Survey of Code Optimizations
Hal Perkins
Winter 2009

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

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
- Larger scope => better optimization but more cost and complexity

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

    | lw $08, 12($fp) | sw $08, 12($fp) | lw $12, 12($fp) | mv $12, 08 |
More Examples: 68K

<table>
<thead>
<tr>
<th>sub sp, 4, sp</th>
<th>mov rl1, -(sp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov l2(fp), rl</td>
<td>inc l2(fp)</td>
</tr>
</tbody>
</table>

- One way to do complex instruction selection

Peephole Optimization of Jumps

- Eliminate jumps to jumps
- Eliminate jumps after conditional branches
- "Adjacent" instructions = "adjacent in control flow"
- Source code
  ```c
  if (a < b) {
    if (c < d) { // do nothing
      stmt1;
    } else {
      stmt2;
    }
  } else {
  }
  ```

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

Local Dead Assignment Elimination

- If l.h.s. of assignment never referenced again before being overwritten, then can delete assignment

Why would this happen? CallQ-up after previous optimizations, often

<table>
<thead>
<tr>
<th>final int count = 10;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = count * 5;</td>
</tr>
<tr>
<td>y = x * 3;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>final int count = 10;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = count * 5;</td>
</tr>
<tr>
<td>y = x * 3;</td>
</tr>
<tr>
<td>z = 7;</td>
</tr>
<tr>
<td>t4 = exp(t4, t5);</td>
</tr>
<tr>
<td>y = t6;</td>
</tr>
</tbody>
</table>

Intermediate code after constant propagation
Local Common Subexpression Elimination

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

... a[i] + b[i] ...

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;

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

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

t1 = b[j];
t2 = 10000;
for (i = 0; i < 10; i = i+1) {
    a[i] = a[i] + t1;
    z = z + t2;
}

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

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

=> transformed to
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);
    ... (Then what?)
    ```
- After inlining:
  ```java
  ...
  double r = 5.0;
  ...
  double a = pi * r * r;
  ```

**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 (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 definition(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**
- Consider two maps of constant information:
  - Map 1: x = 5
  - Map 2: x = 4
- Merging maps:
  - x = 5 (Map 1)
  - x = 4 (Map 2)
- Resulting map: x = ?
Example Merges

```
Example Merges

int x
x := 5
x := f(…)

x ==?
```

How to analyze loops

```
How to analyze loops

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

Optimistic Iterative Analysis

```
Optimistic Iterative Analysis

- 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

```
Loop Terminology

preheader
entry edge
head
back edge
loop
exit edge
```

Example

```
Example

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 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
- ...

Summary

- Enlarging scope of analysis yields better results
  - today, most optimizing compilers work at the intraprocedural (aka global) level
- 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