ISSN No. 0976-5697

Volume 4, No. 8, May-June 2013

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

A Few Good Pattern to Optimize Compiler

Subhendu Guha Roy
Academic Counselor
SOCIS, IGNOU Siliguri SC:2805
IGNOU,Sevoke Road, Siliguri
Dt.-Darjeeling.WB,India
shubha100@live.in

Abstract: Optimization of code is the term that was applied to a process in which a code is tuned to be better in some respects: either speed, memory consumption, Input/output (disk read and writes or network reads and writes), etc. In Mathematics, Optimization means a process in which one finds the values with the best performance. In Computing where programs are very complex, usually optimizing for speed in the mathematical sense is impossible. Instead the term has come to mean just advancing in the direction of better performance in one or more respects. This document will focus on optimizing code to run faster. However, as you will see later, doing this may involve having to optimize the code in a different aspect. Furthermore, often when programmers are trying to optimize one aspect of a program, they are doing so in order to increase speed.

Keywords: compiler, optimization, reduce and faster code, byte code, high performance, abstract quality.

I. INTRODUCTION

In computing and computer science, an optimizing compiler[5] is a compiler that tries to minimize or maximize some attributes of an executable computer program. The most common requirement is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied. The growth of portable computers has created a market for minimizing the power consumed by a program. Compiler optimization is generally implemented using a sequence of *optimizing transformations*, algorithms which take a program and transform it to produce a semantically equivalent output program that uses fewer resources.

It has been shown that some code optimization problems are NP-complete, or even undecidable. In practice, factors such as the programmer's willingness to wait for the compiler to complete its task place upper limits on the optimizations that a compiler implementor might provide. (Optimization is generally a very CPU- and memory-intensive process)[1]. In the past, computer memory limitations were also a major factor in limiting which optimizations could be performed. Because of all these factors, optimization rarely produces "optimal" output in any sense, and in fact an "optimization" may impede performance in some cases; rather, they are heuristic methods for improving resource usage in typical programs.

So, the mean objective of the performance improving is to write the code in such a way that memory and speed both optimize. Several different options exist for measuring application performance[5]. Homegrown timing functions inserted into the code are a more effective way to gather performance data. Other most efficient and accurate ways to gather timing data is to use a good performance profiler, which show the time spent in each function of the program and will also provide an analyses based on this data[1,5].

II. TYPES OF OPTIMIZATIONS

Techniques used in optimization can be broken up among various *scopes* which can affect anything from a single statement to the entire program. Generally speaking, locally scoped techniques are easier to implement than global ones but result in smaller gains[4]. Some examples of scopes include:

a. Peephole optimizations:

Usually performed late in the compilation process after machine code has been generated. This form of optimization examines a few adjacent instructions (like "looking through a peephole" at the code) to see whether they can be replaced by a single instruction or a shorter sequence of instructions. For instance, a multiplication of a value by 2 might be more efficiently executed by left-shifting the value or by adding the value to itself[1]. (This example is also an instance of strength reduction.)

b. Local optimizations:

These only consider information local to a basic block. Since basic blocks have no control flow, these optimizations need very little analysis (saving time and reducing storage requirements), but this also means that no information is preserved across jumps.

c. Global optimizations:

These are also called "intra procedural methods" and act on whole functions.[2] This gives them more information to work with but often makes expensive computations necessary[3]. Worst case assumptions have to be made when function calls occur or global variables are accessed (because little information about them is available).

d. Loop optimizations:

These act on the statements which make up a loop, such as a *for* loop (e.g., loop-invariant code motion). Loop optimizations can have a significant impact because many

programs spend a large percentage of their time inside loops[1].

Interprocedural, whole-program or link-time optimization

These analyze all of a program's source code. The greater quantity of information extracted means that optimizations can be more effective compared to when they only have access to local information (i.e., within a single function). This kind of optimization can also allow new techniques to be performed. For instance function inlining, where a call to a function is replaced by a copy of the function body[5].

e. Machine code optimization:

These analyze the executable task image of the program after all of a executable machine code has been linked. Some of the techniques that can be applied in a more limited scope, such as macro compression (which saves space by collapsing common sequences of instructions), are more effective when the entire executable task image is available for analysis.

A. Some Experiments And Result Analysis

Here I depicted some code optimization techniques with implementation of C code. Although I use C syntax in the examples below, these techniques clearly apply to other languages just as well[2].

a. An example of elimination common sub expression:

Statement of expression is

M=A*LOG(Y)+(LOG(Y)**2)

Elimination common sub expression, introducing an explicit temporary variable t:

t=LOG(Y)

M=A*t+(t**2)

Saves one 'heavy' function call, by an elimination of the common sub-expression LOG(Y), the exponentiation now is:

M = (A + t) * t

b. Declare local functions as "static"

Doing so tells the compiler that the function need not be so general as to service arbitrary general calls from unrelated modules. If the function is small enough, it may be inlined without having to maintain an external copy[3]. If the function's address is never taken, the compiler can try to simplify and rearrange usage of it within other functions.

Before:

```
void swap(int *a, int *b) {
int t;
    t = *b;
    *b = *a;
    *a = t;
 }
```

After:

```
static void swap(int *a, int *b) {
int t;
    t = *b;
    *b = *a;
    *a = t;
 }
```

c. Remove unnecessary if then Else statement:

Let's takes two examples to remove unnecessary if-else statement[4]. It could be simplified to enhance the code's efficiency and reduce its size.

```
void main()
{
boolean b;
void  boolean()
{
if (b)
{
return true;
}
else
{
return false;
}
}
}
```

Should be written as:

```
Void main()
{
boolean b;
void boolean ()
{
return b;
}
}
```

With else, smaller code, but slower one

```
inline int
test(int a)
{
        return a > 0 ? 1 : 0;
}
```

Without else, large code but faster one,

```
 inline int
test(int a)
{
        if (a > 0)
                return 1;
        /* implied else */
        return 0;
}
```

Let's take another,

The slowest expression, compiling and running

```
int
max(int a, int b)
{
        if (a > b)
                return a;
        else
                return b;
}
```

Normal expression with inlining,

```
inline int
max(int a, int b)
{
        return ((a > b) ? a : b);
}
```

d. "Else" clause removal

The performance of if-then-else is one taken jump

no matter what. However, often the condition has a lop-sided probability in which case other approaches should be considered. The elimination of branching is an important concern with today's deeply pipelined processor architectures. The reason is that a "mispredicted" branch often costs many cycles[8].

Before:

```
if( Condition ) {
   Case M;
} else {
   Case N;
}
```

After:

```
Case N;
if( Condition ) {
   Undo Case N;
   Case M;
}
```

Clearly this only works if **Undo *Case N;*** is possible. However, if it is, this technique has the advantage that the jump taken case can be optimized according to the *Condition* probability and **Undo *Case N; Case M;*** might be merged together to be more optimal than executing each separately[7].

Obviously you would swap cases M and N depending on which way the probability goes[7,8]. Also since this optimization is dependent on sacrificing performance of one set of circumstances for another, you will need to time it to see if it is really worth it. (On processors such as the ARM or Pentium II, you can also use conditional *mov* instructions to achieve a similar result.)

e. Use finite differences to avoid multiplies:

Before:

```
for(k=0;k<10;k++) {
   printf("%d\n",k*10);
}
```

After:

```
for(k=0;k<100;k+=10) {
   printf("%d\n",k);
}
```

This one should be fairly obvious, use constant increments instead of multiplies if this is possible. (Believe it or not, however, some C compilers are clever enough to figure this out for you in some simple cases.)

f. Rearrange an array of structures as several arrays:

Instead of processing a single array of aggregate objects[6], process in parallel two or more arrays having the same length.

For example, instead of the following code:

```
const int n = 10000;
struct { double a, b, c; } s[n];
for (int i = 0; i < n; ++i) {
   s[i].a = s[i].b + s[i].c;
}
```

the following code may be faster:

```
const int n = 10000;
double a[n], b[n], c[n];
for (int i = 0; i < n; ++i) {
   a[i] = b[i] + c[i];
}
```

Using this rearrangement, "a", "b", and "c" may be processed by array processing instructions that are significantly faster than scalar instructions. This optimization may have null or adverse results on some (simpler) architectures.

g. Data type considerations:

Often to conserve on space you will be tempted to mix integer data types; chars for small counters, shorts for slightly larger counters and only use longs or ints when you really have to[4]. While this may seem to make sense from a space utilization point of view, most CPUs have to end up wasting precious cycles to convert from one data type to another, especially when preserving sign[3].

Before:

```
char a;
int b;
   b = a;
```

After:

```
int a, b;
   b = a;
```

B. A case of Copy Propagation:

This optimization is similar to constant propagation, but generalized to non-constant values. If we have an assignment **m = n** in our instruction stream, we can replace later occurrences of **m** with **n** (assuming there are no changes to either variable in-between)[2,3]. Given the way we generate TAC code, this is a particularly valuable optimization since it is able to eliminate a large number of instructions that only serve to copy values from one variable to another. The code on the left makes a copy of **a** in **b** and a copy of **c** in **d**. In the optimized version on the right, we eliminated those unnecessary copies and propagated the original variable into the later uses:

```
before:          After Copy Propagation:
b=a;                c=a*a;
c=b*a;              e=c*a;
d=c;                x=e +c;
e=c*b;
x=e + d;
```

III. CONCLUSION

Recent research in code optimization has led to the development of unified optimizing transformations like the generalized code movement transformation of Dhamdhere-Isaac [6] and Morel-Renvoise[9], and the composite hoisting-and-strength reduction transformation of Dhamdhere-Isaac[8] and Joshi-Dhamdhere [9]. Using a good compiler and some knowledge about optimization techniques, developers can much more easily create high performance applications. This alternate recompilation does not affect the correctness of the application because all compilers should be generating correct bytecodes, which means that such a situation allows the application to pass all regression test suites. But you can end up with the production application not running as fast as you expect for reasons that are difficult to track down. Speed and optimizing other resources are one important factor in the

general, abstract quality of programs[4]. If your program is slow, it is likely going to make your users frustrated and unhappy, which will be a failure in your mission as a software developer. So it is important that your program is fast enough, if not very much so[9].

IV. REFERENCES

[1]. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers, Principles, Techniques and Tools. Addison-Wesley, 1988.

[2]. F.E. Allen. Interprocedural data flow analysis. Proceedings of IFIP Congress 1974, pp.398–402. North Holland, 1974.

[3]. John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, pp. 29–41, January 1979.

[4]. S. Muchnick, Advanced Compiler Design and Implementation. San Francisco, CA: Morgan Kaufmann, 1997.

[5]. Optimizing compiler - Wikipedia, the free encyclopedia.mht and Programming Optimization Techniques, examples and discussion.mht.

[6]. CS143 Handout 20, Summer 2008 August 04, 2008, Code Optimization, Handout written by Maggie Johnson.

[7]. Agarwal, R. Metamorphic, authenticated models for neural networks. Journal of Cooperative, Trainable Methodologies 656 (Feb. 1997), 71-88.

[8]. http://www.azillionmonkeys.com/qed/ optimize.html.

[9]. Iverson, K., Kobayashi, L., and Gupta, a. NOSLE: Synthesis of public-private key pairs. In Proceedings of PODS (Jan. 2001).

Author: Subhendu Guha Roy, M.Sc,MCA, have approx. 5 years and above of teaching and counseling experiences in various colleges along with IGNOU,Siliguri Study Centre(2805). He is holding membership of IASTED, IACSIT, IAENG, etc and published some my paper in national and international journal and conferences.