**COMP 181**

Lecture 17  
*Instruction selection*

November 21, 2005

---

**MICRO report**

- Major themes
  - How to utilize more transistors
  - Frequency "wall" – problem is energy
  - Speculation
    - “Helper" threads, run-ahead execution
    - Branch prediction
    - Out-of-order execution
  - Keynote – Norm Jouppi (HP Labs)
    - Processors need to get simpler
    - New themes
      - Manageability
      - Availability
      - Security
      - Languages – Java, C#

---

**Prelude**

- Bose “noise cancelling” headphones
  - How do they work?
    - Measure noise, generate a wave with same frequency, but inverted amplitude
    - Two waves cancel out
  - We may already have this ability
    - *Obvious*: nerves go from ear to brain
    - *Not as obvious*: nerves go from brain to ear
    - Actively change frequency response of the cochlea
  - I’m on a grant to help study this phenomenon

---

**Back end**

- Essentials tasks
  - Instruction selection
    - Map low-level IR to actual machine instructions
  - Register allocation
    - Low-level IR assumes unlimited registers
    - Map to actual resources of machines
    - Goal: maximize use of registers
  - Optimizations
    - General optimizations (from last time)
    - Instruction scheduling
      - “Peep-hole” optimizations

---

**Instruction Selection**

- Low-level IR different from machine ISA
  - Why?
    - Allow different back ends
    - Abstraction – to make optimization easier
- Differences between IR and ISA
  - IR: simple, uniform set of operations
  - ISA: many specialized instructions
  - Often a single instruction does work of several operations in the IR
Instruction Selection

- Easy solution
  - Map each IR operation to a single instruction
  - May need to include memory operations

- Problem: inefficient use of ISA

### Example

- Generate code for:
  \[ a[i+1] = b[j] \]

- Simplifying assumptions
  - All variables are globals (No stack offset computation)
  - All variables are in registers (Ignore load/store of variables)

### Possible Translation

- Address of \(b[j]\):
- Load value \(b[j]\):
- Address of \(a[i+1]\):
- Store into \(a[i+1]\):

### Another Translation

- Address of \(b[j]\):
  - (no load)
- Address of \(a[i+1]\):
- Store into \(a[i+1]\):

### Yet Another Translation

- Index of \(b[j]\):
- (no load)
- Address of \(a[i+1]\):
- Store into \(a[i+1]\):
Different translations

- Why is last translation preferable?
  - Fewer instructions
  - Instructions have different costs
  - Space cost: size of each instruction
  - Time cost: number of cycles to complete

- Example

  - Idioms are cheaper than constituent parts

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r2, r1</td>
<td>1 cycle</td>
</tr>
<tr>
<td>muli c, r1</td>
<td>10 cycles</td>
</tr>
<tr>
<td>load r2, r1</td>
<td>3 cycles</td>
</tr>
<tr>
<td>store r2, r1</td>
<td>3 cycles</td>
</tr>
<tr>
<td>movem r2, r1</td>
<td>4 cycles</td>
</tr>
<tr>
<td>movex r3, r2, r1</td>
<td>5 cycles</td>
</tr>
</tbody>
</table>

Minimizing cost

- Goal:
  - Find instructions with low overall cost

- Difficulty
  - How to find these patterns?
  - Machine idioms may subsume IR operations that are not adjacent

- Idea: back to tree representation
  - Convert computation into a tree
  - Match parts of the tree

Tree Representation

- Build a tree: \[ a[i+1] = b[j] \]

  - **Goal:** find parts of the tree that correspond to machine instructions

  - IR:
    
    \[
    \begin{align*}
    t1 &= j*4 \\
    t2 &= b+t1 \\
    t3 &= *t2 \\
    t4 &= i+1 \\
    t5 &= t4*4 \\
    t6 &= a+t5 \\
    \ast t6 &= t3
    \end{align*}
    \]

Tiles

- Idea: a tile is contiguous piece of the tree that corresponds to a machine instruction

- IR:
  
  \[
  \begin{align*}
  t1 &= j*4 \\
  t2 &= b+t1 \\
  t3 &= *t2 \\
  t4 &= i+1 \\
  t5 &= t4*4 \\
  t6 &= a+t5 \\
  \ast t6 &= t3
  \end{align*}
  \]

Tiling

- **Tiling:** cover the tree with tiles

  - Assembly:
    
    \[
    \begin{align*}
    \text{mul} 4, rj \\
    \text{add} rj, rb \\
    \text{add} 1, ri \\
    \text{mul} 4, ri \\
    \text{add} ri, ra \\
    \text{movem} rb, ra \\
    \end{align*}
    \]
Generating code

- Given a tiling of a tree
  - A tiling implements a tree if:
    - It covers all nodes in the tree
    - The overlap between tiles is exactly one node
- Post-order tree walk
  - Emit machine instructions for each tile
  - Tie boundaries together with registers
  - Note: order of children matters

Tiling

- What’s hard about this?
  - Define system of tiles in the compiler
  - Finding a tiling that implements the tree (Covers all nodes in the tree)
  - Finding a “good” tiling
- Different approaches
  - Ad-hoc pattern matching
  - Automated tools

Algorithms

- **Goal**: find a tiling with the fewest tiles
- **Ad-hoc top-down algorithm**
  - Start at top of the tree
  - Find largest tile that matches top node
  - Tile remaining subtrees recursively

```java
Tile(n) {
  if ((op(n) == PLUS) && (left(n).isNum())) {
    Code c = Tile(right(n));
    c.append(ADDI left(n) right(n))
  }
}
```

Ad-hoc algorithm

- **Problem**: what does tile size mean?
  - Not necessarily the best fastest code (Example: multiply vs add)
  - How to include cost?
- **Idea**:
  - Total cost of a tiling is sum of costs of each tile
- **Goal**: find a minimum cost tiling

Including cost

- **Algorithm**:
  - For each node, find minimum total cost tiling for that node and the subtrees below
- **Key**:
  - Once we have a minimum cost for subtree, can find minimum cost tiling for a node by trying out all possible tiles matching the node
- Use dynamic programming

Dynamic programming

- **Idea**
  - For problems with optimal substructure
  - Compute optimal solutions to sub-problems
  - Combine into an optimal overall solution
- **How does this help?**
  - Use memoization:
    - Save previously computed solutions to sub-problems
    - Sub-problems recur many times
    - Can work top-down or bottom-up
Recursive algorithm

- Memoization
  - For each subtree, record best tiling in a table
  - (Note: need a quick way to find out if we’ve seen a subtree before – some systems use DAGs instead of trees)

- At each node
  - First check table for optimal tiling for this node
  - If none, try all possible tiles, remember lowest cost
  - Record lowest cost tile in table
  - Greedy, top-down algorithm

- We can emit code from table

Pseudocode

```java
Tile(n) {
    if (best(n)) return best(n)
    // -- Check all tiles
    if ((op(n) == STORE) && (op(right(n)) == LOAD) && (op(child(right(n))) == PLUS)) {
        Code c = Tile(left(n))
        c.add(Tile(left(child(right(n)))))
        c.add(Tile(right(child(right(n)))))
        c.append(MOVEX ...)
        if (cost(c) < cost(best(n))
            best(n) = c
    } // . . . and all other tiles . . .
    return best(n)
}
```

Ad-hoc algorithm

- Problem:
  - Hard-codes the tiles in the code generator

- Alternative:
  - Define tiles in a separate specification
  - Use a generic tree pattern matching algorithm to compute tiling
  - Tools: code generator generators
  - Probably overkill for RISC

Code generator generators

- Tree description language
- Represent IR tree as text
- Specification
- IR tree patterns
- Code generation actions
- Generator
- Takes the specification
- Produces a code generator

Tree notation

- Use prefix notation to avoid confusion

Rewrite rules

- Rule
  - Pattern to match and replacement
  - Cost
  - Code generation template
  - May include actions – e.g., generate register name

<table>
<thead>
<tr>
<th>Pattern, replacement</th>
<th>Cost</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>+reg1, reg2 → reg1</td>
<td>1</td>
<td>add r1, r2</td>
</tr>
<tr>
<td>store(reg1, load(reg2)) → done</td>
<td>5</td>
<td>movem r2, r1</td>
</tr>
</tbody>
</table>
Rewrite rules

- Example rules:

<table>
<thead>
<tr>
<th>#</th>
<th>Pattern, replacement</th>
<th>Cost</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+(reg, reg) → reg2</td>
<td>1</td>
<td>add r1, r2</td>
</tr>
<tr>
<td>2</td>
<td>+(reg, reg) → reg2</td>
<td>10</td>
<td>mul r1, r2</td>
</tr>
<tr>
<td>3</td>
<td>+(num, reg) → reg2</td>
<td>1</td>
<td>addi num, r1</td>
</tr>
<tr>
<td>4</td>
<td>+(num, reg) → reg2</td>
<td>10</td>
<td>muli num, r1</td>
</tr>
<tr>
<td>5</td>
<td>store(reg0, load(reg0)) → done</td>
<td>5</td>
<td>movem r2, r1</td>
</tr>
</tbody>
</table>

- What kinds of optimizations can we do?
  - Strength reduction: multiply to shift or add

Example

Assembly

```
muli 4, rj
add rj, rb
addi 1, ri
muli 4, ri
add ri, ra
movem rb, ra
```

Rewriting process

```
store(ra, +(ri, 4)), load(rb)
```

Implementation

- What does this remind you of?
  - Similar to parsing
  - Implement as an automaton
  - Use cost to choose from competing productions

- Provides linear time optimal code generation
  - BURS (bottom-up rewrite system)
  - burg, Twig, BEG

Summary

<table>
<thead>
<tr>
<th>Ad-hoc pattern matchers</th>
<th>Probably reasonable for RISC machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encode matching as automaton</td>
<td>Fast, optimal code generation – requires separate tool</td>
</tr>
<tr>
<td>Use parsers</td>
<td>Can lead to highly ambiguous grammars</td>
</tr>
</tbody>
</table>

Modern processors

- Execution time not sum of tile times
  - Instruction order matters
    - Pipelining: parts of different instructions overlap
    - Bad ordering stalls the pipeline – e.g., too many operations of one type
    - Superscalar: some operations executed in parallel

- Cost is an approximation
  - Instruction scheduling helps
Next time…

- Happy Thanksgiving!
- Monday: register allocation
- Only four more lectures after this one

Top-down algorithm

\[
\text{Tile}(n) \\
\begin{align*}
&\text{Label}(n) \leftarrow \emptyset \\
&\text{if } n \text{ has two children then} \\
&\phantom{\text{Label}(n)} \text{Tile (left child of } n) \\
&\phantom{\text{Label}(n)} \text{Tile (right child of } n) \\
&\text{for each rule } r \text{ that implements } n \\
&\phantom{\text{Label}(n)} \text{if (left} (r) \text{) } \in \text{Label(left}(n)) \text{ and} \\
&\phantom{\text{Label}(n)} \text{(right} (r) \text{) } \in \text{Label(right}(n)) \text{ then} \\
&\phantom{\text{Label}(n)} \text{Label}(n) \leftarrow \text{Label}(n) \cup \{r\} \\
&\text{else if } n \text{ has one child} \\
&\phantom{\text{Label}(n)} \text{Tile(child of } n) \\
&\text{for each rule } r \text{ that implements } n \\
&\phantom{\text{Label}(n)} \text{if (left} (r) \text{) } \in \text{Label(child}(n)) \text{ then} \\
&\phantom{\text{Label}(n)} \text{Label}(n) \leftarrow \text{Label}(n) \cup \{r\} \\
&\text{else } /* n \text{ is a leaf */} \\
&\phantom{\text{Label}(n)} \text{Label}(n) \leftarrow \{\text{all rules that implement } n\}
\end{align*}
\]

This algorithm

- Finds all matches in rule set
- Labels node \( n \) with that set
- Can keep lowest cost match at each point
- Leads to a notion of local optimality — lowest cost at each point
- Spends its time in the two matching loops

Homework

- Due on Monday
- Define code generation schemes for the project:
  - Procedure call
    - Evaluate and assign parameters
    - Retrieve the result
  - Case (switch) statement
    - Use form “if (val == caseval) jump to case code”
    - Use two passes
  - For loop
    - Note differences between Pascal and C
    - Handle both “to” and “downto” loops
- Use \texttt{generate()}, \texttt{emit()}, \texttt{new_temp()}, and \texttt{new_label()} from Lectures 14, 15.