CS 4120
Introduction to Compilers

Andrew Myers
Cornell University

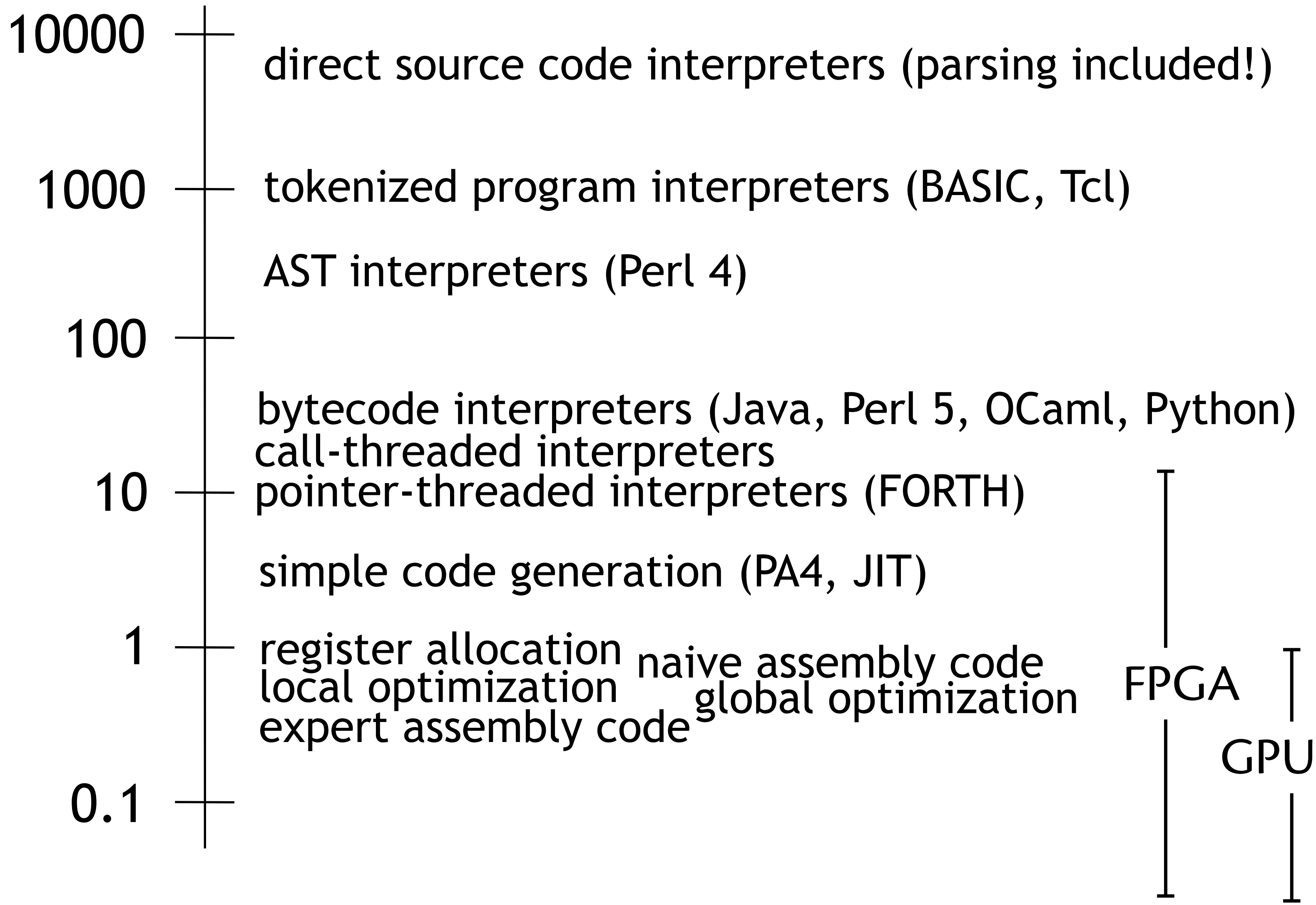
Lecture 19: Introduction to Optimization

2020 March 9

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:
    - Appel, Dragon book
    - Muchnick has 10 chapters of optimization techniques
    - Cooper and Torczon

How fast can you go?



Goal of optimization

- Compile clean, modular, high-level source code to expert assembly-code performance.
- Can't change meaning of program to behavior not allowed by source.
- Different goals:
  - 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 for clarity.
- Source language may make it hard 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 speeds up one loop but slows rest of program down a little.
  - Frequently executed loops with many iterations: space/time tradeoff is generally a win.
  - Infrequently executed code: optimize code space at expense of time, saving space in instruction cache, TLB
- Time–space tradeoffs 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: hoist invariant code out of loop:

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

Preserves meaning?

Faster?

How to write fast programs

1. Pick the right algorithms and data structures: design for few operations, small memory footprint and good locality
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	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	Constant folding& propagation (again) Branch prediction/optimization
LIR		Register allocation Loop unrolling
	Assembly	Cache optimization Peephole optimizations

Register allocation

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

```
mov t1,t2
add t1,[rbp–8]
mov t3,[rbp-16]
mov t4,t3
cmp t1,t4
```

```
mov rax, rbx
add rax, [rbp-8]
mov rbx, [rbp-16]

cmp rax, rbx
```

Try to reuse registers aggressively (e.g., **`rbx = t2, t3, t4`**)

- Coalesce registers (`t3`, `t4`) to eliminate **`mov`**'s
- In general, must **spill** 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 other optimizations

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

$a * 1 \Rightarrow a$ $\quad$ $a * 0 \Rightarrow 0$ **identities**

$a + 0 \Rightarrow a$

$b \mid false \Rightarrow b$ $\quad$ $b \ \& \ true \Rightarrow b$

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

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

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

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

- Be careful with floating point and overflow — algebraic identities may give wrong or less precise answers.
  - E.g., (a+b)+c = a+(b+c) for Java int but not C int
  - (a+b)+c ≠ a+(b+c) in floating point (`float`, `double`) 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.
- ≠ dead code elimination
  ("dead" = reached but useless)

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
- Inlining methods is more difficult — there can only be 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 it's A.m
b = new B(); f(b)  // know it's B.m
```

- Can inline methods when implementation is known
  —e.g., if only one implementing class or exact class is known
- Can specialize inherited methods to receiver class (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

```
x:int = 5;
y:int = x*2;  y:int = 5*2;  y:int = 10;
int z = a[y]; // = MEM(TEMP(a) + y*8)
                = MEM(TEMP(a) + 80)
```

- 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          dead code
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
- Idea: **hoist** unchanging expressions before loop
  - must have no externally visible side effect

```
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

- Replace expensive operations (*,/) by cheap ones (+, –) using **dependent induction variable**

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

$j == 3*i$

```
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 up to ¾ of conditional branches!
- Creates more straight-line code to optimize
- 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.
- The hard problem: when are optimizations **safe** and when are they **effective**?

⇒ **Dataflow analysis**

⇒ **Control flow analysis**

⇒ **Pointer analysis**