EXPERIENCE WITH A PATTERN-MATCHING CODE GENERATOR

by

M A Mulders[1], D A Sewry[2] and W R van Biljon
National Research Institute for Mathematical Sciences, CSIR

1. Central Data Systems, Prime Park, Bedford View, Johannesburg
2. Department of Computer Science, Rhodes University, Grahamstown.

ABSTRACT

An implementation of a Graham-Glanville type pattern-matching code generator for the NRIMS systems programming language SCRAP is discussed. The simple heuristic algorithm on which the pattern matcher is based is presented and compared to more exhaustive and complex algorithms implemented elsewhere. Finally, optimal instruction selection as a method of code improvement is discussed. The pattern matcher has been implemented for the Perkin Elmer 3200 and the Motorola 68000 instruction sets.

1. INTRODUCTION

Code optimization represents an area of major interest in research on languages and code generation. One optimization technique that has been the subject of much work is that of optimal instruction selection. The Graham-Glanville approach of table-driven pattern-matching code generation was first introduced as a method of providing a machine-independent code generation algorithm intended to facilitate the construction of retargetable compilers [4]. Recently, this technique has been advanced as having significant advantages in the area of optimal instruction selection and in the specification and implementation of efficient code generators [1,2,3].

This paper details the experiences of the project team involved in the implementation of a pattern matching code generator for the systems programming language SCRAP (Systems ConstRuction and Applications Programming language) [5] developed and used at NRIMS. The compiler is a bootstrapped implementation (it is written in SCRAP). Two code generators have been implemented for a Perkin-Elmer 3200 series minicomputer and a Motorola MC68000 microprocessor.

Extensive use of the language in a CPU-intensive graphics environment dictated the main design goals of the code generator:

(a) the code generator should be simple to implement, port and maintain;

(b) it should generate efficient code.

2. THE STRUCTURE OF THE CODE GENERATOR

The code generator is divided into three main phases (Figure 1).

2.1 The Pre-Processor

The input to the code generator from the syntax stage is in the form of triples, for example :

```
DATA d;          (* d defines the base address of *)

  INT  a,b;      (* variables in the data block *)

ENDD d;

...

a:=b+4
```

emits, from the semantic analyzer, the triples:

```
1. add address      ADDRESS, (d) OFFSET (a)
2. add address      ADDRESS, (d) OFFSET (b)
```

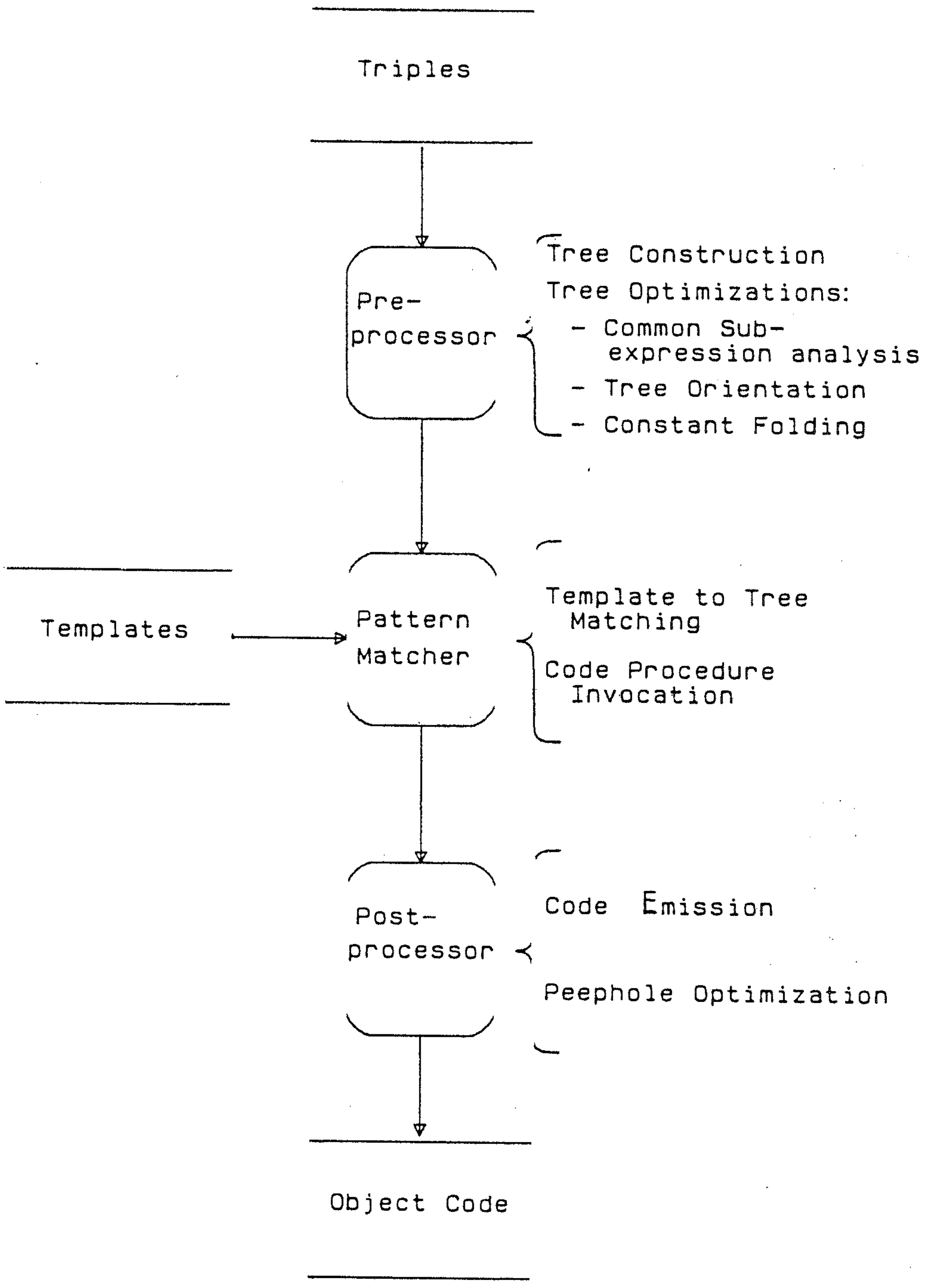

Figure 1 : The Structure of the Code Generator

```
3. value at        (2)
4. add             (3), constant (#4)
5. store at        (4), (1)
```

The triples are built into binary expression trees by the pre-processor (Figure 2).

Each node in the expression tree has the following structure:

node type	- describes the operation or operand
type	- describes the type of the operand or the result.
use_list	- list of tree numbers of trees using this node.
use_count	- number of times that the value represented by this node is referenced.
register	- number of the register that holds this value.
left operand	- pointer to left child.
right operand	- pointer to right child.

A sequence of triples for a whole procedure will result in a forest of trees. The expression trees in the forest are connected at their roots by links to the roots of the previous and next trees. The trees are numbered sequentially as they are built. The left and right operand pointers of a node point to its left and right children. As a result of some simple common sub-expression analysis done in the pre-processor, some pointers may point to nodes in other trees instead - some nodes are then shared between several trees, with the result that the data structure is in fact a directed, acyclic graph (DAG) [6], although it is useful to think of it in terms of trees (Figure 3). The use-count of each node reflects the number of times it is referenced. For common sub-expressions this value will thus be greater than one. The use_list of a node is a list of the numbers of all trees that reference the node.

At this stage advance global register booking is done. Register numbers are booked for nodes with large use counts and for nodes that have been heuristically identified as being "high activity" values. These include FOR loop induction variables and USING statement (analogous to the Pascal WITH) base addresses. Booking a register consists of setting the node's register field to a register number and adding the node's tree number to a list of that type kept for each register. Register allocation is described more fully in a separate paper [7].

An important operation, performed in the pre-processor is that of tree orientation/modification. This will be mentioned again.

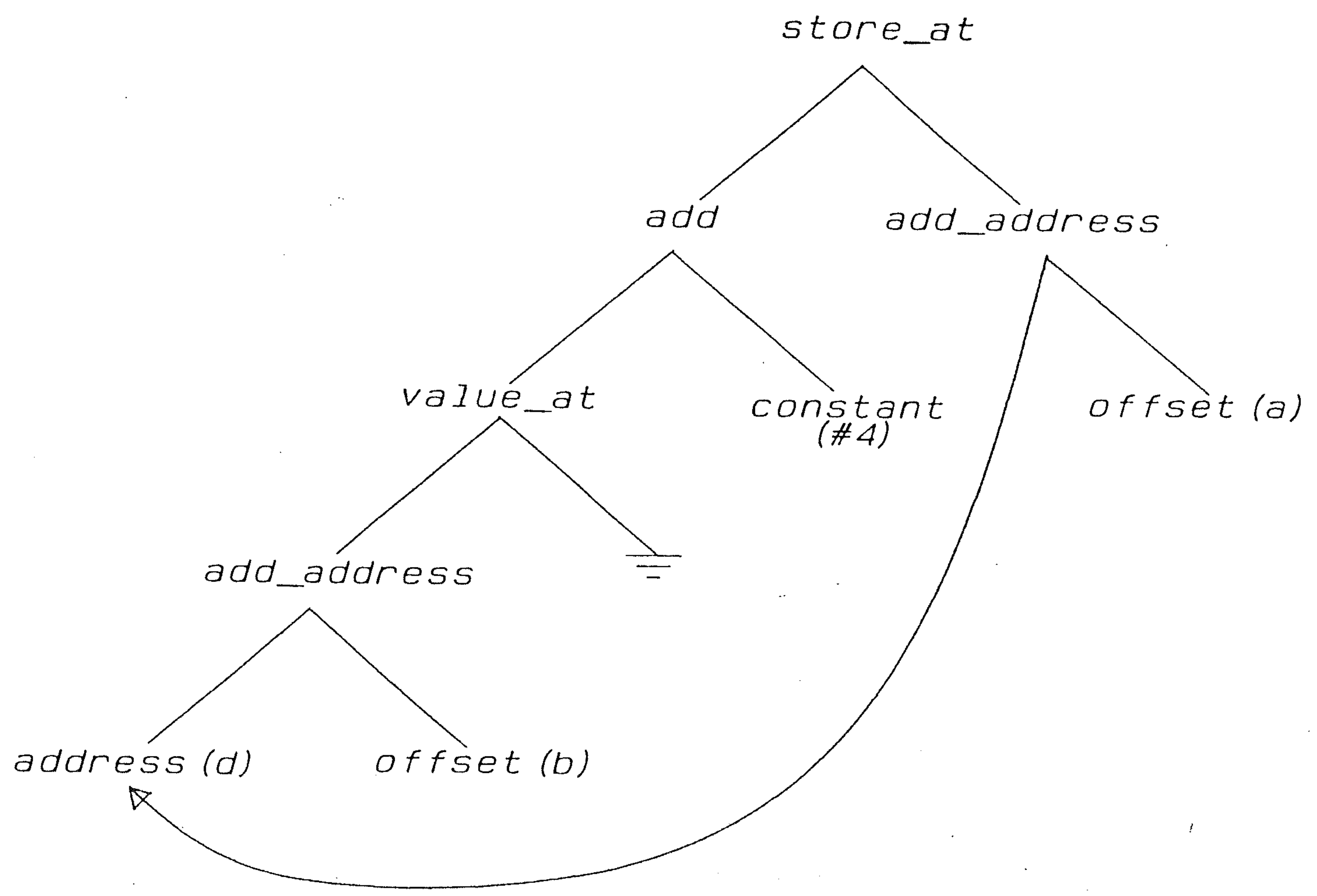

Figure 2 : An Expression Tree

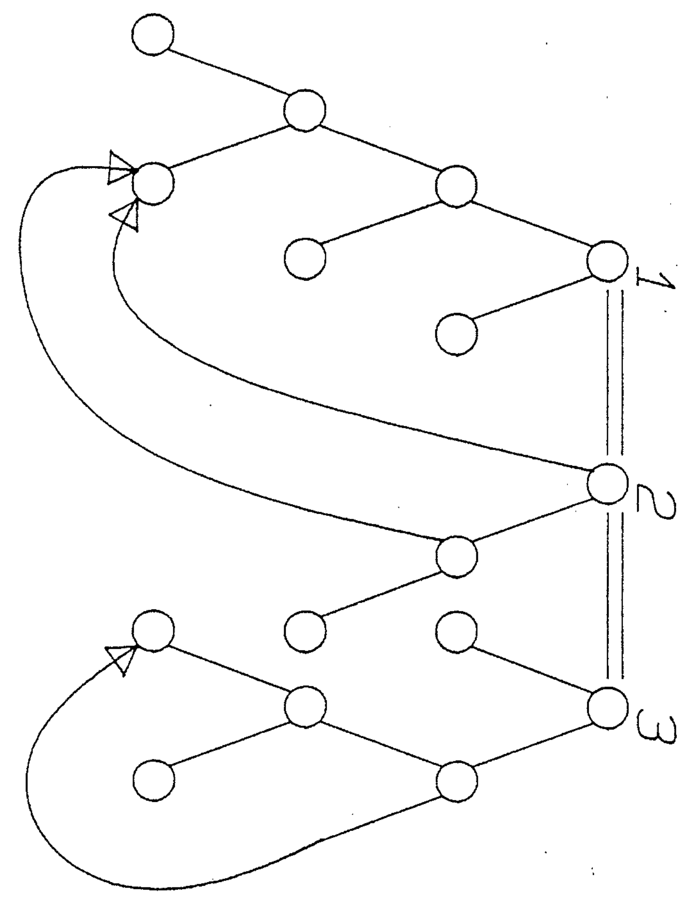

Figure 3 : A Forest of Trees

2.2. The Pattern Matcher

The next phase is that of target instruction selection. For every target machine instruction, an instruction template is created. These templates are stored in the form of binary trees. Each template consists of :

- the type names of each node, stored in post order tree;
- a pointer to a procedure that must be called to generate code for this template when it is matched to an input subtree;
- the location of a template result. This is the location of the result of the operation specified by the template. Result locations fall into three main types :

1. Register (the computation result is left in a machine register).
2. Const (the result is a number).
3. Null result.

The root of each template is an operator (for example, plus, assignment, procedure call). Templates are grouped into lists according to their root operator. Each list will contain all the templates with a particular root operator (and result type).

Finally the lists are grouped into tables according to their result type. A table is set up for each of the possible template result types. All lists with the same result type will be referenced in a table of that type. Lists are placed in the table in the order of their (unique) root operator type.

Matching algorithm:

The forest of binary trees, as constructed by the pre-processor is now matched against the templates, one tree (statement) at a time. The algorithm for matching is as follows:

(a) Choose one of the result tables according to the result type required. For a root node this would be the main index table (null-result).

(b) Use the root operator of the subtree as an index into the table chosen in (a). This will result in a pointer to a list of templates, all of which have that operator as their root.

(c) A match is then attempted against each template in the list, accessing the templates in the order in which they were stored.

(d) Matching proceeds as follows: Take the root node of the input subtree and attempt to match its operands with the operands of the current template. Matching is done first on the left operand and then on the right operand.

(e) If the operands match, replace the subtree with the template result and invoke the template's procedure to generate the corresponding code.

(f) If the left operand does not match, view it as the root node of a new subtree. The template's left operand then becomes the result type required. We then consult the appropriate result table (using the new root operator as an index) for a new template list. If an appropriate result table exists and if a non-null list is found at that index, we proceed (recursively) back to step (c).

(g) If no appropriate result tables exists,or no non-null list is found we have failed to match the current template. This template is then discarded in favour of the next in the current list. If no more templates exist in the current list, we are forced to halt and report a "failure to generate code for this expression" message.

(h) The same process is then repeated for the right operand.

Note that the algorithm is one of first-fit and that no cost function or other suitability guideline is used in the matching process. The crux of the algorithm is the ordering of the templates with the template lists. Here a simple heuristic rule is followed :

Place the templates in descending order of template (tree) size.

The rationale is: the bigger the template that is matched against a given subtree, the more work can be accomplished by a single (corresponding) instruction. This translates into fewer machine instructions per source statement. Since no cost function or desirability expression is required for each instruction (as in [3]), template preparation time is reduced. The matching algorithm is also easier to design and implement.

2.3. The Post-Processor

The post-processor is concerned mainly with the task of code emission. The code, together with directives required by the linker, is emitted to an object file with a prescribed format. The last phase of code optimization is also done here. The code is examined by a simple and crude peephole optimizer. Redundant instructions are removed, and some complex instructions are replaced by their equivalent simpler instructions. For example: for the MC68000 the instruction:

CMPI #0,Dn is replaced by
TST Dn
to compare data register n with zero.

3. OBSERVATIONS AND EXPERIENCE

The quality of instruction selection by this rather simple heuristic pattern matcher was surprising. In almost all cases the template lists could be ordered such that the largest possible subtree would be matched to a suitable target machine instruction. Furthermore, the premise of "fewer instructions - better code" in general proved to be valid for both the Perkin-Elmer 3200 and MC68000 series.

A comparison of code generated with that of other compilers on the Perkin-Elmer minicomputer showed the code to compare favourably with far more complex compilers. The Dhrystone benchmark [8] was coded, and recorded results are:

Perkin-Elmer Pascal compiler	:	888 dhrystones/second
Perkin-Elmer Optimizing Pascal Compiler (10 pass)	:	1379
Scrap III compiler	:	714
Scrap IV compiler	:	721
C	:	625

The Scrap III compiler utilized a primitive code generator based on simple macro-expansion of triples.

The results of the simple heuristic pattern matcher compare very favourably with those of a number of the pattern matchers utilizing more complex rule algorithms. When we compared our generated code to that of Scheuneman (as given in [3]) for equivalent source constructs, the code was virtually identical. Scheunemans algorithm is however, one of exhaustive best-fit (using instruction cost functions).

4. PROBLEMS

The development of the pattern matcher was not without its difficulties, however, and the implementation is not without its shortcomings:

1. The static representation of the templates, lists and tables as initialized constant blocks posed difficulties. Maintenance of the compiler proved to be non-trivial; as the number of templates was increased, the recursive nature of the matching algorithm meant that tracing the flow of control became increasingly arduous. Furthermore, the learning curve for new project members proved to be rather more formidable than we had hoped.

2. Storage requirements for the code generator inevitably resulted in trade-offs. To ensure fast access all templates were kept in memory during compilation. Storage for the trees was allocated on a node-for-node basis as they were built. As trees are reduced by the pattern matcher, this storage becomes available for re-use. However, this piecemeal approach to storage allocation/disposal proved to be too expensive in terms of garbage collection. A mark/release mechanism was implemented instead - storage levels were marked before code generation for a source language block; storage were allocated above that mark as required and released back to that mark (only) after generation for the block was completed. This complicated storage allocation for other structures, which are required throughout code generation (for example, linked lists of linker directives).

3. Storage requirements had repercussions within the matching algorithm itself. It proved too costly to keep a record of input tree configurations at each stage of the matching process. This had the consequent implication that all reductions made to an input tree were irreversible. An attempt to match a tree can result in a reduction in an offspring subtree. If the match of the larger tree later failed, it was impossible to undo the work done on parts of it. Thus an additional constraint was placed on the ordering of templates in the template list. Templates of near equivalent size had to be ordered such that during the match of template n, no part of the input subtree contained in template n+1 was reduced before the entire subtree was matched. This added to the complexity of template preparation.

4. Generality among templates in the pattern matcher was difficult to achieve. In a sense it became a matter of matching a large number of 'special cases'. Ease of maintenance and program clarity versus code quality were trade-offs as an ever growing number of templates were constructed to cover most source statement configurations. For example, the MC68000 addressing mode [d,An,Am,] is found in a large number of instructions and also corresponds to a large number of possible input subtrees (Figure 4). To create a separate template for each instruction type for this one addressing mode would be laborious and would result in longer compilation times as the matcher scan times increased. It was simpler to allow the pre-processor to orientate all input trees containing this addressing subtree into a prescribed configuration (for example, tree 1 in Figure 4) before matching was done.

5. CONCLUSION - Optimal instruction selection versus other forms of code optimization.

The project afforded an opportunity to evaluate instruction selection as an optimization technique. Often the results were disappointing:
(a) Consider the MC68000 - one of its most complex instructions is

```
MOVE [d1, An, Am] , [d2, Ax, Ay]      (cycle time = 32)
```

which could be matched in its entirety to a source language statement of the type

```
ARRAY [0:10] OF BYTE a,b;
INT i,j;

....

a[i] := b[j];
```

The 68000 MOVE instruction allows more than eight addressing modes for both the source and destination operands. Therefore, to enable the pattern matcher to select an instruction of this type would precipitate the construction of a template list of more than 60 items. Once again, this is both laborious and leads to significantly higher matcher scan

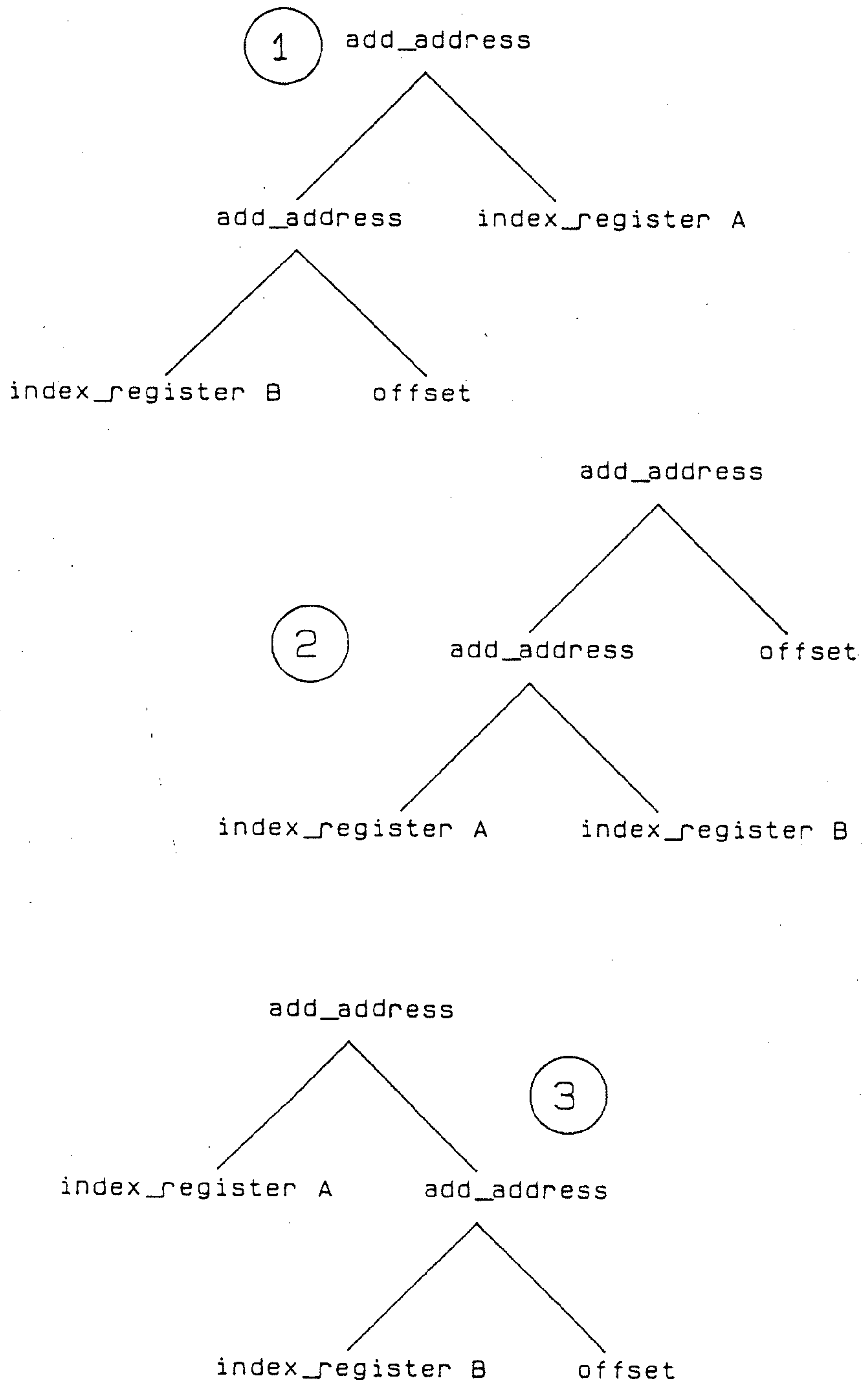

Figure 4 : Some configurations of the (d, An, Am) addressing mode.

times. The alternative is to allow the match to take place in two stages, using a temporary register, that is

```
        MOVE [dl, An,Am] , Da      (Cycle time = 18)
        MOVE Da, [dl, An, Am]      (Cycle time = 18)
```

A comparison of cycle times reveals that we gain a mere 4 cycles out of 36 for the latter sequence, an improvement of only 11%.

(b) Certain instruction selection optimizations are difficult or impossible to effect in our simple pattern matcher. Some machines provide a choice of instructions to implement a source construct. Code quality often depends critically on the selection of the most appropriate instruction. For example; an "increment" can be used instead of an "add 1", that is

```
 INC Dx    versus    ADD #1, Dx        {add 1 to data register x}
```

In order to add a specific template to allow the increment instruction to be selected, we must ensure that our matcher recognizes both a distinct register type (a data register) and distinct constant range (number 1). This would add to the specific nature of the templates and consequently, the matching algorithm. It would further frustrate our attempts to generalize the matching process and to reduce the number of specific templates. In our compiler such optimizations proved easier to implement in the post-processor (peephole) optimizer. The instruction is generated as an ADD by the pattern matcher and later optimized in the post-processor. Other examples include the TST instruction described earlier.

(c) Simple manipulations performed in the pre-processor, such as changing the tree orientation and constant folding (particularly type conversions on constants), also significantly reduced the number of templates and matching times.

It can therefore be concluded that instruction selection represents but a small fraction of the optimization possible in a compiler of this size and perhaps, not the most useful. Furthermore, the current trend to pipelined architectures largely invalidates the "fewer instruction - better code" premise.

Also, the impact of the technique is lessened for RISC architectures where instructions tend to the same order of complexity (or tree size).

6. REFERENCES

1. P.J. Hatcher and T.W. Christopher, 'High Quality Code Generation via Bottom-Up Tree Pattern Matching' Proceedings of the 13th CM Symposium on the Principles of Programming Languages, 119-130 (1986).

2. A.V. Aho and M. Ganapathi, 'Efficient Tree Pattern Matching: An Aid to Code Generation' Proceedings of the 13th Acm Symposium on the Principles of Programming Languages, 334-340 (1986).

3. A. Scheuneman, 'A machine independent approach to automatic code generation' M.Sc. Thesis, School of Computer Science, McGill University, (1982).

4. S.L. Graham and R.S. Glanville, 'The use of a machine description for compiler code generation' Information Technology, North-Holland Publishing Company, 509-514 (1978).

5. M.H. van Rooyen, 'Die Definisie en Implementasie van die taal Scrap' Quaestiones Informaticae 2,2 29-35 (May 1983).

6. A.V. Aho, R. Sethi, J.D. Ullman, 'Compiler Principles, Techniques, and Tools', Addison-Wesley, 1986.

7. W.R. van Biljon, D.A. Sewry and M.A. Mulders, 'Register allocation in a pattern matching code generator' TWISK 492, CSIR, Pretoria, (1986).

8. R.P. Weicker 'Dhrystone: A synthetic Systems Programming benchmark', Comm ACM, Vol 27, no 10, 1984.

9. R.N. Horspool and A. Scheunemann, 'Automating the Selection of Code Templates' Software - Practice and Experience, 15,5 503-514, (May 1985).