# Fast Peephole Optimization Techniques

B. J. McKenzie

University of Canterbury, Christchurch, New Zealand

Postal Address:

Dr. B.J.McKenzie,
Department of Computer Science,
University of Canterbury,
Private Bag, Christchurch,
New Zealand.

Electronic mail address (uucp):

...{mcvax,watmath, vuwcomp}!cantuar!bruce

# Fast Peephole Optimization Techniques

B. J. McKenzie

University of Canterbury, Christchurch, New Zealand

SUMMARY

Techniques for increasing the throughput of a peephole optimizer for intermediate code are presented. An analysis of the optimations to be performed enables an efficient matching and replacement algorithm to be found which minimizes rescanning after a successful replacement. The optimizer uses procedural interfaces; both for the input from the front end phase of the compiler and for the output to the back end. The result is a library module which may optionally be loaded with various other phases of the compiler to provide a flexible range of options regarding compiler size, quality of generated code and compilation speed.

KEY WORDS: intermediate code, peephole optimization,
deterministic finite automata, procedural interfaces, compilers

## INTRODUCTION

It has become increasingly popular in the area of compiler construction to use some form of intermediate code to communicate between the front end (source language specific) and back end (target machine specific) phases of a compiler. Such a design results in a clean interface between these two phases of the compiler as well as expediting portability. It is common to share the same intermediate language for a number of different source languages [1], or a number of different target machine languages [2], or even both [3, 4]. Such a sharing allows tasks which are common to all front and back ends to be isolated in modules that can be shared by all without duplication of effort. Machine independent optimization [5] is an example of such a common task and this paper focuses on the peephole optimizer in the Amsterdam Compiler Kit (ACK) [6].

The ACK is a toolkit for the construction of compilers and cross-compilers for Algol-like languages (e.g. Pascal, C, Occam, Basic, Modula-2), and byte-addressable target machines (e.g. 680x0, pdp, vax, 8086). It uses a stack based intermediate language called EM [4], which is encoded in a binary file for communication between phases of the compiler. Generation of these encoded binary files is simplified by the provision of a library of routines, (one for each distinct form of an EM instruction,) that control the encoding and output of the binary EM code. Similarly a library for reading an encoded EM file provides a routine that reads and decodes the next EM instruction into a C structure variable. To ease debugging of new tools, versions of these libraries which deal with human readable text files are also available.

## EXISTING PEEPHOLE OPTIMIZER

The design of the existing peephole optimizer is described in some detail in [7]. This optmizer reads and stores the EM instructions corresponding to a complete procedure and then breaks this up into a series of basic blocks delimited by labels. For each such block it makes repeated passes searching for optimizations until a complete pass makes no replacements. Finally the blocks are output. There are currently 580 optimizations described by a table of templates of which the following is a typical example:

```
loc  adi w  loc  adi w   →   loc $1+$3  adi w
```

This example asserts that the sequence of EM instructions, `loc adi loc adi` (where `loc` is *load constant* and `adi` is *add integer*), when both the `adi` instructions have arguments with value equal to the word size (`w`), can be replaced by a single `loc` instruction which has an argument with value the sum of the two replaced `loc` instructions' arguments and a single `adi` instruction. The search for possible optimizations thus includes both the search through the block of EM instructions for a pattern of EM instructions corresponding to the left hand side of the template as well as ensuring that any restrictions on and between arguments are satisfied.

To search for potential optimizations the next three EM instruction opcodes are hashed and the resulting value used to index a chained bucket of all possible templates that could match. These are searched in turn attempting to match the LHS instructions of the template. Upon a complete match any associated conditions (which have been encoded in tables) are checked. If this is also successful any expressions required by the RHS are evaluated (again using tables) and the LHS instructions replaced by the RHS instructions. Scanning for subsequent optimizations continues from the beginning of the replacement.

Table 1 presents timings for the phases using the C language front end and sun3 back end while compiling the source of three Unix commands. All times are the sum of the user and system cpu times under SunOS 3.5 on a Sun 3/60 and are averaged over 5 runs. The statistics regarding the source files are after pre-processing and removal of empty lines. It can be seen that the peephole optimizer consumes a significant proportion of the processing resources; perhaps surprisingly similar to that consumed by the front end.

| program source | ar.c | ctags.c | cu.c |
|---|---|---|---|
| no. of lines | 982 | 965 | 1128 |
| no. of words | 3111 | 3149 | 3803 |
| no. of EM instr. before optimization | 3591 | 5611 | 4619 |
| no. of EM instr. after optimization | 2606 | 3818 | 3413 |
| time for C front end | 1.60 | 1.79 | 2.06 |
| time for optimizer | 1.60 | 2.51 | 2.01 |
| time for sun3 back end | 3.23 | 5.96 | 4.72 |
| time for assembler | 6.64 | 9.07 | 8.34 |
| time for loader | 0.85 | 0.81 | 0.84 |

Table 1. Statistics regarding phases of ACK pipeline on Sun 3/60

Figure 1 graphs gives the number of passes required by the peephole optimizer for each basic block summed over all three sources files. It also shows the frequency distribution of the sizes of the basic blocks both before and after optimization.

From this it can be seen that there is a significant reduction in the number of EM instructions, but also that it requires a number of passes through the block to achieve this reduction. The last pass through each block is essentially wasted as it achieves no improvement in the EM code. This effect becomes more pronounced as the number of optimization patterns is increased as shown by figure 2 where the optimization time is graphed against the number of optimized patterns used. This is a result of the increased possibility that a replacement will be made requiring a further pass through the block and also from an increase in the length of the hash collision chains.

## RATIONALE FOR THE NEW OPTIMIZER

The analysis of the statistics presented in the previous section suggests that an algorithm that can both reduce the overhead in detecting optimizations and reduce the requirement of repeatedly re-scanning the basic blocks should increase the throughput of the optimizer.

One way to reduce the re-scan would be to limit the re-scan by buffering the output and after each successful replacement calculating how many instructions are required to be backed up before continuing. For example, a primitive method of doing this would be to back-up L–1 instructions[1] where L is the length of the longest LHS of the patterns.

In practice this primitive estimate can be improved for optimizations with non-null replacements as there is some context information available to limit the back-up. Suppose the pattern:

$$a_1 a_2 \cdots a_m \rightarrow b_1 \cdots b_n$$

has just been successfully matched and replaced. There are four possible forms for LHS's that require back-up; these LHS's and the amount of back-up they require (including the replacement itself) are:

| | LHS pattern | | back-up required |
|---|---|---|---|
| a) | $c_1 \cdots c_i\ b_1 \cdots b_n\ d_1 \cdots d_j$ | $0 \le i\ 0 \le j$ | n+i |
| b) | $c_1 \cdots c_i\ b_1 \cdots b_j$ | $0 \le i\ 1 \le j < n$ | n+i |
| c) | $b_i \cdots b_n\ c_1 \cdots c_j$ | $1 < i \le n\ 0 \le j$ | n-i+1 |
| d) | $b_i \cdots b_j$ | $1 < i < j < n$ | n-i+1 |

[1] L–1 rather than L is all that is necessary as any optimization L or more instructions back will already have been made.

For each replacement an analysis of all possible LHS's can be made to determine conformance with one of the above four forms and the back-up limited to the maximum of the back-ups required.

An analysis of the optimization patterns shows that the size of the back-up ranges from 2 to 13 instructions with a mean of 4.85. More importantly, if the analysis is restricted to those optimizations actually made during the compilation of the three source files considered in table 1, this results in a range from 2 to 8 with an mean back-up of 3.96. This means that the overhead resulting from re-scanning is reduced substantially from the previous method of re-scanning the block completely if a replacement is made.

## ALGORITHMS AND DATA STRUCTURES

New data structures suitable for the matching and replacement of patterns are represented in figure 3. All matching takes place in the fixed size buffer consisting of an array of structures, each structure representing a single decoded EM instruction. The EM instructions are read as required from the input, decoded and then stored in a structure of the buffer.

This enables easy access to the instruction's opcode while searching for possible matches and to its argument (if any) while any expression is being evaluated. The instruction stream in the buffer can be divided into three distinct regions:

a) those instructions that have already been searched for optimizations,

b) those instructions that constitute a valid prefix for one or more patterns, and

c) those instructions that have been backed-up over after a previous replacement or failure.

The final region may contain no instructions (indicated by end=0) in which case attempts to access instructions from this region will cause new instructions to be read

and decoded from the input stream and deposited in the next available structure in the buffer.

The matching process can be controlled by a deterministic finite automaton (DFA) built from the opcodes that constitute the LHS patterns of the optimizations. Figure 4 shows the structure of such a DFA given the following 5 optimization patterns:

| | | |
|---|---|---|
| loc adi w loc sbi w | → | loc $1-$3 adi w |
| inc dec | → | |
| inc loc adi w | → | loc $2+1 adi w |
| inc loc sbi w | → | loc $2-1 sbi w |
| dec loc adi w | → | loc $2-1 adi w |
| dec loc sbi w | → | loc $2+1 sbi w |

Patterns are recognized by starting in state 0 and making transitions in the DFA based on the next instruction's opcode in the matching buffer. The final states, (indicated by double circles in the diagram,) correspond to complete LHS patterns; in these states the success of the match is determined by evaluating any expressions associated with each optimization with the same LHS opcodes. If none of these expressions is satisfied the state will not be regarded as a final state and the matching process continues. If one succeeds then the replacement RHS is constructed in the separate replacement buffer (whose length is known to be that of the longest RHS) using the still available LHS arguments if required. The replacement instructions can then be copied in the matching buffer; if necessary instructions in the back-up region are moved to make room or close up any gap. The need for such a move can be determined when the DFA is constructed, on the basis of the difference in length of the LHS and RHS of the successful pattern.

Figure 5 shows the same DFA as figure 4 but also includes the failure transitions calculated using the methods from the previous section. These are labelled with the amount of back-up required before scanning continues. To simplify the diagram, failure transitions from states that require no back-up and continue scanning from state 0 are not shown. The figure also shows for each final state the back-up required after a successful replacement[1].

The 580 optimization patterns result in a DFA with 699 states. An attempt was made to encode these states as a series of nested C `switch` statements but this exposed various size restrictions in such constructs in a number of available C compilers. Instead, the DFA was encoded in tables using row displacement encoding [8]. Because of the extremely sparse nature of the DFA transition function, such encoding was extremely successful with the 698 transitions being encoded in just 755 table entries.

After each transition to a new state, an array of functions was consulted to determine if the new state was a final state. If this yielded a non-null entry the retrieved function was called to check the corresponding expressions, if any, for all patterns with the same LHS opcodes. These were written as a series of C `if` statements; the body of the `if` statement contained code to construct the replacement instructions and perform any subsequent copying and back-up required before returning to state 0 to continue the search. If none of the *if* statements were valid or the new state was not a final state then the tables encoded the necessary adjustments to the buffer pointers and the state to continue from.

---

[1] In general the back-up required for such states will depend on which RHS replacement is made. This is not required for this simple example as there are no two patterns with the same LHS opcodes.

## EFFECTIVNESS OF THE NEW OPTIMIZER

A measure of the effectiveness of the new optimizer is presented in table 2, where the times for optimizing the three example source files are presented.

| program source | ar.c | ctags.c | cu.c |
|---|---|---|---|
| time for old optimizer | 1.60 | 2.51 | 2.01 |
| time for new optimizer | 0.73 | 1.02 | 0.86 |

Table 2. Timing of old and new optimizers in Sun 3/60 cpu secs.

Averaged over the three source files these figures give throughputs of 2300 and 5300 EM instructions per second for the old and new optimizers respectively, so the throughput has been improved by a factor of 2.3. The effect of the number of patterns on optimization time for the old and new optimizers is presented in figure 5. It will be noted that the new optimizer's time is almost independent of the number of optimization being matched.

The instructions in the matching buffer can be flushed to the output stream whenever an instruction that is not in any LHS pattern is read or generated by an output replacement. In general it is not possible to calculate the largest possible size required for the matching buffer as it is possible to construct combinations of pathological input streams and optimization patterns that would require a matching buffer that could hold the complete input stream before it was flushed. In practice, however, only a very modest size buffer is required. While optimizing the three example source programs containing a total of 12821 EM instructions the buffer was flushed 2671 times. The mean length of the output, pattern and backup queues at the point of flushing are 2.5 , 1, and 0.2

instructions respectively while that of the sum of these is 3.6 instructions. The maximum lengths for these same queues at flushing are 23, 1, 7 and 24 . The mean length of the sum of these queues before each transition in the DFA is 4.6 .

Furthermore, if the buffer ever does become full, it is possible to perform a "half flush". This involves writing out half the instructions waiting in the output queue and then moving left all the remaining output, patterns and back-up instructions and pointers before matching continues. The only effect of such a flush is to possibly miss some optimizations as it limits the back-up possible after a subsequent optimization. In the current version of the optimizer the buffer is able to hold 200 instructions and no such half flush occurred during the optimization of the three test files. Even if such a flush did occur it would still only result in missed optimization opportunities if (cumulative) back-ups of the order of 100 instructions were required which is highly unlikely. An alternative strategy would be to allow an extra passes over the intermediate code in the exceptional situations where a half-flush occurred.

## OPTIMIZER AS OPTIONAL LIBRARY

It was noted in the introduction that generation of encoded EM files by front ends was simplified by the provision of a library of generating routines, one for each distinct EM instruction. To generate a `loc 5` instruction for example, the front end need only make the call `C_loc(5)` and ensure that the corresponding library is included when the front end is loaded.

The design of the new optimizer was carefully chosen to ensure that it was possible to provide a similar procedural interface. This was provided via a call of the form `O_loc(5)`, for example, to output a `loc 5` instruction to the optimizer. Each such routine constructs an appropriate data structure in the next free location in the matching

buffer before calling the DFA matching routine. The state of the DFA is kept between calls to the DFA routine and the DFA loop continues to make transitions, replacements and back-ups until a new instruction is required in the matching buffer. At this point it returns and will continue only when called again from another O_xxx routine acting like a co-routine.

Such a structure allows the optimizer to be conveniently packaged as a library which allows a number of flexible alternatives.

In its simplest form the library can be used to build a simple stand-alone optimizer by providing a main program loop that reads EM instructions (using the reading EM library routines) and then calls the O_xxx routines of the optimizer library. This is then loaded with the EM reading, optimizing and writing routines to build a program that has the same functionality as the old optimizer in the ACK pipeline.

| program source | ar.c (secs) | ctags.c (secs) | cu.c (secs) | throughput (lines/sec) |
|---|---|---|---|---|
| front end + old optimizer | 3.20 | 4.30 | 4.07 | 266 |
| front end + new optimizer | 2.37 | 2.79 | 2.95 | 379 |
| combined front end and optimizer | 1.95 | 2.19 | 2.38 | 472 |

Table 3. Timing for phases of ACK pipeline on Sun 3/60

Alternatively, the front-end sources can be pre-processed to change every C_xxx procedure call to a O_xxx call, and the optimizer library is loaded with the front-end

program. This results in a front-end that produces optimized EM instructions directly without the overhead of encoding, writing, reading and decoding the EM files between the phases. Table 3 presents the timing for such a C language front-end using the three example source files as input. The final column of this table gives the throughput measured in C source lines per cpu second accumulated over the three files. This shows that the throughput achieved by the combined front-end is 1.77 times that of the separate front-end plus old optimizer and 1.25 times that when the new optimizer is executed as a separate pass.

Further flexible combinations are also possible. Any step in the ACK pipeline that outputs EM instructions can be changed to output an optimized stream of EM instructions instead by pre-processing the sources and loading the new library. For example, experiments were conducted with fast code expander back-ends that generate machine code directly by expanding each EM instruction to a group of machine instructions. These were constructed using the same procedural interface used by the EM writing library and so could be loaded directly with the front-ends to produce a complete compiler. These ran extremely fast but the quality of the code produced suffered as a result of the front-ends generating EM code under the assumption that the peephole optimizer would always remove obvious inefficiencies. Again, by pre-processing the front-end sources and loading the optimizer library the peephole optimization can be added to these compilers resulting in a substantial improvement in the size and speed of the generated code with very little effect on the throughput of the compiler.

## CONCLUSIONS

The redesign of a peephole optimizer in the context of the Amsterdam Compiler Kit has been described. By detailed analysis at the time the optimizer is constructed of the individual optimizations to be used, limits were placed on the back-up required after

failure and successful replacement. This achieved substantial improvements in the throughput of the optimizer. The redesigned optimizer was constructed within the context of a Deterministic Finite Automata using a fixed sized matching buffer. This was augmented with code to test any restrictions among the arguments of the intermediate instructions before any successful patterns were matched and replaced in the buffer. By providing a procedural interface, packaged as a library, flexible combinations of front and back-ends could be achieved. The resulting reduction in the overheads in dealing with a file-based intermediate instruction stream resulted in substantial improvements in the throughput of the ACK pipeline.

## ACKNOWLEDGEMENTS

This work was begun while the author was on leave at the Vrije Universiteit, Amsterdam, The Netherlands. Thanks are due to Andy Tanenbaum, Ceriel Jacobs and Dick Grune for helpful discussions about this and other problems. I thank Tim Bell and Rod Harries for useful comments on this paper. The idea of an extra pass after a half-flush was suggested by one of the referees.

## REFERENCES


1. JOHNSTON, S.C. A tour through the portable C compiler AT&T Bell Laboratories, Murray Hill, N.J.

2. AMMANN, U., *The Zurich implementation* in: Barron D.W. (1981) *Pascal - The Language and its Implementation* Chichester: Wiley pp. 63-82

3. STEEL, T.B., JR. UNCOL: The myth and the fact. *Annu. Rev. Autom. Program.* 2 (1960), 325-344.

4. TANENBAUM, A.S., STEVENSON, J.W., AND VAN STAVEREN, H. Description of an experimental machine architecture for use with block structured languages. *Inf. Rapp.* 54, Vrije Univ., Amsterdam, 1980.

5. DAVIDSON, J.W., AND FRASER, C. W. The design and application of a retargetable peephole optimizer. *TOPLAS* 2, 2 (April. 1980), 191-202.

6. TANENBAUM, A.S., VAN STAVEREN, H., AND STEVENSON, J.W. A practical toolkit for making portable compilers. *Commun. ACM* 26, 9 (Sept. 1983), 654-660.

7. TANENBAUM, A.S., VAN STAVEREN, H., AND STEVENSON, J.W. Using peephole optimization on intermediate code. *TOPLAS* 4, 1 (Jan. 1982), 21-36.

8. AHO, A.V., SETHI, R., AND ULLMAN, J.D. *Compilers: Practical techniques and tools*. Reading (Masachusetts): Addison Wesley (1986), 144-146