CS 4120
Introduction to Compilers

Andrew Myers

Cornell University

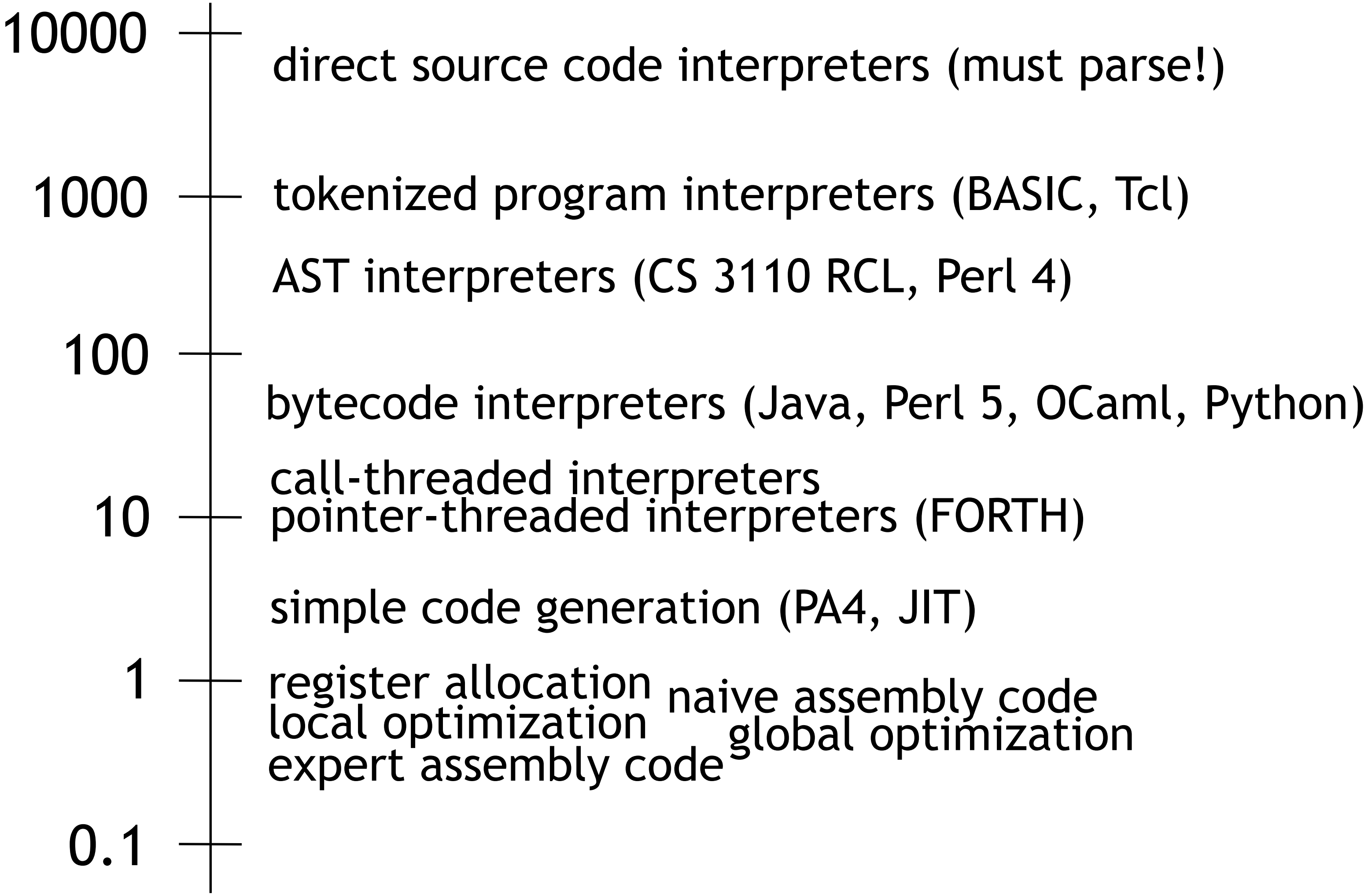
Lecture 18: Introduction to Optimization

9 March 16

Optimization

- Next topic: how to generate better code through **optimization.**
- This course covers the most valuable and straightforward optimizations – much more to learn!
  - Other sources:
    - Muchnick has 10 chapters of optimization techniques
    - Cooper and Torczon also cover optimization

How fast can you go?



direct source code interpreters (must parse!)

tokenized program interpreters (BASIC, Tcl)

AST interpreters (CS 3110 RCL, Perl 4)

bytecode interpreters (Java, Perl 5, OCaml, Python)

call-threaded interpreters

pointer-threaded interpreters (FORTH)

simple code generation (PA4, JIT)

register allocation

naive assembly code

local optimization

global optimization

expert assembly code

Goal of optimization

- Help programmers
  - clean, modular, high-level source code
  - but compile to assembly-code performance
- Optimizations are code transformations
  - can't change meaning of program to behavior not allowed by source.
- Different kinds of optimization:
  - space optimization: reduce memory use
  - time optimization: reduce execution time
  - power optimization: reduce power usage

Why do we need optimization?

- Programmers may write suboptimal code to make it clearer.
- High-level language may make it inconvenient or impossible to avoid redundant computation

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

- Architectural independence.
- Modern architectures assume optimization—hard to optimize by hand.

Where to optimize?

- Usual goal: improve time performance
- But: many optimizations trade off space vs time.
- Example: loop unrolling replaces a loop body with N copies.
  - 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: optimize code space at expense of time, saving instruction cache space
  - Complex optimizations may never pay off!
- Focus of optimization: program **hot spots**

Safety

- Possible opportunity for **loop-invariant code motion:**

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

- Transformation: invariant code out of loop:

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

Preserves meaning?

Faster?

Three aspects of an optimization:

1. the code transformation
2. safety of transformation
3. performance improvement

Writing fast programs in practice

1. Pick the right algorithms and data structures: design for locality and few operations
2. Turn on optimization and **profile** to figure out program hot spots.
3. Evaluate whether design works; if so…
4. Tweak source code until optimizer does "the right thing" to machine code
   - understanding optimizers helps!

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
Peephole optimizations

Register allocation

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

```
mov t1,t2
add t1,-8(rbp)
mov t3,-16(rbp)
mov t4,t3
cmp t1,t4
```

```
mov rax, rbx
add rax, -8(rbp)
mov rbx, -16(rbp)

cmp rax, rbx
```

Try to reuse registers aggressively (e.g., **`rbx`**)

- Coalesce registers (t3, t4) to eliminate **`mov`**’s
- May be impossible without **spilling** some temporaries to stack

Constant folding

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

```
int x = (2 + 3)*4*y;    ⇒    int x = 5*4*y;
                        ⇒    int x = 20*y;
```

- Easy and useful at every stage of compilation
  - Constant expressions are created by translation and by 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 ⇒ if (false) S ⇒ {}

Algebraic simplification

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

$a * 1 \Rightarrow a$ $a * 0 \Rightarrow 0$

$a + 0 \Rightarrow a$

identities

$b \mid \text{false} \Rightarrow b$ $b \;\&\; \text{true} \Rightarrow b$

$(a + 1) + 2 \Rightarrow a + (1 + 2) \Rightarrow a+3$

reassociation

$a * 4 \Rightarrow a \text{ shl } 2$

$a * 7 \Rightarrow (a \text{ shl } 3) - a$

strength reduction

$a / 32767 \Rightarrow a \text{ shr } 15 + a \text{ shr } 30 + a \text{ shr } 45 + a \text{ shr } 60$

- Must be careful with floating point and with overflow - algebraic identities may give wrong or less precise answers.
  - E.g., $(a+b)+c \neq a+(b+c)$ in floating point if a,b small.

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
- Reductions in code size improve cache, TLB performance.

Inlining

- Replace a function call with the body of the function:

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

                  { b:int=1; n:int=0;
                    while (n<a) { b = 2*b; fx=b }}
              return 1 + fx; }
```

- Best done on HIR
- Can inline methods, but more difficult – there can be only one f.
- May need to rename variables to avoid **name capture**—what if f refers to a global variable x?

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 e.g. if only one implementing class
- Can specialize inherited methods (e.g., HotSpot JIT)

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*8)
```

- Interleave with 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;
```

- **Dead variable:** if never read after defn. (exc to update other dead vars)

```
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 (definitely) propagate to

```
x = y
if (x > 1)
  x = x * f(x – 1)
```

⇒

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

Redundancy Elimination

- Common Subexpression Elimination (CSE) combines redundant computations

```
            a[i] = a[i] + 1
⇒ [[a]+i*8] = [[a]+i*8] + 1
⇒ t1 = [a] + i*8; [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: important, effective, and numerous

Loop-invariant code motion

- A 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** before loop

```
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++) {
    ...
}
```

Loop-invariant code motion

- Often useful for array element addressing computations – invariant code not visible at source level
- Requires analysis to identify loop-invariant expressions

Strength reduction

- Replace expensive operations (*,/) by cheap ones (+, –) via **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 }
```

- Eliminate 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/smaller/...
- 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 and when are they effective?

⇒ **Dataflow analysis**

⇒ **Control flow analysis**

⇒ **Pointer analysis**