Instruction Selection and Scheduling

Machine code generation
Machine code generation

- **Input:** intermediate code + symbol tables
  - All variables have values that machines can directly manipulate
  - Each operation has at most two operands
  - Assume program is free of errors
    - Type checking has taken place, type conversion done

- **Output:**
  - Absolute/relocatable machine (assembly) code
  - Architectures
    - RISC machines, CISC processors, stack machines

- **Issues:**
  - Instruction selection
  - Instruction scheduling
  - Register allocation and memory management
Retargetable back-end

- Build retargetable compilers
  - Compilers on different machines share a common IR
    - Can have common front and mid ends
  - Isolate machine dependent information
    - Table-based back ends share common algorithms
- Table-based instruction selector
  - Create a description of target machine, use back-end generator
Instruction Selection

- Based on locations of operands, different instructions may be selected
- Two pattern-matching approaches
  - Generate efficient instruction sequences from the AST
  - Generate naïve code, then rewrite inefficient code sequences
Tree-Pattern Matching

- **Tiling the AST**
  - Use a low-level AST to expose all the impl. details
  - Define a collection of (operation pattern, code generation template) pairs
  - Match each AST subtree with an operation pattern, then select instructions accordingly

- **Given an AST and a collection of operation trees**
  - A tiling is a collection of <ASTnode, op-pattern> pairs, each specifying the implementation for a AST node
  - Storage for result of each AST operation must be consistent across different operation trees

low-level AST for \( w \leftarrow x - 2 + y \)

Tiling an AST for \( G+12 \):

\[
+ \quad \text{Reg}:=+(\text{Reg1},\text{Num2})
\]

\[
\text{Lab}(\@G) \quad \text{Num}(12)
\]

\[
\text{Reg}:=\text{Lab}1
\]
Rules Through Tree Grammar

- Use attributed grammar to define code generation rules
  - Summarize structures of AST through context-free grammar
  - Each production defines a tree pattern in prefix-notation
  - Each production is associated with a code generation template (syntax-directed translation) and a cost
  - Each grammar symbol is associated with a synthesized attribute (location of value) to be used in code generation

<table>
<thead>
<tr>
<th>production</th>
<th>cost</th>
<th>Code template</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Goal := Assign</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2: Assign := &lt;- (Reg1, Reg2)</td>
<td>1</td>
<td>Store r2 =&gt; r1</td>
</tr>
<tr>
<td>3: Assign := &lt;- (+ (Reg1, Reg2), Reg3)</td>
<td>1</td>
<td>storeA0 r3 =&gt; r1, r2</td>
</tr>
<tr>
<td>4: Assign := &lt;- (+ (Reg1, num2), Reg3)</td>
<td>1</td>
<td>storeAI r3 =&gt; r1, n2</td>
</tr>
<tr>
<td>5: Assign := &lt;- (+ (num1, Reg2), Reg3)</td>
<td>1</td>
<td>storeAI r3 =&gt; r2, n1</td>
</tr>
<tr>
<td>6: Reg:=lab1 (a relocatable symbol)</td>
<td>1</td>
<td>loadI lab1 =&gt; rnew</td>
</tr>
<tr>
<td>7: Reg:=val1 (value in reg, e.g. rarp)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8: Reg := Num1 (constant integer value)</td>
<td>1</td>
<td>loadI num1 =&gt; rnew</td>
</tr>
</tbody>
</table>
## Tree Grammar (continued)

<table>
<thead>
<tr>
<th>production</th>
<th>cost</th>
<th>Code template</th>
</tr>
</thead>
<tbody>
<tr>
<td>9: Reg := M(Reg1)</td>
<td>1</td>
<td>Load r1 =&gt; rnew</td>
</tr>
<tr>
<td>10: Reg := M(+ (Reg1,Reg2))</td>
<td>1</td>
<td>loadA0 r1, r2 =&gt; rnew</td>
</tr>
<tr>
<td>11: Reg := M(+ (Reg1,Num2))</td>
<td>1</td>
<td>loadAI r1, n2 =&gt; rnew</td>
</tr>
<tr>
<td>12: Reg := M(+ (Num1,Reg2))</td>
<td>1</td>
<td>loadAi r2, n1 =&gt; rnew</td>
</tr>
<tr>
<td>13: Reg := M(+ (Reg1, Lab2))</td>
<td>1</td>
<td>loadAI r1, l2 =&gt; rnew</td>
</tr>
<tr>
<td>14: Reg := M(+ (Lab1,Reg2))</td>
<td>1</td>
<td>loadAI r2, l1 =&gt; rnew</td>
</tr>
<tr>
<td>15: Reg := - (Reg1,Reg2)</td>
<td>1</td>
<td>Sub r1 r2 =&gt; rnew</td>
</tr>
<tr>
<td>16: Reg := - (Reg1, Num2)</td>
<td>1</td>
<td>subI r1, n2 =&gt; rnew</td>
</tr>
<tr>
<td>17: Reg := +(Reg1, Reg2)</td>
<td>1</td>
<td>add r1, r2=&gt; rnew</td>
</tr>
<tr>
<td>18: Reg := + (Reg1, Num2)</td>
<td>1</td>
<td>addI r1, n2 =&gt; rnew</td>
</tr>
<tr>
<td>19: Reg := + (Num1, Reg2)</td>
<td>1</td>
<td>addI r2, n1 =&gt; rnew</td>
</tr>
<tr>
<td>20: Reg := + (Reg1, Lab2)</td>
<td>1</td>
<td>addI r1, l2 =&gt; rnew</td>
</tr>
<tr>
<td>21: Reg := + (Lab1, Reg2)</td>
<td>1</td>
<td>addI r2, l1 =&gt; rnew</td>
</tr>
</tbody>
</table>
Tree Matching Approach

- Need to select lowest-cost instructions in bottom-up traversal of AST
  - Need to determine lowest-cost match for each storage class

- Automatic tools
  - Hand-coding of tree matching
  - Encode the tree-matching problem as a finite automata
  - Use parsing techniques
    - Need to be extended to handle ambiguity
  - Use string-matching techniques
    - Linearize the tree into a prefix string
    - Apply string pattern matching algorithms
Tiling the AST

- Given an AST and a collection of operation trees, tiling the AST maps each AST subtree to an operation tree.
- A tiling is a collection of <ASTnode, op-tree> pairs, each specifying the implementation for a AST node.
  - Storage for result of each AST operation must be consistent across different operation trees.

```
Reg:= Lab(1)
Reg:= Lab+Num(12)
```

```
Lab(@G)
Num(12)
```
Finding a tiling

- **Bottom-up walk of the AST, for each node** n
  - Label(n) contains the set of all applicable tree patterns

```
Tile(n)
Label(n) := ∅
if n is a binary node then
  Tile(left(n))
  Tile(right(n))
  for each rule r that matches n’s operation
    if left(r) ∈ Label(left(n)) and right(r) ∈ Label(right(n))
      then Label(n) := Label(n) ∪ {r}
else if n is a unary node then
  Tile(left(n))
  for each rule r that matches n’s operation
    if (left(r) ∈ Label(left(n))
      then Label(n) := Label(n) ∪ {r}
else /* n is a AST leaf */
  Label(n) := {all rules that match the operation in n}
```
Finding The Low-cost Tiling

- Tiling can find all the matches in the pattern set
  - Multiple matches exist because grammar is ambiguous
- To find the one with lowest cost, must keep track of the cost in each matched translation

Example: low-level AST for $w \leftarrow x - 2 + y$

```
(18,1) + (17,2)
(7,0) arp (4,5) (2,6) <-
(15,3) + (16,2)
(9,2)(10,2) (11,1) M
(8,1) arp (17,4) (9,2) (11,1)
(18,1) + (17,2) M
(8,1) arp (18,1) (17,2)
(17,4) + (17,2) arp (18,1) (17,2)
(11,1) arp (17,2) (18,1) (17,2)
(7,0) arp (7,0) arp (7,0)
(8,1) (8,1) (8,1)
```

loadAI rarp,8=>r1
subI r1, 2=> r2
loadAI rarp,12=>r3
Add r2, r3 => r4
storeAI r4=>rarp, 4
Peephole optimization

- Use simple scheme to match IR to machine code
  - Discover local improvements by examining short sequences of adjacent operations

- StoreAI r1 => rarp, 8
  - loadAI rarp, 8 => r15
  - storeAI r1 => rarp 8
    - I2i r1 => r15

- addI r2, 0 => r7
  - Mult r4, r7 => r10
  - Mult r4, r2 => r10

- jumpI -> L10
  - L10: jumpI -> L11
  - jumpI -> L11
  - L10: jumpI -> L11
Systematic Peephole Optimization

- **Expander**
  - Rewrites each assembly instruction to a sequence of low-level IRs that represent all the direct effects of operation

- **Simplifier**
  - Examine and improve LLIR operations in a small sliding window
    - Forward substitution, algebraic simplification, constant evaluation, eliminating useless effects

- **Matcher**
  - Match simplified LLIR against pattern library for instructions that best captures the LLIR effects
Peephole optimization example

**Optimizations:**

- Expand:
  - \( \text{mult} \ 2 \ y \Rightarrow t1 \)
  - \( \text{sub} \ x \ t1 \Rightarrow w \)

- Simplify:
  - \( r1 := n1 \)
  - \( r2 := r3 + r1 \)
  - \( r3 := \text{M}(r1) \)
  - \( r1 := r2 + n1 \)
  - \( r3 := \text{M}(r2 + n1) \)
  - \( \text{M}(r2 + n1) := r3 \)

- Match:
  - \( r10 := 2 \)
  - \( r11 := @G \)
  - \( r12 := 12 \)
  - \( r13 := r11 + r12 \)
  - \( r14 := \text{M}(r13) \)
  - \( r15 := r10 \times r14 \)
  - \( r16 := -16 \)
  - \( r17 := \text{rarp} + r16 \)
  - \( r18 := \text{M}(r17) \)
  - \( r19 := \text{M}(r18) \)
  - \( r20 := r19 - r15 \)

- Load:
  - \( \text{loadI} \ 2 \Rightarrow r10 \)
  - \( \text{loadI} \ @G \Rightarrow r11 \)
  - \( \text{loadAI} \ r11 \ 12 \Rightarrow r14 \)
  - \( \text{Mult} \ r10 \ r14 \Rightarrow r15 \)
  - \( \text{loadAI} \ \text{rarp} \ -16 \Rightarrow r18 \)
  - \( \text{Load} \ r18 \Rightarrow r19 \)
  - \( \text{Sub} \ r19 \ r15 \Rightarrow r20 \)

- Store:
  - \( \text{storeAI} \ r20 \Rightarrow \text{rarp} \ 4 \)
Efficiency of Peephole Optimization

- Design issues
  - Dead values
    - May intervene with valid simplifications
    - Need to be recognized in the expansion process
  - Control flow operations
    - Complicates simplifier
      - Clear window vs. special-case handling
  - Physical vs. logical windows
    - Adjacent operations may be irrelevant
    - Sliding window includes ops that define or use common values
- RISC vs. CISC architectures
  - RISC architectures makes instruction selection easier
- Additional issues
  - Automatic tools to generate large pattern libraries for different architectures
  - Front ends that generate LLIR make compilers more portable