DWIT COLLEGE

DEERWALK INSTITUTE OF TECHNOLOGY

Tribhuvan University

Institute of Science and Technology

IMPLEMENTATION OF PEEPHOLE OPTIMIZATION

A PROJECT REPORT

Submitted to

Department of Computer Science and Information Technology

DWIT College

In partial fulfillment of the requirements for the Bachelor's Degree in Computer Science and Information Technology

Submitted by

Subigya Kumar Nepal

Asmit Prasai

Binod Aryal

Roshan Basnet

January, 2017

Table of Contents

Introduction

Peephole optimization is a process that replaces a sequences of consecutive instructions by semantically equivalent but more efficient sequence. Peephole optimization improves the code in terms of memory and time. The optimization technique may be applied

i. In the front end
ii. In the intermediated code
iii. In the back end.

In the front end, development effort will increase since many common optimizations have to be carried out into each front end. In the back end duplication of effort is required. Optimization that is performed on the intermediate code is applicable to all front ends and also to all the machines being used and so, is needed to be carried out only once.

We focused on the intermediate optimization technique.

The intermediate code used for this project is the assembly language of a simple stack machine called EM. It has been designed to be suitable for algebraic languages and bytes addressable target machines.

Each procedure invocation creates a frame on the stack. The Local Base register (LB) points to the base of the current frame, and the Stack Pointer (SP) points to the top of the frame. The External Base register (EB) points to the bottom of the outermost stack frame. Variables at intermediate levels of lexical nesting are accessed by following the static chain backward. All arithmetic instructions fetch their operands from the top of the stack and put their results back on the stack. Expressions are evaluated merely by converting them to reverse Polish. There are no general registers. Instructions are provided for manipulating integers of various lengths, floating-point numbers, pointers, and multi word unsigned quantities (e.g., for representing sets) (Tanenbaum, 1982)

The used instruction in this project are given in the table below.

Mnemonic	Instructions
ADD	Add
MUL	Multiply
ADI	Add immediate
LOC	Load constant
SHL	Shift left
BEG	Begin procedure
NEG	Negate
LOV	Load variable
LDV	Load double variable
LAV	Load address of variable
DIV	Divide
LOI	Load indirect
STV	Store variable
STI	Store Indirect
SUB	Subtract
INV	Increment

Optimization Pattern

The optimizer is driven by a pattern/replacement table consisting of collection of lines. Each line contains a pattern part and a replacement part. A pattern or replacement part is composed of a consecutive sequence of stack machine instruction (EM), 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.

For example,

Pattern		**Replacement**
LOV A INC STA A	==>	INV A
LOC A NEG	==>	LOC - A

In each line the ==> symbol separates the pattern part (left) from the replacement part (right).

Optimization Table

In this section we present and discuss a major portion of the EM optimization table. We have divided the optimization into six major groups, and these are only a few of the instructions that were adapted from the original paper. The original paper has quite a large list of instructions under each optimization group.

Constant Folding

SN.	Pattern			Replacement
1	LOC A	LOC B	MUL	LOC A* B
2	LOC A	LOC B	ADD	LOC A+B

Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime. Terms in constant expressions are typically simple literals, such as the integer literal 2, but they may also be variables whose values are known at compile time.

For example: k= 2 * 3 + 6

EM code is:

```
LOC 2
LOC 3
MUL
LOC 3
ADD
```

First three instructions are replaced by LOC 6 (since 2 and 3 is multiplied) and LOC 6 LOC 3 ADD is replaced by LOC 9 (since 6 and 3 is added). Arbitrary constant expressions involving all the operators can be folded into a single LOC.

Operator strength reduction

SN.	Pattern		Replacement
1	LOC 2	MUL	LOC 1 SHL

Operator strength reduction is a transformation that a compiler uses to replace costly (strong) instructions with cheaper (weaker) ones. The operator strength reduction group replaces multiplications by powers of two with shifts.

Null sequences

SN.	Pattern		Replacement
1	ADI 0		‘ ’
2	BEG 0		‘ ’
3	NEG	NEG	‘ ’
4	LOC 0	ADD	‘ ’
5	LOC 0	SUB	‘ ’
6	LOC 1	MUL	‘ ’
7	LOC 1	MUL	‘ ’

Null Sequence eliminates sequences or partial sequences of code that are redundant. ADI 0 is typically generated when accessing the first field of a record. The front end arranges for the address of the record to appear on the stack and then increases this address by the relative position of the desired field in the record, which for the first field is 0.

Combined moves

SN.	Pattern		Replacement
1	LOV A	LOV A+2	LDV A
2	LDV A	LOV A+4	LAV A LOI 6

The combined moves group tries to combine consecutive push or pop operations into a single one. When the EM code is to be interpreted, replacing two instructions by one is always worth doing.

The basic strategy followed by combined moves is to combine single-word, double word, and multi-word moves (LOV, LDV) into longer units.

Indirect moves

SN.	Pattern		Replacement
1	LAV A	LOI 2	LOV A
2	LAV A	STI 2	STV A

The indirect move group is largely concerned with replacing indirect moves, generated by the general case of the assignment statement, with more efficient direct ones.

Reordering

SN.	Pattern			Replacement
1	ADD	LOC A	ADD	LOC A ADD ADD
2	ADD	LOC A	SUB	LOC A SUB ADD

The reordering group merely reorders the instructions in each pattern without changing them. The given pattern here moves a LOC from the middle of an operation sequence to the start of it. By moving LOC forward, the chances of another optimization becoming possible are increased.

Implementation

The pattern checking and replacement was implemented by simply using regular expressions. Patterns are represented as regular expressions, which provide a format for expressing the sequence of characters to look for. This format allows abstraction over simple characters. If the string represents a valid occurrence of the pattern then a pattern successfully matches an input string and if at any point during matching, the input string does not satisfy the requirement of pattern then the pattern fails to match a string. (Mr. Chirag H. Bhatt, 2013)

Conclusion

As per the paper that this project was adapted from, the peephole optimization led to a median saving of 16 percent i.e. one in six EM instructions were eliminated. Based on this result, we can conclude that peepholing the intermediate code is worthwhile, since the optimizer need only be written once, for all languages and all machines, and it is fast too. (Tanenbaum, 1982)

We have used string based pattern matching in our project. Other approaches to pattern matching such as tree based and object based pattern matching might work better. (Mr. Chirag H. Bhatt, 2013) Also, our approach was to identify inefficient instruction sequences and replace them with equivalent instruction sequences. A better approach in dealing with peephole optimization these days, would be to automatically infer replacement rules from a symbolic description of the target machine. (Sorav Bansal, 2006)

Bibliography

Mr. Chirag H. Bhatt, D. H. (2013, March). Peephole Optimization Technique for analysis and review of Compiler Design and Construction. *IOSR Journal of Computer Engineering (IOSR-JCE)*, 80-86.

Sorav Bansal, A. A. (2006). Automatic generation of peephole superoptimizers. *12th International Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 394-403). San Jose: ACM SIGPLAN Notices 41(11).

Spinellis, D. (1999, February). Declarative peephole optimization using string pattern matching. *ACM SIGPLAN Notices*, 34(2):47–51.

Tanenbaum, A. v. (1982, 01 1). Using Peephole Optimization on Intermediate Code. *ACM Trans. Prog. Lang. and Sys. 3*, pp. 21-36.

Appendix

Code

```
import re
from itertools import izip

def constant_folding(code):
    try:
        op = re.findall('LOC ([0-9]+)\nLOC ([0-9]+)\nMUL', code)
        if op and len(op[0])==2:
            A = int(op[0][0])
            B = int(op[0][1])
            if A!=B:
                code = re.sub('LOC ([0-9]+)\nLOC ([0-9]+)\nMUL', 'LOC ' +
str(A * B), code)

        op = re.findall('LOC ([0-9]+)\nLOC ([0-9]+)\nADD', code)
        if op and len(op[0])==2:
            A = int(op[0][0])
            B = int(op[0][1])
            if A!=B:
                code = re.sub('LOC ([0-9]+)\nLOC ([0-9]+)\nADD', 'LOC ' +
str(A + B), code)
    finally:
        return code

def opr_strength_reduction(code):
    try:
        found = re.findall('LOC 2\nMUL', code)
        if len(found)!=0:
            code = re.sub('LOC 2\nMUL', 'LOC 1\nSHL', code)
    finally:
        return code

def null_seq(code):
    try:
        code = re.sub('ADI 0\n','', code)
        code = re.sub('BEG 0\n','', code)
        code = re.sub('NEG \nNEG','', code)
        code = re.sub('LOC 0\nADD','', code)
        code = re.sub('LOC 0\nSUB','', code)
        code = re.sub('LOC 1\nMUL','', code)
        code = re.sub('LOC 1\nDIV','', code)
    finally:
        return code

def combined_moves(code):
    try:
        op = re.findall('LOV ([0-9]+)\nLOV ([0-9]+)', code)
        if op and len(op[0])==2:
            A = int(op[0][0])
```

```
            B = int(op[0][1])
            if (B-A==2):
                code = re.sub('LOV ([0-9]+)\nLOV ([0-9]+)', 'LDV ' +
str(A), code)

        op = re.findall('LDV ([0-9]+)\nLOV ([0-9]+)', code)

        if op and len(op[0])==2:
            A = int(op[0][0])
            B = int(op[0][1])
            if (B-A==4):
                code = re.sub('LDV ([0-9]+)\nLOV ([0-9]+)', 'LAV ' +
str(A) + '\nLOI 6', code)

    finally:
        return code

def indirect_moves(code):
    try:
        op = re.findall('LAV ([0-9]+)\nLOI 2', code)
        if op:
            A = int(op[0])
            code = re.sub('LAV ([0-9]+)\nLOI 2', 'LOV ' + str(A), code)

        op = re.findall('LAV ([0-9]+)\nSTI 2', code)
        if op:
            A = int(op[0])
            code = re.sub('LAV ([0-9]+)\nSTI 2', 'STV ' + str(A), code)
    finally:
        return code

def reordering(code):
    try:
        op = re.findall('ADD\nLOC ([0-9]+)\nADD', code)
        if op:
            A = int(op[0])
            code = re.sub('ADD\nLOC ([0-9]+)\nADD', 'LOC ' + str(A) +
'\nADD\nADD' , code)

        op = re.findall('ADD\nLOC ([0-9]+)\nSUB', code)
        if op:
            A = int(op[0])
            code = re.sub('ADD\nLOC ([0-9]+)\nSUB', 'LOC ' + str(A) +
'\nSUB\nADD' , code)
    finally:
        return code
```

```
code = '''LOC 3
LOC 5
MUL
LOC 6
ADD
ADI 0
LOC 2
MUL
LOV 2
LOV 4
LAV 2
LOI 2
ADD
LOC 2
ADD'''

con_fold_optimized = constant_folding(code)
opr_str_optimized = opr_strength_reduction(con_fold_optimized)
null_seq_optimized = null_seq(opr_str_optimized)
combined_moves_optimized = combined_moves(null_seq_optimized)
indirect_moves_optimized = indirect_moves(combined_moves_optimized)
reordering_optimized = reordering(indirect_moves_optimized)
print reordering_optimized
```

Example

Input	Output
LOC 3 LOC 5 MUL LOC 6 ADD ADI 0 LOC 2 MUL LOV 2 LOV 4 LAV 2 LOI 2 ADD LOC 2 ADD	LOC 21 LOC 1 SHL LDV 2 LOV 2 LOC 2 ADD ADD

Explanation

SN.	Pattern	Replacement	Reason
1	LOC 3 LOC 5 MUL	LOC 15	Constant Folding
2	LOC 15 LOC 6 ADD	LOC 21	Constant Folding
3	ADI 0	‘ ‘	Null Sequences
4	LOC 2 MUL	LOC 1 SHL	Operator strength reduction
5	LOV 2 LOV 4	LDV 2	Combined moves
6	LAV 2 LOI 2	LOV 2	Indirect moves
7	ADD LOC 2 ADD	LOC 2 ADD ADD	Reordering