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?

- 10000: direct source code interpreters (must parse!)
- 1000: tokenized program interpreters (BASIC, Tcl)
  AST interpreters (CS 3110 RCL, Perl 4)
- 100: bytecode interpreters (Java, Perl 5, OCaml, Python)
  call-threaded interpreters
  pointer-threaded interpreters (FORTH)
- 10: simple code generation (PA4, JIT)
- 1: register allocation
  local optimization
  naive assembly code
  global optimization
  expert assembly code
- 0.1:
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) {
      ...
  }
  ```

Three aspects of an optimization:
1. the code transformation
2. safety of transformation
3. performance improvement

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

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

\[
\begin{align*}
\text{mov} & \ t1, t2 \\
\text{add} & \ t1, -8(\text{rbp}) \\
\text{mov} & \ t3, -16(\text{rbp}) \\
\text{mov} & \ t4, t3 \\
\text{cmp} & \ t1, t4
\end{align*}
\]

\[
\begin{align*}
\text{mov} & \ \text{rax}, \ \text{rbx} \\
\text{add} & \ \text{rax}, \ -8(\text{rbp}) \\
\text{mov} & \ \text{rbx}, \ -16(\text{rbp}) \\
\text{cmp} & \ \text{rax}, \ \text{rbx}
\end{align*}
\]

Try to reuse registers aggressively (e.g., \texttt{rbx})

• Coalesce registers (t3, t4) to eliminate \texttt{mov}'s
• May be impossible without \textit{spilling} some temporaries to stack
Constant folding

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

\[
\text{int } x = (2 + 3) \times 4 \times y; \quad \Rightarrow \quad \text{int } x = 5 \times 4 \times y;
\]

\[
\Rightarrow \quad \text{int } x = 20 \times y;
\]

- Easy and useful at every stage of compilation
  - Constant expressions are created by translation and by optimization

\[
a[2] \Rightarrow MEM(MEM(a) + 2 \times 4)
\]

\[
\Rightarrow MEM(MEM(a) + 8)
\]
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

\[
\begin{align*}
\text{a} \times 1 & \Rightarrow \text{a} & \text{a} \times 0 & \Rightarrow 0 \\
\text{a} + 0 & \Rightarrow \text{a} \\
\text{b} | \text{false} & \Rightarrow \text{b} & \text{b} & \& \text{true} \Rightarrow \text{b} \\
(\text{a} + 1) + 2 & \Rightarrow \text{a} + (1 + 2) \Rightarrow \text{a} + 3 \\
\text{a} \times 4 & \Rightarrow \text{a} \text{ shl} 2 \\
\text{a} \times 7 & \Rightarrow (\text{a} \text{ shl} 3) - \text{a} \\
\text{a} / 32767 & \Rightarrow \text{a} \text{ shr} 15 + \text{a} \text{ shr} 30 + \text{a} \text{ shr} 45 + \text{a} \text{ shr} 60
\end{align*}
\]

- 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:

\[
\begin{align*}
    f(a:\text{int}):\text{int} &= \{ \text{b:int}=1;\quad \text{n:int}=0 \\
                        &\quad \text{while (n<a) \{b = 2*b; \text{return b}\} } \\
    g(x:\text{int}):\text{int} &= \{ \text{return 1 + f(x) } \} \\
\Rightarrow g(x:\text{int}):\text{int} &= \{ \text{fx:int } \{ \text{a:int }= x} \\
                        &\quad \{ \text{b:int}=1; \text{n:int}=0; \\
                        &\quad \quad \text{while (n<a) \{ b = 2*b; fx=b } \} \\
                        &\quad \quad \text{return 1 + fx; } \} \\
\end{align*}
\]

• 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 with 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 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

```plaintext
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

\[ x = y \cdot y; \quad // \text{dead!} \]
\[ \ldots \quad // \text{x unused} \quad \ldots \quad x = z \cdot z; \]

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

```c
int i;
while (m<n) { m++; i = i+1}  \quad \text{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

\[
\begin{align*}
x &= y \\
if (x > 1) & \quad \rightarrow \quad if (y > 1) \\
x &= x \times f(x - 1) & \quad \rightarrow \quad x = y \times f(y - 1)
\end{align*}
\]
Redundancy Elimination

• Common Subexpression Elimination (CSE) combines redundant computations

\[ a[i] = a[i] + 1 \]

\[ \Rightarrow [[a]+i*8] = [[a]+i*8] + 1 \]

\[ \Rightarrow t1 = [a] + i*8; [t1] = [t1]+1 \]

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

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

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

hoisted loop-invariant expression
```

```java
int 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

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

```c
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 \( \frac{3}{4} \) 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