CS415 Compilers

Code Generation - Part 2

Intermediate Representations

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University.
Announcements

Roadmap for the remainder of the course

• Project #2 – Bottom-up parser and compiler
  New due date: Wednesday April 20

• Homework #5 has been posted

• Midterm #1 – Grade challenge deadline is Friday, April 15. Please pick up your exams in recitation

• Final exam on May 10, 1:00pm, (60 minutes in class)

• Grading Scheme
  → Exams: 2 x 30% (best two exams count)
  → Projects: 3 x 10%
  → Homeworks: 5 x 2% (best five homeworks count)
Code Generation

EaC Chapter 7
How should the compiler represent them?
• Answer depends on the target machine

Two classic approaches
• Numerical representation
• Positional (implicit) representation

**Correct choice depends on both context and ISA**
Boolean & Relational Values

Numerical representation
- Assign values to TRUE and FALSE
- Use hardware AND, OR, and NOT operations
- Use comparison to get a boolean from a relational expression

Examples

\[ x < y \quad \text{becomes} \quad \text{cmp\_LT} \ r_x, r_y \Rightarrow r_1 \]

\[
\begin{align*}
\text{if } (x < y) \\
\text{then } \text{stmt}_1 \\
\text{else } \text{stmt}_2
\end{align*}
\quad \text{becomes} \quad \begin{align*}
\text{cmp\_LT} \ r_x, r_y & \Rightarrow r_1 \\
\text{cbr} \ r_1 & \Rightarrow \text{stmt}_1, \text{stmt}_2
\end{align*}
\]
What if the ISA uses a condition code?

- Must use a conditional branch to interpret result of compare
- Necessitates branches in the evaluation

Example: // $r_2$ should contain boolean value of “$x<y$” evaluation

```
cmp r_x, r_y ⇨ cc_1
cbr_\text{⊥T} \text{cc}_1 ⇨ L_T, L_F
```

$x < y$ becomes

- $L_T$: load 1 \(\Rightarrow r_2$
- br \(\rightarrow L_E$
- $L_F$: load 0 \(\Rightarrow r_2$
- $L_E$: ...other stmts...

This “positional representation” is much more complex
The last example actually encodes result in the PC.
If result is used to control an operation, this may be enough.

<table>
<thead>
<tr>
<th>Variations on the ILOC Branch Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Straight Condition Codes</strong></td>
</tr>
<tr>
<td>comp</td>
</tr>
<tr>
<td>r_x, r_y (\Rightarrow) cc_1</td>
</tr>
<tr>
<td>cbr_LT</td>
</tr>
<tr>
<td>cc_1 (\Rightarrow) r_1, L_1, L_2</td>
</tr>
<tr>
<td>L_1: add</td>
</tr>
<tr>
<td>r_c, r_d (\Rightarrow) r_a</td>
</tr>
<tr>
<td>br (\Rightarrow) L_OUT</td>
</tr>
<tr>
<td>L_2: add</td>
</tr>
<tr>
<td>r_e, r_f (\Rightarrow) r_a</td>
</tr>
<tr>
<td>br (\Rightarrow) L_OUT</td>
</tr>
<tr>
<td>L_OUT: nop</td>
</tr>
</tbody>
</table>

Condition code version does not directly produce \((x < y)\)

Boolean version does

Still, there is no significant difference in the code produced.
Conditional move & predication both simplify this code

Example

```plaintext
if (x < y) then a ← c + d else a ← e + f
```

<table>
<thead>
<tr>
<th>Other Architectural Variations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conditional Move</strong></td>
</tr>
<tr>
<td>comp</td>
</tr>
<tr>
<td>rₓ, rᵧ ⇒ c c₁</td>
</tr>
<tr>
<td>add</td>
</tr>
<tr>
<td>rₓ, rᵈ ⇒ r₁</td>
</tr>
<tr>
<td>add</td>
</tr>
<tr>
<td>rₓ, rₓ + r₁</td>
</tr>
<tr>
<td>add</td>
</tr>
<tr>
<td>rₓ, rₓ + r₁</td>
</tr>
</tbody>
</table>

Both versions avoid the branches
Both are shorter than CCs or Boolean-valued compare
Are they better? **What about power?**
Consider the assignment \( x \leftarrow a < b \land c < d \) (short circuiting?)

<table>
<thead>
<tr>
<th>Variations on the ILOC Branch Structure</th>
<th>Boolean Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Straight Condition Codes</strong></td>
<td><strong>Boolean Compare</strong></td>
</tr>
<tr>
<td>comp ( r_a, r_b \Rightarrow c c_1 )</td>
<td>cmp ( \perp ) ( r_a, r_b \Rightarrow r_1 )</td>
</tr>
<tr>
<td>cbr ( \perp ) ( c c_1 \Rightarrow L_1, L_2 )</td>
<td>cmp ( \perp ) ( c c_2 \Rightarrow r_2 )</td>
</tr>
<tr>
<td>L_1:</td>
<td>and ( r_1, r_2 \Rightarrow r_x )</td>
</tr>
<tr>
<td>comp ( r_c, r_d \Rightarrow c c_2 )</td>
<td></td>
</tr>
<tr>
<td>cbr ( \perp ) ( c c_2 \Rightarrow L_3, L_2 )</td>
<td></td>
</tr>
<tr>
<td>L_2:</td>
<td></td>
</tr>
<tr>
<td>loadl ( 0 \Rightarrow r_x )</td>
<td></td>
</tr>
<tr>
<td>br ( \Rightarrow L_{OUT} )</td>
<td></td>
</tr>
<tr>
<td>L_3:</td>
<td></td>
</tr>
<tr>
<td>loadl ( 1 \Rightarrow r_x )</td>
<td></td>
</tr>
<tr>
<td>br ( \Rightarrow L_{OUT} )</td>
<td></td>
</tr>
<tr>
<td>L_{OUT}:</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
</tbody>
</table>

Here, the boolean compare produces much better code.
Intermediate Representation

EaC Chapter 5
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
• Decisions in IR design affect the speed and efficiency of the compiler

• Some important IR properties
  → Ease of generation
  → Ease of manipulation
  → Size
  → Level of abstraction

• The importance of different properties varies between compilers
  → Selecting an appropriate IR for a compiler is critical
Types of Intermediate Representations

Three major categories

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

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

• Hybrid
  → Combination of graphs and linear code

Examples:
- Trees, DAGs
- 3 address code
- Stack machine code
- 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

  Low level linear code:
  Good for address calculation

\[
\begin{align*}
\text{loadI } 1 & \rightarrow r_1 \\
\text{sub } r_j, r_1 & \rightarrow r_2 \\
\text{loadI } 10 & \rightarrow r_3 \\
\text{mult } r_2, r_3 & \rightarrow r_4 \\
\text{sub } r_i, r_1 & \rightarrow r_5 \\
\text{add } r_4, r_5 & \rightarrow r_6 \\
\text{loadI } @A & \rightarrow r_7 \\
\text{Add } r_7, r_6 & \rightarrow r_8 \\
\text{load } r_8 & \rightarrow r_{Aij}
\end{align*}
\]
Level of Abstraction

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

\[
\begin{align*}
& load \\
& + \\
& * \\
& \text{o} \\
& 10 \\
& - \\
& j \\
& 1 \\
& \text{loadArray A, i, j} \\
& \text{High level linear code}
\end{align*}
\]
Abstract Syntax Tree

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

\[
x - 2 * y
\]

- Can use linearized form of the tree
  - in postfix form
    \[
    x 2 y * -
    \]
  - in prefix form
    \[
    - * 2 y x
    \]
- \textit{S}-expressions are (essentially) ASTs (remember functional languages such as Scheme or Lisp!)
A directed acyclic graph (DAG) is an AST with a unique node for each value

- Makes sharing explicit
- Encodes redundancy

Same expression twice means that the compiler might arrange to evaluate it just once!
Stack Machine Code

Originally used for stack-based computers, now Java

- Example:
  \[ x - 2 \times y \]
  becomes
  
  push x
  push 2
  push y
  multiply
  subtract

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)*

Implicit names take up no space, where explicit ones do!
Three Address Code

Several different representations of three address code

• In general, three address code has statements of the form:
  \[ x \leftarrow y \ op z \]
  With 1 operator \( \texttt{op} \) and, at most, 3 names \( \langle x, y, z \rangle \)

Example:
\[ z \leftarrow x \]
\[ 2 \times y \]

becomes
\[ t \leftarrow 2 \times y \]
\[ z \leftarrow x - t \]

Advantages:

• Resembles many machines
• Introduces a new set of names
• Compact form
Naïve representation of three address code

- Table of $k \times 4$ small integers
- Simple record structure
- Easy to reorder
- Explicit names

RISC assembly code (not ILOC)

<table>
<thead>
<tr>
<th>Quadruples</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r1, y</td>
</tr>
<tr>
<td>loadI r2, 2</td>
</tr>
<tr>
<td>mult r3, r2, r1</td>
</tr>
<tr>
<td>load r4, x</td>
</tr>
<tr>
<td>sub r5, r4, r3</td>
</tr>
</tbody>
</table>

The original FORTRAN compiler used “quads”
Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>y</td>
</tr>
<tr>
<td>2</td>
<td>loadI</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>mult</td>
<td>(1) (2)</td>
</tr>
<tr>
<td>4</td>
<td>load</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>sub</td>
<td>(4) (3)</td>
</tr>
</tbody>
</table>

Implicit names take no space!
Control-flow Graph (CFG)

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example

```
if (x = y)
  a ← 2
  b ← 5
  a ← 3
  b ← 4
  c ← a + b
```

Basic blocks — Maximal length sequences of straight-line code
Static Single Assignment Form (SSA)

- The main idea: each name defined exactly once in program
- Introduce $\phi$-functions to make it work

**Original**

$$
\begin{align*}
x & \leftarrow \ldots \\
y & \leftarrow \ldots \\
\text{while } (x < k) & \\
& \quad x \leftarrow x + 1 \\
& \quad y \leftarrow y + x
\end{align*}
$$

**SSA-form**

$$
\begin{align*}
x_0 & \leftarrow \ldots \\
y_0 & \leftarrow \ldots \\
\text{if } (x_0 > k) & \text{ goto next} \\
\text{loop:} & \\
& \quad x_1 \leftarrow \phi(x_0,x_2) \\
& \quad y_1 \leftarrow \phi(y_0,y_2) \\
& \quad x_2 \leftarrow x_1 + 1 \\
& \quad y_2 \leftarrow y_1 + x_2 \\
& \text{if } (x_2 < k) \text{ goto loop} \\
\text{next:} & \quad \ldots
\end{align*}
$$

**Strengths of SSA-form**

- Sharper analysis
- “minimal” $\phi$-functions placement is non-trivial
- (sometimes) faster algorithms
Work on the project!

Compiler Optimizations

Procedure abstraction
Read EaC: Chapter 6.1 - 6.5