

Intro to Optimization

CS 471
November 26, 2007



Code Optimization

Definition: A transformation that we expect to:

1. Improve the running time of the program
2. Decrease its space requirements, or
3. Decrease power consumption

Misnomer

- Produce “improved” code, not “optimal” code
- Can sometimes produce worse code

Classical optimizations

- Reduce the number of instructions
- Reduce the cost of instructions

Modern optimizations

- Change the ordering of instructions (latency)
- Execute instructions in parallel
- Modify data placement (registers, cache, memory)
- Reduce power consumption

1 CS 471 – Fall 2007

Code Optimization

How can optimizations improve code quality?

Machine independent transformations

1. Replace a redundant computation with a reference
2. Move evaluation to a less-frequently executed location
3. Specialize some general-purpose code
4. Expose an opportunity for another optimization

Machine dependent transformations

1. Replace a costly operation with a cheaper one
2. Replace a sequence of instructions with a more powerful one
3. Eliminate or tolerate latency
4. Lower power consumption

2 CS 471 – Fall 2007

Why Do We Need Optimization?

- Programmers don’t always write optimal code
- High-level languages may not allow programmer to avoid redundant computation

$$a[i][j] = a[i][j] + 1$$

- Architectural independence -- don’t have to understand machine
- Modern architectures assume optimization -- too hard to optimize by hand

3 CS 471 – Fall 2007

Where Should We Optimize?

- **Usual goal:** improve time performance
- **Problem:** many optimizations trade off space versus time (example: loop unrolling)
- Increasing code space slows program down a little, speeds up one loop
- Frequently executed code: space/time tradeoff is generally a win
- Infrequently executed code: may want to optimize code space at expense of time
- Complex optimizations may never pay off

4 CS 471 – Fall 2007

Scope of Optimization

Local (or single block)

- Confined to straight-line code
- Simplest to analyze

Intraprocedural (or global)

- Consider the whole procedure

Interprocedural (or whole program)

- Consider the whole program

5 CS 471 – Fall 2007

Analysis and Transformation

Most optimizations require some global understanding of program flow

- Moving, removing, rearranging instructions

Achieve understanding by discovering the control flow of the procedure

- What blocks follow/are reachable from other blocks
- Where loops exist (focus optimization efforts)
- We call this **Control-Flow Analysis**

Also, find and connect definitions and uses of variables

- We call this **Data-Flow Analysis**

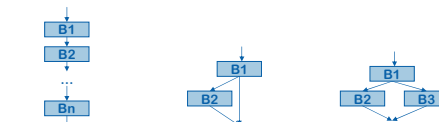
All approaches to control-flow analysis require

- Identification of basic blocks
- Construction of a flow graph of the procedure

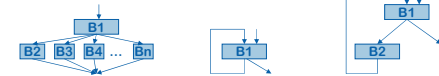
6

CS 471 – Fall 2007

Common Region CFGs



Straight-line code



7

CS 471 – Fall 2007

Structure of an Optimization

Optimization is a code transformation

Applied at some compilation stage (HIR, MIR, LIR)

In general requires some analysis:

- safety analysis to determine where transformation does not change meaning (e.g. live variable analysis)
- cost analysis to determine where it should speed up code (e.g. which variable to spill)

8

CS 471 – Fall 2007

When to Apply Optimization

HIR	AST	Inlining Specialization
	IR	Constant folding Constant propagation Value numbering
MIR	Canonical IR	Dead code elimination Loop-invariant code motion Common sub-expression elimination Strength reduction
	Abstract Assembly	Branch prediction/optimization Register allocation
LIR	Assembly	Loop unrolling Cache optimization

9

CS 471 – Fall 2007

Constant Folding

Idea: if operands are known at compile time, evaluate at compile time.

$\text{int } x = (2 + 3)*y; \Rightarrow \text{int } x = 5*y;$

Performed at various stages during compilation as constant expressions are created (by translation or optimization)

$a[2] \Rightarrow \text{MEM}(\text{MEM}(a) + 2*4)$
 \Rightarrow _____

10

CS 471 – Fall 2007

Constant Folding Conditionals

- > if (true) S \Rightarrow _____
- > if (false) S \Rightarrow _____
- > if (true) S else S' \Rightarrow _____
- > if (false) S else S' \Rightarrow _____
- > while (false) S \Rightarrow _____
- > if (2 > 3) S \Rightarrow _____

11

CS 471 – Fall 2007

Constant Propagation

- If value of variable is known to be a constant, replace use of variable with constant
- Value of variable must be propagated forward from point of assignment

```
int x = 5;
int y = x*2;
int z = a[y]; // = MEM(MEM(a) + y*4)
```

- For full effect, interleave w/ constant folding

12 CS 471 – Fall 2007

Conditional Constant Propagation

Is *j* always equal to 1?

Simple constant propagation missed this opportunity!

13 CS 471 – Fall 2007

Dead-Code Elimination

- If side-effect of a statement can never be observed, can eliminate the statement

```
int x = y*y; // dead!
... // x not used
x = z*z;
```

- Variable is **dead** if never used after def.

```
int i;
while (m < n) { m++; i = i+1; }
```

- Data-flow analysis can determine deadness
- Other optimizations will introduce dead code

14 CS 471 – Fall 2007

Dead-Code Elimination

Dead code is either:

- Unreachable code
- Assignments where the result is never used

Examples

- "y in 1" is _____
- "x in 1" is _____
 - i.e. along path 1-2-4 but not 1-3-4
- "z in 4,5" is _____
 - i.e. never used in relevant computations
 - here: only x, y are relevant

15 CS 471 – Fall 2007

Unreachable Code Elimination

- Basic blocks not contained by any trace leading from starting basic block are **unreachable** and can be eliminated
- Performed at canonical IR or assembly code levels

16 CS 471 – Fall 2007

Copy Propagation

- Given assignment $x = y$, replace subsequent uses of x with y
- May make x a dead variable, result in dead code
- Data-flow analysis needed to determine where copies of y propagate

17 CS 471 – Fall 2007

Common Sub-Expression Elimination

- CSE recognizes the same expression being computed in multiple places, creates a temporary to hold value of expression and reuse it

```
a[ i ] = a[ i ] + 1
[[a]+i*4] = [[a]+i*4] + 1
⇒ t1 = [a] + i*4; [t1] = [t1]+1
```

- Data-flow analysis needed to decide that expression has same value in both places

18

CS 471 – Fall 2007

Strength Reduction

- Replaces expensive operations (multiplies, divides) by cheap ones (adds, subtracts)

```
x = y * 4      ⇒ _____
ADDI r1, r1, 1 ⇒ _____
```

19

CS 471 – Fall 2007

Strength Reduction for Loops

- Replaces expensive operations by cheap ones) by creating **dependent induction variable**

```
for (int i = 0; i < n; i++) {
    a[i*3] = 1;
}
↓
int j = 0;
for (int i = 0; i < n; i++) {
    a[ j ] = 1;
    j = j+3;
}
```

20

CS 471 – Fall 2007

Loop-Invariant Code Motion

- If result of a statement or expression does not change during loop, and it has no side-effect, can move its computation outside loop
- Often useful for array element addressing computations -- not visible in source code

```
while (b) {
    z = y/x; // x, y not
    ...     // assigned
}           // in loop
→
z = y/x;
while (b) {
    ...
}
```

- Requires data-flow analysis to identify loop-invariant expressions

21

CS 471 – Fall 2007

Loop Unrolling

Branches are expensive; **unroll** loop to avoid them

```
for (i = 0; i < n; i++) { S }
↓
int m = n - n%4;
for (i = 0; i < m; i+=4) {S; S; S; S; }
for (i = 0; i < n%4; i++) S;
```

Space-time tradeoff; not a good idea for large S or small n.

22

CS 471 – Fall 2007

Inlining

Replace a call to a function with the body of the function itself:

```
g(x: int):int = 1+ f(x);
f(a: int):int = { b:int=1; n:int = 0;
                while (n<a) { b = 2*b; } b }
↓
g(x:int):int = 1 + {a:int = x; { b:int=1; n:int = 0;
                          while (n<a) { b = 2*b; } b }}
```

May need to rename variables to avoid **name capture** -- consider f refers to a global var x

23

CS 471 – Fall 2007

Summary



- **Many useful optimizations that can transform code to make it faster**
- **Whole is greater than sum of parts:** optimizations should be applied together, sometimes more than once, at different levels
- **Common theme:** need some kind of data-flow analysis to figure out where optimizations are safe

Test Wednesday!
PA7 Due Friday Night

