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 1

1 The nature of the problem 3

2 Optimisation 7

3 Analysis of the code 15

4 Approaches considered 18

5 Approaches adopted 22

6 Conclusions 31

References 34

Appendices

A Implementation of flowgraph and DAGs A-1

B Ad hoc patches B-1
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.
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.
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,

\[
i := 2 + 3
\]

develops into

\[
i := 5
\]

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

\[
i := 2
\]

\[
\cdots
\]

\[
i := i + 3
\]

develops into

\[
i := 5
\]

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

\[
i := i \times 2
\]

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,

\[
\text{for } i := 1 \text{ to } 1000 \text{ do }
\begin{align*}
& \text{begin} \\
& \quad t := 0 ; \\
& \quad \ldots \{ \text{ t unaltered } \} \\
& \text{end ;}
\end{align*}
\]

would be replaced by

\[
\begin{align*}
& t := 0 ; \\
& \text{for } i := 1 \text{ to } 1000 \text{ do } \\
& \quad \ldots
\end{align*}
\]
* 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.

```c
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,

\[
\begin{align*}
\text{LIS} & 10,0 & \{ \text{load } 0 \text{ into reg. } 10 \} \\
\text{ST} & 10,i & \{ i \text{ is an address} \} \\
\text{LIS} & 10,0 \\
\text{ST} & 10,j
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGNME</td>
<td>10</td>
</tr>
<tr>
<td>BLOCK</td>
<td>6</td>
</tr>
<tr>
<td>CALL</td>
<td>74</td>
</tr>
<tr>
<td>CALLNONS</td>
<td>58</td>
</tr>
<tr>
<td>CASESTAT</td>
<td>3</td>
</tr>
<tr>
<td>FACTOR</td>
<td>203</td>
</tr>
<tr>
<td>IFSTATEM</td>
<td>34</td>
</tr>
<tr>
<td>PROFILE</td>
<td>1</td>
</tr>
<tr>
<td>SIMPLEEX</td>
<td>194</td>
</tr>
<tr>
<td>TERM</td>
<td>195</td>
</tr>
<tr>
<td>WITHSTAT</td>
<td>3</td>
</tr>
</tbody>
</table>

Notes:

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

BALR (0)
STCR (0)
BFCR (5)
CLR (0)
OR (1)
XR (0)
LR (0.39)
CR (0)
AR (2)
SR (0)
SRLS (0)
SLLS (1)
MR (0)
JR (0)
BTBS (0)
STFS (26)
SFES (0)
SFFS (99)
LIS (29)
LCS (0)
AIS (0)
SIS (0)
STH (13)
BAL (185)
BTC (14)
BFC (43)
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)
AEL (0)
RTL (0)
REL (0)
LHL (0)
TBT (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
1. Fig 3.3 Distribution of procedures in modules

2. Fig 3.4 Distribution of procedure sizes
**Fig 3.5** Distribution of blocks/proc.

- $n = 103$
- $x_{\text{min}} = 1$
- $x_{\text{max}} = 297$
- $\bar{x} = 54$

**Fig 3.6** Distribution of block sizes

- $n = 500$
- $x_{\text{min}} = 2$
- $x_{\text{max}} = 1642$
- $\bar{x} = ?$

No: 66% $x \leq 10$ bytes
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.
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:

\[ A := B \text{ op } C \]

Note: \( A := B \) and \( A := \text{ op } C \) 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
\[ k := (i+1) \times j + (i+1) \]

(a) \[ T_1 := i + 1 \]
(b) \[ T_2 := T_1 \times j \]
(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
the actual code that would have to be dealt with will be of the form (using similar notation):

\[ a := b + c \]

\[ R10 := b \]
\[ R10 := R10 + c \]
\[ a := R10 \]

(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

\[
\begin{align*}
x & := a[i] \\
a[j] & := y \\
z & := a[i]
\end{align*}
\]

would be optimised to produce code equivalent to

\[
\begin{align*}
x & := a[i] \\
z & := x \\
a[j] & := y
\end{align*}
\]

which is incorrect if \( i = j \) and \( y \neq a[i] \). 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**
\[
\begin{align*}
p &:= \mathbf{^x} \\
\ldots
\end{align*}
\]

\[
\begin{align*}
x &:= i + 1 \\
p^\^ &:= x + 2
\end{align*}
\]

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.
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


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.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.

A-5
program optimiser(input, output, pasobj, pasincl, $pasmsgs, opfile, optout);

INCLUDE GLOBALS.PAS;

procedure optimise( procname : alpha ) ;

const
INCLUDE OPS.CST;
maxaddr = 3000;
stats = false;
list.code = false;
list.blocks = true;
maxblocks = 300;
maxn = 255; { maximum DAG node no. }
trace = false;

type

dagptr = ^dagnodes;
proc.name = packed array [1..20] of char;
opcode = 0..265;
opcode = record
  mnemonic : array [1..5] of char
  format : formats
  class : integer
end;
oper.kinds = (register, constant, variable, indexed, location, result);
instr.ptr = ^instruction;
blockptr = ^block;
reg.range = 0..15;
pred.rec = record
  this_pred : blockptr;
  next: ^pred.rec
end;
operand = record
  case kind : oper.kinds
  of
    register : (number : reg.range);
    constant : (value : integer);
    reg : reg.range;
    variable,
    indexed : (address : boolean;
    base : reg.range;
    index : reg.range;
    offset : integer);
    location : (ptr : instr.ptr);
    result : (result_kind : integer;
    dagnode : dagptr);
end ;
56  end;
57  instruction = record
58    id: integer;
59    op: opcode;
60    format: formats;
61    oper1, oper2: operand;
62    next: instrptr;
63    owner: blockptr;
64    size: integer;
65  end;
66
67  block = record
68    id: integer;
69    instr_head, instr_tail: instrptr;
70    true_successor, false_successor: blockptr;
71    pred_list_head: ^pred_rec;
72    size: integer;
73    ninstrs: integer;
74    case boolean of
75      true: (dummy: 0..255);
76      false: (flags: packed record
77          visited, \#2,\#3,\#4,\#5,\#6,\#7,\#8: boolean;
78        end);
79  end;
80
81  instr_rec = record
82    n, min, max, avg: integer;
83  end;
84
85  string = packed array [1..12] of char;
86
87  dagnodes = record (describing a DAG node)
88    killed: boolean;
89    num: 0..maxn;
90    op: opcode;
91    opclass: integer;
92    oper: array [1..2] of dagptr;
93    first_item: ^item;
94  end;
95
96  item = record (describes an object attached to a node)
97    init_val: boolean;
98    oper: operand;
99    next: ^item;
100  end;
101
102
103 var
104  stores: integer;
105  locs: array [0..maxaddr] of instrptr;
106  dfn: array [1..256] of blockptr;
107  pc: integer;
108  id: integer;
109  bid: integer;
110
optable : array [0..265] of opinfo;
finished : boolean;
traced : boolean;
hi, lo, flag, data, loc, num : integer;
save, previous : instrptr;
temp : integer;
twoto : array [0..31] of integer;
p, head : instrptr;
b, blockhead : blockptr;
pp : "pred_rec"
i, j, stat : integer;
filen : packed array [1..16] of char;
tail : packed array [1..8] of char;
nprocs, nblacs,
totproc, totbloc,
minproc, maxproc, aveproc,
minbloc, maxbloc, avebloc : integer;
newbloc, bloc : blockptr;
instrcnt,
blocksize : stat_rec;

procedure gettime( var tim : timerec );
begin

procedure update_stat( new_n : integer;
var stat : stat_rec );
begin

procedure report_stat( str : string;
var stat : stat_rec );
begin

procedure report_time( str : string );
var

begin
with stat do begin
n := n + 1;
if new_n < min then
min := new_n;
if new_n > max then
max := new_n;
end;
end;

procedure report_stat( str : string;
stat : stat_rec );
begin

end;

timenow := timerec;

begin { report_time }
gettime(timenow);
write(optout, str, ' CPU = ', (timenow.usercpu - timehold.usercpu):6);
write(optout, ' SVC = ', (timenow.oscpu - timehold.oscpu):6);
write(optout, ' WAIT = ', (timenow.wait - timehold.wait):6);
write(optout, ' ROLL = ', (timenow.rolled - timehold.rolled):6);
timehold := timenow;
end { report_time }

procedure opt_error( errno : integer ;
where : proc_name );

begin
if errno > 0 then
begin
  write(optout, '*** Error *** ');
  case errno of
  0: write(' Dummy error. ');
  else: write(' Error number ',errno:1);
      end;
  end
else
begin
  write(optout, '*** Warning *** ');
  case errno of
  0: write(' Dummy warning. ');
  else: write(optout, ' Warning number ',errno:1);
      end;
  end;
writeln(optout, ' [',where,' ]');
end;

procedure operout( oper : operand );

begin
with oper do 
  case kind of
  register : write(optout,number:1);
  constant : begin
    write(optout, ' =',value: 1);
    if reg <> 0 then
      write(optout,' (',reg:' )');
  end;
  variable : write(optout,offset:1, ' (',base:1,' )');
  indexed : write(optout,offset:1, ' (',index:1, ' , ',base:1, ' )');
  location : if ptr <> nil then
    write(optout, '#',ptr^.id:1)
  else
    write(optout, '#');
  result: write(optout, '@',dagnode^.num:1);
  else:write(optout,' ?');
end;
procedure initialise;

var
  i : integer;
  val : integer;

procedure init_stat(var stat : stat_rec);
begin { init_stat }
  with stat do
  begin
    n := 0;
    min := maxint - 1;
    max := -1;
    avg := 0;
  end;
end; { init_stat }

begin
  writeln(optout, 'Optimising ', procname, ' (',ic:1, ' bytes)');
  for i := 0 to maxaddr do
    locs[i] := nil;
  for i := 0 to 31 do
    begin
      twoto[i] := val;
      if val <> 31 then val := val + val;
    end;
  stores := 0;
  nblocs := 0;
  nprocs := 0;
  totbloc := 0;
  totproc := 0;
  minproc := maxint;
  minbloc := maxint;
  maxproc := -1;
  maxbloc := -1;
  tail := '.PRO,ERO';
  init_stat(blocksize);
  init_stat(instrcnt);
end;

procedure initoptable;

var
  i, status : integer;
  j : 0..9;
begin
  assignfl(opfile, 'OPFILE.SRO', status);
  if status = 0 then
    begin

    end;
reset(optfile);
for i := 0 to 265 do
  begin
    for j := 1 to 5 do
      read(optfile,optable[i].mnemonic[j]);
    readln(optfile,j,optable[i].class);
    optable[i].format := formats(j);
  end;
end;

procedure splitbyte( byte : integer );
  var hi, lo : integer ;
begin
  hi := byte div 16;
  lo := byte mod 16;
end;

function fetch( i : integer ) : integer ;
var
  locptr : codesptr ;
begin
  locptr := codesptr[i div codeperseg];
  fetch := ord(locptr.bytes[i mod codeperseg]);
end ; { fetch }

function nextbyte : integer ;
begin
  nextbyte := fetch(pc); 
  pc := pc + 1 ;
  if (pc = 18) and (pmv then 
    pc := pc + 24 ;
  saveA.size := saveA.size + 1 ;
end ; { nextbyte }

function signed( num : integer ;
    bits : integer ) : integer ;
begin
  if num >= twoto[bits - 1] then
    signed := num - twoto[bits]
  else
    signed := num ;
end ;

function value( size : integer ;
    sign : boolean ) : integer ;
var
magic : record
  case boolean of
    true : (word : integer);
    false: (bytes: packed array [1..4] of 0..255);  
  end;
  i : integer;

begin
  magic.word := 0;
  for i := (4 - size + 1) to 4 do
    magic.bytes[i] := nextbyte;
  if sign and (size <> 4) then
    value := signed(magic.word, size*8)
  else
    value := magic.word;
end;

procedure flowgraph;
var
  ip : instrptr;
  endoflist : boolean;
  rxxflag : boolean;

procedure add_predecessor( bloc2, bloc1 : blockptr );
var
  predp : ^pred_rec;
begin { add_predecessor }
  new(prep);
  predp^this_pred := bloc1;
  predp^next := bloc2^pred_list_head;
  bloc2^pred_list_head := predp;
end; { add_predecessor }

procedure swap_predecessors( bloc : blockptr;
  oldp, newp : blockptr );
var
  predp : ^pred_rec;
  swapped : boolean;
begin { swap_predecessors }
  if bloc <> nil then
    begin
      swapped := false;
      predp := bloc^pred_list_head;
      while not swapped and (predp <> nil) do
        begin
          if predp^this_pred = oldp then
            begin
              predp^this_pred := newp;
              swapped := true;
            end;
        end;
    end;
end; { swap_predecessors }
end
else
  predp := predp^next;
end;

end;  \{ swap_predecessors \}

procedure newblock(var bp : blockptr);
begin
  bsize : integer;
ip : instrptr;
preb : blockptr;
trusucc, falsucc : boolean;
tsucc, fsucc : blockptr);

bid := bid + 1;
new(bp);
with bp^do
  begin
    id := bid;
dummy := 0;
  end;

  size := bsize;
instrs := 0;
instr_head := ip;
  true_successor := tsucb;
false_successor := fsucb;
pred_list_head := nil;

if preb <> nil then
  begin
    add_predecessor(bp, preb);
    if trusucc then
      preb^true_successor := bp;
    if falsucc then
      preb^false_successor := bp;
  end;

begin
  pc := 0;
id := 1;
bid := 0;
previous := nil;
finished := false;
xxflag := false;
newblock(bloc,0,nil,nil,false,false,nil,nil);
while not finished do
  begin
    save := locs[pc div 2];
    new(ip);
    if (save <> nil) and (bloc^instr_head <> nil) then
      newblock(bloc,0,nil,bloc^true,true,nil,nil);
    while save <> nil do
      begin

save^ owner^ true_successor := bloc;
add_predecessor(bloc,save^ owner);
p := save^ oper2.ptr;
save^ oper2.ptr := ip;
save := p;
end;

ip^ id := ip;
id := id + 1;
save := ip;
ip^ size := 0;
ip^ next := nil;
ip^ owner := bloc;
bloc^ instr_tail := ip;
if bloc^ instr_head = nil then
bloc^ instr_head := ip;
locs[pc div 2J := ip;

with locs[pc div 2J do
begin
op := nextbyte;
format := optableOpt1.format;
if rrrxflag or (format = RRRX) then
format := RX;
case format of
RR : begin
splitbyte(nextbyte,hi,lo);
oper1.kind := register;
oper1.number := hi;
oper2.kind := register;
oper2.number := lo;
end;
SF : begin
splitbyte(nextbyte,hi,lo);
oper1.kind := register;
oper1.number := hi;
oper2.kind := constant;
oper2.value := lo;
oper2.reg := 0;
end;
RX : begin
splitbyte(nextbyte,hi,lo);
oper1.kind := register;
oper1.number := hi;
temp := nextbyte;
splitbyte(temp,flag,data);
if flag = 4 then ( = 0100 )
format := RX3
else
if flag = B then ( = 1xxx )
format := RX2
else
format := RX1;
case format of
RX1: begin
oper2.kind := variable;
oper2.base := lo;
OP.PAS 10

begin
  splitbyte<nextbyte,hi,lo>
  oper1.kind := register
  oper1.number := hi;
  oper2.kind := constant
  oper2.reg := lo;
  if format = RI1 then
    oper2.value := value< 2, true >
  else
    oper2.value := value< 4, true >
  end;

  RX2: begin
    oper2.kind := variable
    oper2.base := lo
    oper2.offset := signed(temp - 128) * 256 + nextbyte, 15
  end;

  RX3: begin
    oper2.kind := indexed
    oper2.base := lo
    oper2.index := data
    oper2.offset := nextbyte * 32768 + nextbyte * 256 + nextbyte
  end;

  RX1, RX2:
    begin
      splitbyte<nextbyte, hi, lo>
      oper1.kind := register
      oper1.number := hi
      oper2.kind := constant
      oper2.reg := lo
      if format = RX1 then
        oper2.value := value< 2, true >
      else
        oper2.value := value< 4, true >
      end;

      NULL,
      RXRX: ;
    end;

    blocA.size := blocA.size + size
    blocA.ninstrs := blocA.ninstrs + 1
    if not rxrxflag then
      case op of
      M, MH:
        if format = RX2 then
          if oper2.base = 0 then
            begin
              loc := pc + oper2.offset
              num := fetch(loc) * 256 + fetch(loc + 1)
              if (op = M) or (op = D) then
                num := num * 32768 + fetch(loc + 2) + 2
              oper2.kind := constant
              oper2.value := num
              oper2.reg := 0
            end;
          else
            begin
              ST, STH, STB:
                stores := stores + 1
            end
        else
          begin
            BTBS, BFSS
            BTC
            BFC:
              begin
                if format = SF then
                  if (op = BTFS) or (op = BFSS) 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 21];
      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 21];
                        locs[loc div 21] := save;
                        end;
                        newblock(bloc.O.nil.bloc,false,true,nil,nil);
                        end;
                  end
   end
   begin
   if (format = RX3) and (op <> SVCX) then
      if (oper2.base = 0) and (oper2.index = 0) then
         begin { relocatable item }
oper2.kind := location;
if oper2.offset <> 0 then
    oper2.ptr := locs[oper2.offset-2] div 2
else
    oper2.ptr := nil;
end;
if op <> LA then
    newblock(bloc, 0, nil, bloc, true, true, nil, nil);
end;
BFCR: if oper2.number = 15 then
begin
    bloc^.true_successor := nil;
    bloc^.false_successor := nil;
    finished := true;
end;
else
begin
    rrxrxfag := false;
splitbyte(op, hi, lo);
op := BLOP;
if (hi mod 4) <> 0 then
    previous^.op := lo + 260
else
    previous^.op := lo + 256;
end;
if hi in [4..7] then
begin
    temp := oper1.number;
    oper1.kind := constant;
    oper1.value := temp;
    oper1.reg := 0;
end;
if hi in [8..11] then
begin
    temp := previous^.oper1.number;
    previous^.oper1.kind := constant;
    previous^.oper1.value := temp;
    previous^.oper1.reg := 0;
end;
if hi in [12..15] then
begin
    temp := previous^.oper1.number;
    previous^.oper1.kind := constant;
    previous^.oper1.value := temp;
    previous^.oper1.reg := 0;
    temp := oper1.number;
    oper1.kind := constant;
    oper1.value := temp;
    oper1.reg := 0;
end;
if previous <> nil then
previous^.next := save;
procedure intra_block_optimise;

const
maxnodes = 256;  \{ maximum no. of nodes/DAO \}
maxn = 255;

type
opnodes = record
  occupied : boolean;
  opclass : integer;
  opn : array [1..2] of O..maxn;
  dagnode : dagptr;
end;

comp_link = record
  dagnode : dagptr;
  next : ^comp_link;
end;

objects = record \{describes a data item\}
  occupied : boolean;
  oper : operand;
  dagnode : dagptr;
end;

var
object : array [0..maxn] of objects;
nextn : 0..maxn;
first_component, last_component : ^comp_link;
base_of_heap : ^integer;
tempoper : operand;

procedure trip( procedure visit( dagptr, operand ) ;
oper : operand ) ;
{ 'visit's every node in the DAG by doing a post-order traversal of each of the components: i.e. the trees whose roots formed the comp_link list }

var
  cp : ^comp_link; \{ chains down list of roots \}
procedure traverse( root : dagptr ;
  procedure visit( dagptr, operand ) ;
  oper : operand ) ;
{ perform post-order traversal of tree belonging to 'root'. Each node is 'visit'ed }
begin { traverse }
    if root^.oper[1] <> nil then { traverse left subtree }
    traverse(root^.oper[1].visit.oper);
    if root^.oper[2] <> nil then { traverse right subtree }
    traverse(root^.oper[2].visit.oper);
    visit(root.oper); { process this node }
end; { traverse }

begin { trip }
    cp := first_component;
    while cp <> nil do { traverse tree for this component }
        begin
            traverse(cp^.dagnode.visit.oper);
            cp := cp^.next;
        end ; { trip }
    procedure kill( dagnode : dagptr ;
                    oper : operand ) ;
    { kill the indicated node. oper is a dummy arg }
        begin
            dagnode^.killed := true ;
        end ; { kill }
    procedure arraykill( dagnode : dagptr ;
                         oper : operand ) ;
    { kill node if is an INDX node with base address 'oper' }
        begin
            arraykill;
            if dagnode^.op = INDX then
                with dagnode^.oper[1]^ .first_item^.oper do
                    if (base = oper.base) and (offset = oper.offset) then
                        dagnode^.killed := true ;
        end ; { arraykill }
    procedure dump_node( dagnode : dagptr ;
                         oper : operand ) ;
    { Dump the contents of a DAG node. }
        var
            itemp : item ;
        begin { dump_node }
            with dagnode^ do
                begin
                    write(optout,'Node #',num:1:',',optable[op].mnemonic);
                    write(optout,'\n');
                    if oper[1] <> nil then
                        write(optout,oper[1]^ .num:1)
                    else
                        write(optout,'/-');
            end ; { dump_node }
function equal(oper1, oper2 : operand) : boolean;
begin
  equal := true;
  if oper1.kind = oper2.kind then
    case oper1.kind of
      register: if oper1.number <> oper2.number then equal := false;
      constant: begin
        if (oper1.value <> oper2.value) or
           (oper1.reg <> oper2.reg) then equal := false;
      end;
      variable: if (oper1.base <> oper2.base) or
                (oper1.offset <> oper2.offset) then equal := false;
      indexed: if (oper1.base <> oper2.base) or
                (oper1.offset <> oper2.offset) or
                (oper1.index <> oper2.index) then equal := false;
      result: if (oper1.result_kind <> oper2.result_kind) or
                (oper1.dagnode <> oper2.dagnode) then equal := false;
      else: equal := false;
    end;
  else
    equal := false;
  end;
end;

function object_ident(oper : operand) : O..maxn;
var
  id : O..maxn;
begin
  function hash_object(oper : operand) : integer;

  write(optout, ');
  if oper[2] <> nil then
    write(optout, oper[2].num:1);
  else
    write(optout, '-');
  writeln(optout);
  write(optout, ' attached items: ');
  itemp := first_item;
  while itemp <> nil do
    begin
      operout := itemp.oper;
      itemp := itemp.next;
      if itemp <> nil then
        write(optout, ' • ');
    end;
  writeln(optout);
end;

{ dump_node }
begin { hash_object }
with oper do
  case kind of
    register: hash_object := number;
    constant: hash_object := value mod (maxnodes - 16) + 16;
    variable: hash_object :=
      (base * ((maxnodes - 16) div 8) + offset) mod
      (maxnodes - 16) + 16;
    result: hash_object :=
      dagnode^num mod (maxnodes - 16) + 16;
    else: hash_object := maxnodes - 1;
  end;
end { hash_object }

function rehashop( n : integer ) : integer;
begin
  n := (n + 7) mod maxnodes;
end { rehash }

begin { object_ident }
id := hash_object(oper);
while not equal(oper honoring operator) and
  (object[id].occupied) do
  id := rehashop(id);
if not object[id].occupied then
  begin
    object[id].occupied := true;
    object[id].oper := oper;
    object[id].dagnode := nil;
    end;
object_ident := id;
end { object_ident }

function newitem( oper : operand ) : ^item;
begin
  var
    item : ^item;
  begin { newitem }
    if trace then
      begin
        write(out, 'Generating new item for ', oper);
        writeln(out);
      end;
    new(item);
    item^init_val := false;
    item^oper := oper;
    item^next := nil;
    newitem := item;
  end { newitem }
procedure move_item( oper : operand;
from_node, to_node : dagptr );

< Attach item for 'oper' to 'to_node' item list.
Delete from 'from_node' list if appropriate. >

var
item, lastitem : ^item;
found : boolean;
id : 0..maxn;

begin { move_item }

if trace then
begin
writeCoptout, 'Moving '); operout(oper);
writeCoptout, 'from '); operout(oper);
if from_node <> nil then writeCoptout, from_nodeA.num:1)
else writeCoptout, '-'
writeCoptout, 'to '); operout(oper);
if to_node <> nil then writeCoptout, to_nodeA.num:1)
else writeCoptout, '-'
writelnCoptout>
end ;

if from_node = nil then { new item needed }
item := newitem(oper)
else { find and remove current item,}
begin
item := from_nodeA.first_item;
lastitem := nil;
found := false;
while not found and (item <> nil) do
if not equal(item^, oper^, oper) then
begin
lastitem := item;
item := item^, next;
end
else
found := true;
if found then { remove from 'from_node' list }
if lastitem = nil then { remove head of list }
from_nodeA.first_item := item^, next
else
lastitem^, next := item^, next
else { *** error *** }
begin
opt_error(-1, 'move_item ');
item := newitem(oper);
end;
item^, next := nil;
procedure replace( a, b, c : dagptr ) ;
{ Replaces components a and b by component c. This
  currently only requires deletion of a and b. }
var
cp : ^comp_link ;
done : boolean ;
begin { replace }
cp := first_component ;
last_component := nil ;
while cp <> nil do
  begin
done := false ;
  if (cp^.dagnode = a) or (cp^.dagnode = b) then
    begin
      if last_component <> nil then
        last_component^.next := cp^.next
      else
        first_component := cp^.next ;
      done := true ;
    end ;
  if not done then
    last_component := cp ;
  cp := cp^.next ;
end ;
end ; { replace }

procedure build_dag( b : blockptr ) ;
var
opnode : array [0..maxnl] of opnodes ;
np : instrptr ;
tempoper : operand ;
function newnode( op : opcode ;
  opclass : integer ;
  son1, son2 : dagptr ) : dagptr ;
var
np : dagptr ;
cp : ^comp_link ;
begin { newnode }
  if trace then
    begin
      write(optout, 'Creating new node (',nextn:L ')labelled ',

end ;

```pascal
991  opTable[op].mnemonic: ''); (',opclass:=1,'));
992  write(opTable,' with sons ');
993  if son1 <> nil then
994  write(opTable,son1^.num:1);
995  else
996  write(opTable,'nil');
997  write(opTable,' and ');
998  if son2 <> nil then
999  write(opTable,son2^.num:1);
1000  else
1001  write(opTable,'nil');
1002  writeln(opTable);
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  new(cp);
1013  cp^.dagnode := np;
1014  cp^.next := nil;
1015  if last_component <> nil then
1016     last_component^.next := cp;
1017  else
1018     first_component := cp;
1019     last_component := cp;
1020  end;
1021  newnode := np;
1022  end; { newnode }
1023
1024  function intnode( np1, np2 : dagptr ;
1025                  op : opcode ;
1026                  opclass : integer ) : dagptr ;
1027  var
1028     n, nn1, nn2 : 0..maxn ;
1029  begin
1030     function inthash( i1, i2 : integer ) : integer ;
1031     begin
1032         inthash := ( i1 * i2 ) mod maxnodes ;
1033         end; { inthash }
1034     function rehash( n : integer ) : integer ;
1035     begin < rehash >
1036         rehash := ( n + 7 ) mod maxnodes ;
1037         end; < rehash >
1038     begin < intnode >
1039         if trace then
```
begin
write(optout, 'Find/create int. node for ',
  optable[op].mnemonic, ('opclass:1, ') with operands');
if np1 <> nil then
  write(optout, np1^.num:1)
else
  write(optout, 'nil');
write(optout, ' and ');  
if np2 <> nil then
  write(optout, np2^.num:1)
else
  write(optout, 'nil');
writeln(optout);
end;
n1 := np1^.num;
n2 := np2^.num
if np2 <> nil then
  nn2 := 1
else
  nn2 := 1;

n := inthash(nn1, nn2);
while (opnode[n] . occupied) and
  (opnode[n] . opn[1] = nn1) and
  (opnode[n] . opn[2] = nn2) and
  (opnode[n] . opclass = opclass) do
  n := rehash(n);
if opnode[n] . occupied then
  intnode := opnode[n] . dagnode
else
begin
  opnode[n] . occupied := true;
  opnode[n] . dagnode := newnode(op, opclass, np1, np2);
  intnode := opnode[n] . dagnode;
  opnode[n] . opn[1] := nn1;
  opnode[n] . opclass := opclass;
end;

function node( var oper : operand;
  ip : instrptr;
  createnew : boolean ) : dagptr;
var
  temp : operand;
  ni, n2, n3 : dagptr;
  id : 0..maxn;
begin
  node
  if trace then
    begin
      if createnew then
        write(optout, 'Find/create ')
      else
        write(optout, 'Find ');
write(optout, 'node for ');
end;
if createnew and (ip^.op = LA) and (ip^.format <> RX3) then 
begin  
n1 := newnode(LEAF, SPEC, nil, nil);  
moveme_item(oper, nil, n1);  
n2 := newnode(ADDR, SPEC, nil, nil);  
oper.kind := result;  
oper.result_kind := ADDR;  
oper.dagnode := n2;  
end  
else  
if (oper.offset = 0) and (oper.kind = variable) then  
begin  
temp.kind := register;  
temp.number := oper.base;  
n1 := node(temp, ip, true);  
n2 := node(temp, ip, true);  
oper.kind := result;  
oper.result_kind := DREF;  
oper.dagnode := n2;  
node := node(oper, ip, createnew);  
end  
else  
if (oper.kind = indexed) then  
begin  
temp.kind := variable;  
temp.base := oper.base;  
temp.offset := oper.offset;  
n1 := node(temp, ip, true);  
oper.kind := register;  
temp.number := oper.index;  
n2 := node(temp, ip, true);  
n3 := intnode(n1, n2, INDX, SPEC);  
oper.kind := result;  
oper.result_kind := INDX;  
oper.dagnode := n3;  
node := node(oper, ip, createnew);  
end  
else  
begin  
id := object_ident(oper);  
if createnew and (objectidl.dagnode = nil) then  
begin  
objectidl.dagnode :=  
newnode(LEAF, SPEC, nil, nil);  
moveme_item(oper, nil, objectidl.dagnode);  
end  
node := objectidl.dagnode;  
end;  
< node >

procedure add_instr_to_dag( ip : instrptr );
var
  a, b, c : dagptr;
opclass : integer;
tempoper : operand;
begin (add_instr_to_dag)
with ip^ do
  begin
    if trace then
      begin
        if opclass = LOAD then
          begin
            if trace then
              writeln(optout,'It is a LOAD instruction.');
            b := node(oper2, ip. true) ;
            a := node(oper1, ip. false) ;
            move_item(oper2,a,b);
          end
        else if opclass = STORE then
          begin
            if trace then
              writeln(optout,'It is a STORE instruction.');
            b := node(oper1, ip. true) ;
            a := node(oper2, ip. false) ;
            move_item(oper2,a,b);
            if oper2.result_kind = INDX then
              begin
                tempoper :=
                oper2.dagnode^.oper[1]^,first_item^.oper ;
                trip(arraykill,tempoper) ;
              end
          end
        else begin
          if trace then
            writeln(optout,'It is a dyadic instruction.');
          a := node(oper1, ip. true) ;
          b := node(oper2, ip. true) ;
        end
      end
c := intnode(a.b,op.opclass) ;
replace(a,b,c) ;
move_item(op1,a,c) ;
end ;
end ; { add_instr_to_dag }
procedure dag_init ;
var
i : 0..maxn ;
begin
first_component := nil ;
last_component := nil ;
for i := 0 to maxn do
begin
object[i].occupied := false ;
opnode[i].occupied := false ;
end ;
end ;
dag_init ;
nextn := 0 ; { first DAG node number }
ip := bA.instr_head ;
while ip <> bA.instr_tail do
begin
add_instr_to_dag(ip) ;
trip(dump_node,tempoper) ;
ip := ip^.next ;
end ; { build_dag }
procedure squeeze_dag( b : blockptr ) ;
begin
{ Optimise the DAG. }
end ;
procedure code_dag( b : blockptr ) ;
begin
{ Regenerate flowgraph node from DAG. }
end ;
begin
b := locs[O].owner ;
while b <> nil do
begin
mark(base_of_heap) ;
build_dag(b) ;
build_dag(b) ;
report_time('DAG build ') ;
trip(dump_node, tempoper);
squeeze_dag(b);
code_dag(b);
b := b'.false_successor;
end;
{ intra_block_optimise }

procedure depth_first;
var
dfi : 0..256;
begin
procedure search(b : blockptr);
procedure checkson(s : blockptr);
begin
if s <> nil then
if not s'.flags.visited then
search(s);
end;
begin
b'.flags.visited := true;
checkson(b'.true_successor);
checkson(b'.false_successor);
dfn[dfi] := b;
dfi := dfi - 1;
end;
begin
dfi := bid;
if locs[OJ <> nil then
search(locs[OJ'.owner);
end;
procedure print(instr : instruction);
begin
with instr do
begin
write(optout, id:5, ' '); 
write(optout, optable[op].mnemonic:6);
write(optout, ' '); 
operate(oper1);
write(optout, ' '); 
operate(oper2);
writeln(optout);
end;
end;
procedure blokid(bptr : blockptr);
begin
if bptr <> nil then
write(optout, 'B', bptr^.id:1)
else
write(optout, 'NIL')
end;

procedure analyse_graph;
var
  b : blockptr;
begin
  begin { analyse_graph }
    b := locs[0]^.owner;
    while b <> nil do
      begin
        with b^ do
          begin
            if list_blocks then
              begin
                write(optout, ' [B', id:1, ':']);
                blokid(true_successor);
                write(optout, ' : ]');
                blokid(false_successor);
                writeln(optout, ',*, instr_head^.id:1, ', size:1, ']');
              end;
              update_stat(size, blocksize);
              update_stat(ninstrs, instrcnt);
              end;
            b := b^.false_successor
          end;
      end;
  end; { analyse_graph }
begin
  initialise;
  initoptable;
  report_time('Compile ');
  if (ic div 2) <= maxaddr then
    begin
      flowgraph;
      report_time('Flowgraph ');
      analyse_graph;
      report_stat('block size ', blocksize);
      report_stat('instruction', instrcnt);
      intra_block_optimise;
      begin { depth_first }
        if list_code then
          begin
            p := locs[0];
            while p <> nil do
              begin
                print(p^);
                p := p^.next;
              end;
            end;
        end;
    end
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.
PROCEDURE INTEGRH(F1ATTRP, F2ATTRP: ATTRP; FOP: OPERATOR);
VAR OPRX, OPRR: INTEGER;
FOLDED: BOOLEAN;
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
      CASE FOP OF
        PLUS: IF F1ATTRP^.CVAL.IVAL = 0 THEN FOLDED := TRUE;
        MUL: IF F1ATTRP^.CVAL.IVAL = 1 THEN FOLDED := TRUE;
        ELSE: END;
      END;
      IF (F2ATTRP^.KIND = CST) AND (F2ATTRP^.CVAL. CKIND = INT) THEN
        CASE FOP OF
          PLUS: IF F2ATTRP^.CVAL.IVAL = 0 THEN BEGIN
            EXCATTR(F1ATTRP, F2ATTRP);
            FOLDED := TRUE;
          END;
          MUL: IF F2ATTRP^.CVAL.IVAL = 1 THEN BEGIN
            EXCATTR(F1ATTRP, F2ATTRP);
            FOLDED := TRUE;
          END;
          ELSE: END;
        END;
      END;
    END;
  ELSE
    BEGIN
      IF FOP IN (.PLUS, .MUL.) THEN
        CASE FOP OF
          PLUS: BEGIN OPRX := ZA; OPRR := ZA; LOAD(F1ATTRP, F2ATTRP); END;
          MINUS: BEGIN OPRX := ZS; OPRR := ZS; LOAD(F1ATTRP, F2ATTRP); END;
          MUL: BEGIN OPRX := ZM; OPRR := ZM; LOADVF(F1ATTRP, F2ATTRP); END;
          IDIV, IMOD: BEGIN OPRX := ZD; OPRR := ZD; LOADVF(F1ATTRP, F2ATTRP); END;
        END;
      END;
    END;
  END;
END.
WITH F1ATTRP.REXPR DO
BEGIN REGISTER(SUCC(RNO.)): = REGISTER(RNO.);
   REGISTER(RNO.).USED: = FALSE; RNO.:= SUCC(RNO.);
END;
IF FOP=IMOD THEN REGISTER(SUCC(F1ATTRP.REXPR.RNO)).USED := FALSE;
EXCAATTR(F1ATTRP,F2ATTRP);
END; /* IF NOT FOLDED */
END;
PROCEDURE BOOLARITH(F1ATTRP, F2ATTRP: ATTRP; FOP: OPERATOR);
VAR X: INTEGER;
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
        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
        ERROR(400) (* ILLEGAL BOOLEAN OP *)
    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
          ERROR(400) (* ILLEGAL BOOLEAN OP *)
      ELSE
        ERROR(400) (* ILLEGAL BOOLEAN OP *)
  ELSE
    BEGIN (* FORMER BOOLARITH *)
      IF F2ATTRP^.KIND=EXPR THEN EXCATTR(F1ATTRP, F2ATTRP);
      LOWD(F1ATTRP, F2ATTRP);
      IF FOP=ANDOP THEN X:=0 ELSE X:=ZD-ZN;
      OPERATION(F1ATTRP, F2ATTRP, ZN+X, ZNR+X);
      EXCATTR(F1ATTRP, F2ATTRP);
      END; (* OF STANDARD BOOLARITH *)
END;
PROCEDURE RELINT(F1ATTRP,F2ATTRP: ATTRP; FOP: OPERATOR);
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);
        LTEOP: F2ATTRP^CVAL.IVAL := ORD(F1ATTRP^CVAL.IVAL <= F2ATTRP^CVAL.IVAL);
        EGEOP: F2ATTRP^CVAL.IVAL := ORD(F1ATTRP^CVAL.IVAL >= F2ATTRP^CVAL.IVAL);
        EGTOP: F2ATTRP^CVAL.IVAL := ORD(F1ATTRP^CVAL.IVAL > F2ATTRP^CVAL.IVAL);
      END;
      F2ATTRP^TYPTR := BOOLPTR;
    END;
  ELSE
    ERRORC400 (* NOT AN INTEGER *)
  END;
ELSE
  BEGIN
    IF F2ATTRP^KIND=EXPR THEN
      BEGIN FOP:=DUALOP(FOP); EXCATTR(F1ATTRP,F2ATTRP);
    END;
    LOWD(F1ATTRP,F2ATTRP);
    OPERATION(F1ATTRP,F2ATTRP,ZC,ZCR);
    BOOLVALUE(REALREG(F1ATTRP^REXP.RNO.),BMASK(FOP.));
    F1ATTRP^TYPTR := BOOLPTR;
    EXCATTR(F1ATTRP,F2ATTRP);
    END;
  END;
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; (*#2673*)
BEGIN
IF REGSELECT THEN (* use specified register to *)
BEGIN
IF REGISTER[SELREGNO].USED THEN (* reload loop control variable *)
BEGIN
RWORK := SELREGNO ;
KILLREG(RWORK) ;
END
ELSE
ERROR(400)
ELSE
RWORK := SELREGNO
ELSE
CASE T OF
SINGLE: BEGIN
FOR RWORK := R10 TO R13 DO
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 ;
FOR RWORK := R10 TO R13 DO
IF NOT (USING(RWORK,FATTRP) OR SSINDEX[RWORK]) THEN BEGIN
SAVECRWDRK);
GOTO 1 END;
ERROR(400) ;
END;
FLOAT: BEGIN
FOR RWORK := FO TO F6 DO
IF NOT REGISTER(RWORK).USED THEN GOTO 1 ;
FOR RWORK := FO TO F6 DO
IF NOT USING(RWORK,FATTRP) THEN BEGIN
SAVECRWDRK);
GOTO 1 END;
ERROR(400);
IF NOT(REGISTER(R10).USED OR REGISTER(R11).USED) THEN RWORK := R10
ELSE IF NOT(REGISTER(R12).USED OR REGISTER(R13).USED) THEN RWORK := R12
ELSE IF NOT(USING(R10,FATTRP) OR USING(R11,FATTRP)) THEN
BEGIN SAVE(R10); SAVE(R11); RWORK := R10 END
ELSE IF NOT(USING(R12,FATTRP) OR USING(R13,FATTRP)) THEN
BEGIN SAVE(R12); SAVE(R13); RWORK := R12 END
ELSE ERROR(400);
END ; (* DOUBLE *)

1: RMAIN := REALREG(RWORK);

FUNCTION INREG( FATRP : ATTRP ) : BOOLEAN;
(* SETS RWORK AND RETURNS TRUE IF FATRP IS CURRENTLY HELD IN A REGISTER. FATRP MUST BE A FULLWORD CONSTANT OR VARIABLE *)
LABEL
1:
BEGIN (* INREG *)
INREG := FALSE;
WITH FATRP^ DO
IF ((KIND = CST) OR (KIND = VARBL)) AND (TYPR^ 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 = FATRP^ .KIND THEN
IF KIND = CST THEN
IF CVAL.CKIND = FATRP^.CVAL.CKIND THEN
IF CVAL.CKIND = INT THEN
IF CVAL.IVAL = FATRP^.CVAL.IVAL THEN
BEGIN
INREG := TRUE;
GOTO 1;
END
ELSE
ELSE
ELSE
ELSE
IF KIND = VARBL THEN
IF (ACCESS = DIRECT) AND (FATRP^.ACCESS = DIRECT) THEN
IF (VADRS = FATRP^.VADRS) AND (VLEVEL = FATRP^.VLEVEL) THEN
BEGIN
INREG := TRUE;
GOTO 1;
END;
(* INREG *)

BEGIN  (* LOWD *)  
  IF FIATTRP^.TYPR<>NIL
    THEN WITH FIATTRP^.TYPR^ DO 
  END  (*nz*)
BEGIN  (*nz*)
  IF (KIND<>EXPR) OR (REXPR.REGTEMP<>REGIST) THEN 
    BEGIN 
      (* CHECK REGISTERS *) 
      IF NOT INREG(FIATTRP) 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(FIATTRP,REGISTER(RWORK),REGCONT) ; 
                REGISTER(RWORK).NOTEXPR := TRUE ; 
                END 
            ELSE 
              REGISTER(RWORK).REGCONT := FIATTRP ; 
            END ; (* OF NOT INREG(FIATTRP) *)
          END 
          KIND:=EXPR, REXPR.REGTEMP:=REGIST, REXPR.RNO:=RWORK,
          REGISTER(RWORK).USED:=TRUE ; (*REGISTER(RWORK).REGCONT:=FIATTRP ; *)
          BEGIN REGISTER(. SUCC(RWORK)).USED:=TRUE;
          END
          REGFORM := RFORMTMP;
          END;
    END;
  END;
END;
END;
BEGIN
{constant folding}

if debug and (max > 100) then

else

end.

GENERATED CODE LISTING:

0000 D00B 0000 STM 0.0(8)
0004 001B LR 1.8
0006 F000 0050 AL 8.80
000C 41F0 4000 0000 BAL 15.0
0012 5024 4911 4904 2020
001A 3036 2F31 302F 3833
0022 3132 3A31 303A 3331
002A 2401 LIS 10.0
002C 2400 ST 10.64(1)
0030 2400 LIS 10.0
0032 2400 ST 10.68(1)
0036 2400 LHI 10.99
003A 2400 SIS 10.1
003C 2400 ST 10.72(1)
0040 2400 LHI 10.99
0044 2400 CHI 10.100
0048 2400 LA 10.1
004C 2400 BTC 2.*+2
0050 2400 LIS 10.0
0052 2400 NHI 10.1
005A 2400 LSI 3.*+12
005C 2400 LSI 10.64(1)
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0124</td>
<td>048A</td>
<td>NR</td>
<td>11.10</td>
</tr>
<tr>
<td>0126</td>
<td>2114</td>
<td>BTFS</td>
<td>1.4</td>
</tr>
<tr>
<td>0128</td>
<td>C990</td>
<td>CHI</td>
<td>11.1</td>
</tr>
<tr>
<td>012C</td>
<td>2323</td>
<td>BFFS</td>
<td>2.3</td>
</tr>
<tr>
<td>012E</td>
<td>E1B0</td>
<td>SVC</td>
<td>11.**+14</td>
</tr>
<tr>
<td>0132</td>
<td>D2B1</td>
<td>STB</td>
<td>11.77(1)</td>
</tr>
<tr>
<td>0136</td>
<td>5071</td>
<td>ST</td>
<td>7.28(1)</td>
</tr>
<tr>
<td>013A</td>
<td>D101</td>
<td>LM</td>
<td>0.0(1)</td>
</tr>
<tr>
<td>013E</td>
<td>030F</td>
<td>BFCR</td>
<td>0.15</td>
</tr>
<tr>
<td>0140</td>
<td>1408</td>
<td>2711</td>
<td></td>
</tr>
<tr>
<td>0144</td>
<td>0000</td>
<td>0000</td>
<td></td>
</tr>
</tbody>
</table>

* NO ERRORS DETECTED IN PASCAL PROGRAM OPTIMISA *
program optimisation;

const
max = 99;
debug = true;

var
i, j, k : integer;
a, b, c : boolean;

i := 0;
j := 0;

k := max - 1;

if debug and (max > 100) then
begin
  a := (i = j) or (max > 50);
b := (i = j) or (max > 99);
end
else
begin
  a := (i = j) and (max > 50);
b := (i = j) and (max > 99);
end;

GENERATED CODE LISTING:

0000 DOOB 0000 STM 0.0(8)
0004 0B1B LR 1.8
0006 FA00 0000 0050 AI 8.80
000C 41F0 4000 0000 BAL 15.0
0012 502E 4041 49FE 2020
001A 3036 2F31 302F 3833
0022 3132 3A01 313A 3335
002A 24A0 LIS 10.0
002C 50A1 0040 ST 10.64(1)
0030 50A1 0044 ST 10.68(1)
0034 C880 0062 LHI 11.98
0038 5061 0048 ST 11.72(1)
003C 4330 8040 BFC 3.564
0040 59C1 0040 L 12.64(1)
0044 59C1 0044 C 12.68(1)
0048 E560 0001 LA 12.1
004C 4330 8002 BFC 3.562
0050 24C0 LTI 12.0
0052 24C1 LIS 12.1
0054 22C1 004C STB 12.76(1)
0058 59D1 0040 L 13.64(1)
005C 59D1 0044 C 13.68(1)
* NO ERRORS DETECTED IN PASCAL PROGRAM OPTIMISA *