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?

<table>
<thead>
<tr>
<th>Speed</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>direct source code interpreters</td>
</tr>
<tr>
<td>1000</td>
<td>tokenized program interpreters (BASIC, Tcl)</td>
</tr>
<tr>
<td>100</td>
<td>AST interpreters (CS 3110 RCL, Perl 4)</td>
</tr>
<tr>
<td>10</td>
<td>bytecode interpreters (Java, Perl 5, OCaml)</td>
</tr>
<tr>
<td>1</td>
<td>call-threaded interpreters</td>
</tr>
<tr>
<td>1</td>
<td>pointer-threaded interpreters (FORTH)</td>
</tr>
<tr>
<td>1</td>
<td>simple code generation (PA4, JIT)</td>
</tr>
<tr>
<td>0.1</td>
<td>register allocation</td>
</tr>
<tr>
<td></td>
<td>naïve assembly code</td>
</tr>
<tr>
<td></td>
<td>local optimization</td>
</tr>
<tr>
<td></td>
<td>global optimization</td>
</tr>
<tr>
<td></td>
<td>expert assembly code</td>
</tr>
<tr>
<td></td>
<td>naive assembly code</td>
</tr>
</tbody>
</table>

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 avoiding redundant computation inconvenient or impossible
  \[ a[i][j] = a[i][j] + 1 \]
- Architectural independence.
- Modern architectures make it hard to optimize by hand.

Where to optimize?

- Usual goal: improve time performance
- Problem: many optimizations trade off space versus 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) {
    ...
  }
  ```

Three aspects of an optimization:
- the code transformation
- safety of transformation
- 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

<table>
<thead>
<tr>
<th>Stage</th>
<th>AST</th>
<th>Inlining</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Specialization</td>
</tr>
<tr>
<td>HIR</td>
<td>IR</td>
<td>Constant folding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Constant propagation</td>
</tr>
<tr>
<td></td>
<td>IR</td>
<td>Value numbering</td>
</tr>
<tr>
<td></td>
<td>IR</td>
<td>Dead code elimination</td>
</tr>
<tr>
<td>MIR</td>
<td>Canonical IR</td>
<td>Loop-invariant code motion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Common sub-expression elimination</td>
</tr>
<tr>
<td></td>
<td>Abstract IR</td>
<td>Strength reduction</td>
</tr>
<tr>
<td></td>
<td>Assembly</td>
<td>Constant folding &amp; propagation</td>
</tr>
<tr>
<td></td>
<td>Assembly</td>
<td>Branch prediction/optimization</td>
</tr>
<tr>
<td>LIR</td>
<td>Assembly</td>
<td>Register allocation</td>
</tr>
<tr>
<td></td>
<td>Assembly</td>
<td>Loop unrolling</td>
</tr>
<tr>
<td></td>
<td>Assembly</td>
<td>Cache optimization</td>
</tr>
<tr>
<td></td>
<td>Assembly</td>
<td>Peephole optimizations</td>
</tr>
</tbody>
</table>

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 eax, ebx`
- `add eax, [ebp-4]`
- `mov ebx, [ebp-8]`
- `cmp eax, ebx`

Need to reuse registers aggressively (e.g., `ebx`)
- 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;  \Rightarrow  int x = 5*4*y;
```

- Can perform at every stage of compilation
  - Constant expressions are created by translation and by optimization

```
a[2]  \Rightarrow  MEM(MEM(a) + 2*4)
```

```
\Rightarrow  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
  \[ a * 1 ⇒ a \]
  \[ a * 0 ⇒ 0 \]
  \[ a + 0 ⇒ a \]
  \[ b | false ⇒ b \]
  \[ b & true ⇒ b \]
  \[ (a + 1) + 2 ⇒ a + (1 + 2) ⇒ a + 3 \]
  \[ a * 4 ⇒ a \text{ shl} 2 \]
  \[ a * 7 ⇒ (a \text{ shl} 3) - a \]
  \[ a / 32767 ⇒ a \text{ shr} 15 + a \text{ shr} 30 \]

• 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:
  \[ f(a: \text{int}):\text{int} = \{ \text{b:int}=1; \text{n:int}=0; \text{while (n<a) (b = 2*b); return b;} \} \]
  \[ g(x: \text{int}):\text{int} = \{ \text{return 1+ f(x);} \} \]
  ⇒ \[ g(x: \text{int}):\text{int} = \{ \text{fx:int; a:int=x; } \}
  \{ \text{b:int=1; n:int=0; } \text{while (n<a) (b = 2*b); fx=b; } \}
  \text{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—consider 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
  
  ```java
  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
- 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
  
  ```java
  int x = 5;
  int y = x*2;
  int z = a[y]; // = MEM(MEM(a) + y*4)
  ```

- Interleave with constant folding!

**Dead code elimination**

- If side effect of a statement can never be observed, can eliminate the statement
  
  ```java
  x = y*y;  // dead!
  ...     // x unused ...
  x = z*z;     x = z*z;
  ```

- **Dead variable**: if never read after defn.

  ```java
  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
  
  ```java
  x = y
  if (x > 1) x = x * f(x - 1) if (y > 1) {
  x = y * f(y - 1)
  ```
Redundancy Elimination

- Common Subexpression Elimination (CSE) combines redundant computations

\[ a(i) = a(i) + 1 \]
\[ \Rightarrow [[a] + i \times 4] = [[a] + i \times 4] + 1 \]
\[ \Rightarrow t1 = [a] + i \times 4; \; [t1] = [t1] + 1 \]
- Need to determine that expression always has same value in both places

\[ b[j] = a[i] + 1; \; c[k] = a[i] \Rightarrow t1 = a[i]; \; b[j] = t1 + 1; \; c[k] = t1 \]

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
**Strength reduction**

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

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

**Loop unrolling**

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

```c
define 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/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**