Reading Assignment

• Read pages 1-30 of “Automatic Program Optimization,” by Ron Cytron. (Linked from the class Web page.)
Example

Let’s look at instruction selection for

\[ a = b - 1; \]

where \( a \) is a global int, accessed with a 32 bit address and \( b \) is a local int, accessed as an offset from the frame pointer.
We match tree nodes **bottom-up**. Each node is labeled with the nonterminals it can be reduced to, the production used to produce the nonterminal, and the cost to generate the node (and its children) from the nonterminal.

We match leaves first:
We now work upward, considering operators whose children have been labeled. Again, if an operator can be generated by a nonterminal, we mark the operator with the nonterminal, the production used to generate the operator, and the total cost (including the cost to generate all children).

If a nonterminal can generate the operator using more than one production, the least-cost derivation is chosen.

When we reach the root, the nonterminal with the lowest overall cost is used to generate the tree.
\[ \text{uint}:R0:0 \quad \text{reg}:R6:2 \quad \text{int32} \quad * \quad \text{reg}:R9:1 \quad s13 \quad \text{reg}:R8:2 \quad \text{imm}:R4:0 \quad \text{reg}:R5:1 \quad \text{adr}:R3:0 \quad \text{reg}:R1:0 \quad \text{adr}:R2:0 \quad r \quad s13 \quad \text{imm}:R4:0 \quad \text{reg}:R5:1 \quad \text{void}:R10:4 \]
Note that once we know the production used to generate the root of the tree, we know the productions used to generate each subtree too:

```
= Void:R10:4

(UInt:R0:0
 int32
 -

(* Reg:R8:2
  s13 Imm:R4:0

 +

 Adr:R3:0

 Reg:R1:0

 r

 s13 Imm:R4:0)
```
We generate code by doing a depth-first traversal, generating code for a production after all the production’s children have been processed.

We need to do register allocation too; for our example, a simple on-the-fly generator will suffice.

```
1. ld    [%fp+b],%l0
2. sub   %l0,1,%l0
3. sethi %hi(a),%g1
4. st    %l0,[%g1+%lo(a)]
```
Had we translated a slightly difference expression,

\[ a = b - 1000000; \]

we would automatically get a different code sequence (because 1000000 is an int32 rather than an s13):

\[
\begin{align*}
\text{ld} & \quad \text{[fp+b]},%l0 \\
\text{sethi} & \quad \%hi(1000000),%g1 \\
\text{or} & \quad %g1,%lo(1000000),%l1 \\
\text{sub} & \quad %l0,%l1,%l0 \\
\text{sethi} & \quad %hi(a),%g1 \\
\text{st} & \quad %l0,[%g1+%lo(a)]
\end{align*}
\]
Adding New Rules

Since instruction selectors can be automatically generated, it’s easy to add “extra” rules that handle optimizations or special cases. For example, we might add the following to handle addition of a left immediate operand or subtraction of 0 from a register:

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Production</th>
<th>Cost</th>
<th>SPARC Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>R11</td>
<td>Reg → + Imm Reg</td>
<td>1</td>
<td>add Reg, Imm, Reg</td>
</tr>
<tr>
<td>R12</td>
<td>Reg → − Reg Zero</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Improving the Speed of Instruction Selection

As we have presented it, instruction selection looks rather slow—for each node in the IR tree, we must match productions, compare costs, and select least-cost productions. Since compilers routinely generate programs with tens or hundreds of thousands of instructions, doing a lot of computation to select one instruction (even if it’s the best instruction) could be too slow.

Fortunately, this need not be the case. Instruction selection using BURS can be made very fast.
Adding States to BURG

We can precompute a set of states that represent possible labelings on IR tree nodes. A table of node names and subtree states then is used to select a node’s state. Thus labeling becomes nothing more than repeated table lookup.

For example, we might create a state $s_0$ that corresponds to the labeling \{Reg:$R_1$:0, Adr:$R_2$:0\}.

A state selection function, $\text{label}$, defines $\text{label}(r) = s_0$. That is, whenever $r$ is matched as a leaf, it is to be labeled with $s_0$.

If a node is an operator, $\text{label}$ uses the name of the operator and the labeling
assigned to its children to choose the operator’s label. For example,

\[ \text{label}(+, s_0, s_1) = s_2 \]
says that a + with children labeled as \( s_0 \) and \( s_1 \) is to be labeled as \( s_2 \).

In theory, that’s all there is to building a fast instruction selector.

We generate possible labelings, encode them as states, and table all combinations of labelings.

But,

how do we know the set of possible labelings is even finite?

In fact, it isn’t!
Normalizing Costs

It is possible to generate states that are identical except for their costs.

For example, we might have

\( s_1 = \{ \text{Reg:R1:0, Adr:R2:0} \} \),

\( s_2 = \{ \text{Reg:R1:1, Adr:R2:1} \} \),

\( s_3 = \{ \text{Reg:R1:2, Adr:R2:2} \} \), etc.

Here an important insight is needed—the absolute costs included in states aren’t really essential. Rather relative costs are what is important. In \( s_1, s_2, \) and \( s_3 \), Reg and Adr have the same cost. Hence the same decision in choosing between Reg and Adr will be made in all three states.
We can limit the number of states needed by normalizing costs within states so that the lowest cost choice is always 0, and other costs are differences (deltas) from the lowest cost choice.

This observation keeps costs bounded within states (except for pathologic cases).

Using additional techniques to further reduce the number of states needed, and the time needed to generate them, fast and compact BURS instruction selectors are achievable. See “Simple and Efficient BURS Table Generation,” T. Proebsting, 1992 PLDI Conference.
### Example

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>{Reg:R1:0, Adr:R2:0}</td>
</tr>
<tr>
<td>s1</td>
<td>{Imm:R4:0, Reg:R5:1}</td>
</tr>
<tr>
<td>s2</td>
<td>{adr:R3:0}</td>
</tr>
<tr>
<td>s3</td>
<td>{Reg:R9:0}</td>
</tr>
<tr>
<td>s4</td>
<td>{UInt:R0:0}</td>
</tr>
<tr>
<td>s5</td>
<td>{Reg:R8:0}</td>
</tr>
<tr>
<td>s6</td>
<td>{Void:R10:0}</td>
</tr>
<tr>
<td>s7</td>
<td>{Reg:R7:0}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Left Child</th>
<th>Right Child</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td></td>
<td></td>
<td>s0</td>
</tr>
<tr>
<td>s13</td>
<td></td>
<td></td>
<td>s1</td>
</tr>
<tr>
<td>int32</td>
<td></td>
<td></td>
<td>s4</td>
</tr>
<tr>
<td>+</td>
<td>s0</td>
<td>s1</td>
<td>s2</td>
</tr>
<tr>
<td>*</td>
<td>s2</td>
<td></td>
<td>s3</td>
</tr>
<tr>
<td>-</td>
<td>s3</td>
<td>s1</td>
<td>s5</td>
</tr>
<tr>
<td>-</td>
<td>s1</td>
<td>s3</td>
<td>s7</td>
</tr>
<tr>
<td>=</td>
<td>s4</td>
<td>s5</td>
<td>s6</td>
</tr>
</tbody>
</table>
We start by looking up the state assigned to each leaf. We then work upward, choosing the state of a parent based on the parent's kind and the states assigned to the children. These are all table lookups, and hence very fast.

At the root, we select the nonterminal and production based on the state assigned to the root (any entry with 0 cost). Knowing the production used at the root tells us the nonterminal used at each child. Each state has only one entry per nonterminal, so knowing a node’s state and the nonterminal used to generate it immediately tells us the production used. Hence identifying the production used for each node is again very fast.
Step 1 (Label leaves with states):

\[
\begin{align*}
&= \\
&\text{int32} \\
&\ast \\
&+ \\
&s0 \\
&s13 \\
&s1 \\
&s4
\end{align*}
\]

Step 2 (Propagate states upward):

\[
\begin{align*}
&= s6 \\
&\text{int32} \\
&\ast \\
&s3 \\
&s5 \\
&s13 \\
&s1 \\
&s4 \\
&s2 \\
&s0 \\
&r \\
&s13 \\
&s1
\end{align*}
\]
Step 3 (Choose production used at root): R10.

Step 4 (Propagate productions used downward to children):

```
= R10

R0, int32

- R8

R3

R1 R4

R9

s13

s13 R4

r

s13 R4
```
Data Flow Frameworks

- Data Flow Graph:
  Nodes of the graph are basic blocks or individual instructions.
  Arcs represent flow of control.
- Forward Analysis:
  Information flow is the same direction as control flow.
- Backward Analysis:
  Information flow is the opposite direction as control flow.
- Bi-directional Analysis:
  Information flow is in both directions. (Not too common.)
- **Meet Lattice**

  Represents solution space for the data flow analysis.

- **Meet operation** (And, Or, Union, Intersection, etc.)

  Combines solutions from predecessors or successors in the control flow graph.
Transfer Function

Maps a solution at the top of a node to a solution at the end of the node (forward flow)
or
Maps a solution at the end of a node to a solution at the top of the node (backward flow).
Example: Available Expressions

This data flow analysis determines whether an expression that has been previously computed may be reused.

Available expression analysis is a forward flow problem—computed expression values flow forward to points of possible reuse.

The best solution is True—the expression may be reused.

The worst solution is False—the expression may not be reused.
The Meet Lattice is:

- T (Expression is Available)
- F (Expression is Not Available)

As initial values, at the top of the start node, nothing is available. Hence, for a given expression,
\[ \text{AvailIn}(b_0) = F \]

We choose an expression, and consider all the variables that contribute to its evaluation.

Thus for \( e_1 = a + b - c \), a, b and c are \( e_1 \)'s operands.
The transfer function for $e_1$ in block $b$ is defined as:

If $e_1$ is computed in $b$ after any assignments to $e_1$’s operands in $b$

Then $\text{AvailOut}(b) = T$

Elsif any of $e_1$’s operands are changed after the last computation of $e_1$ or $e_1$’s operands are changed without any computation of $e_1$

Then $\text{AvailOut}(b) = F$

Else $\text{AvailOut}(b) = \text{AvailIn}(b)$

The meet operation (to combine solutions) is:

$$\text{AvailIn}(b) = \bigwedge_{p \in \text{Pred}(b)} \text{AvailOut}(p)$$
Example: $e_1 = v + w$
Circularities Require Care

Since data flow values can depend on themselves (because of loops), care is required in assigning initial “guesses” to unknown values. Consider

If the flow value on the loop backedge is initially set to false, it can never become true. (Why?) Instead we should use True, the identity for the AND operation.