Instruction Selection

CSE 501
Lecture 11
May 6, 2013
An optimizing compiler

SL → FE → IL → optimizer → IL → BE → TL

Remember this picture?
We've been focusing on optimization.
Now we'll spend some time looking at the back end, aka the code generator.

We'll focus on cases where the TL = assembly language for some machine.
Code generation

Pursuing our optimistic separation of concerns, we think of code generation as having three parts:

• Instruction selection,
• Register allocation, and
• Instruction scheduling.

And, as usual, whenever there are multiple phases, there's a phase-ordering problem.

For some machines, one part or another won't matter to much.

In other, tougher cases, we'll have to explore new ideas.
Instruction selection

For best optimization, we like an IR that's very low level. Trying to expose every detail of the computation to optimization.

Typically, target machine have instructions that do more than one thing at once. For example, this bit of x86-64 assembly

\[
\text{add } \%r8, \quad \$123(\%r9, \%r10, 2)
\]

which I'd write out as

\[
\begin{align*}
t1 & = \text{r10} \ll 2 \\
t2 & = t1 + \text{r9} \\
t3 & = t3 + 123 \\
t4 & = *t3 \\
t5 & = t4 + \text{r8} \\
*{t3} & = t5
\end{align*}
\]

oh, and there's condition codes too!
An example

Typically, we'll be considering each basic block individually. Here's IR for a simple statement, $i = c + 4$, where $c$ is a char and $i$ is an int.

\[
\begin{align*}
t1 &= &i \\
t2 &= &c \\
t3 &= &t2 \\
t4 &= c2i(t3) \\
t5 &= 4 \\
t6 &= t4 + t5 \\
*t1 &= t6
\end{align*}
\]

The details will vary depending on your IR and your optimizer.

If we're in a hurry, or targeting a RISC machine, we could just choose 7 simple assembly instructions, but we like to do better.
Selection via peephole optimization

An approach pursued by Chris Fraser and his students. Considers pairs of (not necessarily adjacent) instructions and tries to replace them with a single, cheaper instruction. For example,

\[
\begin{align*}
  t2 & = \&c \\
  t3 & = *t2
\end{align*}
\]

might be replaced by \( t3 = c \)

Similarly,

\[
\begin{align*}
  t5 & = 4 \\
  t6 & = t4 + t5
\end{align*}
\]

might become \( t6 = t4 + 4 \)
Which pairs of instructions?

In olden times, we looked at all pairs of adjacent instructions. Fraser noticed that most useful pairs were connected by UD chains.

\[
\begin{align*}
t2 & = & c \\
t3 & = & *t2 \\
t5 & = & 4 \\
t6 & = & t4 + t5
\end{align*}
\]

Suddenly we had a plan:

*Consider each instruction together with the instructions that feed it.*

And since UD chains reach between blocks, our instruction selector can work globally. Works great with SSA too.
Implementation

Reduce a pair of instructions to a schematic, e.g.,

\[
\begin{align*}
  t5 &= 4 \\
  t6 &= 4 + t5
\end{align*}
\]

becomes

\[
\begin{align*}
  \texttt{\%reg1} &= \texttt{\%con} \\
  \texttt{\%reg3} &= \texttt{\%reg2} + \texttt{\%reg1}
\end{align*}
\]

then look it up in a hash table. If found, remove the source instruction and replace the destination.

\[
\begin{align*}
  \texttt{\%reg3} &= \texttt{\%reg2} + \texttt{\%con}
\end{align*}
\]

and fill in the details

\[
\begin{align*}
  t6 &= t4 + 4
\end{align*}
\]
Finding patterns

For a RISC machine, it's plausible to write all the patterns by hand. For a CISC machine, cleverness is required.

Fraser used 2 ideas:
- Build a machine description that can be explored automatically
- When a schematic isn't found in the hash table, look for it in the machine description

Record the results of all MD searches in the hash table.

Naturally, writing a useful MD is an interesting problem too.
Greedy

This scheme is greedy, no attempt to find the best translation. And what's the best anyway? Fastest, smallest, lowest power? Consider

\[
\begin{align*}
  r_2 &= *r_1 \\
  r_3 &= 10 \\
  r_4 &= r_2 + r_3
\end{align*}
\]

Which do we prefer?

\[
\begin{align*}
  r_3 &= 10 & r_2 &= *r_1 \\
  r_4 &= *r_1 + r_3 & r_4 &= r_2 + 10
\end{align*}
\]

We'd like to choose based on some informed notion of "best"
Tools

iburg and burg (and others) use a combination of *dynamic programming* and *tree pattern matching* to find optimal translations.

These tools encapsulate the results of years of research. The results are practical and efficient.

Not really a research area anymore (almost like parsing), although there are certainly opportunities to make better tools.

(Nevertheless, I'll discuss research opportunities towards the end)
Tool chain

spec

iburg

cc

SL → FE → OPT → BE → TL
A tree-based representation

Remember our example?

\[
\begin{align*}
  t1 &= \&i \\
  t2 &= \&c \\
  t3 &= *t2 \\
  t4 &= c2i(t3) \\
  t5 &= 4 \\
  t6 &= t4 + t5 \\
  *t1 &= t6
\end{align*}
\]

A tree-based representation makes the UD chains explicit. Plus, each definition in the tree is used exactly once.

Typically, a basic block would have a sequence of expression trees.
Dynamic programming

Aho & Johnson published an approach, based on dynamic programming, for making the best choices for instruction selection from expression trees.

They work in two passes over the tree:
1. bottom up, from leaves to the root, and
2. top down, from the root back to the leaves.

On the way up, they record costs for all of the alternatives. On the way down, they choose among the low costs.
Tree patterns

An iburg specification is a set of rules, where

\[
\begin{align*}
\text{rule} &= \text{nonterm} : \text{tree} = \text{integer} (\ \text{cost} ) \\
\text{tree} &= \text{term} (\ \text{tree}, \ \text{tree}) \\
| &\ \text{term} (\ \text{tree}) \\
| &\ \text{term} \\
| &\ \text{term} \\
| &\ \text{nonterm}
\end{align*}
\]

Terminals are IR operations and nonterminals are used to name sets of rules.
For example

```plaintext
stmt : ASSIGN(addr, reg) = 1 (1)
stmt : reg = 2 (0)
reg   : ADD(reg, rc) = 3 (1)
reg   : ADD(rc, reg) = 4 (1)
reg   : LD(addr) = 5 (1)
reg   : C2I(LD(addr)) = 6 (1)
reg   : addr = 7 (1)
reg   : con = 8 (1)
addr  : ADD(reg, con) = 9 (0)
addr  : ADD(con, reg) = 10 (0)
addr  : ADDRLP = 11 (0)
rc    : con = 12 (0)
rc    : reg = 13 (0)
con   : CNST = 14 (0)
```

Usually this form would be generated from a higher-level spec.
Bottom up (labelling)

\[
\text{stmt} : \text{reg} = 2 \ (0) \\
\text{reg} : \text{addr} = 7 \ (1) \\
\text{addr} : \text{ADDRLP} = 11 \ (0) \\
\text{rc} : \text{reg} = 13 \ (0)
\]

\[
\begin{array}{cccccc}
\text{stmt} & \text{reg} & \text{addr} & \text{rc} & \text{con} \\
\text{a} & \text{b} & \text{c} & \text{d} & \text{e} \\
\text{f} & \text{g} & (2, 1) & (7, 1) & (11, 0) & (13, 1)
\end{array}
\]
Bottom up (labelling)

\[
\text{stmt : reg} = 2 \ (0)
\]
\[
\text{reg} \ : \ \text{LD(addr)} = 5 \ (1)
\]
\[
\text{rc} \ : \ \text{reg} = 13 \ (0)
\]

<table>
<thead>
<tr>
<th>stmt</th>
<th>reg</th>
<th>addr</th>
<th>rc</th>
<th>con</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>(2,1)</td>
<td>(5,1)</td>
<td>(13,1)</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>(2,1)</td>
<td>(7,1)</td>
<td>(11,0)</td>
<td>(13,1)</td>
</tr>
</tbody>
</table>
Bottom up (labelling)

stmt : reg = 2 (0)
reg : con = 8 (1)
rc : con = 12 (0)
con : CNST = 14 (0)

<table>
<thead>
<tr>
<th>stmt</th>
<th>reg</th>
<th>addr</th>
<th>rc</th>
<th>con</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>(2, 1)</td>
<td>(8, 1)</td>
<td>(12, 0)</td>
<td>(14, 0)</td>
</tr>
<tr>
<td>f</td>
<td>(2, 1)</td>
<td>(5, 1)</td>
<td>(13, 1)</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
</tbody>
</table>

---

\[
\begin{align*}
\text{stmt} & : \text{reg} = 2 \ (0) \\
\text{reg} & : \text{con} = 8 \ (1) \\
\text{rc} & : \text{con} = 12 \ (0) \\
\text{con} & : \text{CNST} = 14 \ (0)
\end{align*}
\]
Bottom up (labelling)

stmt : reg = 2 (0)
reg : C2I(LD(addr)) = 6 (1)
rc : reg = 13 (0)

<table>
<thead>
<tr>
<th>stmt</th>
<th>reg</th>
<th>addr</th>
<th>rc</th>
<th>con</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>(2,1)</td>
<td>(6,1)</td>
<td>(13,1)</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>(2,1)</td>
<td>(8,1)</td>
<td>(12,0)</td>
<td>(14,0)</td>
</tr>
<tr>
<td>f</td>
<td>(2,1)</td>
<td>(5,1)</td>
<td>(13,1)</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>(2,1)</td>
<td>(7,1)</td>
<td>(11,0)</td>
<td>(13,1)</td>
</tr>
</tbody>
</table>
Bottom up (labelling)

stmt : reg = 2 (0)
reg : ADD(reg, rc) = 3 (1)
addr : ADD(reg, con) = 9 (0)
rc : reg = 13 (0)

<table>
<thead>
<tr>
<th>stmt</th>
<th>reg</th>
<th>addr</th>
<th>rc</th>
<th>con</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>(2,2)</td>
<td>(3,2)</td>
<td>(9,1)</td>
<td>(13,2)</td>
</tr>
<tr>
<td>d</td>
<td>(2,1)</td>
<td>(6,1)</td>
<td></td>
<td>(13,1)</td>
</tr>
<tr>
<td>e</td>
<td>(2,1)</td>
<td>(8,1)</td>
<td>(12,0)</td>
<td>(14,0)</td>
</tr>
<tr>
<td>f</td>
<td>(2,1)</td>
<td>(5,1)</td>
<td></td>
<td>(13,1)</td>
</tr>
<tr>
<td>g</td>
<td>(2,1)</td>
<td>(7,1)</td>
<td>(11,0)</td>
<td>(13,1)</td>
</tr>
</tbody>
</table>
Bottom up (labelling)

stmt : reg = 2 (0)
reg : addr = 7 (1)
addr : ADDRLP = 11 (0)
rc : reg = 13 (0)

<table>
<thead>
<tr>
<th>stmt</th>
<th>reg</th>
<th>addr</th>
<th>rc</th>
<th>con</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
<tr>
<td>c</td>
<td>(2, 2)</td>
<td>(3, 2)</td>
<td>(9, 1)</td>
<td>(13, 2)</td>
</tr>
<tr>
<td>d</td>
<td>(2, 1)</td>
<td>(6, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>e</td>
<td>(2, 1)</td>
<td>(8, 1)</td>
<td>(12, 0)</td>
<td>(14, 0)</td>
</tr>
<tr>
<td>f</td>
<td>(2, 1)</td>
<td>(5, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>g</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
</tbody>
</table>
Bottom up (labelling)

stmt : ASSIGN(addr, reg) = 1 (1)

<table>
<thead>
<tr>
<th>stmt</th>
<th>reg</th>
<th>addr</th>
<th>rc</th>
<th>con</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1,3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2,1</td>
<td>7,1</td>
<td>11,0</td>
<td>13,1</td>
</tr>
<tr>
<td>c</td>
<td>2,2</td>
<td>3,2</td>
<td>9,1</td>
<td>13,2</td>
</tr>
<tr>
<td>d</td>
<td>2,1</td>
<td>6,1</td>
<td></td>
<td>13,1</td>
</tr>
<tr>
<td>e</td>
<td>2,1</td>
<td>8,1</td>
<td>12,0</td>
<td>14,0</td>
</tr>
<tr>
<td>f</td>
<td>2,1</td>
<td>5,1</td>
<td></td>
<td>13,1</td>
</tr>
<tr>
<td>g</td>
<td>2,1</td>
<td>7,1</td>
<td>11,0</td>
<td>13,1</td>
</tr>
</tbody>
</table>
Top down (reduction)

stmt : ASSIGN(addr, reg) = 1 (1)

<table>
<thead>
<tr>
<th>stmt</th>
<th>reg</th>
<th>addr</th>
<th>rc</th>
<th>con</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(1, 3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
<tr>
<td>c</td>
<td>(2, 2)</td>
<td>(3, 2)</td>
<td>(9, 1)</td>
<td>(13, 2)</td>
</tr>
<tr>
<td>d</td>
<td>(2, 1)</td>
<td>(6, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>e</td>
<td>(2, 1)</td>
<td>(8, 1)</td>
<td></td>
<td>(12, 0)</td>
</tr>
<tr>
<td>f</td>
<td>(2, 1)</td>
<td>(5, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>g</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
</tbody>
</table>
Top down (reduction)

addr : ADDRLP = 11 (0)

<table>
<thead>
<tr>
<th>stmt</th>
<th>reg</th>
<th>addr</th>
<th>rc</th>
<th>con</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(1, 3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
<tr>
<td>c</td>
<td>(2, 2)</td>
<td>(3, 2)</td>
<td>(9, 1)</td>
<td>(13, 2)</td>
</tr>
<tr>
<td>d</td>
<td>(2, 1)</td>
<td>(6, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>e</td>
<td>(2, 1)</td>
<td>(8, 1)</td>
<td></td>
<td>(12, 0)</td>
</tr>
<tr>
<td>f</td>
<td>(2, 1)</td>
<td>(5, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>g</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
</tbody>
</table>
Top down (reduction)

reg: \text{ADD}(\text{reg}, \text{rc}) = 3 \ (1)

<table>
<thead>
<tr>
<th>stmt</th>
<th>reg</th>
<th>addr</th>
<th>rc</th>
<th>con</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>\text{(1, 3)}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
<tr>
<td>c</td>
<td>(2, 2)</td>
<td>\text{(3, 2)}</td>
<td>(9, 1)</td>
<td>(13, 2)</td>
</tr>
<tr>
<td>d</td>
<td>(2, 1)</td>
<td>(6, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>e</td>
<td>(2, 1)</td>
<td>(8, 1)</td>
<td>(12, 0)</td>
<td>(14, 0)</td>
</tr>
<tr>
<td>f</td>
<td>(2, 1)</td>
<td>(5, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>g</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
</tbody>
</table>
Top down (reduction)

\[ \text{reg} : \text{C2I(LD(addr))} = 6 \ (1) \]
Top down (reduction)

addr : ADDRLP = 11 (0)

<table>
<thead>
<tr>
<th>stmt</th>
<th>reg</th>
<th>addr</th>
<th>rc</th>
<th>con</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(1, 3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
<tr>
<td>c</td>
<td>(2, 2)</td>
<td>(3, 2)</td>
<td>(9, 1)</td>
<td>(13, 2)</td>
</tr>
<tr>
<td>d</td>
<td>(2, 1)</td>
<td>(6, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>e</td>
<td>(2, 1)</td>
<td>(8, 1)</td>
<td></td>
<td>(12, 0)</td>
</tr>
<tr>
<td>f</td>
<td>(2, 1)</td>
<td>(5, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>g</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
</tbody>
</table>
Top down (reduction)

\[ rc : \text{con} = 12 (0) \]

<table>
<thead>
<tr>
<th>stmt</th>
<th>reg</th>
<th>addr</th>
<th>rc</th>
<th>con</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(1, 3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
<tr>
<td>c</td>
<td>(2, 2)</td>
<td>(3, 2)</td>
<td>(9, 1)</td>
<td>(13, 2)</td>
</tr>
<tr>
<td>d</td>
<td>(2, 1)</td>
<td>(6, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>e</td>
<td>(2, 1)</td>
<td>(8, 1)</td>
<td>(12, 0)</td>
<td>(14, 0)</td>
</tr>
<tr>
<td>f</td>
<td>(2, 1)</td>
<td>(5, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>g</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
</tbody>
</table>
Top down (reduction)

con : CNST = 14 (0)

<table>
<thead>
<tr>
<th>stmt</th>
<th>reg</th>
<th>addr</th>
<th>rc</th>
<th>con</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(1, 3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
<tr>
<td>c</td>
<td>(2, 2)</td>
<td>(3, 2)</td>
<td>(9, 1)</td>
<td>(13, 2)</td>
</tr>
<tr>
<td>d</td>
<td>(2, 1)</td>
<td>(6, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>e</td>
<td>(2, 1)</td>
<td>(8, 1)</td>
<td></td>
<td>(12, 0)</td>
</tr>
<tr>
<td>f</td>
<td>(2, 1)</td>
<td>(5, 1)</td>
<td></td>
<td>(13, 1)</td>
</tr>
<tr>
<td>g</td>
<td>(2, 1)</td>
<td>(7, 1)</td>
<td>(11, 0)</td>
<td>(13, 1)</td>
</tr>
</tbody>
</table>
Plug

For my last code generator, I wrote a preprocessor for iburg.
I wanted:
• the iburg rules
• semantic actions
• ways to avoid some of the repetition, and
• some checking.

It turned out pretty well.
An example

Here's how I specified the implementation of gcc's \texttt{plus:DI} operation on the Opteron:

\begin{verbatim}
regex : PLUS_DI(regex, reg) [1, 3]
    name {
        $$->reg = $1->reg;
    },
    dump {
        dumpRR("addq", $2->reg, $$->reg);
    };
\end{verbatim}

Plus will process this and spit out a line to the iburg file, e.g.,

\begin{verbatim}
regs : PLUS(regex, reg) = 1 (259)
\end{verbatim}

plus appropriate lines to \texttt{name.cases} and \texttt{dump.cases}
Commutativity and associativity

I also added a way to take advantage of operations that commute and associate. For example, the pattern

```
rx : plus(r | mem(plus(base| index | imm)))
```

describes an add-from-memory instruction for the Opteron.

Plug expands it into 24 separate iburg rules.
Better yet, ...  

Robert Henry reworked plug, throwing out the positional parameters and using named parameters. So, instead of

```plaintext
regex : PLUS_DI(regx, reg) [1, 3]
dump {
    dumpRR("addq", $2->reg, $$->reg);
};
```

he'd write

```plaintext
regex.dst : PLUS_DI(regx.src1, reg.src2) [1, 3]
dump {
    dumpRR("addq", $src2, $src1);
};
```

Allowed many rules to share the same set of reductions (a giant savings) and supported better checking.
Weaknesses

For the combination of gcc and the Opteron, we needed a lot of rules (2500 rules, expanding to ~100K lines of C).

Writing some C code to canonicalize the expression trees would have save a lot.

Can't handle DAGs (but see Koes & Goldstein)

I had about a zillion rules for handling different-sized constants. Might have been cleaner to handle with dynamic costs.

Certain peephole optimizations would have been enabled by dynamic costs.
Can we do better?

Sure! Koes and Goldstein extend iburg to handle DAGs (versus just trees). Not quite optimal, but not bad. This is what LLVM uses.

Where are they weak?
They can't handle instructions that have multiple results, e.g.,
• divrem, produces quotient and remainder
• sincos, produces both sine and cosine of an argument
• swap, exchanges 2 registers, or perhaps a register and memory

While we can hack around such things, it'd be nicer to have a framework strong enough to handle them naturally.
Other applications

hardware synthesis
data structure selection
relational algebra