Instruction Selection

Leonidas Fegaras
Basic Blocks and Traces

• Many computer architectures have instructions that do not exactly match our IR representations
  – they do not support two-way branching as in CJUMP(op,e1,e2,l1,l2)
  – nested calls, such as CALL(f,[CALL(g,[...])]), will cause interference between register arguments and returned results
  – the nested SEQs, such as SEQ(SEQ(s1,s2),s3), impose an order of evaluation, which restricts optimization
    • if s1 and s2 do not interfere with each other, we want to be able to switch the SEQ(s1,s2) with the SEQ(s2,s1) because it may result to a more efficient program

• We will fix these problems in two phases:
  – transforming IR trees into a list of canonical trees, and
  – transforming unrestricted CJUMPs into CJUMPs that are followed by their false target label
Canonical Trees

• An IR is a *canonical tree* if it does not contain SEQ or ESEQ and the parent node of each CALL node is either an EXP or a MOVE(TEMP(t),...) node
  
  – *Method*: we transform an IR in such a way that all ESEQs are pulled up in the IR and become SEQs at the top of the tree. At the end, we are left with nested SEQs at the top of the tree, which are eliminated to form a list of statements

For example, the IR:

```
SEQ(MOVE(NAME(x),ESEQ(MOVE(TEMP(t),CONST(1)),TEMP(t))),
   JUMP(ESEQ(MOVE(NAME(z),NAME(L)),NAME(z))))
```

is translated into:

```
SEQ(SEQ(MOVE(TEMP(t),CONST(1)),
        MOVE(NAME(x),TEMP(t)))
SEQ(MOVE(NAME(z),NAME(L)),
   JUMP(NAME(z)))
```

which corresponds to a list of statements:

```
[ MOVE(TEMP(t),CONST(1)), MOVE(NAME(x),TEMP(t)),
  MOVE(NAME(z),NAME(L)), JUMP(NAME(z)) ]
```
Some Rules

- \[ \text{SEQ}(s_1, \text{ESEQ}(s_2, e)) = \text{ESEQ}(\text{SEQ}(s_1, s_2), e) \]
- \[ \text{BINOP}(\text{op}, \text{ESEQ}(s, e_1), e_2) = \text{ESEQ}(s, \text{BINOP}(\text{op}, e_1, e_2)) \]
- \[ \text{MEM}(\text{ESEQ}(s, e)) = \text{ESEQ}(s, \text{MEM}(e)) \]
- \[ \text{JUMP}(\text{ESEQ}(s, e)) = \text{SEQ}(s, \text{JUMP}(e)) \]
- \[ \text{CJUMP}(\text{op}, \text{ESEQ}(s, e_1), e_2, l_1, l_2) = \text{SEQ}(s, \text{CJUMP}(\text{op}, e_1, e_2, l_1, l_2)) \]
- \[ \text{CJUMP}(\text{op}, \text{ESEQ}(s, e_1), e_2, l_1, l_2) = \text{SEQ}(\text{MOVE}(\text{temp}(t), e_1), \text{SEQ}(s, \text{CJUMP}(\text{op}, \text{TEMP}(t), e_2, l_1, l_2))) \]
- \[ \text{MOVE}(\text{ESEQ}(s, e_1), e_2) = \text{SEQ}(s, \text{MOVE}(e_1, e_2)) \]
- To handle function calls, we store the function results into a new register:
  \[ \text{CALL}(f, a) = \text{ESEQ}(\text{MOVE}((\text{TEMP}(t), \text{CALL}(f, a))), \text{TEMP}(t)) \]
That way expressions, such as \(+\text{CALL}(f, a), \text{CALL}(g, b))\), would not rewrite each other's result register.
Basic Blocks

- Need to transform any CJUMP into a CJUMP whose false target label is the next instruction after CJUMP
  - this reflects the conditional JUMP found in most architectures
- We will do that using basic blocks
- A basic block is a sequence of statements whose first statement is a LABEL, the last statement is a JUMP or CJUMP, and does not contain any other LABELs, JUMPs, or CJUMPs
  - we can only enter at the beginning of a basic block and exit at the end
Algorithm

• We first create the basic blocks for an IR tree
• then we reorganize the basic blocks in such a way that every CJUMP at the end of a basic block is followed by a block the contains the CJUMP false target label
• A secondary goal is to put the target of a JUMP immediately after the JUMP
  – that way, we can eliminate the JUMP (and maybe merge the two blocks)
• The algorithm is based on *traces*
Traces

• You start a trace with an unmark block and you consider the target of the JUMP of this block or the false target block of its CJUMP
• then, if the new block is unmarked, you append the new block to the trace, you use it as your new start, and you apply the algorithm recursively
• otherwise, you close this trace and you start a new trace by going back to a point where there was a CJUMP and you choose the true target this time
• You continue until all blocks are marked
This is a greedy algorithm

At the end, there may be still some CJUMPs that have a false target that does not follow the CJUMP
  - this is the case where this false target label was the target of another JUMP or CJUMP found earlier in a trace
  - in that case:
    • if we have a CJUMP followed by a true target, we negate the condition and switch the true and false targets
    • otherwise, we create a new block LABEL(L) followed by JUMP(F) and we replace CJUMP(op,a,b,T,F) with CJUMP(op,a,b,T,L)

Also, if there is a JUMP(L) followed by a LABEL(L), we remove the JUMP
Instruction Selection

- After IR trees have been put into a canonical form, they are used in generating assembly code.
- The obvious way to do this is to macro-expand each IR tree node.
- For example,
  \[
  \text{MOVE(MEM(+(TEMP(fp),CONST(10))),CONST(3))}
  \]
  is macro-expanded into the pseudo-assembly code:

  \[
  \begin{align*}
  &\text{TEMP(fp)} & t1 := fp \\
  &\text{CONST(10)} & t2 := 10 \\
  &+(\text{TEMP(fp),CONST(10)}) & t3 := t1+t2 \\
  &\text{CONST(3)} & t4 := 3 \\
  &\text{MOVE(MEM(\ldots),CONST(3))} & M[t3] := t4
  \end{align*}
  \]

  where \(t_i\) stands for a temporary variable.

- This method generates very poor quality code.
- It can be done using only one instruction in most architectures: 
  \[
  M[fp+10] := 3
  \]
Maximum Munch

- Maximum munch generates better code, especially for RISC machines
- The idea is to use tree pattern matching to map a tree pattern (a fragment of an IR tree) into a list of assembly instructions
  - these tree patterns are called *tiles*
- For RISC we always have one-to-one mapping (one tile to one assembly instruction)
  - for RISC machines the tiles are small (very few number of IR nodes)
  - for CISC machines the tiles are usually large since the CISC instructions are very complex
Tiles

- The following is the mapping of some tiles into MIPS code:

<table>
<thead>
<tr>
<th>IR</th>
<th>Tile</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONST(c)</td>
<td>li 'd0, c</td>
</tr>
<tr>
<td>+(e0,e1)</td>
<td>add 'd0, 's0, 's1</td>
</tr>
<tr>
<td>+(e0,CONST(c))</td>
<td>add 'd0, 's0, c</td>
</tr>
<tr>
<td>*(e0,e1)</td>
<td>mult 'd0, 's0, 's1</td>
</tr>
<tr>
<td>*(e0,CONST(2^k))</td>
<td>sll 'd0, 's0, k</td>
</tr>
<tr>
<td>MEM(e0)</td>
<td>lw 'd0, ('s0)</td>
</tr>
<tr>
<td>MEM((e0,CONST(c)))</td>
<td>lw 'd0, c('s0)</td>
</tr>
<tr>
<td>MOVE(MEM(e0),e1)</td>
<td>sw 's1, ('s0)</td>
</tr>
<tr>
<td>MOVE(MEM((e0,CONST(c))),e1)</td>
<td>sw 's1, c('s0)</td>
</tr>
<tr>
<td>JUMP(NAME(X))</td>
<td>b X</td>
</tr>
<tr>
<td>JUMP(e0)</td>
<td>jr 's0</td>
</tr>
<tr>
<td>LABEL(X)</td>
<td>X: nop</td>
</tr>
</tbody>
</table>
Tiling

• To translate an IR tree into assembly code, we perform tiling:
  – we cover the IR tree with non-overlapping tiles
  – we can see that there are many different tilings

• eg, the IR for \( a[i] := x \) is:
  \[
  \text{MOVE(MEM(+(MEM(+fp,CONST(20))),*(TEMP(i),CONST(4))),}
  \text{MEM(+fp,CONST(10))))}
  \]

• The following are two possible tilings of the IR:

\[
\begin{align*}
\text{l} & \text{w r} 1, 20($fp)  & \text{a} & \text{d} & \text{d r} 1, $fp, 20 \\
\text{s} & \text{l} & \text{l r} 2, r 1, 2  & \text{l} & \text{w r} 1, (r 1) \\
\text{a} & \text{d} & \text{d r} 1, r 1, r 2  & \text{s} & \text{l} & \text{l r} 2, r 1, 2 \\
\text{l} & \text{w r} 2, 10($fp)  & \text{a} & \text{d} & \text{d r} 1, r 1, r 2 \\
\text{s} & \text{w} & \text{r} 2, (r 1)  & \text{a} & \text{d} & \text{d r} 2, $fp, x \\
\text{l} & \text{w} & \text{r} 2, (r 2)  & \text{s} & \text{w} & \text{r} 2, (r 1)
\end{align*}
\]

• The left tiling is obviously better since it can be executed faster
Optimum Tiling

• It's highly desirable to do optimum tiling:
  – to generate the shortest instruction sequence
  – alternatively the sequence with the fewest machine cycles

• This is not easy to achieve

• Two main ways of performing optimum tiling
  – using maximal munch (a greedy algorithm):
    • you start from the IR root and from all matching tiles
    • you select the one with the maximum number of IR nodes
    • you go to the children of this tile and apply the algorithm recursively until you reach the tree leaves
  – using dynamic programming:
    • it works from the leaves to the root
    • it assigns a cost to every tree node by considering every tile that matches the node and calculating the minimum value of:
      \[
      \text{cost of a node} = (\text{number of nodes in the tile}) + (\text{total costs of all the tile children})
      \]
Maximal Munch Example

A lw r1, fp
B lw r2, 8(r1)
C lw r3, i
D sll r4, r3, 2
E add r5, r2, r4
F lw r6, fp
G lw r7, 16(r6)
H add r8, r7, 1
I sw r8, (r5)