Agenda

- Survey some code “optimizations” (improvements)
  - Get a feel for what’s possible
- Some organizing concepts
  - Basic blocks
  - Control-flow and dataflow graph
Optimizations

- Use added passes to identify inefficiencies in intermediate or target code
- Replace with equivalent ("has the same externally visible behavior") but better sequences
- Target-independent optimizations best done on IL code
- Target-dependent optimizations best done on target code
- "Optimize" overly optimistic: "usually improve" is generally more accurate
  - And "clever" programmers can outwit you!
An example

\[
x = a[i] + b[2];
c[i] = x - 5;
\]

```plaintext
t1 = *(fp + ioffset);  // i
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t5 = 2;
t6 = t5 * 4;
t7 = fp + t6;
t8 = *(t7 + boffset);  // b[2]
t9 = t4 + t8; *(fp + xoffset) = t9;  // x = ...
t10 = *(fp + xoffset);  // x
t11 = 5;
t12 = t10 - t11;
t13 = *(fp + ioffset);  // i
t14 = t13 * 4;
t15 = fp + t14;
*(t15 + coffset) = t12;  // c[i] := ...
```
Kinds of optimizations

- peephole: look at adjacent instructions
- local: look at straight-line sequence of statements
- intraprocedural: look at whole procedure
  - Commonly called “global”
- interprocedural: look across procedures
  - “whole program” analysis
  - “link time optimization” is a version of this
- Larger scope => usually better optimization but more cost and complexity
  - Analysis is often less precise because of more possibilities
Peephole Optimization

- After target code generation, look at adjacent instructions (a “peephole” on the code stream)
  - try to replace adjacent instructions with something faster

```
sw $8, 12($fp)
lw $12, 12($fp)
```

```
sw $8, 12($fp)
mv $12, $8
```
More Examples: 68K

- One way to do complex instruction selection

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Instruction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub sp, 4, sp</td>
<td>mov r1, -(sp)</td>
</tr>
<tr>
<td>mov r1, 0(sp)</td>
<td></td>
</tr>
<tr>
<td>mov 12(fp), r1</td>
<td>inc 12(fp)</td>
</tr>
<tr>
<td>add r1, 1, r1</td>
<td></td>
</tr>
<tr>
<td>mov r1, 12(fp)</td>
<td></td>
</tr>
</tbody>
</table>
Peephole Optimization of Jumps

- Eliminate jumps to jumps
- Eliminate jumps after conditional branches
- “Adjacent” instructions = “adjacent in control flow”

Source code
```java
if (a < b) {
    if (c < d) { // do nothing
    } else {
        stmt1;
    }
} else {
    stmt2;
}
```
Algebraic Simplification

- “constant folding”, “strength reduction”
  - `z = 3 + 4;`
  - `z = x + 0;`
  - `z = x * 1;`
  - `z = x * 2;`
  - `z = x * 8;`
  - `z = x / 8;`

- `double x, y, z;`
- `z = (x + y) - y;`
- Can be done by peephole optimizer, or by code generator
- Why do these examples happen?
Local Optimizations

- Analysis and optimizations within a basic block
- *Basic block*: straight-line sequence of statements
  - no control flow into or out of middle of sequence
- Better than peephole
- Not too hard to implement

- Machine-independent, if done on intermediate code
Local Constant Propagation

- If variable assigned a constant, replace downstream uses of the variable with constant

```java
final int count = 10;
...
x = count * 5;
y = x ^ 3;
```

```java
t1 = 10;
t2 = 5;
t3 = t1 * t2;
x = t3;
t4 = x;
t5 = 3;
t6 = exp(t4, t5);
y = t6;
```
Local Dead Assignment Elimination

- If l.h.s. of assignment never referenced again before being overwritten, then can delete assignment
  - Why would this happen? Clean-up after previous optimizations, often

```java
final int count = 10;
...
x = count * 5;
y = x ^ 3;
x = 7;
```

```
t1 = 10;
t2 = 5;
t3 = 50;
x = 50;
t4 = 50;
t5 = 3;
t6 = 125000;
y = 125000;
x = 7;
```

Intermediate code after constant propagation
Local Common Subexpression Elimination

- Avoid repeating the same calculation
- Eliminate redundant loads
- Keep track of available expressions

\[ \ldots a[i] + b[i] \ldots \]

\[
\begin{align*}
  t1 &= *(fp + ioffset); \\
  t2 &= t1 * 4; \\
  t3 &= fp + t2; \\
  t4 &= *(t3 + aoffset); \\
  t5 &= *(fp + ioffset); \\
  t6 &= t5 * 4; \\
  t7 &= fp + t6; \\
  t8 &= *(t7 + boffset); \\
  t9 &= t4 + t8;
\end{align*}
\]
Intraprocedural optimizations

- Enlarge scope of analysis to whole procedure
  - more opportunities for optimization
  - have to deal with branches, merges, and loops
- Can do constant propagation, common subexpression elimination, etc. at “global” level
- Can do new things, e.g. loop optimizations
- Optimizing compilers usually work at this level (-O2)
Code Motion

- Goal: move loop-invariant calculations out of loops
- Can do at source level or at intermediate code level

```plaintext
for (i = 0; i < 10; i = i+1) {
    a[i] = a[i] + b[j];
    z = z + 10000;
}

for (i = 0; i < 10; i = i+1) {
    a[i] = a[i] + t1;
    z = z + t2;
}
```
for (i = 0; i < 10; i = i+1) {
    a[i] = b[j];
}

*(fp + ioffset) = 0;
label top;
    t0 = *(fp + ioffset);
    iffalse (t0 < 10) goto done;
    t1 = *(fp + joffset);
    t2 = t1 * 4;
    t3 = fp + t2;
    t4 = *(t3 + boffset);
    t5 = *(fp + ioffset);
    t6 = t5 * 4;
    t7 = fp + t6; *(t7 + aoffset) = t4;
    t9 = *(fp + ioffset);
    t10 = t9 + 1;
    *(fp + ioffset) = t10;
    goto top;
label done;
Loop Induction Variable Elimination

- For-loop index is induction variable
  - incremented each time around loop
  - offsets & pointers calculated from it
- If used only to index arrays, can rewrite with pointers
  - compute initial offsets/pointers before loop
  - increment offsets/pointers each time around loop
  - no expensive scaling in loop
  - can then do loop-invariant code motion

```c
for (i = 0; i < 10; i = i+1) {
    a[i] = a[i] + x;
}
```

=> transformed to
```c
for (p = &a[0]; p < &a[10]; p = p+4) {
    *p = *p + x;
}
```
Interprocedural Optimization

- Expand scope of analysis to procedures calling each other
- Can do local & intraprocedural optimizations at larger scope
- Can do new optimizations, e.g. inlining
Inlining: replace call with body

- Replace procedure call with body of called procedure
- Source:

  ```java
  final double pi = 3.1415927;
  double circle_area(double radius) {
      return pi * (radius * radius);
  }
  ...
  double r = 5.0;
  ...
  double a = circle_area(r);
  ```

- After inlining:

  ```java
  ...
  double r = 5.0;
  ...
  double a = pi * r * r;
  ```

- (Then what?)
Intraprocedural (Global) Optimizations

- Need a convenient representation of procedure body
- Control flow graph (CFG) captures flow of control
  - nodes are IL statements, or whole basic blocks
  - edges represent (all possible) control flow
  - node with multiple successors = branch/switch
  - node with multiple predecessors = merge
  - loop in graph = loop
- Data flow graph (DFG) capture flow of data, e.g. def/use chains:
  - nodes are def(inition)s and uses
  - edge from def to use
  - a def can reach multiple uses
  - a use can have multiple reaching defs
Analysis and Transformation

- Each optimization is made up of
  - some number of analyses
  - followed by a transformation
- Analyze CFG and/or DFG by propagating info forward or backward along CFG and/or DFG edges
  - edges called program points
  - merges in graph require combining info
  - loops in graph require iterative approximation
- Perform improving transformations based on info computed
  - have to wait until any iterative approximation has converged
- Analysis must be conservative/safe/sound so that transformations preserve program behavior
Example: Constant Propagation, Folding

- Can use either the CFG or the DFG
- CFG analysis info: table mapping each variable in scope to one of:
  - a particular constant
  - NonConstant
  - Undefined
- Transformation at each instruction:
  - if reference a variable that the table maps to a constant, then replace with that constant (constant propagation)
  - if r.h.s. expression involves only constants, and has no side-effects, then perform operation at compile-time and replace r.h.s. with constant result (constant folding)
- For best analysis, do constant folding as part of analysis, to learn all constants in one pass
Merging data flow analysis info

- Constraint: merge results must be sound
  - if something is believed true after the merge, then it must be true no matter which path we took into the merge
  - only things true along all predecessors are true after the merge

- To merge two maps of constant information, build map by merging corresponding variable information

- To merge information about two variable
  - if one is Undefined, keep the other
  - if both same constant, keep that constant
  - otherwise, degenerate to NonConstant
Example Merges

int x
x := 5

int x
x := 5
x := 5
x ==?

int x
x := 5
x := 4
x ==?
Example Merges

```
int x

x := 5

x ==?
```

```
int x

x := 5

x := f(…)

x ==?
```
How to analyze loops

```c
i = 0;
x = 10;
y = 20;
while (...) {
    // what’s true here?
    ...
    i = i + 1;
    y = 30;
}
// what’s true here?
... x ... i ... y ...
```

- Safe but imprecise: forget everything when we enter or exit a loop
- Precise but unsafe: keep everything when we enter or exit a loop
- Can we do better?
Loop Terminology
Optimistic Iterative Analysis

- Assuming information at loop head is same as information at loop entry
- Then analyze loop body, computing information at back edge
- Merge information at loop back edge and loop entry
- Test if merged information is same as original assumption
  - If so, then we’re done
  - If not, then replace previous assumption with merged information,
  - and go back to analysis of loop body
Example

```plaintext
i = 0;
x = 10;
y = 20;
while (...) {
    // what’s true here?
    ...
    i = i + 1;
    y = 30; }
// what’s true here?
... x ... i ... y ...
```
Why does this work?

- Why are the results always conservative?
  - Because if the algorithm stops, then
    - the loop head info is at least as conservative as both the loop entry info and the loop back edge info
    - the analysis within the loop body is conservative, given the assumption that the loop head info is conservative

- Why does the algorithm terminate?
  - It might not!
  - But it does if:
    - there are only a finite number of times we could merge values together without reaching the worst case info (e.g. NotConstant)
More interprocedural analyses

- Needed to support interprocedural optimizations

- Alias analysis
  - Different references referring to the same memory locations
  - may-alias vs. must-alias, context- and flow-sensitivity

- Escape analysis (pointers that are live on exit from procedures), shape analysis (static analysis of the properties of dynamic data structures), …
Supporting representations include

- Call graph
- Program dependence graph
- …
Enlarging scope of analysis yields better results today, most optimizing compilers work at the intraprocedural (a\k\a global) level
  Changing though, e.g., gcc LTO (link-time optimization)
Optimizations organized as collections of passes, each rewriting IL in place into better version
Presence of optimizations makes other parts of compiler (e.g. intermediate and target code generation) easier to write