American Journal of Engineering Research (AJER)
e-ISSN : 2320-0847 p-ISSN : 2320-0936
Volume-02, Issue-05, pp-140-144
www.ajer.us

Research Paper Open Access

Technical Review of peephole Technique in compiler to optimize intermediate code

Vaishali Sanghvi[1], Meri Dedania[2], Kaushik Manavadariya[3,] Ankit Faldu[4]
[1](Department of MCA/ Atmiya Institute of Technology & Science-Rajot, India)
[2](Department of MCA/ Atmiya Institute of Technology & Science-Rajot, India)
[3](Department of MCA/ Atmiya Institute of Technology & Science-Rajot, India)
[4](Department of MCA/ Atmiya Institute of Technology & Science-Rajot, India)

Abstract: Peephole optimization is a efficient and easy optimization technique used by compilers sometime called window or peephole is set of code that replace one sequence of instructions by another equivalent set of instructions, but shorter, faster. Peephole optimization is traditionally done through String pattern matching that is using regular expression. There are some the techniques of peephole optimization like constant folding, Strength reduction, null sequences, combine operation, algebraic laws, special case instructions, address mode operations.

The peephole optimization is applied to several parts or section of program or code so main question is where to apply it before compilation, on the intermediate code or after compilation of the code .The aim of this dissertation to show the current state of peephole optimization and how apply it to the IR (Intermediate Representation) code that is generated by any programming language.

Keywords: *Compiler, peephole, Intermediate code, code generation, PO (peephole optimization)*

I. INTRODUCTION

Compilers take a source code as input, and typically produce semantically correspondent target machine instructions as output. A compiler (or a language translator) performs two operations: analysis and synthesis. The analysis phase determines the meaning of the source text, and the synthesis phase creates an equivalent program in a different language.

In the case of compilers, automatic generation of language translation tools requires and facilitates understanding of language translation. Syntactic analysis especially is well understood, encompasses a sizeable formal literature, and is habitually done mechanically by programs like **yacc** and **lex.**

A common method for implementing a group of languages on a collection of machines is to have one front end per language and one back end per machine. Each front end translates from its source language to a common intermediate code, called an UNCOL, and each back end translates from the common intermediate code to a specific target machine's assembly language.

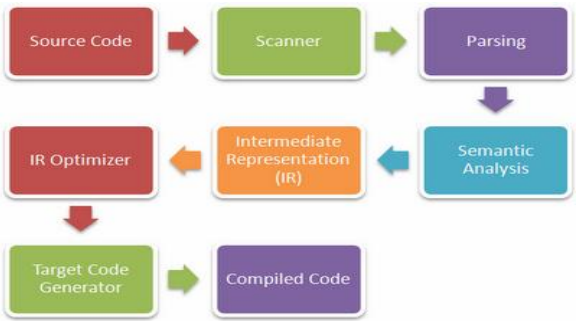

Fig 1: Compiler Process

Thus a "compiler" consists of a front end, a back end, and possibly programs that optimize the

intermediate code. When this model has been adopted, adding a new language or new machine only requires writing one new program (front end or back end) to maintain the property that all languages are available on all machines.

The question of where to perform the optimization arises. There are three theoretical possibilities:

1. in the front ends;
2. on the intermediate code;
3. in the back ends.

If the first option is chosen, many common optimizations will have to be programmed into each front end, increasing development effort.

Similarly, putting the optimizations in the back ends will also require a duplication of effort.

Though, any optimizations that can be performed on the intermediate code itself only need be done once, with the results being applicable to all front ends and all machines being used.

The compiler's task is to provide the translation. An ***'optimal translation'*** would ideally require minimal CPU time and memory.

This is usually not possible. But optimizing compilers can approach this by improving generated code with various techniques. Therefore, code optimization aims at producing more proficient, faster, and shorter code without changing its effects.

The problem with naive code generation is that resulting code often contains redundancies when juxtaposed. These can easily be reduced with an effective optimization technique – **peephole optimization.**

The peephole optimizer is repeatedly passed over the Modified assembly code, performing optimizations until no further changes are possible.

Peephole optimization is an effective technique for locally improving the target code. small sequences of target code instructions are examined and replacement by faster sequences wherever possible.Common techniques applied in peephole optimization.

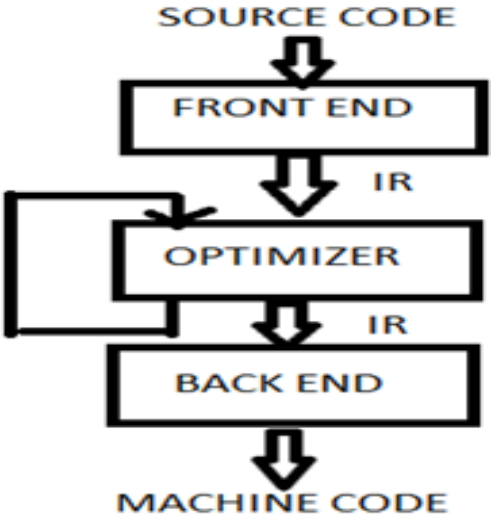

Fig 2: Optimization Level

1) Constant Folding
2) Strength Reduction
3) Null sequences
4) Combine Operations
5) Algebraic Laws
6) Special Case Instructions
7) Address Mode Operations
8) Copy Propagation

II. STRATEGIES

In peephole optimization technique, this part presents a formal analysis of the pattern matching strategies that have been implemented

1.1Strategy Part

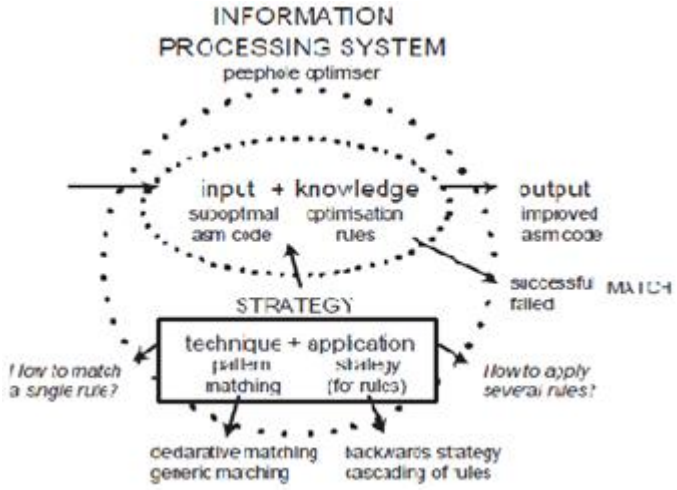

Fig 3: Peephole optimization as information processing problem

The main focus of this work is the strategy part shown in Fig 3, which is divided into two parts:

1.1.1 Pattern matching strategy:

This component particularly deals with the nature of the application of a single optimization rule to the input Matching strategies that have been Implemented in this work is the more primitive declarative, regular Expressions-based matching strategy, and a more abstract, object-oriented generic matching strategy.

1.1.2 Rule application strategy:

For this component, the grouping and ordering of the rules is analyzed as for a certain aim. In the implementation, the *backwards strategy* has been employed for both the *regular expressions strategy* and *the generic matching strategy*. The general matching policy has been extended with the ability to cascade rules, i.e. to control a number of rule application sequences.

III. CODE GENERATION STRETEGY

In order to understand the significance of the optimizations, it is necessary to understand something about typical code sequences produced by front ends for EM.

The traditional way to generate good code for commonly occurring statements is to build a myriad of special cases into all the front ends. For example, the statement N := N + 1 occurs often in many programs; so EM has a special INCREMENT VARIABLE instruction.

The normal approach would be to have the front end check all assignments to see if they can use this instruction. It is our belief that this approach is a mistake and that recognition and replacement of important instruction sequences should be done by the optimizer.

In fact, our basic intermediate file optimizer performs only this kind of peephole optimization; data flow and other optimizations can and should be done by other programs, cascading them in the manner of UNIX filters.

Coming back to the case of assignment statements, in general, assignment statements can be hard, such as A[I + 1].FIELD1 := B[J * K].FIELD2[M]

Although such statements measure rare, front ends should be ready to handle them consequently, the common strategy used by our front ends is to judge the address of the right-hand side, shove this address onto the stack, and then do a LOAD INDIRECT n instruction, which fetch the address through pop the address and pushes the n bite-long object starting at the address.

After that, the address of the left-hand part is also evaluated and pushed onto the stack, and then a STORE INDIRECT n instruction, which fetches the intented address and n-byte object from the stack and performs the store operation, is executed.

There are many special cases of the assignment statement that can be optimized, but the front ends ignore most of them, leaving the work to the optimizer.

Next to assignment statements, if statements are most common. Statements of the form if A = B then. Occur far more frequently than other types; so the obvious EM code sequence consists of instructions to push A and B onto the stack followed by a BNE instruction, which pops 2 operands and branches to the else part if they are unequal.

At first look it might appear that six Bxx directions would be required within the EM architecture, for

xx = EQ, NE, LT, LE, GT, and GE, but fact is that many more are needed, since a entire set is needed for single-precision integers, pointers,floating-point numbers, double precision integers, unsigned integers, sets, etc.

To avoid this large number, EM has one compare instruction for each data type that pops two operands and replaces them with a -1, 0, or +1, depending on whether the first is greater than, equal or less than the second.

Then there are six Txx instructions for replacing this number with true (represented by 1) or false (represented by 0), it is based on the relational operator.

IV. OPTIMIZATION PATTERNS

The optimizer is driven by a pattern/substitute table consisting of a collection of lines. Each line contains a pattern part and a replacement part. A pattern or substitute part is composed of a consecutive sequence of EM instructions, all of which designate an opcode and some of which designate an operand. (By design, no EM instruction has more than one operand.)

The operands can be constants, references to other operands within the line, or expressions involving both

Pattern Replacement

(1) LOV A INC STV A ~ INV A
(2) LOV A LOV A + 2 ~ LDV A
(3) LOC A NEG ~ LOC –A

It often occurs that the output of one optimization produces a pattern that itself can be optimized. In reality, this principle has been extensively used in the design of the optimization table to reduce its size. By repeating the matching process until no more matches can be found, patterns much longer than the longest optimization table entry can ultimately be replaced. After a replacement, the code pointer is moved back a distance equal to the longest pattern to make sure that no newly created matches are missed.

V. MEASURED RESULT

To measure the effect of the peephole optimizer, we have run two tests. In the first we compared the number of machine instructions in each optimized EM program with the number in the unoptimized EM program. Thus, for each program we have a number between 0.00 and 1.00 giving the number of instructions in the optimized program as a fraction of the original. This metric was chosen since it is independent of both the source language and the target machine and directly measures what the optimizer primarily attempts to do, namely, eliminate EM instructions. This metric can also be protected on theoretical grounds. EM code is really just glorified reverse Polish, in other words, the parse tree linearized into postfix order. Removal of an EM instruction typically corresponds to a removal of a node in the parse tree. Since object code size is typically proportional to parse tree size, such elimination normally has a straight impact on the final object code size. The dimensions presented below bear this out.

The occurrence frequencies per 1000 optimizations are shown in Table III which is labeled column EM. The median saving is 16 percent: one in six EM instructions is eliminated. The second test consisted of translating the optimized and unoptimized EM code into PDP-11 object code and comparing the number of bytes produced in each case. These results are given in Table III in the column labeled PDP-11.

The median reduction in the object code is 14 percent, close to the EM result. This closeness suggests that nearly all the EM optimizations are indeed reflected in the final object code. In 2 test programs, the optimized PDP-11 code was increased by 2 percent over the unoptimized code due to optimization 50; this was traced to a design error in the (original) EM to PDP-11 back end. (With the optimization the operands were actually stacked, whereas without it they were not.) This fault can merely be fixed, however.

On the basis of these results, we believe peep holing the intermediate code to be worthwhile, since the optimizer need only be written once, for all languages and every machines, and it in no way inhibits extra, more sophisticated optimizers, e1ither on the source code, on the EM code, or on the target code. Moreover, the peephole optimizer is fast: 1140 EM instructions per CPU second on a PDP-11/ 45 excluding certain overhead not related to peephole optimization and 650 instructions per CPU second including all overhead. This speed was achieved without any special effort to tune the program.

It could easily be made faster still by hashing the pattern table instead of examining all patterns starting with the current op code

TABLE 4: DISTRIBUTION OF AMOUNT OF REDUCTION IN SIZE

Ratio	EM	PDP-11
<0.60	0	12
0.60-0.64	3	6
0.65-0.69	22	25
0.70-0.74	24	35
0.75-0.79	205	71
0.80-0.84	283	191
0.85-0.89	298	333
0.90-0.94	126	181
0.95-1.00	39	141
>1.00	0	4
Total	1,000	999a

a Not 1000 due to round off.[19].

VI. SUGGESTED RESEARCH SCHOP

Further suggestions for interesting extensions and improvements:

5.1 Labels table:

Since the labels table has proven to be useful, it should be utilized more. The current implementation of the look ahead matching procedure using the labels table covers two cases only.
These should be extended and more should be added. Then, as generic matching is currently employed at level 1 only, 'deeper' generic matching by means of the labels table might help finding complex structures like chains of jump instructions and label definitions, or other constructions.
Since this 'table' approach is a good way of accessing information quickly, another extension might consist of researching other constructs than jump instructions and label definitions for which a table might be useful. The aim here is to do more intelligent matching.

5.2 Extended generic strategy:

Basic improvements can be made concerning the maintenance of the options. For more challenging improvements first the look-ahead behavior should be analyzed closer with a good rules set and suitable test files. An extension might then attempt to automatically find the best look-ahead setting (or cascade) for the given assembly input before applying it. This would be a 'cascade of rule sequences'

.

5.3 New strategies:

The analysis of pattern matching strategies does not end with the generic ones. Other rule application and pattern matching strategies might be explored: it would be particularly interesting to combine generic matching with AI methods as a next step.

REFERENCE

[1]. McKee man, W. M. (1965) Peephole optimization. CACM 8(7):443-444.
[2]. Lamb, D. A. (1981) Construction of a Peephole Optimizer. Software-Practice & Experience 11(6):639–647
[3]. Fraser, C. W. (1982) Copt, a simple, retargetable peephole optimizer. Software, available at ftp://ftp.cs.princeton.edu/pub/lcc/contrib/ copt.shar (up 24/02/2005).
[4]. Davidson, J. W., Fraser, C. W. (1980) The design and application of a retargetable peephole optimizer. ACM TOPLAS 2(2):191-202.
[5]. Fraser, C. W. (1979) A compact, machine-independent peephole optimizer. POPL'79:1-6.
[6]. Tanenbaum, A. S., van Staveren, H., Stevenson, J. W. (1982) Using Peephole Optimization on Intermediate Code. ACM TOPLAS 4(1):21-36.
[7]. Davidson, J. W., Fraser, C. W. (1987) Automatic Inference and Fast Interpretation of Peephole Optimization Rules. Software - Practice & Experience 17(11):801-812.
[8]. Davidson, J. W., Fraser, C. W. (1984) Automatic generation of peephole E optimizations. CC84:111-116.
[9]. Kessler, P. B. (1986) Discovering machine-specific code improvements.CC86:249-254.
[10]. Kessler, R. R. (1984) Peep - An architectural description driven peephole optimizer. CC84:106-110.
[11]. STEEL, T.B., JR. UNCOL: The myth and the fact. Annu. Rev. Autom. Program. 2 (1960), 325-344.
[12]. Warfield, J. W., Bauer, III, H. R. (1988) An Expert System for a Retargetable Peephole Optimizer. ACM SIGPLAN Notices 23(10):123–130.
[13]. Davidson, J. W., Fraser, C. W. (1984) Register allocation and exhaustive peephole optimization. Software - Practice & Experience 14(9):857-865.
[14]. Davidson, J. W., Fraser, C. W. (1984) Register allocation and exhaustive peephole optimization. Software - Practice & Experience 14(9):857-865.
[15]. Aho, A. V., Ganapathi, M., Tjiang, S. W. K. (1989) Code Generation Using Tree Matching and Dynamic Programming. ACM TOPLAS11(4):491-516.
[16]. Fraser, C. W., Wendt, A. L. (1986) Integrating Code Generation and Optimization. Proceedings of the SIGPLAN'86 symposium on Compiler Construction, pp. 242–248.