CS415 Compilers

More Code generation
Optimizations (CSE)
Intermediate representations

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

- Homework #7 will be posted today.

- Project 2 is due on Monday, April 22.
  removed GT and GEQ from language description
  no need to change provided parse.y

- Final exam: May 14, noon – 3:00pm, SEC-118 (our current room)
  CONFLICTS?
  If you have a conflict, please send me the details of your conflict:
  class, email of instructor, time of scheduled exam

- My office hours cancelled tomorrow; moved to
  Monday, April 22, 11:00-12:30pm, CoRE 318
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
Loop Implementation Code

for (i = 1; i < 100; i++) {
    \textit{body}

    next statement

\begin{align*}
\text{Initialization} & \quad \text{Pre-test} & \quad \text{Post-test} \\
\text{L}_1: & \quad \text{body} & \quad \text{L}_2: & \quad \text{next statement} \\
\text{loadI} & \quad 1 \Rightarrow r_1 & \quad \text{add} & \quad r_1, r_2 \Rightarrow r_1 \\
\text{loadI} & \quad 1 \Rightarrow r_2 & \quad \text{cmp\_LT} & \quad r_1, r_3 \Rightarrow r_5 \\
\text{loadI} & \quad 100 \Rightarrow r_3 & \quad \text{cbr} & \quad r_4 \Rightarrow L_2, L_1 \\
\text{cmp\_GT} & \quad r_1, r_3 \Rightarrow r_4 & \quad \text{cbr} & \quad r_5 \Rightarrow L_1, L_2
\end{align*}
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!
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

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

\[ \text{if (} x < y \text{)} \quad \text{then stmt}_1 \quad \text{becomes} \quad \text{cmp}_{\text{LT}} \ r_x, r_y \Rightarrow r_1 \\
\text{else stmt}_2 \quad \text{becomes} \quad \text{cbr} \ r_1 \Rightarrow \_\text{stmt}_1, \_\text{stmt}_2 \]
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

\[
\begin{align*}
\text{cmp} & \quad r_x, r_y \Rightarrow cc_1 \\
\text{cbr}_{\perp} & \quad cc_1 \rightarrow L_T, L_F \\

x < y & \quad \text{becomes} \\
L_T & : \quad \text{loadl} \quad 1 \Rightarrow r_2 \\
\text{br} & \quad \rightarrow L_E \\
L_F & : \quad \text{loadl} \quad 0 \Rightarrow r_2 \\
L_E & : \quad \ldots \text{other stmts} \ldots
\end{align*}
\]

This “positional representation” is much more complex
Boolean & Relational Values

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>
</tr>
</thead>
<tbody>
<tr>
<td>if ((x &lt; y))</td>
</tr>
<tr>
<td>then (a \leftarrow c + d)</td>
</tr>
<tr>
<td>else (a \leftarrow e + f)</td>
</tr>
</tbody>
</table>

### Other Architectural Variations

<table>
<thead>
<tr>
<th>Conditional Move</th>
<th>Predicated Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp (r_x, r_y \Rightarrow cc_1)</td>
<td>cmp_LT (r_x, r_y \Rightarrow r_1)</td>
</tr>
<tr>
<td>add (r_c, r_d \Rightarrow r_1)</td>
<td>((r_1)?) add (r_c, r_d \Rightarrow r_a)</td>
</tr>
<tr>
<td>add (r_e, r_f \Rightarrow r_2)</td>
<td>((-r_1)?) add (r_e, r_f \Rightarrow r_a)</td>
</tr>
<tr>
<td>i2i_&lt; (cc_1, r_1, r_2 \Rightarrow r_a)</td>
<td></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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Straight Condition Codes</strong></td>
</tr>
<tr>
<td>comp</td>
</tr>
<tr>
<td>cbr_LT ( cc_1 \Rightarrow L_1, L_2 )</td>
</tr>
<tr>
<td>( L_1 ): comp ( r_c, r_d \Rightarrow cc_2 )</td>
</tr>
<tr>
<td>cbr_LT ( cc_2 \Rightarrow L_3, L_2 )</td>
</tr>
<tr>
<td>( L_2 ): loadl 0 \rightarrow r_x \Rightarrow L_{OUT}</td>
</tr>
<tr>
<td>br</td>
</tr>
<tr>
<td>( L_3 ): loadl 1 \rightarrow r_x \Rightarrow L_{OUT}</td>
</tr>
<tr>
<td>br</td>
</tr>
<tr>
<td>( L_{OUT} ): nop</td>
</tr>
</tbody>
</table>

Here, the boolean compare produces much better code.
- tries to improve quality of code (may fail in some cases)
- optimizer typically consists of multiple passes
- different optimization (code improvement) objectives:
  - execution time reduction
  - reduction in resource requirements (memory, registers)
  - (peak) power and energy reduction

- criteria for effectiveness of optimizations
  - safety - program semantics must be preserved
  - opportunity - how often can it be applied?
  - profitability - how much improvement?
We will focus on two optimizations:

1. Common subexpression elimination (CSE - local, ILOC level)

2. Vectorization / parallelization (source level) - will do this later if time allows

Local CSE reference: ALSU, chapter 8.5.2
Optimization: **Local Common Subexpression Elimination (CSE)**

Source code: \[ a(i) \quad (1\text{-based indexing}) \]

\[
\begin{align*}
4 & : \quad t_1 = \text{addr}(a) - 4 \\
5 & : \quad t_2 = i \times 4 \\
6 & : \quad t_3 = t_1[t_2] \\
& \quad \ldots
\end{align*}
\]
Optimization: Local Common Subexpression Elimination (CSE)

Source code: \( a(i) \times a(i) \) (1-based indexing)

\[
\begin{align*}
4. & \quad t_1 = \text{addr}(a) - 4 \\
5. & \quad t_2 = i \times 4 \\
6. & \quad t_3 = t_1[t_2] \\
7. & \quad t_4 = \text{addr}(a) - 4 \\
8. & \quad t_5 = i \times 4 \\
9. & \quad t_6 = t_4[t_5] \\
10. & \quad t_7 = t_3 \times t_6 \\
\ldots
\end{align*}
\]
Optimization: Local Common Subexpression Elimination (CSE)

Source code: \( a(i) \ast a(i) \) (1-based indexing)

\[
\begin{align*}
4. & \quad t1 = \text{addr}[a] - 4 \\
5. & \quad t2 = i \ast 4 \\
6. & \quad t3 = t1[t2] \\
7. & \quad t4 = \text{addr}[a] - 4 \\
8. & \quad t5 = i \ast 4 \\
9. & \quad t6 = t4[t5] \\
10. & \quad t7 = t3 \ast t6 \\
& \hspace{1cm} \ldots
\end{align*}
\]

Basic Block DAG Construction

<table>
<thead>
<tr>
<th>Code generated:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t1 = \text{addr}[a] - 4 )</td>
</tr>
<tr>
<td>( t2 = i \ast 4 )</td>
</tr>
<tr>
<td>( t3 = t1[t2] )</td>
</tr>
<tr>
<td>( t7 = t3 \ast t3 )</td>
</tr>
</tbody>
</table>

\[\begin{array}{c}
\ast, t7 \\
[ ], t3, t6 \\
\ast, t2, t5 \\
\end{array}\]
How to add a subexpression into a partially constructed DAG:

\[ A = B + C \]

Is there a node already for \( B + C \)?
- If so, add \( A \) to its list of labels.
- If not:
  - Is there a node labeled \( B \) already?
    - If not, create a leaf labeled \( B \).
  - Is there a node labeled \( C \) already?
    - If not, create a leaf labeled \( C \).
  - Create a node labeled \( A \), for \(+\), with left child \( B \) and right child \( C \).

How to do this? **HASHING** <op, node(opd1), node(opd2)>
DAG Construction Algorithm

How to add a subexpression into a partially constructed DAG:

\[ A = B + C \]

Is there a node already for \( B + C \)? \(<+ , \text{node}(B) , \text{node}(C) > \) defined?

- If so, add \( A \) to its list of labels.
- If not:
  - is there a node labeled \( B \) already? \( \text{node}(B) \) defined?
    If not, create a leaf labeled \( B \).
  - Is there a node labeled \( C \) already? \( \text{node}(C) \) defined?
    If not, create a leaf labeled \( C \).
  - Create a node labeled \( A \), for \(+\), with left child \( B \) and right child \( C \).

How to do this? HASHING \(<\text{op} , \text{node}(\text{opd1}) , \text{node}(\text{opd2}) \>)
DAG Construction Algorithm

Summary:
- every expression is assigned a value number
  examples:  node(a),
             node(4),
             node(<+, valNum1, ValNum2>)
- assignment changes value number associated with LHS variable

- implementation of value numbers
  - use pointers of nodes in DAG
  - use virtual register numbers (code shape encoding!)

You could do this in a single pass in our compiler!
Source code: \( A(i) + A(i) \)

```
loadI 1024 => r0

loadAI r0, 8 => r1 // assume 8 is base address of i
loadI 1 => r2
sub r1, r2 => r3 // first element of A is A(1)
loadI 4 => r4
mult r3, r4 => r5 // offset of A(i) in bytes
loadAO r0, r5 => r6 // A(i)

loadAI r0, 8 => r7 // assume 8 is base address of i
loadI 1 => r8
sub r7, r8 => r9 // first element of A is A(1)
loadI 4 => r10
mult r9, r10 => r11 // offset of A(i) in bytes
loadAO r0, r11 => r12 // A(i)

add r6, r12 => r13 // A(i) + A(i)
```
ILOC common subexpressions

**Source code:** $A(i) + A(i)$

- `loadI 1024 => r0`
- `loadAI r0, 8 => r1`
- `loadI 1 => r2`
- `sub r1, r2 => r3`
- `loadI 4 => r4`
- `mult r3, r4 => r5`
- `loadAO r0, r5 => r6`
- `loadAI r0, 8 => r7`
- `loadI 1 => r8`
- `sub r7, r8 => r9`
- `loadI 4 => r10`
- `mult r9, r10 => r11`
- `loadAO r0, r11 => r12`

**Idea:** Use register numbers as value numbers

- `Hash(<loadI, 1024>) = undef; gen_code; set to r0, return r0`
- `Hash(<loadAI, r0, 8>) = undef; gen_code; set to r1, return r1`
- `Hash(<loadI, 1>) = undef; gen_code; set to r2; return r2`
- `Hash(<sub, r1, r2>) = undef; gen_code; set to r3; return r3`
- `Hash(<loadI, 4> = undef; gen_code; set to r4; return r4`
- `Hash(<mult, r3, r4>) = undef; gen_code; set to r5; return r5`
- `Hash(<loadAO, r0, r5>) = undef; gen_code; set to r6; ret. r6`
- `Hash(<loadAI, r0, 8>) = r1; no gen_code; return r1`
- `Hash(<loadI, 1>) = r2; no gen_code; return r2`
- `Hash(<sub, r1, r2>) = r3; no gen_code; return r3`
- `Hash(<loadI, 4>) = r4; no gen_code; return r4`
- `Hash(<mult, r3, r4>) = r5; no gen_code; return r5`
- `Hash(<loadAO, r0, r5>) = r6; no gen_code; return r6`
- `Hash(<add, r6, r6>) = undef; gen_code; set to r7`
ILOC common subexpressions

**Source code:** \( A(i) + A(i) \)

- `loadI 1024 => r0`  
  Hash(<loadI, 1024>) = undef; gen_code; set to \( r0 \), return \( r0 \)

- `loadAI r0, 8 => r1`  
  Hash(<loadAI, r0, 8>) = undef; gen_code; set to \( r1 \), return \( r1 \)

- `loadI 1 => r2`  
  Hash(<loadI, 1>) = undef; gen_code; set to \( r2 \); return \( r2 \)

- `sub r1, r2 => r3`  
  Hash(<sub, r1, r2>) = undef; gen_code; set to \( r3 \); return \( r3 \)

- `loadI 4 => r4`  
  Hash(<loadI, 4>) = undef; gen_code; set to \( r4 \); return \( r4 \)

- `mult r3, r4 => r5`  
  Hash(<mult, r3, r4>) = undef; gen_code; set to \( r5 \); return \( r5 \)

- `loadAO r0, r5 => r6`  
  Hash(<loadAO, r0, r5>) = undef; gen_code; set to \( r6 \); ret. \( r6 \)

- `add r6, r6 => r7`  
  Hash(<add, r6, r6>) = undef; gen_code; set to \( r7 \)
ILOC common subexpressions

Source code: \((A(i) + A(i)) \times (A(i) + A(i))\)

\[
\begin{align*}
\text{loadI} & \ 1024 => r0 \\
\text{loadAI} & \ r0, 8 => r1 \\
\text{loadI} & \ 1 => r2 \\
\text{sub} & \ r1, r2 => r3 \\
\text{loadI} & \ 4 => r4 \\
\text{mult} & \ r3, r4 => r5 \\
\text{loadAO} & \ r0, r5 => r6 \\
\text{loadAI} & \ r0, 8 => r7 \\
\text{loadI} & \ 1 => r8 \\
\text{sub} & \ r7, r8 => r9 \\
\text{loadI} & \ 4 => r10 \\
\text{mult} & \ r9, r10 => r11 \\
\text{loadAO} & \ r0, r11 => r12 \\
\text{add} & \ r6, r12 => r13 \\
\end{align*}
\]

How would the CSE code look like?
Source code: \((A(i) + A(i)) \times (A(i) + A(i))\)

```
loadI 1024 => r0
loadAI r0, 8 => r1
loadI 1 => r2
sub r1, r2 => r3
loadI 4 => r4
mult r3, r4 => r5
loadAO r0, r5 => r6
add r6, r6 => r7
mult r7, r7 => r8
```

How would the CSE code look like?

That’s it!
Work on the project!

Intermediate representations
Read EaC: Chapter 5

Procedure abstraction
Read EaC: Chapter 6.1 - 6.5