CS415 Compilers

Code Generation

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

- Second project - Please continue working on it
- Sixth homework has been posted; deadline extension Wednesday, April 21
- Quiz 3 will be posted tonight
Code Generation

EaC Chapter 7
A compiler is a lot of fast stuff followed by some hard problems

→ The hard stuff is mostly in code generation and optimization
→ For superscalars, its allocation & scheduling that is particularly important
The key code quality issue is holding values in registers

- When can a value be safely allocated to a register?
  - When only 1 name can reference its value (no aliasing)
  - Pointers, parameters, aggregates & arrays all cause trouble
- When should a value be allocated to a register?
  - When it is both safe & profitable

Encoding this knowledge into the IR (register-register model)

- Use code shape to make it known to every later phase
- Assign a virtual register to anything that can go into one
- Load or store the others at each reference

Relies on a strong register allocator
Recursive Treewalk vs. Ad-hoc SDT

```c
int expr(node) {
    int result, t1, t2;
    switch (type(node)) {
        case ×,÷,+,—:
            t1 ← expr(left child(node));
            t2 ← expr(right child(node));
            result ← NextRegister();
            emit(op(node), t1, t2, result);
            break;
        case IDENTIFIER:
            t1 ← base(node);
            t2 ← offset(node);
            result ← NextRegister();
            emit(loadAO, t1, t2, result);
            break;
        case NUMBER:
            result ← NextRegister();
            emit(loadI, val(node), none, result);
            break;
    }
    return result;
}
```

Goal :
Expr { $$ = $1; } ;
Expr PLUS Term
{ t = NextRegister();
    emit(add,$1,$3,t); $$ = t; }
| Expr MINUS Term {...}
| Term { $$ = $1; } ;
Term TIMES Factor
{ t = NextRegister();
    emit(mult,$1,$3,t); $$ = t; }
| Term DIVIDES Factor {...}
| Factor { $$ = $1; } ;
Factor:
NUMBER
{ t = NextRegister();
    emit(loadI, val($1),none, t );
    $$ = t; }
| ID
{ t1 = base($1);
    t2 = offset($1);
    t = NextRegister();
    emit(loadAO,t1,t2,t);
    $$ =  t; }
```
Handling Assignment (just another operator)

\[ \text{lhs} \leftarrow \text{rhs} \]

Strategy

- Evaluate \textit{rhs} to a value \hspace{1cm} (an \textit{rvalue})
- Evaluate \textit{lhs} to a location (memory address) \hspace{1cm} (an \textit{lvalue})
  \[ \rightarrow \text{\textit{lvalue} is an address} \Rightarrow \text{store rhs} \]
- If \textit{rvalue} & \textit{lvalue} have different types
  \[ \rightarrow \text{Evaluate} \textit{rvalue} \text{ to its } \text{“natural” type} \]
  \[ \rightarrow \text{Convert that value to the type of} \ \textit{lhs} \text{ value, if possible} \]

Unambiguous scalars may go into registers (no aliasing)
Ambiguous scalars or aggregates go into memory (possible aliasing)
Handling Assignment

What if the compiler cannot determine the rhs’s type?

- This is a property of the language & the specific program
- If type-safety is desired, compiler must insert a run-time check
- Add a tag field to the data items to hold type information

Code for assignment becomes more complex

```plaintext
evaluate rhs
If lhs.type_tag ≠ rhs.type_tag
then
    convert rhs to type(lhs) or signal a run-time error
lhs ← rhs
```

This is much more complex than if it knew the types
Handling Assignment

Compile-time type-checking

- **Goal is to eliminate both the runtime check & the tag**
- **Determine, at compile time, the type of each subexpression**
- **Use compile-time types to determine if a run-time check is needed**

Optimization strategy

- **If compiler knows the type, move the check to compile-time**
- **Unless tags are needed for garbage collection, eliminate them**
- **If check is needed, try to overlap it with other computation (superscalar or multi-core architectures)**
Handling Assignment (with reference counting)

Garbage Collection

The problem with reference counting

- Must adjust the count on each pointer assignment
- Overhead is significant, relative to assignment

Code for assignment becomes

```plaintext
evaluate rhs
lhs→count ← lhs→count - 1
lhs ← addr(rhs)
rhs→count ← rhs→count + 1
```

This adds 1 +, 1 -, 2 loads, & 2 stores

With extra functional units & large caches, this may become either cheap or free. What about power consumption?
First, must agree on a storage scheme

**Row-major order**  
(most languages)  
Lay out as a sequence of consecutive rows  
Rightmost subscript varies fastest  
A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]

**Column-major order**  
(Fortran)  
Lay out as a sequence of columns  
Leftmost subscript varies fastest  
A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]

**Indirection vectors**  
(Java)  
Vector of pointers to pointers to ... to values  
Takes much more space, trades indirection for arithmetic  
Not amenable to analysis
Laying Out Arrays

The Concept

Row-major order

<table>
<thead>
<tr>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
</tr>
</tbody>
</table>

Column-major order

<table>
<thead>
<tr>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
</tr>
</tbody>
</table>

Indirection vectors

These have distinct & different cache behavior
Computing an Array Address

Declaration: \( A[\text{low} .. \text{high}] \) of ...

\( A[i] \)
- \( @A + (i - \text{low}) \times \text{sizeof}(A[1]) \)
- In general: base(A) + (i - low) \times \text{sizeof}(A[1])
Computing an Array Address

Declaration: A[low .. high] of ...

A[i]
- \( @A + (i - low) \times sizeof(A[1]) \)
- In general: base(A) + (i - low) \times sizeof(A[1])

\[
\text{int } A[1:10] \Rightarrow \text{low is 1} \\
\text{Make low 0 for faster access (saves a -)}
\]

Almost always a power of 2, known at compile-time \( \Rightarrow \) use a shift for speed
Computing an Array Address

Declaration: \( A[\text{low1 .. high1, low2 .. high2}] \) of ...

\[ A[ i ] \]
- \( @A + (i - \text{low}) \times \text{sizeof}(A[1]) \)
- In general: \( \text{base}(A) + (i - \text{low}) \times \text{sizeof}(A[1]) \)

What about \( A[i_1,i_2] \)?

**Row-major order, two dimensions**
\[ @A + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times \text{sizeof}(A[1]) \]

**Column-major order, two dimensions**
\[ @A + ((i_2 - \text{low}_2) \times (\text{high}_1 - \text{low}_1 + 1) + i_1 - \text{low}_1) \times \text{sizeof}(A[1]) \]

**Indirection vectors, two dimensions**
\[ *(A[i_1])[i_2] \] — where \( A[i_1] \) is, itself, a 1-d array reference

This stuff looks expensive! Lots of implicit +, -, \( \times \) ops
In row-major order
\[ @A + (i - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) \times w + (j - \text{low}_2) \times w \]

Which can be factored into
\[ @A + i \times (\text{high}_2 - \text{low}_2 + 1) \times w + j \times w \]
\[ - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w) + (\text{low}_2 \times w) \]

If \( \text{low}_i, \text{high}_i, \) and \( w \) are known, the last term is a constant
Define \(@A_0\) as
\[ @A - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w + \text{low}_2 \times w \]
And \( \text{len}_2 \) as \((\text{high}_2 - \text{low}_2 + 1)\)

Then, the address expression becomes
\[ @A_0 + (i \times \text{len}_2 + j) \times w \]

where \( w = \text{sizeof}(A[1,1]) \)

Compile-time constants
One possible approach for code generation:

Loops
- Evaluate condition before loop (if needed)
- Evaluate condition after loop
- Branch back to the top (if needed)

Merges test with last block of loop body

while, for, do, & until all fit this basic model
for (i = 1; i < 100; i++) {} 

next statement

Initialization

Pre-test

Post-test

LoadI 1 ⇒ r₁
LoadI 1 ⇒ r₂
LoadI 100 ⇒ r₃
Cmp_GT r₁, r₃ ⇒ r₄
Cbr r₄ ⇒ L₂, L₁

L₁: body
Add r₁, r₂ ⇒ r₁
Cmp_LT r₁, r₃ ⇒ r₅
Cbr r₅ ⇒ L₁, L₂

L₂: next statement
Many modern programming languages include a **break**

- Exits from the innermost control-flow statement
  - Out of the innermost loop
  - Out of a case statement

Translates into a jump
- Targets statement outside control-flow construct
- Creates multiple-exit construct
- **Skip** in loop goes to next iteration
Case Statements
1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case

Parts 1, 3, & 4 are well understood, part 2 is the key
Case Statements
1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case (use break)

Parts 1, 3, & 4 are well understood, part 2 is the key

Strategies
- Linear search (nested if-then-else constructs)
- Build a table of case expressions & binary search it
- Directly compute an address (requires dense case set: jump table)

Surprisingly many compilers do this for all cases!
Boolean & Relational Values

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

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

\[
\begin{align*}
&\text{if } (x < y) \\
&\text{then stmt}_1 \\
&\text{else stmt}_2
\end{align*}
\]

\[ becomes \quad \text{cmp}_\text{LT} \ r_x, r_y \Rightarrow r_1 \\
\text{cbr } r_1 \Rightarrow \text{stmt}_1, \text{stmt}_2 \]
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  ; set cc1
cbr \textit{LT}  ; cc1 \rightarrow L_T, L_F
```

$x < y \quad \text{becomes} \quad L_T: \quad \text{loadl } 1 \rightarrow r_2
\text{br} \rightarrow L_E
L_F: \quad \text{loadl } 0 \rightarrow r_2
L_E: \quad \text{...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>cbr_LT</td>
</tr>
<tr>
<td>L_1: add</td>
</tr>
<tr>
<td>br</td>
</tr>
<tr>
<td>L_2: add</td>
</tr>
<tr>
<td>br</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

<table>
<thead>
<tr>
<th>Example</th>
<th>Other Architectural Variations</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x &lt; y)</td>
<td></td>
</tr>
<tr>
<td>then a ← c + d</td>
<td>Conditional Move</td>
</tr>
<tr>
<td>else a ← e + f</td>
<td>Predicated Execution</td>
</tr>
<tr>
<td>comp</td>
<td>cmp_LT</td>
</tr>
<tr>
<td>add</td>
<td>(r_1)? add</td>
</tr>
<tr>
<td>add</td>
<td>(¬r_1)? add</td>
</tr>
<tr>
<td>i2i_&lt;</td>
<td>cc_1,r_1,r_2⇒r_a</td>
</tr>
<tr>
<td>r_x,r_y⇒cc_1</td>
<td>r_x,r_y⇒r_1</td>
</tr>
<tr>
<td>r_c,r_d⇒r_1</td>
<td>r_c,r_d⇒r_a</td>
</tr>
<tr>
<td>r_e,r_f⇒r_2</td>
<td>r_e,r_f⇒r_a</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 \( x \leftarrow a < b \land c < d \)

<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</td>
<td>cmp⊥T ( r_a, r_b \Rightarrow r_1 )</td>
</tr>
<tr>
<td>cbr⊥T ( \text{cc}_1 \Rightarrow L_1, L_2 )</td>
<td>( r_c, r_d \Rightarrow r_2 )</td>
</tr>
<tr>
<td>( L_1: ) comp</td>
<td>and ( r_1, r_2 \Rightarrow r_x )</td>
</tr>
<tr>
<td>( \text{cc}_2 \Rightarrow L_3, L_2 )</td>
<td></td>
</tr>
<tr>
<td>( L_2: ) loadl ( 0 \Rightarrow r_x )</td>
<td></td>
</tr>
<tr>
<td>br ( \Rightarrow L_{\text{OUT}} )</td>
<td></td>
</tr>
<tr>
<td>( L_3: ) loadl ( 1 \Rightarrow r_x )</td>
<td></td>
</tr>
<tr>
<td>br ( \Rightarrow L_{\text{OUT}} )</td>
<td></td>
</tr>
<tr>
<td>( L_{\text{OUT}}: ) nop</td>
<td></td>
</tr>
</tbody>
</table>

Here, the boolean compare produces much better code.
Things to do and next class

Work on the project!

Intermediate representations
Read EaC: Chapter 5

Procedure abstraction
Read EaC: Chapter 6.1 - 6.5