Trivial register allocation

- Can convert abstract assembly to real assembly easily (but generate bad code)
- Allocate every temporary in stack frame rather than to a real register
  - \( t_1 = [\text{ebp}-4], t_2 = [\text{ebp}-8], t_3 = [\text{ebp}-12], \ldots \)
- Every temporary stored in different place -- conflict is impossible
- Up to three registers needed to shuttle data in and out of stack frame (max. # registers used by one instruction): \( e.g., \text{eax}, \text{ecx}, \text{edx} \)

Rewriting abstract code

- Given instruction, replace every temporary in instruction with one of three registers \( e[\text{acd}]x \)
- Add \text{mov} instructions before instruction to load registers properly
- Add \text{mov} instructions after to put data back into stack frame (as necessary)

\[ \begin{align*}
\text{push } t_1 & \quad \Rightarrow \quad \text{mov eax, [ebp-4]; push eax} \\
\text{add } t_2, t_3 & \quad \Rightarrow \quad ?
\end{align*} \]
**Result**

- Simple way to get working code – will use for Programming Assignment 4
- Code is bigger and slower than necessary
- Refinement: allocate temporaries to registers until registers run out (3 temporaries on IA-32, 20+ on MIPS, PowerPC)
- Code generation technique actually used by some compilers when all optimization turned off (-O0)

**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>Code Type</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>10</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>naive assembly code</td>
</tr>
<tr>
<td></td>
<td>local optimization</td>
</tr>
<tr>
<td></td>
<td>expert assembly code</td>
</tr>
<tr>
<td></td>
<td>global optimization</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 assume optimization—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>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></td>
<td>IR</td>
<td>Constant propagation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value numbering</td>
</tr>
<tr>
<td>MIR</td>
<td>Canonical IR</td>
<td>Dead code elimination</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Loop-invariant code motion</td>
</tr>
<tr>
<td></td>
<td>IR</td>
<td>Common sub-expression elimination</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Strength reduction</td>
</tr>
<tr>
<td></td>
<td>Abstract</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></td>
<td>Loop unrolling</td>
</tr>
<tr>
<td></td>
<td></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)
  
  \[
  \text{mov t1, t2} \quad \Rightarrow \quad \text{mov eax, ebx}
  \]
  
  \[
  \text{add t1, [bp–4]} \quad \Rightarrow \quad \text{add eax, [ebp–4]}
  \]
  
  \[
  \text{mov t3, [bp–8]} \quad \Rightarrow \quad \text{mov ebx, [ebp–8]}
  \]
  
  \[
  \text{mov t4, t3} \quad \Rightarrow \quad \text{cmp eax, ebx}
  \]

Need to reuse registers aggressively (e.g., \texttt{ebx})

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

Constant folding

- Idea: if operands are known at compile time, evaluate at compile time when possible.
  
  \[
  \text{int x} = (2 + 3)*4*y; \quad \Rightarrow \quad \text{int x} = 5*4*y;
  \]
  
  \[
  \Rightarrow \quad \text{int x} = 20*y;
  \]

- Can perform at every stage of compilation
  - Constant expressions are created by translation and by optimization
    
    \[
    \text{a[2]} \Rightarrow \text{MEM(MEM(a) + 2*4)}
    \]
    
    \[
    \Rightarrow \text{MEM(MEM(a) + 8)}
    \]
Constant folding conditionals

if (true) \( S \) \( \Rightarrow S \)
if (false) \( S \) \( \Rightarrow ; \)
if (true) \( S \) else \( S' \) \( \Rightarrow S \)
if (false) \( S \) else \( S' \) \( \Rightarrow S' \)
while (false) \( S \) \( \Rightarrow ; \)
if (2 > 3) \( S \) \( \Rightarrow \) if (false) \( S \) \( \Rightarrow ; \)

Algebraic simplification

- More general form of constant folding: take advantage of simplification rules
  
  \[ a \cdot 1 \Rightarrow a \]
  
  \[ a \cdot 0 \Rightarrow 0 \]
  
  \[ a + 0 \Rightarrow a \]
  
  \[ b \mid false \Rightarrow b \]
  
  \[ b \mid true \Rightarrow b \]
  
  \[ a + 1 \Rightarrow a + (1 + 2) \Rightarrow a + 3 \]
  
  \[ a \cdot 4 \Rightarrow a \text{shl} 2 \]
  
  \[ a \cdot 7 \Rightarrow (a \text{shl} 3) - a \]
  
  \[ a / 32767 \Rightarrow a \text{shr} 15 + a \text{shr} 30 \]

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

- 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 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 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

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

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

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

- 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
if (x > 1) { x = x * f(x - 1) }
```

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

• Common Subexpression Elimination (CSE) combines redundant computations
  \[ a(i) = a(i) + 1 \]
  \[ \Rightarrow [[a]+i*4] = [[a]+i*4] + 1 \]
  \[ \Rightarrow t1 = [a] + i*4; [t1] = [t1]+1 \]

• Need to determine that expression always has same value in both places
  \[ b[j]=a[i]+1; c[k]=a[i] \Rightarrow t1=a[i]; b[j]=t1+1; c[k]=t1 \]

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

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

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

```c
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:

```c
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 \( \frac{1}{4} \) 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