CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 24: Introduction to Optimization
30 Mar 01

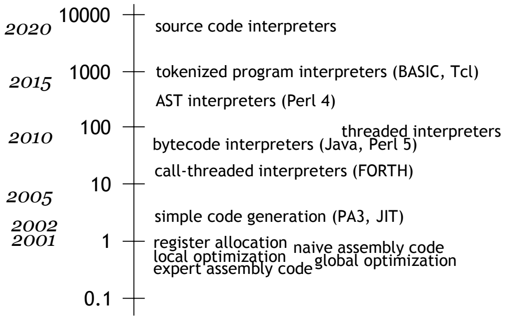
Administration

- Programming Assignment 4 due Wednesday, April 4
- Optional reading: Muchnick 11

Optimization

- This course covers the most valuable and straightforward optimizations – much more to learn!
- Muchnick (optional text) has 10 chapters of optimization techniques

How fast can you go?



Goal of optimization

- Help programmers
  - clean, modular, high-level source code
  - compile to assembly-code performance
- Optimizations are code transformations
  - must be *safe*; can't change meaning of program
- Different kinds of optimization:
  - space optimization: reduce memory use
  - time optimization: reduce execution time

Where to 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 with long loops: 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!
- Want to optimize program *hot spots*

Safety

- Opportunity for *loop-invariant code motion*:

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

- *Hoist* invariant code out of loop:

```
z = y/x;
while (b) {
        …
}
```

Safe?
Faster?

- code transformation
- safety of transformation
- performance improvement

Writing fast programs in practice

- Pick the right algorithms and data structures: reduce operations, memory usage, indirections
- Turn on optimization and *profile* to figure out program hot spots
- Evaluate whether design works; if so…
- Tweak source code until optimizer does “the right thing” to machine code

Structure of an optimization

- Optimization is a code transformation
- Applied at some stage of compiler (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 ought to speed up code (e.g. which variable to spill)

When to apply optimization

HIR
AST
IR

MIR
Canonical IR
Abstract Assembly

LIR
Assembly

Inlining
Specialization
Constant folding
Constant propagation
Value numbering
Dead code elimination
Loop-invariant code motion
Common sub-expression elimination
Strength reduction
Constant folding & propagation
Branch prediction/optimization
Register allocation
Loop unrolling
Cache optimization

Why do we need optimization

- Programmers don’t always write optimal code – can recognize ways to improve code (*e.g.* avoid recomputing same expression)
- High-level language may make avoiding redundant computation inconvenient or impossible

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

- Architectural independence
- Modern architectures assume optimization – too hard to optimize by hand

Register allocation

- Goal: convert abstract assembly (infinite no. of registers) into real assembly (6 registers)

```
mov t1, t2
add t1, [bp-4]
mov t3, [bp-8]
mov t4, t3
cmp t1, t4
```

⇨

```
mov ax, bx
add ax, [bp-4]
mov bx, [bp-8]

cmp ax, bx
```

- Need to reuse registers aggressively (*e.g.,* **bx**)
- Want to coalesce registers (t3, t4) to eliminate **mov**’s
- May be impossible without *spilling* to stack

Constant folding

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

```
int x = (2 + 3)*y;   ⇒   int x = 5*y;
b & false            ⇒   false
```

- Performed at every stage of compilation
  - Constant expressions created by translation or optimization

```
a[2] ⇒ MEM(MEM(a) + 2*4)
     ⇒ MEM(MEM(a) + 8)
```

Constant folding conditionals

```
if (true) S ⇒ S
if (false) S ⇒ ;
if (true) S else S’ ⇒ S
if (false) S else S’ ⇒ S’
while (false) S ⇒ ;

if (2 > 3) S ⇒ ;
```

Algebraic simplification

- More general form of constant folding: take advantage of usual simplification rules

```
a * 1 ⇒ a        a * 0 ⇒ 0
a + 0 ⇒ a
b | false ⇒ b          b & true ⇒ b
```

identities

```
(a + 1) + 2 ⇒ a + (1 + 2) ⇒ a+3
```

reassociation

```
a * 4 ⇒ a shl 2
a * 7 ⇒ (a shl 3) — a
a / 32767 ⇒ a shr 15 + a shr 30
```

strength reduction

- Must be careful with floating point!

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
- Improves cache, TLB performance

Inlining

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

```
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 if f refers to a global var x
- Can inline methods, but more difficult
- Best done on HIR

Specialization

- Idea: create specialized versions of functions (or methods) that are called from different places w/ different args

```
class A implements I { m( ) {…} }
class B implements I { m( ) {…} }
f(x: I) { x.m( ); }      // don’t know which m
a = new A(); f(a)        // know A.m
b = new B(); f(b)        // know B.m
```

- Can inline methods when implementation is known
- Impl known if only one implementing class

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

Dead code elimination

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

```
x = y*y;          // dead!
...               // x unused   ⇨ ...
x = z*z;                          x = z*z;
```

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

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

- Other optimizations create dead statements, variables

Copy propagation

- Given assignment `x = y`, replace subsequent uses of `x` with `y`
- May make `x` a dead variable, result in dead code
- Need to determine where copies of `y` propagate to

```
x = y                   ~~x = y~~
if (x > 1)         ⇨    if (y > 1) {
  x = x * f(x - 1)        x = y * f(y - 1)
```

Redundancy Elimination

- Common Subexpression Elimination folds redundant computations together

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

- Need to determine that expression always has same value in both places

```
b[j]=a[i]+1; c[k]=a[i] ⇒ t1=a[i]; b[j]=t1+1; c[k]=t1 ?
```

Loops

- Program hot spots are usually loops (exceptions: OS kernels, compilers)
- Most execution time in most programs is spent in loops: 90/10 is typical
- Loop optimizations are important, effective, and numerous

Loop-invariant code motion

- Another form of redundancy elimination
- If result of a statement or expression does not change during loop, and it has no externally-visible side-effect (!), can *hoist* its computation before loop
- Often useful for array element addressing computations – invariant code not visible at source level
- Requires analysis to identify loop-invariant expressions

Example

```
for (i = 0; i < a.length; i++) {
  // a not assigned in loop
}
```

hoisted loop-invariant expression

```
t1 = a.length ;
for (i = 0; i < t1; i++) {
  …
}
```

Strength reduction

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

Loop unrolling

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

```
for (i = 0; i< n; i++) { S }
```

```
for (i = 0; i < n-3; i+=4) {S; S; S; S;}
for (      ; i < n; i++) S;
```

- Gets rid of ¾ of conditional branches!
- Space-time tradeoff: not a good idea for large *S* or small *n*.

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
- Problem: when are optimizations are safe?

⇒**Dataflow analysis**

⇒**Control flow analysis**

⇒**Alias analysis**