**CS412/413**

Introduction to Compilers  
Radu Rugina

Lecture 31: Instruction Selection  
14 Apr 06

---

### Backend Optimizations

- **Instruction selection**  
  - translate low-level IR to assembly instructions  
  - A machine instruction may represent multiple IR instructions  
  - Especially applicable to CISC architectures

- **Register Allocation**  
  - Place variables into registers  
  - Avoid spilling variables on stack

---

### Instruction Selection

- Different sets of instructions in low-level IR and in the target machine
- **Instruction selection** = translate low-level IR to assembly instructions on the target machine
- **Straightforward solution**: translate each low-level IR instruction to a sequence of machine instructions
- **Example**:  
  \[ x = y + z \]  
  - `mov y, r1`  
  - `mov z, r2`  
  - `add r2, r1`  
  - `mov r1, x`

---

### Example

- Consider the computation:  
  \[ a[i+1] = b[j] \]
- Assume \(a, b, i, j\) are variables  
  \(\text{register } ra\) holds address of \(a\)  
  \(\text{register } rb\) holds address of \(b\)  
  \(\text{register } rl\) holds value of \(i\)  
  \(\text{register } rj\) holds value of \(j\)

<table>
<thead>
<tr>
<th>IR code:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t1 = j+4</code></td>
<td></td>
</tr>
<tr>
<td><code>t2 = b+t1</code></td>
<td></td>
</tr>
<tr>
<td><code>t3 = t2</code></td>
<td></td>
</tr>
<tr>
<td><code>t4 = i+1</code></td>
<td></td>
</tr>
<tr>
<td><code>t5 = t4+t6</code></td>
<td></td>
</tr>
<tr>
<td><code>t6 = a+t6</code></td>
<td></td>
</tr>
<tr>
<td><code>t6 = t3</code></td>
<td></td>
</tr>
</tbody>
</table>

---

### Possible Translation

- Address of \(b[j]\):  
  - `mulc 4, rj`  
  - `add rj, rb`  
  - `t3 = t2`  
  - \(t4 = t1+1\)  
  - `t5 = t4+t6`  
  - `t6 = a+t5`  
  - `t6 = t3`

- Load value \(b[j]\):  
  - `load rb, r1`

- Address of \(a[i+1]\):  
  - `add 1, rl`
  - `mulc 4, ri`  
  - `add ri, ra`

- Store into \(a[i+1]\):  
  - `store rl, ra`
Another Translation

- Address of b[j]: mulc 4, rj
  - IR code: t1 = j+4
  - t2 = b+t1
  - t3 = *t2
  - t4 = i+1
  - t5 = t4+i
  - t6 = a+t5
  - *t6 = t3

- Address of a[i+1]: add 1, ri
  - mulc 4, ri
  - add ri, ra
- Store into a[i+1]: movem rb, ra

Yet Another Translation

- Index of b[j]: mulc 4, rj
  - IR code: t1 = j+4
  - t2 = b+t1
  - t3 = *t2
  - t4 = i+1
  - t5 = t4+i
  - t6 = a+t5
  - *t6 = t3

- Address of a[i+1]: add 1, ri
  - mulc 4, ri
  - add ri, ra
- Store into a[i+1]: movem rj, rb, ra

Instruction Costs

- Different machine instructions have different costs
  - Time cost: how fast instructions are executed
  - Space cost: how much space instructions take
- Example: cost = number of cycles
  - add r2, r1, cost=1
  - mulc c, r1, cost=10
  - load r2, r1, cost=3
  - store r2, r1, cost=3
  - movem r2, r1, cost=4
  - movem r3, r2, r1, cost=5
- Goal: find translation with smallest cost

How to Solve the Problem?

- Difficulty: low-level IR instruction matched by a machine instructions may not be adjacent
- Example: movem rb, ra
- Idea: use tree representation
  - Easier to detect matching instructions

Tree Representation

- Goal: determine parts of the tree which correspond to machine instructions

IR code:

- t1 = j+4
- t2 = b+t1
- t3 = *t2
- t4 = i+1
- t5 = t4+i
- t6 = a+t5
- *t6 = t3

Tiles

- Tile = tree patterns (subtrees) corresponding to machine instructions

IR code:

- t1 = j+4
- t2 = b+t1
- t3 = *t2
- t4 = i+1
- t5 = t4+i
- t6 = a+t5
- *t6 = t3
Tiling

- Tiling = cover the tree with disjoint tiles

```
movem rb, ra
store
load
```

Assembly:
```
mulc 4, rj
add rj, rb
add 1, ri
mulc 4, ri
add ri, ra
movem rb, ra
```

Different Tilings

```
store rb, ra
movex rj, rb, ra
store
load
```

Directed Acyclic Graphs

- Tree representation: appropriate for instruction selection
  - Subtrees → machine instructions
- DAG construction (aka Value Numbering)
  - Common sub-expressions represented by the same node
  - Tile the expression DAG

Example:
```
t = y+1
y = z*t
z = t*y
```

Big Picture

- What the compiler has to do:
  1. Translate three-address code into a DAG representation
  2. Then find a good tiling of the DAG
     - Maximal munch algorithm
     - Dynamic programming algorithm

Value Numbering

- Input: a sequence of low IR instructions in a basic block
- Output: an expression DAG for the block

- Idea:
  - Label each DAG node with variable which holds that value
  - Build DAG bottom-up
- A variable may have multiple values in a block
- Use different variable indices for different values of the variable: \( t_0, t_1, t_2 \), etc.

Value Numbering Algorithm

```
index(v) = 0 for each variable v
For each instruction I (in the order they appear)
  For each v ∈ use[I], with \( s = index(v) \)
    if node \( v_s \) doesn’t exist
      create node \( v_s \)
    Create node for instruction I, with children
      \( v_s \mid v \in use[I] \)
  For each v ∈ def[I]
    index(v) = index(v) + 1
  If I is of the form \( z = \ldots \) and \( n = index[x] \)
    label the new node with \( x_n \)
```
Next: Tiling

- **Goal**: find a good covering of DAG with tiles
- **Issue**: need to know what variables are in registers
- **Assume abstract assembly**:
  - Machine with infinite number of registers
  - Temporary/local variables stored in registers
  - Parameters/heap variables: use memory accesses

Example Tiling

- Consider the instruction $a = a + i$
  
  $a = $local variable
  
  $i = $parameter

- Need new temporary registers between tiles (unless operand node is labeled with temporary)

- Result code:
  
  ```
  mov %ebp, %eax
  sub $20, %eax
  mov (%eax), %eax
  add %eax, %eax
  ```

Problems

- **Classes of registers**
  - Registers may have specific purposes
  - Example: Pentium multiply instruction
    - multiply register eax by contents of another register
    - store result in eax (low 32 bits) and edx (high 32 bits)
    - need extra instructions to move values into eax

- **Two-address machine instructions**
  - Three-address low-level code
  - Need multiple machine instructions for a single tile

- **CISC versus RISC**
  - Complex instruction sets: multiple possible typings