A Novel Approach for a Specific Optimization Problem in Compilers

Hai Lin[1a] and Baoliang Mu[1b]

[1]College of Software, Shenyang Normal University 253 Huanghe North Street, Shenyang, 110034, China

[a]jlu_linhai@163.com, [b]87090110@qq.com

Keywords: data flow analysis, constant propagation

Abstract. In this paper, we consider the problem of constant propagation. If some variable can only take a constant value in a given program, then we can replace all the occurrences of that variable with the constant value. That does not affect the semantics of the program, the program can be accelerated at run time. In order to present our method, first we propose a tiny language. We give the full syntax and semantics of that language. All the optimization is done on the source code of this language. We then present our optimization method as a set of inference rules.

Introduction

Code optimization techniques are widely used in compilers [1,2]. The goal of code optimization is to produce more efficient code. In other words, code optimization transforms the original program to some other program. The minimum requirement is that the transformation must preserve the semantics of the program. And the transformed program runs more faster than the original program.

Many forms of code optimizations are done in compilers, e.g. dead code elimination, constant propagation [3,4,5,6]. In this paper, we consider a specific form of optimization. Here is a very simple example, illustrating the key idea of our optimization. Let us consider the following piece of C code.

```
1    int x;
2    x = 11;
3    y=8;
4    z= 283*x + y*92;
```

Figure 1

In the above code fragment, x and y are both constants. That means we can replace x and y with their specific values in the code. As a consequence, z becomes a constant. The replacements do not change the semantics of the program. The advantage is that the resulting program runs faster.

The rest of the paper is structured as follows. In the next section, we introduce a tiny language. All the optimization is done on the source code level of this tiny language. And then we present our optimization method in the form of inference rules. Finally we conclude in the last section.

A Tiny Language

For simplicity, we design a new tiny language and present our optimization technique for this language. This language has 5 kinds of statements, assignment statement, conditional statement, while statement, output statement, and compound statement. In this new language, variables can be used without declarations, variables are implicitly integers. We throw away function calls, pointers, and Boolean expressions, for simplicity. Our optimization method can be easily generalized to real-world programs.


```
program         :   statementSeq
;
statementSeq    :   statementSeq statement
|   statement
;
statement       :   assignmentStmt
|   conditionalStmt
|   whileStmt
|   outputStmt
|   compoundStmt
;
assignmentStmt  :   IDENTIFIER ASSIGN expression SEMI
;
conditionalStmt :   IF LPAREN condition RPAREN statement ELSE statement
;
whileStmt       :   WHILE LPAREN conditional RPAREN statement
;
outputStmt      :   PRINT IDENTIFIER SEMI
;
compoundStmt    :   LCURLY statementSeq RCURLY
;
```

Figure 2 demonstrates a piece of code in our tiny language, which computes the product of all the integers between 1 and 10, and outputs the result.

```
total = 1;
num = 1;
while(num < 11){
    total = total * num;
    num = num + 1;
}
print total;
```

Figure 2

Our Optimization Method

In this section, we present our optimization method in the form of inference rules. Here is the basic idea. We keep track of the number of values for each variable in scope. We use x=B to denote that x can take no value, we use x=C to denote that x can take a constant value C, and we use x=T to denote that x can take more than one values at a particular program point.

We define a binary operator $\oplus$ on {B, C, T}, which is as follows.

Table 1

x	y	$x \oplus y$
B	B	B
B	C	B
B	T	B
C	C	C
C	T	T
T	T	T

At each program point, we keep a mapping M, which maps from each variables in scope to its values from {B, C, T}. We use M(x) to denote the value of x under the mapping M. We can use a substitution to modify the mapping, which is defined as follows.

M[x=V](x) = V
M[x=V](y) = M(y), if x≠y.

And finally M1 ⊕ M2 is defined such that for all variables x, M1 ⊕ M2(x) = M1(x) ⊕ M2(x). We use the following inference rules to compute the mapping M at each program point. Rule 1 means that if before x gets a constant value C the mapping is M, then after that statement M becomes M[x=C]. Rule 2 means that if before x gets an expression involving y1, y2,…,yn, the mapping is M, then after that statement M becomes M[x= y1 ⊕ y2 ⊕ … ⊕ yn]. Rule 3 means that at any merge point M1, M2 becomes M1 ⊕ M2.

M; x=C; M[x=C] (Rule 1)
M; x=e{y_1, y_2,…,y_n}; M[x= $y_1 \oplus y_2 \oplus \ldots \oplus y_n$] (Rule 2)
M_1, M_2 ➔ $M_1 \oplus M_2$ (Rule 3)

We can use the above rules to compute the number of values for each variables at all program points. If at some program point, a variable x can only take one constant value C, then we can use C to replace x at that program point.

Conclusions

In this paper, we proposed a new method for constant propagation. This method is useful for produce faster code without changing the semantics of the program. We designed a tiny programming language and present our optimization method for this language. We use a set of inference rules to compute the number of values for each variable at all program point. If at some program point, a value can only take a constant value, then we can replace the variable with that constant value.

Acknowledgement

This work is supported by Liaoning Provincial Natural Science Foundation under grant 201202202, Scientific Research Foundation of Liaoning Provincial Education Department under grant L2012388.

References

[1] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2008.

[2] Traub, O., Schechter, S., and Smith, M. 2000. Ephemeral Instrumentation for Lightweight Program Profiling. Tech. rep., Harvard University.

[3] Bala, V., Duesterwald, E., and Banerjia, S. 1999. Transparent Dynamic Optimization: The Design and Implementation of Dynamo. Tech. Rep. HPL-1999-78, Hewlett Packard Laboratories. June.

[4] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pages 1-8, March 2007.

[5] Fisher, J. A. 1981. Trace Scheduling: A Technique for Global Microcode Compaction. IEEE Trans. Comput. C-30, 7, 478--490.

[6] Mustafa, D.; Eigenmann, R. "Portable section-level tuning of compiler parallelized applications", High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for, On page(s): 1 – 11