CS415 Compilers,

Code Generation,

Local Common Subexpression Elimination Optimization

Intermediate Representations

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

- Homework 7 deadline tonight
- Homework 8 has been posted, due Wednesday, April 25
- First project grades have been posted
- Second project new due date: Wednesday, April 25
- Third project (local CSE) will be posted later this week, due May 2
- Midterms have been returned
  If you have any concerns with respect to the midterm, please talk to TA and/or instructor by Monday, April 23. No more challenges after that date.
- Thinking about taking 515 in the fall 2018 as an undergrad? Please see our class website for information.
Announcements

**Final exam:** Tuesday, May 8, noon-3:00pm, Physics Lecture Hall, Busch Campus

**Conflicts?**

If you have a conflict, please send me the details of your conflict: class, email of instructor, time of scheduled exam
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

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

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

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 \Rightarrow cc_1
\text{cbr}_{\bot}T cc_1 \rightarrow L_T, L_F
```

$x < y \quad \text{becomes} \quad L_T: \; \text{loadl} \; 1 \Rightarrow r_2
\text{br} \quad \rightarrow L_E
L_F: \; \text{loadl} \; 0 \Rightarrow r_2
L_E: \; \ldots \text{other stmts} \ldots$

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.

**Example**

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

**Variations on the ILOC Branch Structure**

<table>
<thead>
<tr>
<th></th>
<th>Straight Condition Codes</th>
<th>Boolean Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>comp</strong></td>
<td>rₓ, rᵧ =⇒ cc₁</td>
<td>cmp_LT</td>
</tr>
<tr>
<td><strong>cbr_LT</strong></td>
<td>cc₁ → L₁, L₂</td>
<td>cbr</td>
</tr>
<tr>
<td><strong>L₁:</strong> <strong>add</strong></td>
<td>rₓ, rᵧ =⇒ ra</td>
<td>r₁ → L₁, L₂</td>
</tr>
<tr>
<td><strong>br</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>L₂:</strong> <strong>add</strong></td>
<td>rₓ, rᵧ =⇒ ra</td>
<td></td>
</tr>
<tr>
<td><strong>br</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>L_OUT:</strong> <strong>nop</strong></td>
<td></td>
<td></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
Boolean & Relational Values

Conditional move & predication both simplify this code

Example

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

Other Architectural Variations

<table>
<thead>
<tr>
<th>Conditional Move</th>
<th>Predicated Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp x, y → cc₁</td>
<td>cmp_LT rₓ, rᵧ → r₁</td>
</tr>
<tr>
<td>add rₓ, rₙ → r₁</td>
<td>(r₁)? add rₓ, rₙ → r₁</td>
</tr>
<tr>
<td>add rₙ, rₓ → r₂</td>
<td>(¬r₁)? add rₑ, rₓ → r₁</td>
</tr>
<tr>
<td>i2i_&lt; cc₁, r₁, r₂ → rₐ</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>
<th>Straight Condition Codes</th>
<th>Boolean Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>comp</strong></td>
<td>( r_a, r_b \Rightarrow cc_1 )</td>
<td>( r_a, r_b \Rightarrow r_1 )</td>
</tr>
<tr>
<td><strong>cbr_L</strong></td>
<td>( cc_1 \rightarrow L_1, L_2 )</td>
<td>( r_c, r_d \Rightarrow r_2 )</td>
</tr>
<tr>
<td><strong>L_1</strong>: comp</td>
<td>( r_c, r_d \Rightarrow cc_2 )</td>
<td><strong>and</strong> ( r_1, r_2 \Rightarrow r_x )</td>
</tr>
<tr>
<td><strong>cbr_L</strong></td>
<td>( cc_2 \rightarrow L_3, L_2 )</td>
<td></td>
</tr>
<tr>
<td><strong>L_2</strong>: loadl</td>
<td>0 ( \Rightarrow r_x )</td>
<td></td>
</tr>
<tr>
<td>br</td>
<td>( \rightarrow L_{OUT} )</td>
<td></td>
</tr>
<tr>
<td><strong>L_3</strong>: loadl</td>
<td>1 ( \Rightarrow r_x )</td>
<td></td>
</tr>
<tr>
<td>br</td>
<td>( \rightarrow L_{OUT} )</td>
<td></td>
</tr>
<tr>
<td><strong>L_{OUT}</strong>: nop</td>
<td></td>
<td></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: \( a(i) \) (1-based indexing)

\[
\begin{align*}
4. & \quad t1 = \text{addr}(a) - 4 \\
5. & \quad t2 = i \times 4 \\
6. & \quad t3 = t1[t2]
\end{align*}
\]

...
Optimization: **Local Common Subexpression Elimination (CSE)**

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

\[
\begin{align*}
4. & \quad t1 = \text{addr}(a) - 4 \\
5. & \quad t2 = i \times 4 \\
6. & \quad t3 = t1[t2] \\
7. & \quad t4 = \text{addr}(a) - 4 \\
8. & \quad t5 = i \times 4 \\
9. & \quad t6 = t4[t5] \\
10. & \quad t7 = t3 \times t6 \\
\end{align*}
\]
Optimization: **Local Common Subexpression Elimination (CSE)**

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

```
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i * 4
9. t6 = t4[t5]
10. t7 = t3 * t6
```

Basic Block DAG Construction

```
[ ]], t3, t6
```

```
\[ t1 = addr[a] - 4 \\
t2 = i * 4 \\
t3 = t1[t2] \\
t4 = addr[a] - 4 \\
t5 = i * 4 \\
t6 = t4[t5] \\
t7 = t3 * t6
```

```
\[ t1 = addr[a] - 4 \\
t2 = i * 4 \\
t3 = t1[t2] \\
t7 = t3 * t3
```

\( * \), \( t7 \)
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 \( C \).

Create node(+) with children \( \text{node}(B), \text{node}(C) \)

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!
ILOC common subexpressions

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

Idea: Use register numbers as value numbers

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

```plaintext
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
add r6, r12 => r13
```

Hash(<loadI, 1024>) = undefined; gen_code; set to r0, return r0

Hash(<loadAI, r0, 8>) = undefined; gen_code; set to r1, return r1

Hash(<loadI, 1>) = undefined; gen_code; set to r2; return r2

Hash(<sub, r1, r2>) = undefined; gen_code; set to r3; return r3

Hash(<loadI, 4>) = undefined; gen_code; set to r4; return r4

Hash(<mult, r3, r4>) = undefined; gen_code; set to r5; return r5

Hash(<loadAO, r0, r5>) = undefined; gen_code; set to r6; return 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>) = undefined; gen_code; set to r7
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
add r6, r6 => r7

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
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

loadAI r0, 8 => r7
loadI 1 => r8
sub r7, r8 => r9
loadI 4 => r10
mult r9, r10 => r11
loadAO r0, r11 => r12

add r6, r12 => r13

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

```assembly
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!
Intermediate Representations
(EaC Chapter 5)

- 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
  → Size
  → Level of abstraction

• The importance of different properties varies between compilers
  → Selecting an appropriate IR for a compiler is critical
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:
- Structural: Trees, DAGs
- Linear: 3 address code, Stack machine code
- Hybrid: Control-flow graph
The level of detail exposed in an IR influences the profitability and feasibility of different optimizations.

Two different representations of an array reference:

```
l chí 1 => r₁
sub r_j, r₁ => r₂
loadI 10 => r₃
mult r₂, r₃ => r₄
sub r_i, r₁ => r₅
add r₄, r₅ => r₆
loadI @A => r₇
Add r₇, r₆ => r₈
load r₈ => r_{Aij}
```

High level AST:
Good for memory disambiguation

Low level linear code:
Good for address calculation
Level of Abstraction

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

```
load
+
+
*  
/
-
-
-  10  
|   |
+  
/  
i  1
```

Low level AST

```
loadArray A, i, j
```

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

\[
\begin{align*}
x - 2 * y
\end{align*}
\]

- Can use linearized form of the tree
  - Easier to manipulate than pointers
    - \[x \ 2 \ y \ * \ -\] in postfix form
    - \[- \ * \ 2 \ y \ x\] in prefix form
- \(S\)-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

The expression $z \leftarrow x - 2 \times y$ and $w \leftarrow x / 2$ appears twice, meaning 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

  \[
  \begin{align*}
  &\text{push } x \\
  &\text{push } 2 \\
  &\text{push } y \\
  &\text{multiply} \\
  &\text{subtract}
  \end{align*}
  \]

Advantages

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

Useful where code is transmitted over slow communication links (the \textit{net})

Implicit names take up no space, where explicit ones do!
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 (\(\text{op}\)) and, at most, 3 names (\(x, y, z\))

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

\[
\begin{align*}
\text{load} & \quad r1, y \\
\text{loadI} & \quad r2, 2 \\
\text{mult} & \quad r3, r2, r1 \\
\text{load} & \quad r4, x \\
\text{sub} & \quad r5, r4, r3
\end{align*}
\]

**RISC assembly code**

<table>
<thead>
<tr>
<th>Quadruples</th>
</tr>
</thead>
<tbody>
<tr>
<td>load \ 1 \ Y</td>
</tr>
<tr>
<td>loadI \ 2 \ 2</td>
</tr>
<tr>
<td>mult \ 3 \ 2 \ 1</td>
</tr>
<tr>
<td>load \ 4 \ X</td>
</tr>
<tr>
<td>sub \ 5 \ 4 \ 2</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

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**

$$
x \leftarrow \ldots
$$

$$
y \leftarrow \ldots
$$

while $(x < k)$

$$
x \leftarrow x + 1
$$

$$
y \leftarrow y + x
$$

**SSA-form**

$$
x_0 \leftarrow \ldots
$$

$$
y_0 \leftarrow \ldots
$$

if $(x_0 > k)$ goto next

**loop:**

$$
x_1 \leftarrow \phi(x_0, x_2)
$$

$$
y_1 \leftarrow \phi(y_0, y_2)
$$

$$
x_2 \leftarrow x_1 + 1
$$

$$
y_2 \leftarrow y_1 + x_2
$$

if $(x_2 < k)$ goto loop

**next:**

$$
\ldots
$$

**Strengths of SSA-form**

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

Procedure abstraction
Read EaC: Chapter 6.1 - 6.5