Trans. Nat. Acad. Sci. & Tech. (Phils.)
1987.9:65-75

A FLEXIBLE PEEPHOLE OPTIMIZATION METHOD FOR INTERMEDIATE CODES[1]

Eliezer A. Albacea
Computer Science Laboratory
Institute of Mathematical Sciences and Physics
University of the Philippines at Los Baños

ABSTRACT

Object codes generated by compilers are usually not optimal. Several methods of improving object codes have been devised to further reduce the execution times of programs. This paper describes a new method under the class of peephole optimization technique. The method was implemented using a one-pass Pascal compiler and the intermediate language SLIM. The latter is described briefly in this paper. To evaluate the new method, a comparison of several methods under the peephole optimization technique is presented. In addition, the codes generated with and without the proposed method are compared.

Introduction

Tanenbaum *et al.* [9] stated that in a compiler consisting of a front end that translates to a common intermediate language and a back end that translates to a machine's assembly language, improvement of object code can be performed in three conceptual places.

The first conceptual place is to do the improvement in the front end. The decision to do it in the front end would consequently require that the translator be "highly specialized". What we mean by a "highly specialized" translator is a translator that attempts to generate the best code that it can possibly generate for a particular source code fragment. Usually this type of translator is too complicated to construct and thus would require a high development effort. In addition, such translators will increase the compilation time of source programs because the compiler will have to carry out numerous tests to get better object code for a certain source code fragment. But no matter how specialized the translator is, it will still miss some possible improvements in the source code fragments translated separately by the translator. For example, the Pascal statements a:= b + c and d:= a + d will be translated by a highly specialized translator (assuming that it translates each

[1]Research supported by the National Research Council of the Philippines (NRCP) under Project Number B-45.

statement separately) to Lb + c Sa followed by La +d SD. Obviously, the translator will fail to detect the possible improvement of the instructions Sa La to simply Sa. To catch these possible improvements, however, the compiler should do further improvement on the intermediate code. This brings us to the second conceptual place, i.e., doing the improvement on the intermediate code.

Since doing the improvement in the front end may still require another pass through the intermediate code to catch every possible improvement, it is usually advisable to do all the code improvement on the intermediate code and merely construct a simple front end translator. In this way, the development of the translator will not involve too much effort. Moreover, since the intermediate language does not change, the optimization procedures will be the same for all front ends or back ends.

The last conceptual place is to do the improvement in the back end. This possibility seems to be the most profitable. The reason is that if the objective is to catch all possible improvements in the code, then improvement should be done in the code that is finally executed. But, this would mean that for every new back end, a new code improver must be written. Note that a possible improvement in one machine may not be possible in the other. For example, the sequence of instructions MOVE.L 4(A1), D0 and MOVE.L D0, 10(A0) are Motorola MC68000 instructions which can be improved to MOVE.L 4(A1),10(A0). But a similar set of instructions in another machine that does not allow memory to memory copy can not be improved at all.

Improvement is usually done on the intermediate code to avoid the greater development effort in doing the improvement in the front end or the back end. Although doing the improvement on the intermediate code will not catch all possible improvements, the difference compared to doing it in the back end is usually slight. This is because each intermediate code is usually mapped to the most efficient actual machine code.

There are several classes of methods devised to improve intermediate codes. But, we shall be concentrating only on one class, the peephole optimization technique. After considering the technique in general, a discussion of the implementations of a peephole optimizer including the new method will be given. To investigate the advantage of the new method, a time comparison of several programs compiled with and without the code improver will be presented.

The intermediate code Stack Language for Intermediate Machines (SLIM) and the SLIM code generated by a one-pass Pascal compiler will be used as basis for the discussion of the new optimization method. Hence, in the following sections the SLIM machine and its instructions will first be introduced before the discussion of the new method.

The SLIM Machine

SLIM is a simple one-accumulator, stack-oriented hypothetical machine. The

machine was first proposed by Fox[4], but was later enhanced by Peck[7]. The machine was designed with the following principal aims:

1. To reflect current machine architecture, if possible;
2. To obtain a reasonably simple machine such that it can be used for teaching the elements of computing;
3. To obtain a machine that is suitable as target machine for high-level languages such as BCPL;
4. To obtain a tool for achieving portability of systems programs; and
5. To obtain a machine on which it is possible to have an operating system.

SLIM is a machine very similar to a conventional computer in that it consists of a memory and a processor.

The memory of SLIM is a sequence of cells. Each cell contains 'n' bits with the value of 'n' implementation dependent. It may be 16 bits, 32 bits, or more, but the choice depends entirely on the number of bits required for an address on the target machine and the memory available. The cells are addressed consecutively starting from 0 to 'm'. The value of 'm' as with the number of bits per cell, is machine dependent. However, the size of 32K cells was found to be a comfortable size for many SLIM implementations, e.g., on Perkin Elmer 3230 Motorola MC68000, IBM 370 (Amdahl 470), etc.

SLIM has a total of seven (7) registers. These registers and their functions are given in Table 1.

Table 1. SLIM Register

Symbol	Register	Function
A	Accumulator	This is where all arithmetic and logical operations take place
E	Environment	Holds a pointer to the environment of a procedure
H	High Point	Points to the last useful cell on the stack
C	Program Counter	Holds a pointer to the instruction to be executed
G	Global	Holds a pointer to the first cell of a sequence of cells reserved for global variables
N	Interrupt	Holds information that is used to recover from an interrupt
S	Stack Limit	Holds a pointer to the last cell in the stack

Typical SLIM instructions may contain at most three fields – the operator, the operand modifier, and the raw operand. The operator field is always present in the instruction while the raw operand and the operand modifier may or may not be, depending on the type of instruction.

An operation may be L for load, i.e., move data from memory to accumulator (A), S for store, i.e., move data from accumulator (A) to memory, + for add, J for jump, and so on.

A raw operand is either an unsigned or signed number, a character, the H register, or a label (@n where 'n' is an integer).

The remaining field is the operand modifier. It is used, if it is part of the instruction, to qualify the meaning of the raw operand. An operand modifier is either E (modified by environment), G (modified by global), I (modified by indirection), IE (combination of environment and indirection), or IG (combination of global and indirection).

Table 2 shows a list of all the SLIM operators. The table gives a brief description of the operator, and the microcode for each operator to help visualize its actions. The microcode is written in BCPL. The table does not show all the possible operands and operand modifiers; instead we represent the operand, modified or unmodified, by the letter W.

Table 2. SLIM Operators

Instruction	Mnemonic	Microcode
Load cell	LW	A := W
Store cell	SW	!W := A
Load cell subscripted	L!W	A := A!W
Store cell subscripted	S!W	LET V = !H; H −:= 1; A!W := V
Load byte	L%W	A := A%W
Store byte	S%W	LET V = !H; H −:= 1; A%W := V
Load field	L:W	A := W of A
Store field	S:W	LET V = !H; H −:= 1; W OF A := V
Load device	L$r	A := A!r
Store device	S$r	LET V = !H; H −:= 1; A!r := V
Push and load cell	PLW	H +:= 1; !H := A; A := W
Jump	JW	C := W
True jump	TW	IF A = TRUE THEN C := W
False jump	FW	IF A = FALSE THEN C := W
Modify high point	MW	H +:= W
<dop>	<dop>W	A := A <op> W
<mop>	<mop>	A := <mop> A
Procedure call	CW	
Procedure return	R	C := E!O; H := E!(−2); E := E!(−1)
Push	P	H +:= 1; !H := A
Exchange	X	W := H!(−1); H!(−1) := A; A := W
Originate	O	
Void	V	no operation
Quit	Q	exit
Switchon sequential	?S	
Switchon indexed	?I	
Non-local access	U	

<dop> – dyadic SLIM operators
<mop> – monadic SLIM operators

Peephole Optimization

Peephole optimization has been used to improve intermediate and actual machine codes. The method works by looking at a small range of instructions, at least two instructions, and replacing them by more efficient instructions. This small range of instructions is referred to as the peephole. The code in the peephole may be continguous, e.g., the peephole SE2 LIE2 is replaced by SE2 or scattered, e.g., the peephole LE2 P . . . SH is replaced by . . . SE2. The nature of the technique is that the replacement code for a sequence of instructions can be used for further improvement. An example is given in Table 3.

Table 3. An illustration of peephole optimization

Sequence of Instruction	Peephole	Replacement
LIE2 P LIE3 +H <inst>	P LIE3	PLIE3
LIE2 PLIE3 +H <inst>	PLIE3 +H	+IE3
LIE2 +IE3 <inst>	+IE3 <inst>	

<inst> – remaining instructions in the sequence

One of the aims of code improvement is to improve the code in a manner that the run-time improvement is greater than the overhead introduced by the improvement procedures at compile time. The next section will discuss how this objective is approached by showing several methods (including the new method) adopted to implement a peephole optimizer.

Implementation of a Peephole Optimizer

Davidson and Fraser [3] described a method for improving assembler codes using two instructions in the peephole. The method works by examining a pair of instructions in the peephole and replacing them, if possible, with one instruction which has the same action. In case the pair of instructions can not be reduced to one instruction, the first of the two instructions gets emitted. The new instructions in the peephole then are the second instruction of the previous peephole and the instruction immediately following the previous pair of instructions. For example, consider the sequence of PDP-11 instructions MOV @R3, R2 and ADD #2 R3 which can be replaced by an equivalent one instruction MOV (R3)+, R2.

Tanenbaum *et al.* [9] developed a method which allows the number of instruction in the peephole to vary from one to any number greater than one. The method was used to improve the intermediate code EM and it employs the use of a pattern/replacement table. The table consists of a collection of lines, each line having a pattern part (peephole) and a replacement part. In contrast to the approach by Davidson and Fraser [3], which uses a constant number of instructions in the peephole, the pattern part (peephole) vary in number of instructions. Their method works by simply constructing the patterns and replacements in advance and

these are looked up in the table during compilation. To avoid missing new patterns created by the replacements, the method repeats the matching process until no more match is found. Examples of pattern and replacement lines are given in Table 4.

Table 4. Patterns and replacements in EM

Pattern	Replacement	Comment
LOC A LOC B ADD	LOC(A + B)	Add constants A and B
LOC 2 MUL	LOC 1 SHL	Change multiplication to shift

Note that the length of the pattern (number of instructions) varies and the replacement is not necessarily shorter in length than the pattern. It may have the same length but the replacement is known to be executed faster than the pattern, e.g., the change from multiplication to shifting.

The New Method

This new method can be used to improve the intermediate code SLIM.In fact, it is theoretically possible to use the method to improve other intermediate codes like PCODE, JANUS , or any intermediate code generated by a one-pass translator.

The method is an extension of the method employed by Davidson and Fraser [3] . The difference is that the number of instructions in the peephole is allowed to increase depending on the kind of source code the translator is translating. The method, therefore, combines the advantages of the methods by Davidson and Fraser [3] and Tanenbaum *et al.* [9].

The extension allowing more than two instructions in some code fragments is essential because the translator generates code which is impossible to improve with only two instructions in the peephole. For example, given the code fragment, a – b, where 'a' and 'b' are the first two local variables of the procedure, then the translation of the given code fragment is LIE2 P LIE3 PLH –H. Using only two instructions in the peephole, this can be improved to LIE2 PLIE3 PLH –H. The subsequent translation, however, can be improved to LIE2 –IE3 if three instructions are used in the peephole.

The problem of determining the number of instructions in the peephole for a particular source code fragment can actually be decided by the manner the translator translates the code fragment. Take for example the same code fragment, i.e., a – b, and suppose that the translator translates the right operand first, this would mean that only two instructions in the peephole are enough to improve the code to its best possible form. To illustrate this point, consider the translation when the right operand is translated first. The translation will be LIE3 P LIE2 –H which can be improved to LIE3 PLIE2 –H and finally to LIE+ –IE2.

Note that the foregoing is always true only for a "highly specialized" translator but not for a one-pass translator. A one-pass translator will usually translate code fragments from left to right; thus requiring longer patterns in the peephole.

The code improver will initially assume a size of two instructions in the peephole. Whenever a code fragment requiring more than two (2) instructions in the peephole is translated, the size of the instructions in the peephole should correspondingly increase. What arrangements then are necessary in order to allow the code improver to change the size of the peephole?

The approach taken in the implementation of this new method was to require the translator to send a signal to the code improver. The signal will inform the code improver that the translator is about to translate another source code fragment. Further, the signal can be in the form of a number. This number will give the code improver the necessary information on the number of instructions required in the peephole in order to improve the code fragment being translated to its best possible form. Another signal must be sent to the code improver once the translation of the current source fragment is through. The size of the peephole then must be sent back to the previous size, i.e., size of the peephole before the latest signal was received by the code improver. Note that the previous size is not necessarily equal to two.

Several advantages can be identified in having the foregoing arrangement. One is that the search for the patterns and their corresponding replacements will only be limited on the patterns whose size is equal to the current size of the peephole. This would consequently mean a significant reduction in search time compared to searching the pattern from all the available patterns that can possibly be improved.

In the case of a one-pass Pascal translator and SLIM as intermediate code, it was found that most translation of Pascal code fragments can actually be improved using only two instructions in the peephole. But for Pascal expressions, two instructions in the peephole are not enough. Three instructions in the peephole were necessary to improve expressions to their most efficient form.

One might argue that since the set of expressions is the only type of code fragment that needs more than two instructions in the peephole, why not use two instructions all throughout? This question makes sense, but if one can reduce the translation of an expression from five (5) to two (2) using three (3) instructions in the peephole, instead of from five (5) to four (4) using two (2) instructions in the peephole, then this will be a great improvement to the code. The reason is that an expression is actually a sub-fragment of almost all Pascal code fragments (not only Pascal but also C, BCPL, Algol 60, Algol 68, PL/I, and other high-level languages).

Adding a code improver of this nature to a one-pass translator will preserve its one-pass property and therefore overhead due to the introduction of the code improver during compilation will be tolerable. The translator will translate source code as before, but everytime an object code is generated it is first pass through the code improver which decides whether the code can already be emitted or not. In short, the code improver is just acting as a filter. The code improver maintains a peephole wherein everytime the translator produces a code it includes this code in the peephole. If the new peephole can be improved, it is replaced by its more efficient equivalent. Otherwise, the oldest instruction in the peephole will be emitted. The code improver will then wait for the translator to generate another object

code. The process is repeated until the translator generates the end of program code.

Patterns and Replacements

There are so many patterns in SLIM that can be improved but only those patterns that can possibly be generated by a one-pass Pascal translator will be shown. Usually, it is the compiler writer who has full knowledge of the patterns of instructions that the compiler generates. It is therefore his responsibility to identify all these patterns and replacements that can be included in the code improver. This is in addition to the requirement that he should know the number of instructions in the peephole necessary to improve a certain source code fragment.

Table 5 shows a summary of all the patterns and their corresponding replacements incorporated in the code improver employing the new method. The type of improvement can be classified into four general classes, namely: folding, rearrangement, strength reduction, and null sequences.

The first group of patterns and replacements constitute those instructions with operands whose values are known at compile time. When the values of all operands in an expression are known at compile time, that expression can be folded, i.e., replaced by a single value. For example, the instructions M5 M4 can be replaced by M9. The values of the operands 4 and 5 were replaced by their sum.

The group on rearrangement has the usual purpose of reducing the amount of temporary storage required during the evaluation of an expression. For example, the code L100 PL50 +H is improved to L100 +50. The unimproved pattern will first load the value 100 into the accumulator; push the value in the accumulator onto the stack; load the value 50 into the accumulator; and finally add whatever is stored in the accumulator with whatever is on top of stack. Compare this to the improved code where the value 100 is loaded into the accumulator and then it is added to the operand of the + operator, i.e., to 50. The use of temporary storage, i.e., the use of the stack, was eliminated.

The strength reduction group of patterns and replacements has the objective of replacing an expensive operation by a cheaper one. The replacement may not necessarily be shorter than the pattern of instructions to be replaced. One example is the replacement of multiplication by the shift operation, e.g., LIE2 *2 to LIE2 >>1.

Finally, the last group is the null sequences group. These instructions can well be deleted from the translation of the source program without affecting the correctness of the translation. In short, this is composed of instructions whose total effect is null. One example is the pattern LIE2 SIE2. The first instruction loads the value of the first local variable into the accumulator and the second instruction stores it back to where it came from.

It will be shown in the next section that the code improver described introduces a negligible overhead to the compilation of source programs but improves the execution time by a reasonable amount.

Table 5. Patterns and replacements in SLIM

Pattern	*Replacement*
Folding	
Lc–	L–c
Mb Mc	M(b+c)
Lφ ∿	L–1
L–1 ∿	LO
Rearrangement	
PLm <di>H	<di>m
PLmPLH <di>H	<di>m
Strength Reduction	
Lm *2	Lm >>1
Lm /2	Lm <<1
Lm *4	Lm >>2
Lm /4	Lm <<2
Lm *8	Lm >>3
Lm /8	Lm <<3
*–1	–
#*–1.0	–
/–1	–
#/–1.0	–
Null Sequences	
J@1 @1:	@1:
J@k J@1	J@k
P Lm	PLm
Mc R	R
R R	R
SEc LIEc	SEc
LIEc SEc	
+0	
# + 0.0	
–0	
# – 0.0	
*1	
# * 1.0	
/1	
# / 1.0	

b & c – integer constants
k & 1 – labels

m – SLIM modifier
<di> – dyadic SLIM operator

Comparison of Improved and Unimproved SLIM Code

The Pascal translator was used to compile several programs to investigate the effect of the code improver on the compilation and execution times of source programs. Of course, attempting to improve the code will almost certainly degrade the compilation of source programs. But, if the improvement will decrease the execution time by at least the same amount as the increase in compilation time then the improvement carried out on the code is certainly worthwhile.

To see whether the method used by the code improver results in an improvement, i.e., the decrease in execution time is greater than the increase in compilation time, the translator was used to translate the following programs:

1. Implementation of the date of Easter algorithm by Amman [2].
2. Sorting of 1000 data items using the quicksort algorithm.
3. Implementation of the eight queens problem by Wirth [10].
4. Multiplying a 20 by 20 matrix.

The programs were translated (with and without the code improver) and executed. The summary of compilation and execution times are given in Table 6 and Table 7, respectively.

Table 6. Compilation times in (seconds)

Program	*With the Code Improver*	*Without the Code Improver*
Date of Easter	0.46	0.41
Quicksort	0.71	0.67
Eight Queens	0.61	0.61
Matrix Multiplication	0.33	0.33

Table 7. Execution times (in seconds)

Program	*Improved SLIM Code*	*Unimproved SLIM Code*
Date of Easter	0.38	0.69
Quicksort	1.36	1.78
Eight Queens	3.16	4.07
Matrix Multiplication	0.96	1.11

It is clear from Table 6 and Table 7 that the degradation in compilation due to the introduction of the code improver is less than 0.1 second. But, the improvement in execution time is much more than 0.1 second. This shows that computing cost can be reduced with the introduction of the code improver.

The significant decrease in execution time can be attributed to the significant decrease in number of instructions in the improved SLIM equivalent. This is illustrated in the date of Easter program where the number of SLIM instructions is reduced from 235 instructions to 131 instructions. A significant difference of 104 instructions. This difference is 44.26% of the original number of instructions.

Conclusion

Although the new method was tested using a one-pass Pascal translator and SLIM as intermediate code, it is worth noting that the method can also be used for other high-level language translators generating intermediate codes. In addition, it can also be added to compilers generating assemblers instead of intermediate codes. But, of course, the method should only be employed in compilers generating assembler codes when it was previously determined that it will be impossible to improve the assembler equivalent of some source code fragments using only two instructions in the peephole.

Introducing the method to existing compilers will require the modification of the compiler itself and the writing of the code improver. The modification on the compiler is necessary because the method requires the translator (compiler) to send a signal about the type of source code it is about to translate.

References

Albacea, E.A. 1985. An Intermediate Code for the Translation of Pascal and BCPL. The University Wollongong MSc(Hons) Thesis, Wollongong, N.S.W., Australia.

Amman. U. 1977. "On code generation in a Pascal compiler," Software – Practice and Experience, 7(3):391-423.

Davidson, J. W., and C.W. Fraser, 1980. "The design and application of a retargetable peephole optimizer," ACM Transactions on Programming Languages and Systems, 2(2):191-202.

Fox, M. 1978. Machine Architecture and the Programming Language BCPL, University of British Columbia MSc Thesis, Vancouver, Canada.

Knuth, D. E. 1968. The Art of Computer Programming, Volume 1, Addition-Wesley, New York.

McKeenan, W.M. 1965. "Peephole Optimization," Comm. ACM, 8(7):443-444.

Peck, J.E.L. The Essence of Portable Programming, in Preparation.

Peck, J.E.L. Private communications.

Tanenbaun, A.S., H. van Stavaren and J.W. Stevenson. 1982. "Using Peephole Optimization on Intermediate Code," ACM Transactions on Programming Languages and Systems, 4(1):23-36.

Wirth, N. 1976. Algorithms + Data Structures = Programs, Prentice-Hall, Inc., Englewood Cliffs, N.J.