Compiler Construction 2009/2010
Instruction Selection

Peter Thiemann

December 1, 2009
Optimal vs Optimum Tiling

**Optimal Tiling**
No two adjacent tiles can be replaced by a larger tile of lower cost.

**Optimum Tiling**
The total cost of the tiling is minimal among all possible tilings.

- Tiling is optimum $\Rightarrow$ tiling is optimal
Temp munchExpr (Tree.Exp e) {
  test patterns from largest to smallest

  choose the first matching pattern
  with instruction INS

  foreach (e_i : wildcard (pattern, e))
    recursively invoke temp_i = munchExpr (e_i)

  emit INS using temp_i as arguments
  putting result into new temp_0

  return temp_0
}
MEM
  +
    /\    /
  CONST1 CONST2

<table>
<thead>
<tr>
<th>pattern</th>
<th>instr</th>
<th>tile cost</th>
<th>wildcard cost</th>
<th>total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONST</td>
<td>ADDI</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Pattern</td>
<td>Instruction</td>
<td>Tile Cost</td>
<td>Wildcard Cost</td>
<td>Total Cost</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-----------</td>
<td>---------------</td>
<td>------------</td>
</tr>
<tr>
<td>+</td>
<td>ADD</td>
<td>1</td>
<td>1+1</td>
<td>3</td>
</tr>
<tr>
<td>+</td>
<td>ADDI</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CONST</td>
<td>ADDI</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>CONST</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Optimum Tiling

#### Example (cont’d)

<table>
<thead>
<tr>
<th>pattern</th>
<th>instr</th>
<th>tile cost</th>
<th>wildcard cost</th>
<th>total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM</td>
<td>LOAD</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>LOAD</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>LOAD</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CONST</td>
<td>ADDI</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>LOAD</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>LOAD</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>LOAD</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>LOAD</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Optimum Tiling

Emitted Code

\[
\text{ADDI } r_1 \leftarrow r_0 + 1 \\
\text{LOAD } r_1 \leftarrow M[r_1 + 2]
\]
Implementation of Optimum Tiling
Dynamic Programming (Bottom Up)

```java
void matchExpr (Tree.Exp e) {
    for (Tree.Exp kid : e.kids())
        matchExpr (kid);

    cost = INFINITY
    for each pattern P_i
        if (P_i.matches (e)) {
            cost_i = cost(P_i)
                + sum ((wildcard (P_i, e)).mincost)
            if (cost_i < cost) { cost = cost_i; choice = i; }
        }
    e.matched = P_{choice}
    e.mincost = cost
}
```
Temp emission (Tree.Exp e) {
    foreach (e_i : wildcard (e.matched, e)) {
        temp_i = emission (e_i)
    }

    emit INS using temp_i as arguments
    putting result into new temp_0

    return temp_0
}
Implementation of Pattern Matching

- Additional side conditions (e.g., size of constants, special constants)
- Matching of patterns can be done with a decision tree that avoids checking the same node twice
- The bottom up matcher can remember partial matches and avoid rechecking the same nodes
  \[ \Rightarrow \text{tree automata} \]
A *bottom-up tree automaton* is $\mathcal{M} = (Q, \Sigma, \delta, F)$ where

- $Q$ is a finite set of states
- $\Sigma$ a ranked alphabet (the tree constructors)
- $\delta \subseteq \Sigma^{(n)} \times Q^n \times Q$ (\forall n) the transition relation
- $F \subseteq Q$ the set of final states

$\mathcal{M}$ is deterministic if $\delta$ is a function.

Define $\Rightarrow$ on $T_{\Sigma^*Q}$ by

$$t[F(q_1, \ldots, q_n)] \Rightarrow t[q_0] \quad \text{if} \quad (F, q_1, \ldots, q_n, q_0) \in \delta$$

$t \in L(\mathcal{M})$ if $t \Rightarrow^* q$ with $q \in F$
Tree automaton for

MEM

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>CONST</td>
</tr>
</tbody>
</table>

Q = \{q_t, q_c, q_a, q_m\}

F = \{q_m\}

\[ \delta = \sum \begin{array}{c|cc|c} & q_1 & q_2 & q_{out} \\ \hline \text{CONST} & & & q_c \\ \text{TEMP} & q_c & q_t & q_a \\ & + & & \\ \text{MEM} & q_a & & q_m \end{array} \]
Generate a bu tree automaton for each pattern
Simulate them in parallel on expression tree
At each node
  - determine all patterns whose root matches the current node
  - compute their cost and mark the node with the minimum cost pattern
There are tools to compile a pattern specification to such an automaton ⇒ BURG (Fraser, Hanson, Proebsting)
Extension: Different pattern sets leading to different kinds of results

Some architectures have different kinds of registers that obey different restrictions

Set of patterns for each kind of register

Example: M680x0 distinguishes data and address registers, only the latter may serve for address calculations and indirect addressing

⇒ Tree grammar needed
A **context-free tree grammar** is defined by $G = (N, \Sigma, P, S)$ where

- $N$ is a finite set of non-terminals
- $\Sigma$ is a ranked alphabet
- $S \in N$ is the start symbol
- $P \subseteq N \times T_{\Sigma+N}$

Define $\Rightarrow$ on $T_{\Sigma+N}$ by

$$t[A] \Rightarrow t[r] \quad \text{in} \quad A \rightarrow r \in P$$

$t \in L(G)$ if $S \Rightarrow^* t \in T_\Sigma$
Tree Grammars
Example: The Schizo-Jouette Architecture (Excerpt)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>( d_i \leftarrow d_j + d_k )</td>
<td>( D \rightarrow D + D )</td>
</tr>
<tr>
<td>ADDI</td>
<td>( d_i \leftarrow d_j + c )</td>
<td>( D \rightarrow D + \text{CONST} )</td>
</tr>
<tr>
<td>MOVEA</td>
<td>( d_i \leftarrow a_j )</td>
<td>( D \rightarrow A )</td>
</tr>
<tr>
<td>MOVED</td>
<td>( a_i \leftarrow d_j )</td>
<td>( A \rightarrow D )</td>
</tr>
<tr>
<td>LOAD</td>
<td>( d_i \leftarrow M[a_j + c] )</td>
<td>( A \rightarrow \text{CONST} )</td>
</tr>
</tbody>
</table>
Efficiency of Tiling

- $N$ number of nodes in input tree
- $T$ number of patterns
- $K$ average number of labeled nodes in pattern
- $K'$ maximum number of nodes to check for a match
- $T'$ average number of patterns that match at each node

**Maximal munch.** Each match consumes $K$ nodes: test for matches at $N/K$ nodes. At each candidate node, choose pattern with $K' + T'$ tests. $(K' + T')N/K$ steps on average. Worst case: $K = 1$.

**Dynamic programming.** Tests every pattern at every node: $(K' + T')N$.

⇒ same linear worst-case complexity. $(K' + T')/K$ is constant, anyway.
### CISC vs RISC

#### Challenges for Instruction Selection and Register Allocation

<table>
<thead>
<tr>
<th>RISC</th>
<th>CISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>few registers (16, 8, 6)</td>
</tr>
<tr>
<td>one class of registers</td>
<td>different classes with restricted operations</td>
</tr>
<tr>
<td>ALU instructions only between registers</td>
<td>ALU operations with memory operands</td>
</tr>
<tr>
<td>three-address instructions</td>
<td>two-address instructions</td>
</tr>
<tr>
<td>$r_1 \leftarrow r_2 \oplus r_3$</td>
<td>$r_1 \leftarrow r_1 \oplus r_2$</td>
</tr>
<tr>
<td>one addressing mode for load/store</td>
<td>several addressing modes</td>
</tr>
<tr>
<td>every instruction 32 bits long</td>
<td>different instruction lengths</td>
</tr>
<tr>
<td>one result / instruction</td>
<td>instructions w/ side effects</td>
</tr>
</tbody>
</table>


## CISC Examples

### Pentium / x86 (32-bit)
- six GPR, \(sp, bp\)
- multiply / divide only on \(eax\)
- generally two-address instructions

### MC 680x0 (32-bit)
- 8 data registers, 7 address registers, 2 stack registers
- ALU operations generally on data registers, indirect addressing only through address registers
- generally two-address instructions
- esoteric addressing modes (68030)
Challenges

- **[Few Registers]** generate temporaries and rely on register allocation
- **[Restricted Registers]** generate extra moves and hope that register allocation can get rid of them. Example:
  - Multiply on Pentium requires one operand and destination in `eax`
  - Most-significant word of result stored to `edx`

Hence for \( t_1 \leftarrow t_2 \cdot t_3 \) generate

```assembly
mov eax, t2       eax ← t2
mul t3           eax ← eax · t3; edx ← garbage
mov t1, eax      t3 ← eax
```
[Two-address instructions]
Generate extra move instructions.
For $t_1 \leftarrow t_2 + t_3$ generate

\[
\begin{align*}
\text{mov} & \quad t_1, t_2 \quad t_1 \leftarrow t_2 \\
\text{add} & \quad t_1, t_3 \quad t_1 \leftarrow t_1 + t_3;
\end{align*}
\]

[Special addressing modes]
Example: memory addressing

\[
\begin{align*}
\text{mov} & \quad eax, [ebp-8] \\
\text{add} & \quad eax, ecx \quad \text{add} \ [ebp-8], ecx \\
\text{mov} & \quad [ebp-8], eax
\end{align*}
\]

Two choices:
1. Ignore and use separate load and store instructions. Same speed, but an extra register gets trashed.
2. Avoid register pressure and use addressing mode. More work for the pattern matcher.
[Variable-length instructions]  
No problem for instruction selection or register allocation. Assembler deals with it.

[Instructions with side effects]  
Example: autoincrement after memory fetch (MC 680x0)

\[ r_2 \leftarrow M[r_1]; \quad r_1 \leftarrow r_1 + 4 \]

Hard to incorporate in tree-pattern based instruction selection.

1. Ignore. . .
2. Ad-hoc solution
3. Different algorithm for instruction selection
Class hierarchy for representing instructions

Each instruction specifies a

- set of defined temporaries
- set of used temporaries
- set of branch targets

each of which may be empty
new OPER ("LOAD 'd0 <- M['s0+8]",
    L (new Temp(), null),// targets: defined
    L (frame.FP, null)); // sources: used

- Independent of register allocation and jump labels
An operation’s def and use set must account for all defined and used registers.

- Example: the multiplication instruction on Pentium
  ```java
  new OPER ("mul 's0",
            L (pentium.EAX, L (pentium.EDX, null))
            L (argTemp, L (pentium.EAX, null)));
  ```

- Example: a procedure call trashes many registers (see the calling convention of the architecture)
  - return address
  - return-value register
  - caller-save registers