^comp_link ;
986
987          begin   { newnode }
988          if trace then
989             begin
990             write(optout,'Creating new node (',nextn:1,') labelled ',
```

```
 991                optable[op].mnemonic,' (',opclass:1,')') ;
 992             write(optout,' with sons ') ;
 993             if son1 <> nil then
 994                write(optout,son1^.num:1)
 995             else
 996                write(optout,'nil') ;
 997             write(optout,' and ') ;
 998             if son2 <> nil then
 999                write(optout,son2^.num:1)
1000             else
1001                write(optout,'nil') ;
1002             writeln(optout) ;
1003             end ;
1004          new(np) ;
1005          np^.num := nextn ;
1006          nextn := nextn + 1 ;
1007          np^.op := op ;
1008          np^.opclass := opclass ;
1009          np^.oper[1] := son1 ;
1010          np^.oper[2] := son2 ;
1011          np^.first_item := nil ;
1012
1013          new(cp) ;
1014          cp^.dagnode := np ;
1015          cp^.next := nil ;
1016          if last_component <> nil then
1017             last_component^.next := cp
1018          else
1019             first_component := cp ;
1020          last_component := cp ;
1021
1022          newnode := np ;
1023          end ;    { newnode }
1024
1025       function intnode( np1, np2 : dagptr ;
1026                         op : opcode ;
1027                         opclass : integer ) : dagptr ;
1028
1029          var
1030             n, nn1, nn2 : 0..maxn ;
1031
1032          function inthash( i1, i2 : integer ) : integer ;
1033
1034             begin
1035             inthash := (i1 * i2) mod maxnodes ;
1036             end ;    { inthash }
1037
1038          function rehash( n : integer ) : integer ;
1039
1040             begin    { rehash }
1041             rehash := (n + 7) mod maxnodes ;
1042             end ;    { rehash }
1043
1044          begin    { intnode }
1045          if trace then
```

```
1046            begin
1047            write(optout,'Find/create int. node for ',
1048               optable[op].mnemonic,' (',opclass:1,') with operands') ;
1049            if np1 <> nil then
1050               write(optout,np1^.num:1)
1051            else
1052               write(optout,'nil') ;
1053            write(optout,' and ') ;
1054            if np2 <> nil then
1055               write(optout,np2^.num:1)
1056            else
1057               write(optout,'nil') ;
1058            writeln(optout) ;
1059            end ;
1060         nn1 := np1^.num ;
1061         if np2 <> nil then
1062            nn2 := np2^.num
1063         else
1064            nn2 := 1 ;
1065         n := inthash(nn1,nn2) ;
1066         while (opnode[n].occupied) and
1067               (opnode[n].opn[1] = nn1) and
1068               (opnode[n].opn[2] = nn2) and
1069               (opnode[n].opclass = opclass) do
1070            n := rehash(n) ;
1071         if opnode[n].occupied then
1072            intnode := opnode[n].dagnode
1073         else
1074            begin
1075            opnode[n].occupied := true ;
1076            opnode[n].dagnode := newnode(op,opclass,np1,np2) ;
1077            intnode := opnode[n].dagnode ;
1078            opnode[n].opn[1] := nn1 ;
1079            opnode[n].opn[2] := nn2 ;
1080            opnode[n].opclass := opclass ;
1081            end ;
1082         end ;  { intnode }
1083
1084      function node( var oper : operand ;
1085                     ip : instrptr ;
1086                     createnew : boolean ) : dagptr ;
1087
1088         var
1089            temp : operand ;
1090            n1, n2, n3 : dagptr ;
1091            id : 0..maxn ;
1092
1093         begin { node }
1094         if trace then
1095            begin
1096            if createnew then
1097               write(optout,'Find/create ')
1098            else
1099               write(optout,'Find ') ;
1100            write(optout,'node for ') ;
```

```
1101            operout(oper) ;
1102            writeln(optout) ;
1103            end ;
1104
1105        if createnew and (ip^.op = LA) and (ip^.format <> RX3) then
1106            begin
1107            n1 := newnode(LEAF,SPEC,nil,nil) ;
1108            move_item(oper,nil,n1) ;
1109            n2 := newnode(ADDR,SPEC,n1,nil) ;
1110            oper.kind := result ;
1111            oper.result_kind := ADDR ;
1112            oper.dagnode := n2 ;
1113            end
1114        else
1115            if (oper.offset = 0) and (oper.kind = variable) then
1116                begin
1117                temp.kind := register ;
1118                temp.number := oper.base ;
1119                n1 := node(temp,ip,true) ;
1120                n2 := newnode(DREF,SPEC,n1,nil) ;
1121                oper.kind := result ;
1122                oper.result_kind := DREF ;
1123                oper.dagnode := n2 ;
1124                node := node( oper, ip, createnew ) ;
1125                end
1126            else
1127                if (oper.kind = indexed) then
1128                    begin
1129                    temp.kind := variable ;
1130                    temp.base := oper.base ;
1131                    temp.offset := oper.offset ;
1132                    n1 := node(temp,ip,true) ;
1133                    temp.kind := register ;
1134                    temp.number := oper.index ;
1135                    n2 := node(temp,ip,true) ;
1136                    n3 := intnode(n1,n2,INDX,SPEC) ;
1137                    oper.kind := result ;
1138                    oper.result_kind := INDX ;
1139                    oper.dagnode := n3 ;
1140                    node := node( oper, ip, createnew ) ;
1141                    end
1142                else
1143                    begin
1144                    id := object_ident( oper ) ;
1145                    if createnew and (object[id].dagnode = nil) then
1146                        begin
1147                        object[id].dagnode :=
1148                            newnode(LEAF,SPEC,nil,nil) ;
1149                        move_item(oper,nil,object[id].dagnode) ;
1150                        end ;
1151                    node := object[id].dagnode ;
1152                    end ;
1153        end ;    { node }
1154
1155    procedure add_instr_to_dag( ip : instrptr ) ;
```

```
1156
1157          var
1158             a, b, c : dagptr ;
1159             opclass : integer ;
1160             tempoper : operand ;
1161
1162          begin   { add_instr_to_dag }
1163          with ip^ do
1164             begin
1165             if trace then
1166                begin
1167                write(optout,'Adding ',optable[op].mnemonic) ;
1168                operout(oper1) ;
1169                write(optout,', ') ;
1170                operout(oper2) ;
1171                writeln(optout,' to DAG.') ;
1172                end ;
1173             if optable[op].class > 17 then
1174                opclass := op
1175             else
1176                opclass := -optable[op].class ;
1177             if trace then
1178                writeln(optout,'Class is ',opclass:1) ;
1179             if (opclass = LOAD) then
1180                begin
1181                if trace then
1182                   writeln(optout,'It is a LOAD instruction.') ;
1183                b := node(oper2,ip,true) ;
1184                a := node(oper1,ip,false) ;
1185                move_item(oper1,a,b) ;
1186                end
1187             else
1188                if (opclass = STORE) then
1189                   begin
1190                   if trace then
1191                      writeln(optout,'It is a STORE instruction.') ;
1192                   b := node(oper1,ip,true) ;
1193                   a := node(oper2,ip,false) ;
1194                   move_item(oper2,a,b) ;
1195                   if oper2.kind = result then
1196                      if oper2.result_kind = INDX then
1197                         begin
1198                         tempoper :=
1199                            oper2.dagnode^.oper[1]^.first_item^.oper ;
1200                         trip(arraykill,tempoper) ;
1201                         end
1202                      else
1203                         trip(kill,tempoper) ;
1204                   end
1205                else
1206                   begin
1207                   if trace then
1208                      writeln(optout,'It is a dyadic instruction.') ;
1209                   a := node(oper1,ip,true) ;
1210                   b := node(oper2,ip,true) ;
```

```
1211                    c := intnode(a,b,op,opclass) ;
1212                    replace(a,b,c) ;
1213                    move_item(oper1,a,c) ;
1214                    end ;
1215              end ;
1216           end ;    { add_instr_to_dag }
1217
1218        procedure dag_init ;
1219
1220           var
1221              i : 0..maxn ;
1222
1223            begin
1224            first_component := nil ;
1225            last_component := nil ;
1226            for i := 0 to maxn do
1227               begin
1228               object[i].occupied := false ;
1229               opnode[i].occupied := false ;
1230               end ;
1231            end ;
1232
1233        begin
1234        dag_init ;
1235        nextn := 0 ; { first DAG node number }
1236        ip := b^.instr_head ;
1237        while ip <> b^.instr_tail do
1238           begin
1239           add_instr_to_dag(ip) ;
1240           trip(dump_node,tempoper) ;
1241           ip := ip^.next ;
1242           end ;
1243        end ;    { build_dag }
1244
1245     procedure squeeze_dag( b : blockptr ) ;
1246
1247        { Optimise the DAG. }
1248
1249        begin
1250        end ;
1251
1252     procedure code_dag( b : blockptr ) ;
1253
1254        { Regenerate flowgraph node from DAG. }
1255
1256        begin
1257        end ;
1258
1259     begin
1260     b := locs[0]^.owner ;
1261     while b <> nil do
1262        begin
1263        mark(base_of_heap) ;
1264        build_dag(b) ;
1265        report_time('DAG build    ') ;
```

```
1266        trip(dump_node,tempoper) ;
1267        squeeze_dag(b) ;
1268        code_dag(b) ;
1269        b := b^.false_successor ;
1270        end ;
1271     end ;    { intra_block_optimise }
1272
1273
1274  procedure depth_first ;
1275
1276     var
1277        dfi : 0..256 ;
1278
1279     procedure search( b : blockptr ) ;
1280
1281        procedure checkson( s : blockptr ) ;
1282
1283           begin
1284           if s <> nil then
1285              if not s^.flags.visited then
1286                 search(s) ;
1287           end ;
1288
1289        begin
1290        b^.flags.visited := true ;
1291        checkson(b^.true_successor) ;
1292        checkson(b^.false_successor) ;
1293        dfn[dfi] := b ;
1294        dfi := dfi - 1 ;
1295        end ;
1296
1297     begin
1298     dfi := bid ;
1299     if locs[0] <> nil then
1300        search(locs[0]^.owner) ;
1301     end ;
1302
1303  procedure print( instr : instruction ) ;
1304
1305     begin
1306     with instr do
1307        begin
1308        write(optout,id:5,') ') ;
1309        write(optout,optable[op].mnemonic:6) ;
1310        write(optout,'  ') ;
1311        operout(oper1) ;
1312        write(optout,',') ;
1313        operout(oper2) ;
1314        writeln(optout) ;
1315        end ;
1316     end ;
1317
1318  procedure blokid( bptr : blockptr ) ;
1319
1320     begin
```

```
1321     if bptr <> nil then
1322        write(optout,'B',bptr^.id:1)
1323     else
1324        write(optout,'NIL') ;
1325     end ;
1326
1327 procedure analyse_graph ;
1328
1329 var
1330        b : blockptr ;
1331
1332     begin   { analyse_graph }
1333     b := locs[0]^.owner ;
1334     while b <> nil do
1335        begin
1336        with b^ do
1337           begin
1338           if list_blocks then
1339              begin
1340              write(optout,' [B',id:1,',') ;
1341              blokid(true_successor) ;
1342              write(optout,',') ;
1343              blokid(false_successor) ;
1344              writeln(optout,',*',instr_head^.id:1,',',size:1,']') ;
1345              end ;
1346           update_stat(size,blocksize) ;
1347           update_stat(ninstrs,instrcnt) ;
1348           end ;
1349        b := b^.false_successor ;
1350        end ;
1351     end ;    { analyse_graph }
1352
1353 begin
1354 initialise ;
1355 initoptable ;
1356 report_time('Compile      ') ;
1357 if (ic div 2) <= maxaddr then
1358     begin
1359     flowgraph ;
1360     report_time('Flowgraph    ') ;
1361     analyse_graph ;
1362     report_stat(' block size ',blocksize) ;
1363     report_stat(' instruction',instrcnt) ;
1364     intra_block_optimise ;
1365 {   depth_first ; }
1366    if list_code then
1367     begin
1368     p := locs[0] ;
1369     while p <> nil do
1370        begin
1371        print(p^) ;
1372        p := p^.next ;
1373        end ;
1374     end ;
1375     end
```

Appendix B

Ad hoc patches

This appendix describes the patches made to the compiler to do optimisation. Also included are listings of parts of the compiler that were substantially altered. Finally, there is a listing of a program and the code generated by the original and modified compilers to demonstrate the effect of the patches.

The first patches made were to procedures INTARITH, BOOLARITH, and RELINT. These patches perform folding of constants in integer, boolean, and relational expressions respectively. They also simplify the (rare) case where only one operand is constant but is an identity for the operation.

The remaining patches were all made in an attempt to implement register management for fullword constants and variables. Modifications to LOWD check to see if an item is already in a register before actually loading it. If a load is necessary then the item is remembered as being in the register. Further modifications are made to REGSEARCH so that if it runs out of registers it will reuse a register occupied by a constant or variable rather than aborting compilation.

As described earlier, it is vital that the contents of the registers are not assumed to be intact over basic block boundaries. For this reason, patches have been made in all the parsing procedures where a basic block is likely to begin (e.g. at the start of a 'while' statement). This did not prove stringent enough since lower levels of the compiler may generate

calls to run-time routines which may also invalidate the registers. The simple solution to this, namely killing the registers (via KILLALL) when a subroutine call (BAL) is generated, is quite easily done but unfortunately leads to errors in some places where the compiler uses knowledge of the behaviour of run-time routines to explicitly assume the validity of registers across such calls.

```
(*2957*)    PROCEDURE INTARITH(F1ATTRP,F2ATTRP:ATTRP; FOP:OPERATOR);
(*2958*)      VAR OPRX,OPRR : INTEGER;                                                    (*nz*)
                  FOLDED : BOOLEAN ;
(*2959*)      BEGIN
              FOLDED := FALSE ;
              (* FOLD CONSTANTS AND CATCH OPERATIONS ON IDENTITIES *)
              IF (F1ATTRP^.KIND = CST) AND (F2ATTRP^.KIND = CST) THEN
                 IF (F1ATTRP^.CVAL.CKIND = INT) AND (F2ATTRP^.CVAL.CKIND = INT) THEN
                    BEGIN
                    CASE FOP OF
                       PLUS :    F2ATTRP^.CVAL.IVAL := F1ATTRP^.CVAL.IVAL + F2ATTRP^.CVAL.IVAL ;
                       MINUS:    F2ATTRP^.CVAL.IVAL := F1ATTRP^.CVAL.IVAL - F2ATTRP^.CVAL.IVAL ;
                       MUL  :    F2ATTRP^.CVAL.IVAL := F1ATTRP^.CVAL.IVAL * F2ATTRP^.CVAL.IVAL ;
                       IDIV :    F2ATTRP^.CVAL.IVAL := F1ATTRP^.CVAL.IVAL DIV F2ATTRP^.CVAL.IVAL ;
                       IMOD :    F2ATTRP^.CVAL.IVAL := F1ATTRP^.CVAL.IVAL MOD F2ATTRP^.CVAL.IVAL ;
                       END ;
                    FOLDED := TRUE
                    END
                 ELSE
              ELSE
                 BEGIN
                 IF (F1ATTRP^.KIND = CST) AND (F1ATTRP^.CVAL.CKIND = INT) THEN
                    CASE FOP OF
                       PLUS :    IF F1ATTRP^.CVAL.IVAL = 0 THEN
                                    FOLDED := TRUE ;
                       MUL  :    IF F1ATTRP^.CVAL.IVAL = 1 THEN
                                    FOLDED := TRUE ;
                       ELSE :    ;
                       END ;
                 IF (F2ATTRP^.KIND = CST) AND (F2ATTRP^.CVAL.CKIND = INT) THEN
                    CASE FOP OF
                       PLUS,
                       MINUS:    IF F2ATTRP^.CVAL.IVAL = 0 THEN
                                    BEGIN
                                    EXCATTR(F1ATTRP, F2ATTRP) ;
                                    FOLDED := TRUE ;
                                    END ;
                       MUL,
                       IDIV :    IF F2ATTRP^.CVAL.IVAL = 1 THEN
                                    BEGIN
                                    EXCATTR(F1ATTRP, F2ATTRP) ;
                                    FOLDED := TRUE ;
                                    END ;
                       ELSE :    ;
                       END ;
                 END ;
               IF NOT FOLDED THEN
                BEGIN
(*2960*)        IF FOP IN (.PLUS,MUL.) THEN
(*2961*)          IF F2ATTRP@.KIND=EXPR THEN
(*2962*)            IF F2ATTRP@.REXPR.REGTEMP=REGIST THEN EXCATTR(F1ATTRP,F2ATTRP);
(*2963*)        CASE FOP OF
(*2964*)          PLUS:   BEGIN OPRX := ZA; OPRR := ZAR; LOWD(F1ATTRP,F2ATTRP); END;      (*nz*)
(*2965*)          MINUS:  BEGIN OPRX := ZS; OPRR := ZSR; LOWD(F1ATTRP,F2ATTRP); END;      (*nz*)
(*2966*)          MUL:    BEGIN OPRX := ZM; OPRR := ZMR; LOADEVENODD(F1ATTRP,F2ATTRP,1); END;  (*nz*)
(*2967*)          IDIV,IMOD:  BEGIN OPRX := ZD; OPRR := ZDR; LOADEVENODD(F1ATTRP,F2ATTRP,0);  (*nz*)
                              GENRRP(ZLR,SUCC(F1ATTRP^.REXPR.RNO),F1ATTRP^.REXPR.RNO);  (*nz*)
                              GENRI1P(ZSRA,F1ATTRP^.REXPR.RNO,0,31);                    (*nz*)
                              REGISTER[SUCC(F1ATTRP^.REXPR.RNO)].USED := TRUE;          (*nz*)
                              REGISTER[SUCC(F1ATTRP^.REXPR.RNO)].REGCONT := F1ATTRP;    (*nz*)
(*2969*)                    END
(*2970*)        END;
```

```
(*2977*)            WITH F1ATTRP@.REXPR DO
(*2978*)              BEGIN REGISTER(.SUCC(RNO).):=REGISTER(.RNO.);
(*2979*)                    REGISTER(.RNO.).USED:=FALSE; RNO:=SUCC(RNO);
(*2980*)              END;
                    IF FOP=IMOD THEN  REGISTER[SUCC(F1ATTRP^.REXPR.RNO)].USED := FALSE;                    (*nz*)
(*2981*)            EXCATTR(F1ATTRP,F2ATTRP);
                    END ;   (* IF NOT FOLDED *)
(*2982*)          END;
(*2983*)
```

```
(*3042*)    PROCEDURE BOOLARITH(F1ATTRP,F2ATTRP: ATTRP; FOP: OPERATOR);
(*3043*)      VAR X:INTEGER;
(*3044*)      BEGIN
              (* FOLD CONSTANTS *)
              IF (F1ATTRP^.KIND = CST) AND (F1ATTRP^.CVAL.CKIND = INT) THEN
                 IF (F2ATTRP^.KIND = CST) AND (F2ATTRP^.CVAL.CKIND = INT) THEN
                    IF FOP = ANDOP THEN
                       F2ATTRP^.CVAL.IVAL := F1ATTRP^.CVAL.IVAL* F2ATTRP^.CVAL.IVAL
                    ELSE
                       IF FOP = OROP THEN
                          IF (F1ATTRP^.CVAL.IVAL = 1) AND (F2ATTRP^.CVAL.IVAL = 0) THEN
                             F2ATTRP^.CVAL.IVAL := 1
                       ELSE
                          ERROR(400) (* ILLEGAL BOOLEAN OP *)
                 ELSE
                    IF FOP = ANDOP THEN
                       IF F1ATTRP^.CVAL.IVAL = 0 THEN
                          BEGIN
                          F2ATTRP^.KIND := CST ;
                          F2ATTRP^.CVAL.CKIND := INT ;
                          F2ATTRP^.CVAL.IVAL := 0 ;
                          END
                       ELSE
                    ELSE
                       IF FOP = OROP THEN
                          IF F1ATTRP^.CVAL.IVAL = 1 THEN
                             BEGIN
                             F2ATTRP^.KIND := CST ;
                             F2ATTRP^.CVAL.CKIND := INT ;
                             F2ATTRP^.CVAL.IVAL := 1 ;
                             END
                          ELSE
                       ELSE
                          ERROR(400) (* ILLEGAL BOOLEAN OP *)
               ELSE
                 IF (F2ATTRP^.KIND = CST) AND (F2ATTRP^.CVAL.CKIND = INT) THEN
                    IF FOP = ANDOP THEN
                       IF F2ATTRP^.CVAL.IVAL = 1 THEN
                          EXCATTR( F1ATTRP, F2ATTRP )
                       ELSE
                    ELSE
                       IF FOP = OROP THEN
                          IF F2ATTRP^.CVAL.IVAL = 0 THEN
                             EXCATTR( F1ATTRP, F2ATTRP )
                          ELSE
                       ELSE
                          ERROR(400) (* ILLEGAL BOOLEAN OP *)
                 ELSE
                    BEGIN (* FORMER BOOLARITH *)
(*3045*)             IF F2ATTRP@.KIND=EXPR THEN EXCATTR(F1ATTRP,F2ATTRP);
(*3046*)             LOWD(F1ATTRP,F2ATTRP);
(*3047*)             IF FOP=ANDOP THEN X:=0 ELSE X:=ZO-ZN;
(*3048*)             OPERATION(F1ATTRP,F2ATTRP,ZN+X,ZNR+X);
(*3049*)             EXCATTR(F1ATTRP,F2ATTRP);
                     END ; (* OF STANDARD BOOLARITH *)
(*3050*)      END;
```

```
(*3058*)    PROCEDURE RELINT(F1ATTRP,F2ATTRP: ATTRP; FOP: OPERATOR);
(*3059*)      BEGIN
              (* FOLD CONSTANTS *)
              IF (F1ATTRP^.KIND = CST) AND (F2ATTRP^.KIND = CST) THEN
                 IF (F1ATTRP^.CVAL.CKIND = INT) AND (F2ATTRP^.CVAL.CKIND = INT) THEN
                    BEGIN
                    CASE FOP OF
                       LTOP : F2ATTRP^.CVAL.IVAL := ORD(F1ATTRP^.CVAL.IVAL < F2ATTRP^.CVAL.IVAL) ;
                       LEOP : F2ATTRP^.CVAL.IVAL := ORD(F1ATTRP^.CVAL.IVAL <= F2ATTRP^.CVAL.IVAL) ;
                       GEOP : F2ATTRP^.CVAL.IVAL := ORD(F1ATTRP^.CVAL.IVAL >= F2ATTRP^.CVAL.IVAL) ;
                       GTOP : F2ATTRP^.CVAL.IVAL := ORD(F1ATTRP^.CVAL.IVAL > F2ATTRP^.CVAL.IVAL) ;
                       NEOP : F2ATTRP^.CVAL.IVAL := ORD(F1ATTRP^.CVAL.IVAL <> F2ATTRP^.CVAL.IVAL) ;
                       EQOP : F2ATTRP^.CVAL.IVAL := ORD(F1ATTRP^.CVAL.IVAL = F2ATTRP^.CVAL.IVAL) ;
                       END ;
                    F2ATTRP^.TYPTR := BOOLPTR ;
                    END
                 ELSE
                    ERROR(400) (* NOT AN INTEGER *)
              ELSE
                 BEGIN
(*3060*)         IF F2ATTRP@.KIND=EXPR THEN
(*3061*)           BEGIN FOP:=DUALOP(.FOP.); EXCATTR(F1ATTRP,F2ATTRP);
(*3062*)           END;
(*3063*)         LOWD(F1ATTRP,F2ATTRP);
(*3064*)         OPERATION(F1ATTRP,F2ATTRP,ZC,ZCR);
(*3065*)         BOOLVALUE(REALREG(.F1ATTRP@.REXPR.RNO.),BMASK(.FOP.));
(*3066*)         F1ATTRP@.TYPTR := BOOLPTR;
(*3067*)         EXCATTR(F1ATTRP,F2ATTRP);
                 END ;
(*3068*)      END;
```

```
            PROCEDURE KILLREG( R : REGNO ) ;

               BEGIN   (* KILLREG *)
               WITH REGISTER[R] DO
                  BEGIN
                  ATTRDISP(REGCONT) ;
                  REGCONT := NIL ;
                  NOTEXPR := FALSE ;
                  USED := FALSE ;
                  END ;
               END ;    (* KILLREG *)

            PROCEDURE KILLALL ;

            (* KILL ALL REGISTERS HOLDING RECENT VARIABLES *)

               VAR
                  R : REGNO ;

               BEGIN   (* KILLALL *)
               FOR R := R10 TO R13 DO
                  IF REGISTER[R].USED AND REGISTER[R].NOTEXPR THEN
                     KILLREG(R) ;
               END ;    (* KILLALL *)

(*2671*)    PROCEDURE REGSEARCH(FATTRP:ATTRP; T:REGKIND);
(*2672*)      LABEL 1;

              BEGIN                                                                    (*nz*)
                IF REGSELCT                                (* use specified register to    *)   (*nz*)
                  THEN                                     (* reload loop control variable *)   (*nz*)
                    IF REGISTER[SELREGNO].USED THEN                                    (*nz*)
                       IF REGISTER[SELREGNO].NOTEXPR THEN
                          BEGIN
                          RWORK := SELREGNO ;
                          KILLREG(RWORK) ;
                          END
                       ELSE
                          ERROR(400)
                    ELSE
                       RWORK := SELREGNO
                  ELSE                                                                 (*nz*)
(*2673*)          CASE T OF                                                            (*nz*)
(*2674*)          SINGLE: BEGIN
                             FOR RWORK := R10 TO R13 DO
(*2675*)                        IF NOT REGISTER(.RWORK.).USED THEN GOTO 1 ;
                             FOR RWORK := R10 TO R13 DO
                               IF REGISTER[RWORK].NOTEXPR THEN
                                  BEGIN
                                  KILLREG(RWORK) ;
                                  GOTO 1 ;
                                  END ;
(*2676*)                     FOR RWORK:=R10 TO R13 DO
(*2677*)                        IF NOT (USING(RWORK,FATTRP) OR SSINDEX[RWORK])       (*nz*)
                                  THEN BEGIN SAVE(RWORK); GOTO 1 END;                (*nz*)
(*2678*)                     ERROR(400);
(*2679*)                  END;
(*2680*)          FLOAT:  BEGIN FOR RWORK:=F0 TO F6 DO
(*2681*)                        IF NOT REGISTER(.RWORK.).USED THEN GOTO 1;
(*2682*)                  FOR RWORK:=F0 TO F6 DO
(*2683*)                     IF NOT USING(RWORK,FATTRP) THEN BEGIN SAVE(RWORK); GOTO 1 END;
(*2684*)                  ERROR(400);
```

```
(*2686*)                IF NOT(REGISTER(.R10.).USED OR REGISTER(.R11.).USED) THEN RWORK:=R10
(*2687*)                  ELSE IF NOT(REGISTER(.R12.).USED OR REGISTER(.R13.).USED) THEN RWORK:=R12
(*2688*)                  ELSE IF NOT(USING(R10,FATTRP) OR USING(R11,FATTRP)) THEN
(*2689*)                          BEGIN SAVE(R10); SAVE(R11); RWORK:=R10 END
                          ELSE IF NOT(USING(R12,FATTRP) OR USING(R13,FATTRP)) THEN                (*nz*)
(*2690*)                          BEGIN SAVE(R12); SAVE(R13); RWORK:=R12 END                      (*nz*)
                          ELSE  ERROR(400);                                                       (*nz*)
                        END ;   (* DOUBLE *)
(*2691*)          END;
(*2692*)       1: RMAIN:=REALREG(.RWORK.);
(*2693*)        END;
(*2694*)

              FUNCTION INREG( FATTRP : ATTRP ) : BOOLEAN ;

              (* SETS RWORK AND RETURNS TRUE IF FATTRP IS
                 CURRENTLY HELD IN A REGISTER.  FATTRP MUST
                 BE A FULLWORD CONSTANT OR VARIABLE *)

                 LABEL
                    1 ;

                 BEGIN   (* INREG *)
                 INREG := FALSE ;
                 WITH FATTRP^ DO
                    IF ((KIND = CST) OR (KIND = VARBL)) AND (TYPTR^.FORM <> POWER) THEN
                       FOR RWORK := R10 TO R13 DO (* LATER TO BE EXPANDED TO USE A REGSET AND FORALL *)
                          IF REGISTER[RWORK].USED THEN
                             WITH REGISTER[RWORK].REGCONT^ DO
                                IF KIND = FATTRP^.KIND THEN
                                   IF KIND = CST THEN
                                      IF CVAL.CKIND = FATTRP^.CVAL.CKIND THEN
                                         IF CVAL.CKIND = INT THEN
                                            IF CVAL.IVAL = FATTRP^.CVAL.IVAL THEN
                                               BEGIN
                                               INREG := TRUE ;
                                               GOTO 1 ;
                                               END
                                            ELSE
                                         ELSE
                                      ELSE
                                   ELSE
                                      IF KIND = VARBL THEN
                                         IF (ACCESS = DIRECT) AND (FATTRP^.ACCESS = DIRECT) THEN
                                            IF (VADRS = FATTRP^.VADRS) AND (VLEVEL = FATTRP^.VLEVEL) THEN
                                               BEGIN
                                               INREG := TRUE ;
                                               GOTO 1 ;
                                               END ;
              1: END ;    (* INREG *)
```

```
(*2742*)      BEGIN     (* LOWD *)                                                    (*nz*)
                IF F1ATTRP^.TYPTR<>NIL                                               (*nz*)
                  THEN  WITH F1ATTRP^,TYPTR^ DO                                      (*nz*)
(*2743*)        BEGIN
(*2744*)          IF (KIND<>EXPR) OR (REXPR.REGTEMP<>REGIST) THEN
(*2745*)            BEGIN
                    (* CHECK REGISTERS *)
                    IF NOT INREG(F1ATTRP) THEN
                      BEGIN
                      ....
                      (* Normal LOWD code *)
                      ....
                      HOLD := FALSE ;
                      IF RKIND = SINGLE THEN
                         IF KIND = CST THEN
                            HOLD := TRUE
                         ELSE
                            IF (KIND = VARBL) AND (ACCESS = DIRECT) THEN
                               HOLD := TRUE ;
                      IF HOLD THEN
                         BEGIN
                         ATTRNEW(REGISTER[RWORK].REGCONT) ;
                         COPYATTR(F1ATTRP,REGISTER[RWORK].REGCONT) ;
                         REGISTER[RWORK].NOTEXPR := TRUE ;
                         END
                      ELSE
                         REGISTER[RWORK].REGCONT := F1ATTRP ;
                      END ; (* OF NOT INREG(F1ATTRP) *)
(*2787*)               KIND:=EXPR; REXPR.REGTEMP:=REGIST; REXPR.RNO:=RWORK;
(*2788*)               REGISTER(.RWORK.).USED:=TRUE; (*REGISTER(.RWORK.).REGCONT:=F1ATTRP; *)
(*2789*)               IF RKIND=DOUBLE THEN
(*2790*)                 BEGIN REGISTER(.SUCC(RWORK).).USED:=TRUE;
(*2791*)                       REGISTER(.SUCC(RWORK).).REGCONT:=F1ATTRP;
                               REGFORM := RFORMTMP;                                  (*nz*)
(*2792*)                 END;
(*2793*)            END;
(*2794*)        END;
(*2795*)      END;
```

```
0040 --      1  program optimisations ;
0040 --      2
0040 --      3  {$C+}
0040 --      4
0040 --      5  const
0040 --      6     max = 99 ;
0040 --      7     debug = true ;
0040 --      8
0040 --      9  var
0040 --     10     i, j, k : integer ;
004C --     11     a, b, c : boolean ;
004F --     12
0000 0-     13  begin
002A --     14
002A --     15  { Register management }
002A --     16
002A --     17  i := 0 ;
0030 --     18  j := 0 ;
0036 --     19
0036 --     20  { constant folding }
0036 --     21
0036 --     22  k := max - 1 ;
0040 --     23
0040 --     24  if debug and (max > 100) then
005A 1-     25     begin
005A --     26     a := (i = j) or (max > 50) ;
0090 --     27     b := (i = j) or (max > 99) ;
00C6 -1     28     end
00C6 --     29  else
00CA 1-     30     begin
00CA --     31     a := (i = j) and (max > 50) ;
0100 --     32     b := (i = j) and (max > 99) ;
0136 -1     33     end ;
0136 --     34
0136 -0     35  end.
GENERATED CODE LISTING:
  0000   D008  0000          STM     0,0(8)
  0004   0818                LR      1,8
  0006   FA80  0000  0050    AI      8,80
  000C   41F0  4000  0000    BAL     15,0
  0012   502E  4D41  494E  2020
  001A   3036  2F31  302F  3833
  0022   3132  3A31  303A  3331
  002A   24A0                LIS     10,0
  002C   50A1  0040          ST      10,64(1)
  0030   24A0                LIS     10,0
  0032   50A1  0044          ST      10,68(1)
  0036   C8A0  0063          LHI     10,99
  003A   27A1                SIS     10,1
  003C   50A1  0048          ST      10,72(1)
  0040   C8A0  0063          LHI     10,99
  0044   C9A0  0064          CHI     10,100
  0048   E6A0  0001          LA      10,1
  004C   4220  8002          BTC     2,*+2
  0050   24A0                LIS     10,0
  0052   C4A0  0001          NHI     10,1
  0056   4330  8070          BFC     3,*+112
  005A   58A1  0040          L       10,64(1)
```

```
005E  59A1  0044        C       10,68(1)
0062  E6A0  0001        LA      10,1
0066  4330  8002        BFC     3,*+2
006A  24A0              LIS     10,0
006C  C8B0  0063        LHI     11,99
0070  C9B0  0032        CHI     11,50
0074  E6B0  0001        LA      11,1
0078  4220  8002        BTC     2,*+2
007C  24B0              LIS     11,0
007E  06BA              OR      11,10
0080  2114              BTFS    1,4
0082  C9B0  0001        CHI     11,1
0086  2323              BFFS    2,3
0088  E1B0  80B4        SVC     11,*+180
008C  D2B1  004C        STB     11,76(1)
0090  58A1  0040        L       10,64(1)
0094  59A1  0044        C       10,68(1)
0098  E6A0  0001        LA      10,1
009C  4330  8002        BFC     3,*+2
00A0  24A0              LIS     10,0
00A2  C8B0  0063        LHI     11,99
00A6  C9B0  0063        CHI     11,99
00AA  E6B0  0001        LA      11,1
00AE  4220  8002        BTC     2,*+2
00B2  24B0              LIS     11,0
00B4  06BA              OR      11,10
00B6  2114              BTFS    1,4
00B8  C9B0  0001        CHI     11,1
00BC  2323              BFFS    2,3
00BE  E1B0  807E        SVC     11,*+126
00C2  D2B1  004D        STB     11,77(1)
00C6  4300  806C        BFC     0,*+108
00CA  58A1  0040        L       10,64(1)
00CE  59A1  0044        C       10,68(1)
00D2  E6A0  0001        LA      10,1
00D6  4330  8002        BFC     3,*+2
00DA  24A0              LIS     10,0
00DC  C8B0  0063        LHI     11,99
00E0  C9B0  0032        CHI     11,50
00E4  E6B0  0001        LA      11,1
00E8  4220  8002        BTC     2,*+2
00EC  24B0              LIS     11,0
00EE  04BA              NR      11,10
00F0  2114              BTFS    1,4
00F2  C9B0  0001        CHI     11,1
00F6  2323              BFFS    2,3
00F8  E1B0  8044        SVC     11,*+68
00FC  D2B1  004C        STB     11,76(1)
0100  58A1  0040        L       10,64(1)
0104  59A1  0044        C       10,68(1)
0108  E6A0  0001        LA      10,1
010C  4330  8002        BFC     3,*+2
0110  24A0              LIS     10,0
0112  C8B0  0063        LHI     11,99
0116  C9B0  0063        CHI     11,99
011A  E6B0  0001        LA      11,1
011E  4220  8002        BTC     2,*+2
0122  24B0              LIS     11,0
```

```
0124   04BA             NR      11,10
0126   2114             BTFS    1,4
0128   C9B0  0001       CHI     11,1
012C   2323             BFFS    2,3
012E   E1B0  800E       SVC     11,*+14
0132   D2B1  004D       STB     11,77(1)
0136   5071  001C       ST      7,28(1)
013A   D101  0000       LM      0,0(1)
013E   030F             BFCR    0,15
0140   140B  2711
0144   0000  0000
```

* NO ERRORS DETECTED IN PASCAL PROGRAM OPTIMISA *

```
0040 --      1  program optimisations ;
0040 --      2
0040 --      3  {$C+}
0040 --      4
0040 --      5  const
0040 --      6     max = 99 ;
0040 --      7     debug = true ;
0040 --      8
0040 --      9  var
0040 --     10     i, j, k : integer ;
004C --     11     a, b, c : boolean ;
004F --     12
0000 0-     13  begin
002A --     14
002A --     15  { Register management }
002A --     16
002A --     17  i := 0 ;
0030 --     18  j := 0 ;
0034 --     19
0034 --     20  { constant folding }
0034 --     21
0034 --     22  k := max - 1 ;
003C --     23
003C --     24  if debug and (max > 100) then
0040 1-     25     begin
0040 --     26     a := (i = j) or (max > 50) ;
0058 --     27     b := (i = j) or (max > 99) ;
007C -1     28     end
007C --     29  else
0080 1-     30     begin
0080 --     31     a := (i = j) and (max > 50) ;
00A4 --     32     b := (i = j) and (max > 99) ;
00BC -1     33     end ;
00BC --     34
00BC -0     35  end.
GENERATED CODE LISTING:
  0000   D008  0000          STM    0,0(8)
  0004   0818                LR     1,8
  0006   FA80  0000  0050    AI     8,80
  000C   41F0  4000  0000    BAL    15,0
  0012   502E  4D41  494E  2020
  001A   3036  2F31  302F  3833
  0022   3132  3A31  313A  3335
  002A   24A0                LIS    10,0
  002C   50A1  0040          ST     10,64(1)
  0030   50A1  0044          ST     10,68(1)
  0034   C8B0  0062          LHI    11,98
  0038   50B1  0048          ST     11,72(1)
  003C   4330  8040          BFC    3,*+64
  0040   58C1  0040          L      12,64(1)
  0044   59C1  0044          C      12,68(1)
  0048   E6C0  0001          LA     12,1
  004C   4330  8002          BFC    3,*+2
  0050   24C0                LIS    12,0
  0052   24C1                LIS    12,1
  0054   D2C1  004C          STB    12,76(1)
  0058   58D1  0040          L      13,64(1)
  005C   59D1  0044          C      13,68(1)
```

```
0060   E6D0  0001          LA      13,1
0064   4330  8002          BFC     3,*+2
0068   24D0                LIS     13,0
006A   08DD                LR      13,13
006C   2114                BTFS    1,4
006E   C9D0  0001          CHI     13,1
0072   2323                BFFS    2,3
0074   E1B0  8054          SVC     11,*+84
0078   D2D1  004D          STB     13,77(1)
007C   4300  803C          BFC     0,*+60
0080   58A1  0040          L       10,64(1)
0084   59A1  0044          C       10,68(1)
0088   E6A0  0001          LA      10,1
008C   4330  8002          BFC     3,*+2
0090   24A0                LIS     10,0
0092   08AA                LR      10,10
0094   2114                BTFS    1,4
0096   C9A0  0001          CHI     10,1
009A   2323                BFFS    2,3
009C   E1B0  8028          SVC     11,*+40
00A0   D2A1  004C          STB     10,76(1)
00A4   58A1  0040          L       10,64(1)
00A8   59A1  0044          C       10,68(1)
00AC   E6A0  0001          LA      10,1
00B0   4330  8002          BFC     3,*+2
00B4   24A0                LIS     10,0
00B6   24A0                LIS     10,0
00B8   D2A1  004D          STB     10,77(1)
00BC   5071  001C          ST      7,28(1)
00C0   D101  0000          LM      0,0(1)
00C4   030F                BFCR    0,15
00C6   0000
00C8   140A  2711
00CC   140D  2711
```

* NO ERRORS DETECTED IN PASCAL PROGRAM OPTIMISA *