Instruction Selection

15-745 Optimizing Compilers
Spring 2007

Compiler Backend

Intermediate Representations

Front end - produces an intermediate representation (IR)
Middle end - transforms the IR into an equivalent IR that runs more efficiently
Back end - transforms the IR into native code

IR encodes the compiler’s knowledge of the program
Middle end usually consists of several passes
Intermediate Representations

Decisions in IR design affect the speed and efficiency of the compiler.

Some important IR properties:
- Ease of generation
- Ease of manipulation
- Procedure size
- Freedom of expression
- Level of abstraction

The importance of different properties varies between compilers:
- Selecting an appropriate IR for a compiler is critical.

Types of Intermediate Representations

Structural
- Graphically oriented
- Heavily used in source-to-source translators
- Tend to be large

Examples: Trees, DAGs

Linear
- Pseudo-code for an abstract machine
- Level of abstraction varies
- Simple, compact data structures
- Easier to rearrange

Examples: 3 address code, Stack machine code

Hybrid
- Combination of graphs and linear code

Example: Control-flow graph

Level of Abstraction

The level of detail exposed in an IR influences the profitability and feasibility of different optimizations.

Two different representations of an array reference:

- **High level AST:** Good for memory disambiguation
  - `load 1 => r_1`
  - `sub r_2, r_1 => r_3`
  - `load 10 => r_3`
  - `mult r_2, r_3 => r_4`
  - `sub r_5, r_4 => r_6`
  - `add r_4, r_6 => r_7`
  - `load @A => r_7`
  - `add r_5, r_7 => r_8`
  - `load r_8 => r_{ij}

- **Low level linear code:** Good for address calculation
  - `loadArray A, i, j`

Level of Abstraction

Structural IRs are usually considered high-level.
Linear IRs are usually considered low-level.
Not necessarily true:

- `load A, i, j`
- `10`
- `J`
- `1`
Abstract Syntax Tree

An abstract syntax tree is the procedure’s parse tree with the nodes for most non-terminal nodes removed.

Directed Acyclic Graph

A directed acyclic graph (DAG) is an AST with a unique node for each value.

When is an AST a good IR?

Structural IR

Directed Acyclic Graph

Directed Acyclic Graph

Structural IR

Stack Machine Code

Originally used for stack-based computers, now Java.

Example:

```
x - 2 * y
```

Advantages

- Compact form
- Introduced names are implicit, not explicit
- Simple to generate and execute code

Useful where code is transmitted over slow communication links (the net)
Three Address Code

Three address code has statements of the form:

\[ x \leftarrow y \ op \ z \]

With 1 operator \((\text{op})\) and, at most, 3 names (\(x, y, \text{& } z\))

Example:

\[ z \leftarrow x - 2 \ast y \text{ becomes } t \leftarrow 2 \ast y \]

\[ z \leftarrow x - t \]

Advantages:
- Resembles many machines (RISC)
- Compact form

Two Address Code

Allows statements of the form

\[ x \leftarrow x \ op \ y \]

Has 1 operator \((\text{op})\) and, at most, 2 names (\(x\) and \(y\))

Example:

\[ z \leftarrow x - 2 \ast y \text{ becomes } t_1 \leftarrow 2 \]
\[ t_2 \leftarrow \text{load } y \]
\[ t_2 \leftarrow t_2 \ast t_2 \]
\[ z \leftarrow \text{load } x \]
\[ z \leftarrow z - t_2 \]

- Can be very compact
- Destructive operations make reuse hard
- Good model for machines with destructive ops (x86)

Control-flow Graph

Models the transfer of control in the procedure

Nodes in the graph are basic blocks
- Straight-line code
- Either linear or structural representation

Edges in the graph represent control flow

Example:

\[
\begin{align*}
\text{if } (x = y) & \rightarrow \text{Basic blocks} = \text{Maximal length sequences of straight-line code} \\
\text{a} \leftarrow 2 & \\
\text{b} \leftarrow 5 & \\
\text{a} \leftarrow 3 & \\
\text{b} \leftarrow 4 & \\
\text{c} \leftarrow \text{a} \ast \text{b} &
\end{align*}
\]
Instruction selection example

Suppose we have

\[
\text{MOVE(TEMP } r, \\
\text{MEM(BINOP(TIMES,TEMP s,CONST c)))}
\]

We can generate the x86 code...

\[
\begin{align*}
\text{movl } &\%(_s,c), \%r \\
\text{imull } &\$c, \%s, \%r \\
\text{movl } &\%r, \%r
\end{align*}
\]

…if \( c = 1, 2, \text{ or } 4; \) otherwise...

\[
\begin{align*}
\text{testl } &\$0,(_s,c) \\
\text{je } &L
\end{align*}
\]

Selection dependencies

We can see that the selection of instructions can depend on the constants.

The context of an IR expression can also affect the choice of instruction.

Consider

\[
\begin{align*}
\text{MOVE(TEMP } r, \\
\text{MEM(BINOP(TIMES,TEMP s,CONST c)))}
\end{align*}
\]

Example, cont’d

For

\[
\text{MEM(BINOP(TIMES,TEMP s,CONST c))}
\]

we might sometimes want to generate

\[
\begin{align*}
\text{testl } &\$0,(_s,c) \\
\text{je } &L
\end{align*}
\]

What context might cause us to do this?
Instruction selection as tree matching

In order to take context into account, instruction selectors often use pattern-matching on IR trees – each pattern specifies what instructions to select.

Sample tree-matching rules

<table>
<thead>
<tr>
<th>IR pattern</th>
<th>code</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINOP(PLUS,i,j)</td>
<td>leal (i,j),r</td>
<td>1</td>
</tr>
<tr>
<td>BINOP(TIMES,i,j)</td>
<td>movl j,r</td>
<td>2</td>
</tr>
<tr>
<td>BINOP(TIMES,i,CONST c)</td>
<td>imull i,r</td>
<td></td>
</tr>
<tr>
<td>MEM(BINOP(PLUS,i,CONST c))</td>
<td>movl c(i),r</td>
<td>1</td>
</tr>
<tr>
<td>MOVE(MEM(BINOP(TIMES,i,j)),k)</td>
<td>movl k,(i,j)</td>
<td>1</td>
</tr>
<tr>
<td>MEM(BINOP(TIMES,i,CONST c))</td>
<td>movl c,r</td>
<td>2</td>
</tr>
<tr>
<td>MEM(i)</td>
<td>movl (i),r</td>
<td>1</td>
</tr>
<tr>
<td>MOVE(MEM(i),j)</td>
<td>movl j,(i)</td>
<td>1</td>
</tr>
<tr>
<td>MOVE(MEM(i),MEM(j))</td>
<td>movl (j),t</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>movl t,(i)</td>
<td></td>
</tr>
</tbody>
</table>

If c is 1, 2, or 4

Tiling an IR tree, v.1

a[x] = *y;

leal $a(%ebp),r1
movl (r1),r2
leal (,x,$4),r3
leal (r2,r3),r4
movl (y),r5
movl r5,(r4)

(assume a is a formal parameter passed on the stack)

Tiling an IR tree, v.2

a[x] = *y;

movl $a(%ebp),r1
leal (,x,4),r2
movl (y),r3
movl r3,(r1,r2)
Tiling choices
In general, for any given tree, many tilings are possible
  – each resulting in a different instruction sequence
We can ensure pattern coverage by covering, at a minimum, all atomic IR trees

The best tiling?
We want the “lowest cost” tiling
  – usually, the shortest sequence
  – but can also take into account cost/delay of each instruction

Optimum tiling
  – lowest-cost tiling
Locally Optimal tiling
  – no two adjacent tiles can be combined into one tile of lower cost

Locally optimal tilings
Locally optimal tiling is easy
  – A simple greedy algorithm works extremely well in practice:
    – Maximal munch
      • start at root
      • use “biggest” match (in # of nodes)
        – use cost to break ties

Maximal munch
Choose the largest pattern with lowest cost, i.e., the “maximal munch”

<table>
<thead>
<tr>
<th>IR pattern</th>
<th>code</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE(MEM(BINOP(PLUS,i,j)),k)</td>
<td>movl k,(i,j)</td>
<td>1</td>
</tr>
<tr>
<td>MOVE(MEM(i),j)</td>
<td>movl j,(i)</td>
<td>1</td>
</tr>
<tr>
<td>MOVE(MEM(i),MEM(j))</td>
<td>movl (j),t</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>movl t,(i)</td>
<td></td>
</tr>
</tbody>
</table>
Maximal munch

Maximal munch does not necessarily produce the optimum selection of instructions
But:
– it is easy to implement
– it tends to work well for current instruction-set architectures

Maximal munch is not optimum

Consider what happens, for example, if two of our rules are changed as follows:

<table>
<thead>
<tr>
<th>IR pattern</th>
<th>code</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM(BINOP(PLUS,i,CONST c))</td>
<td>movl c,r</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>addl i,r</td>
<td></td>
</tr>
<tr>
<td></td>
<td>movl (r),r</td>
<td></td>
</tr>
<tr>
<td>MOVE(MEM(BINOP(PLUS,i,j)),k)</td>
<td>movl j,r</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>addl i,r</td>
<td></td>
</tr>
<tr>
<td></td>
<td>movl k,(r)</td>
<td></td>
</tr>
</tbody>
</table>

Sample tree-matching rules

<table>
<thead>
<tr>
<th>Rule #</th>
<th>IR pattern</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TEMP t</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>CONST c</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>BINOP(PLUS,i,j)</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>BINOP(TIMES,i,j)</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>BINOP(PLUS,i,CONST c)</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>MEM(BINOP(PLUS,i,CONST c))</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>MOVE(MEM(BINOP(PLUS,i,j)),k)</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>BINOP(TIMES,i,CONST c)</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>BINOP(TIMES,i,CONST c)</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>MEM(i)</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>MOVE(MEM(i),j)</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>MOVE(MEM(i),MEM(j))</td>
<td>2</td>
</tr>
</tbody>
</table>

If c is 1, 2, or 4

Tiling an IR tree, new rules

a[x] = *y;

movl $a,r1
addl %ebp,r1
movl (r1),r1
leal (x,4),r2
movl (y),r3
movl r2,r4
addl r1,r4
movl r3,(r4)
**Optimum selection**

To achieve optimum instruction selection, we must use a more complex algorithm

– *dynamic programming*

In contrast to maximal munch, the trees are matched bottom-up

**Dynamic programming**

The idea is fairly simple

– Working bottom up…

– Given the optimum tilings of all subtrees, generate optimum tiling of the current tree
  • consider all tiles for the root of the current tree
  • sum cost of best subtree tiles and each tile
  • choose tile with minimum total cost

Second pass generates the code using the results from the bottom-up pass

**Bottom-up CG, pass 1**

Note: \((r,c)\) means rule \(r\), total cost \(c\)

**Bottom-up CG, pass 2**

Note: \((r,c)\) means rule \(r\), total cost \(c\)
**Bottom-up code generation**

How does the running time compare to maximal munch?
Memory usage?

---

**Tools**

A lot of tools have been developed to automatically generate instruction selectors
- TWIG [Aho,Ganapathi,Tjiang, ’86]
- BURG [Fraser,Henry,Proebsting, ’92]
- BEG [Emmelmann,Schroer,Landwehr, ’89]
- …
These generate bottom-up instruction selectors from tree-matching rule specifications

---

**Code-generator generators**

Twig, Burg, and Beg use the dynamic programming approach
- A lot of work has gone into making these cggs highly efficient
There are also grammar-based cggs that use LR(k) parsing to perform the tree-matching

---

**Tiling a DAG**

How would you tile

![Tiling a DAG Diagram]
Peephole Matching

Basic idea
Compiler can discover local improvements locally
– Look at a small set of adjacent operations
– Move a “peephole” over code & search for improvement

Classic example was store followed by load

Original code
\[
\text{storeAI } r_1 \Rightarrow r_0,8
\]
\[
\text{loadAI } r_0,8 \Rightarrow r_{15}
\]

Improved code
\[
\text{storeAI } r_1 \Rightarrow r_0,8
\]
\[
\text{i2I } r_1 \Rightarrow r_{15}
\]

Simple algebraic identities

Original code
\[
\text{addI } r_2,0 \Rightarrow r_7
\]
\[
\text{mult } r_4, r_7 \Rightarrow r_{10}
\]

Improved code
\[
\text{addI } r_2,0 \Rightarrow r_7
\]
\[
\text{mult } r_4, r_7 \Rightarrow r_{10}
\]

Jump to a jump

Original code
\[
L_{10}: \text{jumpI } \rightarrow L_{10}
\]
\[
L_{10}: \text{jumpI } \rightarrow L_{11}
\]

Improved code
\[
L_{10}: \text{jumpI } \rightarrow L_{11}
\]

Implementing it
Early systems used limited set of hand-coded patterns
Window size ensured quick processing

Modern peephole instruction selectors

Break problem into three tasks

Apply symbolic interpretation & simplification systematically

(Davidson)
Peephole Matching

Expander
Turns IR code into a low-level IR (LLIR) such as RTL
Operation-by-operation, template-driven rewriting
LLIR form includes all direct effects  \( \text{e.g., setting } cc \)  
Significant, albeit constant, expansion of size

\[
\begin{array}{cccc}
\text{IR} & \xrightarrow{\text{Expander}} & \text{LLIR} & \xrightarrow{\text{Simplifier}} & \text{LLIR} & \xrightarrow{\text{Matcher}} & \text{ASM} \\
\end{array}
\]

Simplifier
Looks at LLIR through window and rewrites it
Uses forward substitution, algebraic simplification, local constant propagation, and dead-effect elimination
Performs local optimization within window

This is the heart of the peephole system
− Benefit of peephole optimization shows up in this step

Matcher
Compares simplified LLIR against a library of patterns
Picks low-cost pattern that captures effects
Must preserve LLIR effects, may add new ones  \( \text{e.g., set } cc \)  
Generates the assembly code output

\[
\begin{array}{cccc}
\text{IR} & \xrightarrow{\text{Expander}} & \text{LLIR} & \xrightarrow{\text{Simplifier}} & \text{LLIR} & \xrightarrow{\text{Matcher}} & \text{ASM} \\
\end{array}
\]

Example

Original IR Code

<table>
<thead>
<tr>
<th>OP</th>
<th>Arg(_1)</th>
<th>Arg(_2)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>2</td>
<td>Y</td>
<td>t(_1)</td>
</tr>
<tr>
<td>sub</td>
<td>x</td>
<td>t(_1)</td>
<td>w</td>
</tr>
</tbody>
</table>

LLIR Code

\[
\begin{align*}
& r_{10} \leftarrow 2 \\
& r_{11} \leftarrow @y \\
& r_{12} \leftarrow r_0 * r_{11} \\
& r_{13} \leftarrow \text{MEM}(r_{12}) \\
& r_{14} \leftarrow r_{10} * r_{13} \\
& r_{15} \leftarrow @x \\
& r_{16} \leftarrow r_0 + r_{15} \\
& r_{17} \leftarrow \text{MEM}(r_{16}) \\
& r_{18} \leftarrow r_{17} - r_{14} \\
& r_{19} \leftarrow @w \;
\end{align*}
\]

\( \text{MEM}(r_{20}) \leftarrow r_{18} \)
Example

LLIR Code

\[
\begin{align*}
\text{r}_{10} &\leftarrow 2 \\
\text{r}_{11} &\leftarrow @y \\
\text{r}_{12} &\leftarrow \text{r}_0 + \text{r}_{11} \\
\text{r}_{13} &\leftarrow \text{MEM}(\text{r}_{12}) \\
\text{r}_{14} &\leftarrow \text{r}_{10} \times \text{r}_{13} \\
\text{r}_{15} &\leftarrow @x \\
\text{r}_{16} &\leftarrow \text{r}_0 \times \text{r}_{15} \\
\text{r}_{17} &\leftarrow \text{MEM}(\text{r}_{16}) \\
\text{r}_{18} &\leftarrow \text{r}_{12} \times \text{r}_{14} \\
\text{MEM}(\text{r}_{20}) &\leftarrow \text{r}_{18}
\end{align*}
\]

Simplify

Example

LLIR Code

\[
\begin{align*}
\text{r}_{13} &\leftarrow \text{MEM}(\text{r}_{10} + @y) \\
\text{r}_{14} &\leftarrow 2 \times \text{r}_{13} \\
\text{r}_{17} &\leftarrow \text{MEM}(\text{r}_0 + @x) \\
\text{r}_{18} &\leftarrow \text{r}_{12} - \text{r}_{14} \\
\text{MEM}(\text{r}_{20} + @w) &\leftarrow \text{r}_{18}
\end{align*}
\]

Match

ILoc Code

\[
\begin{align*}
\text{loadAI } \text{r}_0, @y &\Rightarrow \text{r}_{13} \\
\text{multI } 2 \times \text{r}_{13} &\Rightarrow \text{r}_{14} \\
\text{loadAI } \text{r}_0, @x &\Rightarrow \text{r}_{17} \\
\text{sub } \text{r}_{17} - \text{r}_{14} &\Rightarrow \text{r}_{18} \\
\text{storeAI } \text{r}_{18} &\Rightarrow \text{r}_0, @w
\end{align*}
\]

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
\text{r}_{10} &\leftarrow 2 \\
\text{r}_{11} &\leftarrow @y \\
\text{r}_{12} &\leftarrow \text{r}_0 + \text{r}_{11} \\
\text{r}_{13} &\leftarrow \text{MEM}(\text{r}_{12}) \\
\text{r}_{14} &\leftarrow \text{r}_{10} \times \text{r}_{13} \\
\text{r}_{15} &\leftarrow @x \\
\text{r}_{16} &\leftarrow \text{r}_0 \times \text{r}_{15} \\
\text{r}_{17} &\leftarrow \text{MEM}(\text{r}_{16}) \\
\text{r}_{18} &\leftarrow \text{r}_{12} \times \text{r}_{14} \\
\text{MEM}(\text{r}_{20}) &\leftarrow \text{r}_{18}
\end{align*}
\]

3-operation window

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
\text{r}_{10} &\leftarrow 2 \\
\text{r}_{11} &\leftarrow @y \\
\text{r}_{12} &\leftarrow \text{r}_0 + \text{r}_{11} \\
\text{r}_{13} &\leftarrow \text{MEM}(\text{r}_{12}) \\
\text{r}_{14} &\leftarrow \text{r}_{10} \times \text{r}_{13} \\
\text{r}_{15} &\leftarrow @x \\
\text{r}_{16} &\leftarrow \text{r}_0 \times \text{r}_{15} \\
\text{r}_{17} &\leftarrow \text{MEM}(\text{r}_{16}) \\
\text{r}_{18} &\leftarrow \text{r}_{12} \times \text{r}_{14} \\
\text{MEM}(\text{r}_{20}) &\leftarrow \text{r}_{18}
\end{align*}
\]

3-operation window
Steps of the Simplifier

LLIR Code

\[
\begin{align*}
r_{10} & \leftarrow 2 \\
r_{11} & \leftarrow \theta y \\
r_{12} & \leftarrow r_0 + r_{11} \\
r_{13} & \leftarrow \text{MEM}(r_{12}) \\
r_{14} & \leftarrow r_0 \times r_{13} \\
r_{15} & \leftarrow \theta x \\
r_{16} & \leftarrow r_0 + r_{16} \\
r_{17} & \leftarrow \text{MEM}(r_{16}) \\
r_{18} & \leftarrow r_0 + r_{14} \\
r_{19} & \leftarrow \theta w \\
r_{20} & \leftarrow r_0 + r_{19} \\
\text{MEM}(r_{20}) & \leftarrow r_{18}
\end{align*}
\]

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
r_{10} & \leftarrow 2 \\
r_{11} & \leftarrow \theta y \\
r_{12} & \leftarrow r_0 + r_{11} \\
r_{13} & \leftarrow \text{MEM}(r_{12}) \\
r_{14} & \leftarrow r_0 \times r_{13} \\
r_{15} & \leftarrow \theta x \\
r_{16} & \leftarrow r_0 + r_{16} \\
r_{17} & \leftarrow \text{MEM}(r_{16}) \\
r_{18} & \leftarrow r_0 + r_{14} \\
r_{19} & \leftarrow \theta w \\
r_{20} & \leftarrow r_0 + r_{19} \\
\text{MEM}(r_{20}) & \leftarrow r_{18}
\end{align*}
\]
Steps of the Simplifier

LLIR Code

\[
\begin{align*}
 r_{10} &\leftarrow 2 \\
r_{11} &\leftarrow @y \\
r_{12} &\leftarrow r_0 + r_{11} \\
r_{13} &\leftarrow MEM(r_{12}) \\
r_{14} &\leftarrow r_0 \times r_{13} \\
r_{15} &\leftarrow @x \\
r_{16} &\leftarrow r_0 + r_{15} \\
r_{17} &\leftarrow MEM(r_{16}) \\
r_{18} &\leftarrow r_{14} - r_{12} \\
r_{19} &\leftarrow @w \\
r_{20} &\leftarrow r_0 + r_{19} \\
MEM(r_{20}) &\leftarrow r_{18}
\end{align*}
\]

→

\[
\begin{align*}
 r_{14} &\leftarrow 2 \times r_{13} \\
r_{15} &\leftarrow @x \\
r_{16} &\leftarrow r_0 + @x \\
r_{17} &\leftarrow MEM(r_{16}) \\
r_{18} &\leftarrow r_{14} - r_{12} \\
r_{19} &\leftarrow @w \\
r_{20} &\leftarrow r_0 + r_{19} \\
MEM(r_{20}) &\leftarrow r_{18}
\end{align*}
\]
Steps of the Simplifier

LLIR Code
\[\begin{align*}
& r_{10} \leftarrow 2 \\
& r_{11} \leftarrow \theta y \\
& r_{12} \leftarrow r_0 + r_{11} \\
& r_{13} \leftarrow \text{MEM}(r_{12}) \\
& r_{14} \leftarrow r_0 \times r_{13} \\
& r_{15} \leftarrow \theta x \\
& r_{16} \leftarrow r_0 + r_{15} \\
& r_{17} \leftarrow \text{MEM}(r_{16}) \\
& r_{18} \leftarrow r_{13} - r_{14} \\
& r_{20} \leftarrow r_0 + r_{19} \\
& \text{MEM}(r_{20}) \leftarrow r_{18} \\
& \text{MEM}(r_{20}) \leftarrow r_{18}
\end{align*}\]

Example

LLIR Code
\[\begin{align*}
& r_{10} \leftarrow 2 \\
& r_{11} \leftarrow \theta y \\
& r_{12} \leftarrow r_0 + r_{11} \\
& r_{13} \leftarrow \text{MEM}(r_{12}) \\
& r_{14} \leftarrow r_0 \times r_{13} \\
& r_{15} \leftarrow \theta x \\
& r_{16} \leftarrow r_0 + r_{15} \\
& r_{17} \leftarrow \text{MEM}(r_{16}) \\
& r_{18} \leftarrow r_{13} - r_{14} \\
& r_{19} \leftarrow \theta w \\
& r_{20} \leftarrow r_0 + r_{19} \\
& \text{MEM}(r_{20}) \leftarrow r_{18} \\
& \text{MEM}(r_{20}) \leftarrow r_{18}
\end{align*}\]
Making It All Work

Details
- LLIR is largely machine independent (RTL)
- Target machine described as LLIR → ASM pattern
- Actual pattern matching
  - Use a hand-coded pattern matcher (gcc)
  - Turn patterns into grammar & use LR parser (VPO)
- Several important compilers use this technology
- It seems to produce good portable instruction selectors

Key strength appears to be late low-level optimization

SIMD Instructions

- We’ve assumed each tile is connected
- What about SIMD instructions?
- Ex. TI C62x add2
- Subword parallelism

Automatic Vectorization

Loop level
- for (i = 0; i < 1024; i++)
  - C[i] = A[i]*B[i];
- for (i = 0; i < 1024; i+=4)
  - C[i:i+3] = A[i:i+3]*B[i:i+3];

Basic block level
- x = a+b;
- y = c+d;
- (x,y) = (a,c)+(b,d);

looking for independent, isomorphic operations