© 2014 IJIRT | Volume 1 Issue 5 | ISSN : 2349-6002

SIGNIFICANCE OF CODE OPTIMISATION

Priya Kumari, Neha Agrawal
Student, Department Of Information Technology, Dronacharya College Of Engg.

***Abstract*- Optimization is the field where most compiler research is prepared today. The tasks of the Front-end (scanning, parsing, semantic analysis) are well implicit and optimized Code generation is moderately straightforward. In this paper we have given a brief introduction on criteria for code improvement in which we have discussed about Correctness of the code, When and Where To Optimize etc. likewise topics like the principal sources of optimization, code motion, Operator Strength Reduction etc. High-quality optimization is more of an art than a science. Compilers for mature languages aren't judged by how well they analyse or analyse the code—you just suppose it to do it right with a minimum of hassle—but Instead by the quality of the object code they produce. These developments can result in reduced run time and/or space for the object program.**

I. INTRODUCTION

Optimization is still a sizable amount of theology. High-quality optimization is more of an art
than a science. Compilers for mature languages aren't refereed by how well they parse
or analyse the code—you just suppose it to do it right with a minimum of hassle.
Many optimization difficulties are NP-complete and thus most optimization algorithms
rely on heuristics and estimates. It may be possible to come up with a case where
a particular algorithm fails to create better code or perhaps even makes it worse.
However, the algorithms tend to do relatively well overall.
Consider the following two code snippets
where each walks through an array and set every element to one. Which one is faster?

```
int arr[10000];
void Binky() {
int i;
for (i=0; i < 10000; i++)
arr[i] = 1;
}
int arr[10000];
void Winky() {
register int *p;
for (p = arr; p < arr + 10000; p++)
*p = 1;
}
```

You will perpetually encounter people who think the second one is faster. And they are
Probably right….if using a compiler without optimization. But, many modern
Compilers produce the same object code for both, by use of clever techniques (in particular, this one is called "loop-induction variable elimination") that effort particularly well on
Dogmatic usage.

II. CRITERIA FOR CODE IMPROVEMENT

2.1 Correctness of the code

It is obvious that optimization should not change the correctness of the generated code. Transforming the code to something that runs faster but incorrectly is of no value. It is anticipated that the unoptimized and optimized variants give the identical output for all inputs. This may not hold for an incorrectly written program.

2.2 When and Where To Optimize

There are a many tactics for attacking optimization. Some techniques are applied to intermediate code, to restructure, rearrange, compress, etc. in an effort to reduce the size of the abstract syntax tree or shrink the number of TAC instructions. Others are applied as part of final code generation—choosing which instructions to emit, how to allocate registers and when/what to drop. And still other optimizations
may occur after final code generation, attempting to redraft the assembly code itself into something more resourceful.

i. **Profitability :-**Optimization leads to improvement in code quality.

ii. **Others :--** A transformation must, on the average, speed up a program by a measurable amount. Code-optimization for the code that

run rarely or during debugging should be avoided.

iii. Factors affecting optimization

2.2.3.1 The machine itself

Decision about which optimizations should be done depend on the characteristics of the target machine. It is sometimes possible to parameterize some of these machine dependent factors, so that a single piece of compiler code can be used to optimize different machines just by changing the machine description parameters.

2.2.3.2 The architecture of the target CPU

Number of CPU registers, affect the choice of optimization. To a certain extent, the more registers, the easier it is to optimize on the basis of performance. Local variables could be allocated in the registers and not on the stack. Temporary/intermediate results can be left in registers without writing to and reading back from memory.

RISC versus CISC: CISC instruction sets often have a larger number of possible instructions that can be used, and each instruction could take differing amounts of time. RISC instruction sets are usually constant length, with few exceptions, there are usually fewer combinations of registers and memory operations.

2.2.3.3 The architecture of the machine

Cache_size (256 kiB–12 MiB) and type (direct mapped, 2-/4-/8-/16-way associative, fully associative) affects optimization. Techniques such as inline expansion and loop unrolling may increase the size of the generated code and decrease code locality. The program may slow down drastically if a highly utilized section of code (like inner loops in various algorithms) cannot fit in the cache memory. Also, caches which are not fully associative have higher likelihood of cache collisions even in an unfilled cache.

iv. Categories 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. Some examples of scopes include:

2.3 Local Optimizations

Optimizations performed solely within a basic block are called "local optimizations". They are typically the easiest to perform since we never consider any control-flow information, we just work with the statements within the block. Many of the local optimizations have corresponding global optimizations that operate on the same principle, but require additional analysis to perform. ***Local optimization*** techniques normally are concerned with transformations on small sections of code (involving only a few instructions) and usually operate on the machine language instructions which are formed by the code generator.

2.4 Global Optimizations

Optimizations which are generally concerned with larger blocks of code, or even multiple blocks or modules are called "global optimizations". Worst case assumptions have to be made when function calls occur or global variables are accessed (because little information about them is available).

2.5 Inter-procedural optimizations

Inter-procedural optimization mechanism on the whole program, across procedure and file boundaries. It is accepted out with the cooperation of a local part and global part. Characteristic inter-procedural optimizations are: procedure inlining, inter-procedural dead code elimination, inter-procedural constant propagation, and procedure reordering. As usual, the compiler needs to accomplish inter-procedural analysis before its actual optimizations. Inter-procedural analyses include alias analysis, array access analysis, and the creation of a call graph.

Due to the extra time and space required by inter-procedural analysis, most compilers do not perform it by default. Users must use compiler options clearly to tell the compiler to enable inter-procedural analysis and other exclusive optimizations.

2.6 Register Allocation

One machine optimization of particular position is register allocation, which is
perhaps the only most effective optimization for all architectures. Registers are the
Fastest kind of memory available, but as a resource, they can remain scarce. The problem is
How to reduce traffic between the registers and what lies beyond them in the
Memory hierarchy to eliminate time wasted sending data back and forth through the bus
And the different levels of caches.

2.7 Instruction Scheduling

Another particularly important optimization of the final code generator is instruction
scheduling. Because many machines, with most RISC architectures, have some
Sort of pipelining capability, effectively harnessing that capability needs judicious
Ordering of instructions.
In MIPS, each instruction is issued in one cycle, but some take several cycles to
Complete. It takes an additional cycle before the value of a load is accessible and two
Cycles for a branch to reach its destination, but an instruction can be located in the "delay
Slot" after a branch and executed in that slack time. On the left is one procedure of a
Set of instructions that requires 7 cycles. It accepts no hardware interlock and thus
Explicitly stalls between the second and third slots while the load concludes and has a
15
dead cycle after the branch because the delay slot holds a noon. On the right, a more
Favorable rearrangement of the same guidelines will execute in 5 cycles with no dead
cycles.

```
lw $t2, 4($fp)
lw $t3, 8($fp)
noop
add $t4, $t2, $t3
subi $t5, $t5, 1
goto L1
```

III. THE PRINCIPAL SOURCES OF OPTIMIZATION

There are several ways in which a compiler can improve a program without affecting the function it computes. Following are function-preserving transformations:

1. Code Motion
2.Operator Strength Reduction
3. Dead-Code Elimination
4. Algebraic Simplification And Reassociation

IV. CODE MOTION

Code motion (also called *code hoisting*) combines sequences of code common to one or more
basic blocks to reduce code size and possibly avoid expensive re-evaluation. The
most common form of code motion is *loop-invariant* code motion that transfers statements
That evaluate to the same value every iteration of the loop to somewhere outside the loop. What declarations inside the following TAC code can be moved outside the loop
body?

```
L0:
tmp1 = tmp2 + tmp3 ;
tmp4 = tmp4 + 1 ;
PushPram tmp4 ;
LCall _PrintInt ;
PopParams 4;
tmp6 = 10 ;
tmp5 = tmp4 == tmp6 ;
IfZ tmp5 Goto L0 ;
```

We have an intuition of what makes a loop in a flowgraph, but here is a more recognized
definition. A loop is a set of basic blocks which mollifies two conditions:

1. All are *strongly connected*, i.e. there is a path between any two blocks.
2. The set has a unique *entry point*, i.e. every path from outside the loop that spreads
any block inside the loop enters done a single node. A block n *dominates* m if
all paths from the starting block to m must travel complete n. Every block
dominates itself.

13

For loop L, moving invariant statement s in block B which describes variable v outside the
loop is a safe optimization if:

1. B governs all exits from L
2. No other statement allocates a value to v
3. All uses of v inside L are from the definition in s.

Loop invariant code can be moved towards just above the entry point to the loop.

V. OPERATOR STRENGTH REDUCTION

Operator strength reduction changes an operator by a "less expensive" one. Given each
group of identities below, which operations are the most and least costly, assuming
f is a float and **i** is an int? (Trick question: it entirely depends on the architectures—you
need to know your target machine to optimize well!)

```
i*2 = 2*i = i+i = i << 1
i/2 = (int)(i*0.5)
0-1 = -i
f*2 = 2.0 * f = f + f
```

8

```
f/2.0 = f*0.5
```

Strength reduction is often executed as part of *loop-induction variable elimination*. An

idiomatic loop to zero all the elements of an array capacity look like this in Decaf and its
Corresponding TAC:

```
while (i < 100) {
arr[i] = 0;
i = i + 1;
}
L0:_tmp2 = i < 100;
IfZ _tmp2 Goto _L1 ;
_tmp4 = 4 * i ;
_tmp5 = arr + _tmp4 ;
*(_tmp5) = 0 ;
i = i + 1 ;
L1:
```

Each time finished the loop, we multiply i by 4 (the element size) and add to the array
base. Instead, we could retain the address to the current element and instead just
add 4 each time:

```
_tmp4 = arr ;
L0:_tmp2 = i < 100;
IfZ _tmp2 Goto _L1 ;
*_tmp4 = 0;
_tmp4 = _tmp4 + 4;
i = i + 1 ;
L1:
```

This eliminates the multiplication entirely besides reduces the need for an extra temporary.
By re-writing the loop termination test in terms of art, we might remove the variable i
Entirely and not bother tracking and incrementing it at all.

- **Dead Code Elimination**

If an instruction's result is never used, the instruction is measured "dead" and can be
removed from the instruction stream. So if we have

```
tmp1 = tmp2 + tmp3 ;
```

and **tmp1** is never used again, we can remove this instruction altogether. However, we
have to be a little suspicious about making assumptions, for example, if **tmp1** holds the
Result of a function call:

```
tmp1 = LCall _Binky;
```

Even if **tmp1** is never used again, we cannot remove the instruction because we can't
be sure that called function has no side-effects. Dead code can happen in the original
source program but is more likely to have caused from some of the optimization
Techniques run previously.

For example, the program in under contains an assignment to the variable which has no effect on the output since a is not used subsequently, but prior to extra assignment to the variable a.

```
{
a = b + c *d; //This statement has no effect and can be removed
b = c*d/e;
c = b - 3;
a = b - c;
cout << a << b << c ;
}
```

- **Algebraic Simplification And Reassociation**

Simplifications use algebraic properties or specific operator-operand combinations to
simplify expressions. Reassociation refers to using things such as associativity,
commutativity and distributives to rearrange an expression to allow other
optimizations such as constant-folding or loop-invariant code motion.
The most clear of these are the optimizations that can remove useless instructions
entirely via algebraic identities. The rules of arithmetic can come in nearby when
looking for redundant calculations to remove. Consider the examples below, which
allow you to replace an expression on the left with a humbler equivalent on the right:

```
x+0 = x
0+x = x
x*1 = x
1*x = x
0/x = 0
x-0 = x
b && true = b
b && false = false
b || true = true
b || false = b
```

The use of algebraic rearrangement can rearrange an expression to enable constantfolding
or corporate sub-expression elimination and so on. Consider the Decaf code on
the far left, unoptimized TAC in middle, and reorganized and constant-folded TAC on
far right:

```
b = 5 + a + 10 ; _tmp0 = 5 ;
_tmp1 = _tmp0 + a ;
_tmp2 = _tmp1 + 10 ;
b = _tmp2 ;
```

```
_tmp0 = 15 ;
_tmp1 = a + _tmp0 ;
b = _tmp1 ;
```

- **Shortcomings of optimization**

Optimizing compilers are by no means perfect. There is no way that a compiler can guarantee that, for all program source code, the fastest (or smallest) possible equivalent compiled program is output; such a compiler is fundamentally impossible because it would solve the halting problem.

There are a number of other more practical issues with optimizing compiler technology:

- Optimizing compilers focus on relatively trivial constant-factor performance improvements and do not typically improve the algorithmic complexity of a solution. For example, a compiler will not change an implementation of bubble sort to use merge sort instead.
- Compilers usually have to support a variety of contradictory objectives, such as cost of implementation, compilation speed and quality of generated code.
- A compiler typically only deals with a part of a program at a time, often the code contained within a single file or module; the result is that it is unable to consider relative information that can only be obtained by processing the other files.
- The overhead of compiler optimization: Any extra work takes time; whole-program optimization is time consuming for big programs.

- **Optimization Soup**

You might wonder about the interactions among the various optimization techniques.

Some transformations may description possibilities for others, and even the reverse is true,

one optimization may unclear or remove possibilities for others. Algebraic

16 rearrangement might allow for common subexpression elimination or code motion.

Constant folding usually paves the way for constant propagation and then it goes out to be valuable to run another round constant-folding and so on.

CONCLUSION

As compiler technologies have improved, good compilers can often generate better code than human programmers.

Work to improve optimization technology continues. One approach is the use of so-called post-pass optimizers. A good post pass optimizer can improve highly hand-optimized code even further. These tools take the executable output by an "optimizing" compiler and optimize it even further. Post pass optimizers usually work on the assembly language or machine code level (contrast with compilers that optimize intermediate representations of programs). Performance of post pass compilers are restricted by the fact that much of the information available in the original source code is not always accessible to them.

Optimizations often interact with each other and need to be combined in precise ways. Some optimizations may need to be applied multiple times. E.g., dead code elimination, redundancy elimination, copy folding.

REFRENCES

A. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Reading, MA: Addison-Wesley, 1986.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

R. Mak, Writing Compilers and Interpreters. New York, NY: Wiley, 1991.

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

Pyster, Compiler Design and Construction. Reinhold, 1988

Wikipedia