CS415 Compilers

Code Generation & Introduction to Code Optimizations

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Review: Computing an Array Address

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

\( A[i] \)

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

\( \text{int } A[1:10] \Rightarrow \text{low is 1} \)

Make low 0 for faster access (saves a - )

Almost always a power of 2, known at compile-time

\( \Rightarrow \text{use a shift for speed} \)
Review: 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: base(A) + ( i - low ) \times sizeof(A[1])

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

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

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

**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

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

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

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 \( \text{low}_1, \text{high}_1, \) 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

\[ @A_0 + (i \times \text{len}_2 + j) \times w \]

Compile-time constants
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
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

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

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

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

Example:

\[
\begin{align*}
\text{cmp} & \quad r_x, r_y \rightarrow \text{cc}_1 \\
\text{cbr}_{LT} & \quad \text{cc}_1 \rightarrow L_T, L_F \\
x < y & \quad \text{becomes} \\
\end{align*}
\]

\[
\begin{align*}
L_T: & \quad \text{loadI} \ 1 \Rightarrow r_2 \\
& \quad \text{br} \rightarrow L_E \\
L_F: & \quad \text{loadI} \ 0 \Rightarrow r_2 \\
L_E: & \quad \ldots \text{other stmts} \ldots
\end{align*}
\]

This “positional representation” is much more complex
What if the ISA uses a condition code?
• Must use a conditional branch to interpret result of compare
• Necessitates branches in the evaluation

Example:

\[ \text{cmp } r_x, r_y \Rightarrow cc_1 \]
\[ \text{cbr}_\text{LT } cc_1 \Rightarrow L_T, L_F \]

\[ x < y \text{ becomes } L_T: \text{loadI} 1 \Rightarrow r_2 \]
\[ \text{br } \Rightarrow L_E \]
\[ L_F: \text{loadI} 0 \Rightarrow r_2 \]
\[ L_E: \text{...other stmts...} \]

Condition codes
• are an architect’s hack
• allow ISA to avoid some comparisons
  \( (x < y \lor y < z) \)
• complicates code for simple cases

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>L₁: add</td>
</tr>
<tr>
<td>br</td>
</tr>
<tr>
<td>L₂: 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

- **Example**
  - \( (x < y) \)
  - \( \text{then } a \leftarrow c + d \)
  - \( \text{else } a \leftarrow e + f \)

<table>
<thead>
<tr>
<th></th>
<th><strong>Conditional Move</strong></th>
<th><strong>Predicated Execution</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>comp</strong></td>
<td>( r_x, r_y \Rightarrow cc_1 )</td>
<td>( r_x, r_y \Rightarrow r_1 )</td>
</tr>
<tr>
<td><strong>add</strong></td>
<td>( r_c, r_d \Rightarrow r_1 )</td>
<td>( (r_1) ? ) add ( r_c, r_d \Rightarrow r_a )</td>
</tr>
<tr>
<td><strong>add</strong></td>
<td>( r_e, r_f \Rightarrow r_2 )</td>
<td>( (\neg r_1) ? ) add ( r_e, r_f \Rightarrow r_a )</td>
</tr>
<tr>
<td><strong>i2i_\text{&lt;}</strong></td>
<td>( 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?
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 ( r_a, r_b \Rightarrow cc_1 )</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: ) loadI ( 0 \Rightarrow r_x )</td>
</tr>
<tr>
<td>br</td>
</tr>
<tr>
<td>( L_3: ) loadI ( 1 \Rightarrow r_x )</td>
</tr>
<tr>
<td>br</td>
</tr>
<tr>
<td>( L_{OUT}: ) nop</td>
</tr>
</tbody>
</table>

Here, the boolean compare produces much better code.
Control Flow

If-then-else
- Follow model for evaluating relationals & booleans with branches

Branching versus predication
- Frequency of execution
  - Uneven distribution ⇒ do what it takes to speed common case
- Control flow inside the construct
  - Any branching activity within the case base complicates the predicates and makes branches attractive
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++) {
    body
    next statement
}

Initialization

loadI 1 ⇒ r_1
loadI 1 ⇒ r_2
loadI 100 ⇒ r_3
cmp_GE r_1, r_3 ⇒ r_4
cbr r_4 ⇒ L_2, L_1

Pre-test

L_1: body
    add r_1, r_2 ⇒ r_1
cmp_LT r_1, r_3 ⇒ r_5
cbr r_5 ⇒ L_1, L_2

Post-test

L_2: 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/continue** in loop goes to next iteration
Control Flow

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

Surprisingly many compilers do this for all cases!
Compiler Optimization

- 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. Vectorization / parallelization (*source level*)
2. Common subexpression elimination (CSE - local, *ILOC level*) - Next Class
Data Dependences ⇒ defined on memory locations / registers and not values

Statement/instruction b data depends on statement/instruction a if there exists:

- **true** or flow dependence
  a writes a location/register that b later reads (RAW conflict)

- **anti** dependence
  a reads a location/register that b later writes (WAR conflict)

- **output** dependence
  a writes a location/register that b later writes (WAW conflict)

\[
\begin{array}{ccc}
\text{true} & \text{anti} & \text{output} \\
 a = & a = a & a = a = a \\
= a & a = & a =
\end{array}
\]
Data / Control Dependences


A statement $S_1$ control depends on statement $S_2$ iff
(a) $S_2$ computes a conditional branch and
(b) the execution of $S_1$ depends on this branch (in one case $S_1$ will be executed, but not in the other).

Data and control dependences define ORDER CONSTRAINTS that need to be respected in order to generate correct code.

**Fundamental Theorem of Dependence**

**Theorem**
Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
Our Goal: Find out whether a loop can be parallelized or vectorized based on data dependence analysis (assume: body of a loop is a basic block)

array $a(1:100)$ of float
for $i := 3, 99$
  $a(i) = a(i+1) + 1$

Iteration space

anti-dependences with dependence distance = 1
Parallelization vs Vectorization

Our Goal: Find out whether a loop can be parallelized or vectorized based on data dependence analysis (assume: body of a loop is a basic block)

array a(1:100) of float
for i := 3, 99
  a(i) = a(i+1) + 1

Iteration space

| 3 | 4 | 5 | ... | 98 | 99 |

Iteration space

∀ i := 3, 99

a(i) = a(i+1) + 1

Loop cannot be parallelized since no order constraints among loop iterations => anti-dep. may be violated!

Loop can be vectorized since RHS is read before LHS is written => anti-dep. is preserved!
Our Goal: Find out whether a loop can be parallelized or vectorized based on data dependence analysis (assume: body of a loop is a basic block)

array a(1:100) of float
for i := 3, 99
    a(i) = a(i) + 1

Iteration space

3  4  5  ...  98  99

forall i := 3, 99
    a(i) = a(i) + 1

only intra-statement anti-dependences which are preserved by statement execution semantics

a(3:99) = a(3:99) + 1
Parallelization vs Vectorization

Our Goal: Find out whether a loop can be parallelized or vectorized based on data dependence analysis (assume: body of a loop is a basic block)

array a(1:100) of float
for i := 3, 99
  a(i) = a(i-1) + 1

iteration space

forall i := 3, 99
  a(i) = a(i-1) + 1

a(3:99) = a(2:98) + 1

Loop cannot be parallelized since each array assignment depends on previous array assignment => inherently sequential execution

Loop cannot be vectorized

Lecture 21
Perfectly nested loop, $A$ is $m$-dimensional array

\[
\begin{align*}
& \text{DO } i_1 = L_1, U_1 \\
& \quad \text{DO } i_2 = L_2, U_2 \\
& \quad \quad \ldots \\
& \quad \text{DO } i_n = L_n, U_n \\
& S_1 \quad A(f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)) = \ldots \\
& S_2 \quad \ldots = A(g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n)) \\
& \text{ENDDO} \\
& \quad \ldots \\
& \text{ENDDO} \\
& \text{ENDDO}
\end{align*}
\]

\[
f_i(\alpha) = g_i(\beta) \text{ for all } i, 1 \leq i \leq m
\]
\[
\alpha < \beta
\]

Finding integer solutions for this set of simultaneous equations with constraints is an integer programming problem, which is NP-complete.
Our assumptions:

- Singly nested loop
- Single assignment statement
- All arrays are one-dimensional
- No aliasing among arrays
- \( f(i) \) is an affine function of the induction variable \( i \): \( a \times i + c \), with positive integer \( a > 0 \), and \( c \) an integer

\[ \Rightarrow \text{Anti and output dependences don’t matter here for vectorization} \]

Goal of dependence testing:

1. Prove that true dependence does not exist;
2. If cannot do that, show that dependence exists with a fixed distance vector;
3. If cannot do that, assume existence of dependence

Dependence testing algorithm:

For each pair of LHS and RHS array references \( <X(f(i)), Y(g(i))> \) do perform dependence test.

\[ \text{for } i := lb, ub \]
\[ X(f(i)) = \ldots Y(g(i)) \ldots \]
for $i := lb, ub$

\[ X(f(i)) = \ldots Y(g(i)) \ldots \]

**Input**: \(<X(f(i)), Y(g(i))>\)

**Output**: no dependence, dependence with distance vector $d$, or dependence

**Method**: (cascading tests)

\[
\text{if } X \neq Y \text{ then report “no dependence” else}
\]

\[
\quad \text{if } f(i) = i + c \text{ and } g(i) \text{ is a constant (or visa versa) then}
\]

\[
\quad \quad \text{apply simple ZIV test else}
\]

\[
\quad \quad \quad \text{if } f(i) = a \times i + c_1 \text{ and } g(i) = a \times i + c_2 \text{ then}
\]

\[
\quad \quad \quad \quad \text{apply strong SIV test else}
\]

\[
\quad \quad \quad \quad \quad \text{report “dependence”}
\]
for $i := lb, ub$

$$X(f(i)) = \ldots X(g(i)) \ldots$$

$f(i) = i + c_1$ and $g(i) = c_2$, i.e., $g(i)$ is a constant; this means that the same memory location $X(c_2)$ is read in all iterations.

Examples:

\[
\begin{align*}
\langle A(i), A(5) \rangle \\
\langle A(i+1), A(7) \rangle
\end{align*}
\]

No dependence exists if $(i + c_1 - c_2) \neq 0$ for all $lb \leq i \leq ub$:

\[
lb \leq |c_1 - c_2| \leq ub \implies \text{dependence}
\]

Note: This is a special case of the weak-zero SIV test.
Simple Zero Induction Variable (ZIV) Test

for i := lb, ub
    \( X(f(i)) = ... X(g(i)) ... \)

\( f(i) = i + c_1 \) and \( g(i) = c_2 \), i.e., \( g(i) \) is a constant; this means that the same memory location \( X(c_2) \) is read in all iterations.

Examples:
\[ A(i), A(5) \]
\[ A(i+1), A(7) \]

No dependence exists if \( (i + c_1 - c_2) \neq 0 \) for all \( lb \leq i \leq ub \):

\[ lb \leq |c_1 - c_2| \leq ub \implies \text{dependence} \]

Note: This is a special case of the weak-zero SIV test.

Strong Single Induction Variable (SIV) Test

for $i := lb, ub$

$X(f(i)) = \ldots X(g(i)) \ldots$

$f(i) = a \times i + c_1$ and $g(i) = a \times i + c_2$

Examples:

- $< A(i), A(i - 1) >$
- $< A(4i + 2), A(4i - 1) >$

Dependence exists with distance $d$ if there exists an integer solution to the following equation:

- $f(i) = a \times i + c_1$ and $g(i') = a \times i' + c_2$

- $d = i' - i = \frac{c_1 - c_2}{a}$ and $d$ has to be (1) a positive integer and (2) $d \leq (ub-lb)$

- if dependence exists, report “dependence with distance $d$”
for $i := lb, ub$

$$X(f(i)) = \ldots X(g(i)) \ldots$$

$$(i') - i = \frac{c_1 - c_2}{a}$$

$$d \leq ub - lb$$
More Code Optimizations
Read EaC: Chapter 6.1 - 6.5