CSE 504: Compiler Design

Instruction Selection

Pradipta De
pradipta.de@sunykorea.ac.kr
Current Topic

- Instruction Selection techniques
  - Tree Walk
  - Tiling based approach
  - Peephole Optimization
Instruction Selection

- Difficulty of Instruction Selection
  - Map IR (high-level or low-level IRs) → assembly code
    - A pattern matching problem
  - Multiple choices to derive IR → ASM
    - Different operations to achieve same task in ISA
    - Different addressing modes in ISA
    - Find the “best” combination of ASM instructions

Goal: Compile writer writes a description of the target ISA → a tool constructs a selector for use at compile time
Illustrative Example

- Register-to-register copy in ILOC (ISA)
  \[ r_i \rightarrow r_j \]

- Simple choice: \( i2i, r_i, r_j \)

- Other ASM instructions in ILOC:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addI ( r_i, 0 \rightarrow r_j )</td>
<td>subI ( r_i, 0 \rightarrow r_j )</td>
<td>lshiftI ( r_i, 0 \rightarrow r_j )</td>
</tr>
<tr>
<td>multI ( r_i, 1 \rightarrow r_j )</td>
<td>divI ( r_i, 1 \rightarrow r_j )</td>
<td>rshiftI ( r_i, 0 \rightarrow r_j )</td>
</tr>
<tr>
<td>orI ( r_i, 0 \rightarrow r_j )</td>
<td>xorI ( r_i, 0 \rightarrow r_j )</td>
<td>( \ldots ) and others ( \ldots )</td>
</tr>
</tbody>
</table>

Ideally examine all the instructions and pick the best one for the specific context.

Need a systematic way to explore the space of alternative code sequences, or limit the search using pre-computed information to enable deep search.
Simple TreeWalk

- Tree structured IR is common in compilers
- Can we represent the target ISA using trees?

\[
\begin{align*}
\text{add } r_i, r_j & \Rightarrow r_k \\
\text{addI } r_i, c_j & \Rightarrow r_k
\end{align*}
\]

- \( r_i, r_j \) represent registers, and \( c_j \) represents a known constant

- Match ISA trees against IR trees
- Level of detail in IR tree leads to better matching
  - High level IR \( \Rightarrow \) Instruction selector must provide additional details to make better choices
  - Low level IR \( \Rightarrow \) Easier for selector to map choices
ILOC ISA (subset of instructions)

<table>
<thead>
<tr>
<th>Arithmetic Operations</th>
<th>Memory Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>add ( r_1, r_2 \rightarrow r_3 )</td>
<td>store ( r_1 \rightarrow r_2 )</td>
</tr>
<tr>
<td>addI ( r_1, c_2 \rightarrow r_3 )</td>
<td>storeAO ( r_1 \rightarrow r_2, r_3 )</td>
</tr>
<tr>
<td>sub ( r_1, r_2 \rightarrow r_3 )</td>
<td>storeAI ( r_1 \rightarrow r_2, c_3 )</td>
</tr>
<tr>
<td>subI ( r_1, c_2 \rightarrow r_3 )</td>
<td>loadI ( c_1 \rightarrow r_3 )</td>
</tr>
<tr>
<td>rsubI ( r_2, c_1 \rightarrow r_3 )</td>
<td>load ( r_1 \rightarrow r_3 )</td>
</tr>
<tr>
<td>mult ( r_1, r_2 \rightarrow r_3 )</td>
<td>loadAO ( r_1, r_2 \rightarrow r_3 )</td>
</tr>
<tr>
<td>multiI ( r_1, c_2 \rightarrow r_3 )</td>
<td>loadAI ( r_1, c_2 \rightarrow r_3 )</td>
</tr>
<tr>
<td>e \times f</td>
<td>e \times 2</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>IDENT (e, ARP, 4) \times IDENT (f, ARP, 8)</td>
<td>IDENT (e, ARP, 4) \times NUM (2)</td>
</tr>
</tbody>
</table>

### Generated Code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI 4</td>
<td>r5</td>
</tr>
<tr>
<td>loadA0 rarp, r5</td>
<td>r6</td>
</tr>
<tr>
<td>loadI 8</td>
<td>r7</td>
</tr>
<tr>
<td>loadA0 rarp, r7</td>
<td>r8</td>
</tr>
<tr>
<td>mult r6, r8</td>
<td>r9</td>
</tr>
<tr>
<td>loadI 4</td>
<td>r5</td>
</tr>
<tr>
<td>loadA0 r5, r6</td>
<td>r7</td>
</tr>
<tr>
<td>loadI @G</td>
<td>r5</td>
</tr>
<tr>
<td>loadI 4</td>
<td>r6</td>
</tr>
<tr>
<td>loadA0 r5, r6</td>
<td>r7</td>
</tr>
<tr>
<td>loadI @H</td>
<td>r5</td>
</tr>
<tr>
<td>loadI 4</td>
<td>r6</td>
</tr>
<tr>
<td>loadA0 r8, r9</td>
<td>r10</td>
</tr>
<tr>
<td>mult r7, r10</td>
<td>r11</td>
</tr>
</tbody>
</table>

### Desired Code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadA1 rarp, 4</td>
<td>r5</td>
</tr>
<tr>
<td>loadA1 rarp, 8</td>
<td>r6</td>
</tr>
<tr>
<td>mult r5, r6</td>
<td>r7</td>
</tr>
<tr>
<td>loadA1 rarp, 4</td>
<td>r5</td>
</tr>
<tr>
<td>loadA1 r5, @G</td>
<td>r6</td>
</tr>
<tr>
<td>loadA1 r5, @H</td>
<td>r7</td>
</tr>
<tr>
<td>mult r6, r7</td>
<td>r8</td>
</tr>
</tbody>
</table>
Exposing low level detail

Expression: \( a \leftarrow b - 2 \times c \)

Front-end can produce the low level IR

Val: value that resides in register, e.g. ARP
Lab: relocatable symbol, e.g. labels
◆ : level of indirection ➔ child is an address, and it produces the value stored at that address

Apply postorder treewalk, and match against ISA trees
Convenient Notation for Trees

Prefix notation for the low level IR tree:

\[ \leftarrow (+ (Val_1, Num_1), \right) \]
\[ \quad (- (\star (\star (+ (Val_2, Num_2))), \right) \]
\[ \quad \times (Num_3, \star (+ (Lab_1, Num_4)))) \]

Prefix notation for the ILOC instructions:

add \quad (+ (r_i, r_j))

addl \quad (+ (r_i, c_j))

Commutative variant of addl \quad (+ (c_i, r_j))

Given an AST and a collection of operation trees, the goal is to map the AST to operations by constructing a \textit{tiling} of the AST with operation trees.
Tiling

• Goal is to “tile” the AST with operation trees
• A tile is a collection of <ast-node, op-tree>
  – A tile implies that op-tree can implement the ast-node

• A tiling implements a AST if,
  – It implements every operation (subtree in AST)
  – Each tile connects with its neighbors

• Given a tiling that implements a AST, the compiler can generate assembly code in a bottom-up traversal
  – Associate costs with operation trees, and produce minimal cost tiling
Rewrite Rules

• The relationship between operation trees and subtrees in the AST are encoded as rewrite rules

• Rewrite rules consists of:
  – A production in a tree grammar
  – A code template
  – Associated cost
<table>
<thead>
<tr>
<th>Production</th>
<th>Cost</th>
<th>Code Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Goal $\rightarrow$ Assign</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 Assign $\rightarrow$ $\leftarrow (Reg_1,Reg_2)$</td>
<td>1</td>
<td>store $r_2$ $\Rightarrow r_1$</td>
</tr>
<tr>
<td>3 Assign $\rightarrow$ $\leftarrow (+ (Reg_1,Reg_2),Reg_3)$</td>
<td>1</td>
<td>storeAO $r_3$ $\Rightarrow r_1, r_2$</td>
</tr>
<tr>
<td>4 Assign $\rightarrow$ $\leftarrow (+ (Reg_1,Num_2),Reg_3)$</td>
<td>1</td>
<td>storeAI $r_3$ $\Rightarrow r_1, n_2$</td>
</tr>
<tr>
<td>5 Assign $\rightarrow$ $\leftarrow (+ (Num_1,Reg_2),Reg_3)$</td>
<td>1</td>
<td>storeAI $r_3$ $\Rightarrow r_2, n_1$</td>
</tr>
<tr>
<td>6 Reg $\rightarrow$ Lab$_1$</td>
<td>1</td>
<td>loadI $l_1$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>7 Reg $\rightarrow$ Val$_1$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8 Reg $\rightarrow$ Num$_1$</td>
<td>1</td>
<td>loadI $n_1$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>9 Reg $\rightarrow$ $\uplus (Reg_1)$</td>
<td>1</td>
<td>load $r_1$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>10 Reg $\rightarrow$ $\uplus (+ (Reg_1,Reg_2))$</td>
<td>1</td>
<td>loadAO $r_1, r_2$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>11 Reg $\rightarrow$ $\uplus (+ (Reg_1,Num_2))$</td>
<td>1</td>
<td>loadAI $r_1, n_2$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>12 Reg $\rightarrow$ $\uplus (+ (Num_1,Reg_2))$</td>
<td>1</td>
<td>loadAI $r_2, n_1$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>13 Reg $\rightarrow$ $\uplus (+ (Reg_1,Lab_2))$</td>
<td>1</td>
<td>loadAI $r_1, l_2$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>14 Reg $\rightarrow$ $\uplus (+ (Lab_1,Reg_2))$</td>
<td>1</td>
<td>loadAI $r_2, l_1$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>15 Reg $\rightarrow$ $+$ $(Reg_1,Reg_2)$</td>
<td>1</td>
<td>add $r_1, r_2$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>16 Reg $\rightarrow$ $+$ $(Reg_1,Num_2)$</td>
<td>1</td>
<td>addI $r_1, n_2$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>17 Reg $\rightarrow$ $+$ $(Num_1,Reg_2)$</td>
<td>1</td>
<td>addI $r_2, n_1$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>18 Reg $\rightarrow$ $+$ $(Reg_1,Lab_2)$</td>
<td>1</td>
<td>addI $r_1, l_2$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>19 Reg $\rightarrow$ $+$ $(Lab_1,Reg_2)$</td>
<td>1</td>
<td>addI $r_2, l_1$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>20 Reg $\rightarrow$ $-$ $(Reg_1,Reg_2)$</td>
<td>1</td>
<td>sub $r_1, r_2$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>21 Reg $\rightarrow$ $-$ $(Reg_1,Num_2)$</td>
<td>1</td>
<td>subI $r_1, n_2$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>22 Reg $\rightarrow$ $-$ $(Num_1,Reg_2)$</td>
<td>1</td>
<td>rsbI $r_2, n_1$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>23 Reg $\rightarrow$ $\times (Reg_1,Reg_2)$</td>
<td>1</td>
<td>mult $r_1, r_2$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>24 Reg $\rightarrow$ $\times (Reg_1,Num_2)$</td>
<td>1</td>
<td>multI $r_1, n_2$ $\Rightarrow r_{new}$</td>
</tr>
<tr>
<td>25 Reg $\rightarrow$ $\times (Num_1,Reg_2)$</td>
<td>1</td>
<td>multI $r_2, n_1$ $\Rightarrow r_{new}$</td>
</tr>
</tbody>
</table>
How to apply rewrite rules

- The first match in the sequence recognizes that the left leaf (a Lab node) matches rule 6 \(\rightarrow\) rewrite it as a Reg
- The rewritten tree now matches the right-hand side of rule 11 \(\rightarrow\) rewrite the entire subtree as Reg
- This sequence, denoted \(<6,11>\), reduces the entire subtree to a Reg.
Ambiguity in Rewrite Rule Selection

\[\begin{array}{cccc}
\langle 6,11 \rangle & \langle 8,14 \rangle & \langle 6,8,10 \rangle & \langle 8,6,10 \rangle \\
6 & + & 11 & 14 \\
Lab @G & Num 12 & 8 & 8 \\
\langle 6,16,9 \rangle & \langle 8,19,9 \rangle & \langle 6,8,15,9 \rangle & \langle 8,6,15,9 \rangle \\
6 & + & 16 & 19 \\
Lab @G & Num 12 & 8 & 8 \\
\end{array}\]
The instruction selector should choose a tiling that produces the lowest-cost assembly-code sequence.

Both <6,11> and <8,14> produce the lowest cost of two ➔ pick any
Other sequences are more costly ➔ hence rejected
How to tile?

• Need an algorithm to construct tiling that produces efficient code

• Assumptions:
  – Each operation has at most two operands
  – A rule’s RHS contains at most one operation
  – Easy to generalize:

\[
\alpha \rightarrow \text{op}_1(\beta, \text{op}_2(\gamma, \delta)) \quad \text{and} \quad \alpha' \rightarrow \text{op}_1(\beta, \alpha') \text{ and } \alpha' \rightarrow \text{op}_2(\gamma, \delta)
\]
How to tile?

- The compiler computes sequences of rule numbers, or patterns, for each node in a postorder traversal of the tree.

```
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 leaf */
        Label(n) ← {all rules that match the operation in n}
```

Finds tiling for a tree rooted at node n in the AST

Label(n) is a set of rule numbers that is applicable for tree rooted at n
Speeding up Tiling

• Most time spent in computing matches in the two for loops

• Improvement: replace for loop with table lookup
  – Precompute all possible matches and store the result in a 3-dim table
    • \(<\text{operation, label set (left tree), label set (right tree)}>\>

• Find lowest cost match
  – accumulate cost while tiling \(\rightarrow\) at each node, retain the lowest cost matches (locally optimal tiling)
## Code Generator Implementations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hand coded matcher</td>
<td>Avoids large space; tedious</td>
</tr>
<tr>
<td>Encode matching as an automaton</td>
<td>O(1) cost per node; Tools – BURS, BURG</td>
</tr>
<tr>
<td>Use parsing techniques</td>
<td>Standard techniques used in parsing; Have to deal with ambiguous grammar</td>
</tr>
<tr>
<td>Linearize tree into strings</td>
<td>Finds all matches</td>
</tr>
</tbody>
</table>
Peephole Optimization

• Discover local improvements by examining short sequences of adjacent instructions
  – Takes LLIR (or ASM) as input, and produces target machine code (ASM)

• Intuition
  – Use a sliding window or peephole over the code
  – Examine the operations within the window and search for specific patterns that can be improved
  – Rewrite the pattern that can be improved with the better code

• Limited pattern sets and small window size can lead to fast processing
Illustrative Examples

\[
\begin{align*}
\text{storeAI } r_1 & \Rightarrow r_{\text{arp}}, 8 \\
\text{loadAI } r_{\text{arp}}, 8 & \Rightarrow r_{15}
\end{align*}
\Rightarrow
\begin{align*}
\text{storeAI } r_1 & \Rightarrow r_{\text{arp}}, 8 \\
i2i & \quad r_1 \Rightarrow r_{15}
\end{align*}
\]

This code may even make the storeAI operation “dead” \(\Rightarrow\) can eliminate storeAI

Simple Algebraic Identities

\[
\begin{align*}
\text{addI } r_2, 0 & \Rightarrow r_7 \\
\text{mul } r_4, r_7 & \Rightarrow r_{10}
\end{align*}
\Rightarrow
\begin{align*}
\text{mul } r_4, r_2 & \Rightarrow r_{10}
\end{align*}
\]

Target of branch is another branch
Structure of Peephole Optimizer

Simple peephole optimizer:
- Use a limited set of hand-coded patterns
- Use exhaustive search to match the patterns

Expander: recognizes input code in IR form and builds an internal representation
Simplifier: rewriting operations on the IR
  - forward substitution
  - algebraic simplification
  - evaluating constant valued expressions
  - eliminate useless effects (creation of unused condition code)
Matcher: transforms the IR into target machine code
\[
\begin{array}{|c|c|c|c|}
\hline
\text{Op} & \text{Arg}_1 & \text{Arg}_2 & \text{Result} \\
\hline
\times & 2 & c & t_1 \\
- & b & t_1 & a \\
\hline
\end{array}
\]

Expand $\Rightarrow$

\[
\begin{align*}
\text{r10} & \leftarrow 2 \\
\text{r11} & \leftarrow @G \\
\text{r12} & \leftarrow 12 \\
\text{r13} & \leftarrow \text{r11} + \text{r12} \\
\text{r14} & \leftarrow \text{M(r13)} \\
\text{r15} & \leftarrow \text{r10} \times \text{r14} \\
\text{r16} & \leftarrow -16 \\
\text{r17} & \leftarrow \text{rarp} + \text{r16} \\
\text{r18} & \leftarrow \text{M(r17)} \\
\text{r19} & \leftarrow \text{M(r18)} \\
\text{r20} & \leftarrow \text{r19} - \text{r15} \\
\text{r21} & \leftarrow 4 \\
\text{r22} & \leftarrow \text{rarp} + \text{r21} \\
\text{M(r22)} & \leftarrow \text{r20}
\end{align*}
\]

Simplify

\[
\begin{align*}
\text{loadI} & \; 2 \Rightarrow \text{r10} \\
\text{loadI} & \; @G \Rightarrow \text{r11} \\
\text{loadAI} & \; \text{r11},12 \Rightarrow \text{r14} \\
\text{mult} & \; \text{r10},\text{r14} \Rightarrow \text{r15} \\
\text{loadAI} & \; \text{rarp},-16 \Rightarrow \text{r18} \\
\text{load} & \; \text{r18} \Rightarrow \text{r19} \\
\text{sub} & \; \text{r19},\text{r15} \Rightarrow \text{r20} \\
\text{storeAI} & \; \text{r20} \Rightarrow \text{rarp},4
\end{align*}
\]

Match $\Leftarrow$
Matcher

- Final pattern matching step uses familiar concepts
  - Hand-coded matchers
  - Use LR() parsers
Summary

- Two instruction selection techniques
  - Tiling based approach on IR trees
  - Peephole optimization