CS415 Compilers

Code Generation

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

Roadmap

- Code Generation
  - → Expression
  - → Handling Assignment
  - → Array
  - → Boolean and Relational Values
  - → Control Flow

How does the compiler handle A[i,j] ?

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

A				
	1,1	1,2	1,3	1,4
	2,1	2,2	2,3	2,4

These have distinct & different cache behavior

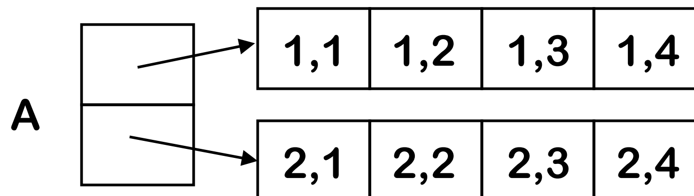
Row-major order

A								
	1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

Column-major order

A								
	1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4

Indirection vectors



Computing an Array Address

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

A[i]

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

Computing an Array Address

Declaration: A[low .. high] of …

A[i]

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

int A[1:10] ⇒ low is 1
Make low 0 for faster access (saves a -)

Almost always a power of 2, known at compile-time ⇒ use a shift for speed

Computing an Array Address

Declaration: A[low1 .. high1, low2 .. high2] of …

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

What about $A[i_1,i_2]$?

This stuff looks expensive!
Lots of implicit +, -, x ops

Row-major order, two dimensions

$@A + ((i_1 - low_1) \times (high_2 - low_2 + 1) + i_2 - low_2) \times sizeof(A[1])$

Column-major order, two dimensions

$@A + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times sizeof(A[1])$

Indirection vectors, two dimensions

$*(A[i_1])[i_2]$ — where $A[i_1]$ is, itself, a 1-d array reference

Optimizing Address Calculation for A[i,j]

where $w = sizeof(A[1,1])$

In row-major order

$$@A + (i-low_1)(high_2-low_2+1) \times w + (j - low_2) \times w$$

Which can be factored into

$$@A + i \times (high_2-low_2+1) \times w + j \times w$$
$$- (low_1 \times (high_2-low_2+1) \times w) + (low_2 \times w)$$

If low_i, $high_i$, and w are known, the last term is a constant

Define $@A_0$ as

$$@A - (low_1 \times (high_2-low_2+1) \times w + low_2 \times w$$

And len_2 as $(high_2-low_2+1)$

Then, the address expression becomes

$$@A_0 + (i \times len_2 + j) \times w$$

Example: Array Address Calculations in a Loop

FORTRAN: column-major order

```
DO J = 1, N
    A[I,J] = A[I,J] + B[I,J]
END DO
```

- Naïve: Perform the address calculation twice

DO J = 1, N

$R1 = @A_0 + (J \times len_1 + I) \times floatsize$

$R2 = @B_0 + (J \times len_1 + I) \times floatsize$

MEM(R1) = MEM(R1) + MEM(R2)

END DO

Example: Array Address Calculations in a Loop

```
DO J = 1, N
    A[I,J] = A[I,J] + B[I,J]
END DO
```

- Sophisticated: Move comon calculations out of loop

```
R1 = I x floatsize
c = len1 x floatsize    ! Compile-time constant
R2 = @A0 + R1
R3 = @B0 + R1
DO J = 1, N
    a = J x c
    R4 = R2 + a
    R5 = R3 + a
    MEM(R4) = MEM(R4) + MEM(R5)
END DO
```

Example: Array Address Calculations in a Loop

```
DO J = 1, N
    A[I,J] = A[I,J] + B[I,J]
END DO
```

- Very sophisticated: Convert multiply to add (Operator Strength Reduction)

```
R1 = I x floatsize
c = len_1 x floatsize  ! Compile-time constant
R2 = @A_0 + R1
R3 = @B_0 + R1
DO J = 1, N
    R2 = R2 + c
    R3 = R3 + c
    MEM(R2) = MEM(R2) + MEM(R3)
END DO
```

Today's topics

- Code Generation
  - → Expression
  - → Handling Assignment
  - → Array
  - → Boolean and Relational Values
  - → Control Flow

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	*becomes*	cmp_LT $r_x, r_y \Rightarrow r_1$
if (x < y) then $stmt_1$ else $stmt_2$	*becomes*	cmp_LT $r_x, r_y \Rightarrow r_1$ cbr $r1 \rightarrow _stmt_1, _stmt_2$

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:

x < y *becomes*

```
     cmp     rx, ry ⇒ cc1
     cbr_LT  cc1 → LT,LF
LT:  loadI   1 ⇒ r2
     br        → LE
LF:  loadI   0 ⇒ r2
LE:  ...other stmts...
```

This "positional representation" is much more complex

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:

x < y *becomes*

```
        cmp     r_x, r_y ⇒ cc_1
        cbr_LT  cc_1 → L_T, L_F
L_T:    loadI   1 ⇒ r_2
        br        → L_E
L_F:    loadI   0 ⇒ r_2
L_E:    ...other stmts...
```

Condition codes

- are an architect's hack
- allow ISA to avoid some comparisons
- complicates code for simple cases

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

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

<table>
<tr><th colspan="6">VARIATIONS ON THE ILOC BRANCH STRUCTURE</th></tr>
<tr><th colspan="3">Straight Condition Codes</th><th colspan="3">Boolean Compares</th></tr>
<tr><td></td><td>comp</td><td>$r_x,r_y \Rightarrow cc_1$</td><td></td><td>cmp_LT</td><td>$r_x,r_y \Rightarrow r_1$</td></tr>
<tr><td></td><td>cbr_LT</td><td>$cc_1 \rightarrow L_1,L_2$</td><td></td><td>cbr</td><td>$r_1 \rightarrow L_1,L_2$</td></tr>
<tr><td>L_1:</td><td>add</td><td>$r_c,r_d \Rightarrow r_a$</td><td>L_1:</td><td>add</td><td>$r_c,r_d \Rightarrow r_a$</td></tr>
<tr><td></td><td>br</td><td>$\rightarrow L_{OUT}$</td><td></td><td>br</td><td>$\rightarrow L_{OUT}$</td></tr>
<tr><td>L_2:</td><td>add</td><td>$r_e,r_f \Rightarrow r_a$</td><td>L_2:</td><td>add</td><td>$r_e,r_f \Rightarrow r_a$</td></tr>
<tr><td></td><td>br</td><td>$\rightarrow L_{OUT}$</td><td></td><td>br</td><td>$\rightarrow L_{OUT}$</td></tr>
<tr><td>L_{OUT}:</td><td>nop</td><td></td><td>L_{OUT}:</td><td>nop</td><td></td></tr>
</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					
Conditional Move		***Predicated Execution***			
comp	$r_x, r_y \Rightarrow cc_1$		cmp_LT	$r_x, r_y \Rightarrow r_1$
add	$r_c, r_d \Rightarrow r_1$	(r_1)?	add	$r_c, r_d \Rightarrow r_a$
add	$r_e, r_f \Rightarrow r_2$	$(\neg r_1)$?	add	$r_e, r_f \Rightarrow r_a$
i2i_<	$cc_1, r_1, r_2 \Rightarrow r_a$			

Both versions avoid the branches

Both are shorter than CCs or Boolean-valued compare

Are they better? What about power?

Boolean & Relational Values

Consider the assignment $x \leftarrow a < b \wedge c < d$

VARIATIONS ON THE ILOC BRANCH STRUCTURE	
Straight Condition Codes	***Boolean Compare***
comp $r_a,r_b \Rightarrow cc_1$	cmp_LT $r_a,r_b \Rightarrow r_1$
cbr_LT $cc_1 \rightarrow L_1,L_2$	cmp_LT $r_c,r_d \Rightarrow r_2$
L_1: comp $r_c,r_d \Rightarrow cc_2$	and $r_1,r_2 \Rightarrow r_x$
cbr_LT $cc_2 \rightarrow L_3,L_2$	
L_2: loadI $0 \Rightarrow r_x$	
br $\rightarrow L_{OUT}$	
L_3: loadI $1 \Rightarrow r_x$	
br $\rightarrow L_{OUT}$	
L_{OUT}: nop	

Here, the boolean compare produces much better code

Boolean & Relational Values

Consider the assignment $x \leftarrow a < b \wedge c < d$

Variations on the Iloc Branch Structure					
Straight Condition Codes			***Boolean Compare***		
	comp	$r_a, r_b \Rightarrow cc_1$	cmp_LT	$r_a, r_b \Rightarrow r_1$	
	cbr_LT	$cc_1 \rightarrow L_1, L_2$	cmp_LT	$r_c, r_d \Rightarrow r_2$	
L_1:	comp	$r_c, r_d \Rightarrow cc_2$	and	$r_1, r_2 \Rightarrow r_x$	
	cbr_LT	$cc_2 \rightarrow L_3, L_2$			
L_2:	loadI	$0 \Rightarrow r_x$			
	br	$\rightarrow L_{OUT}$			
L_3:	loadI	$1 \Rightarrow r_x$			
	br	$\rightarrow L_{OUT}$			
L_{OUT}:	nop				

Potential Problem: Optimized boolean expression

Here, the boolean compare produces much better code

Today's topics

- Review on error detection/recovery
- Review on type expression
- Code Generation
  - → Expression
  - → Handling Assignment
  - → Array
  - → Boolean and Relational Values
  - → <u>Control Flow</u>

Control Flow

If-then-else

- Follow model for evaluating relationals & booleans with branches

Branching versus predication *(e.g., IA-64)*

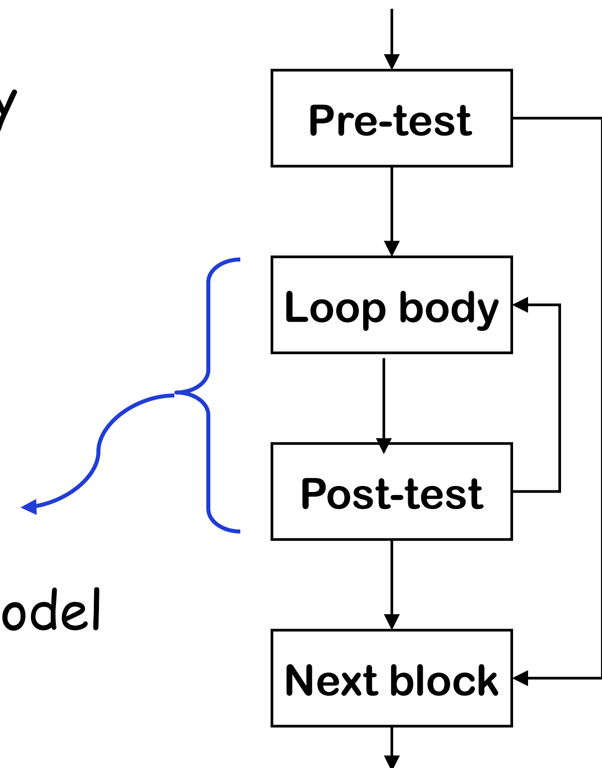
- Frequency of execution
  - → Uneven distribution ⇒ do what it takes to speed common case
- Amount of code in each case
  - → Unequal amounts means predication may waste issue slots
- Control flow inside the construct
  - → Any branching activity within the case base complicates the predicates and makes branches attractive

Control Flow

Loops

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

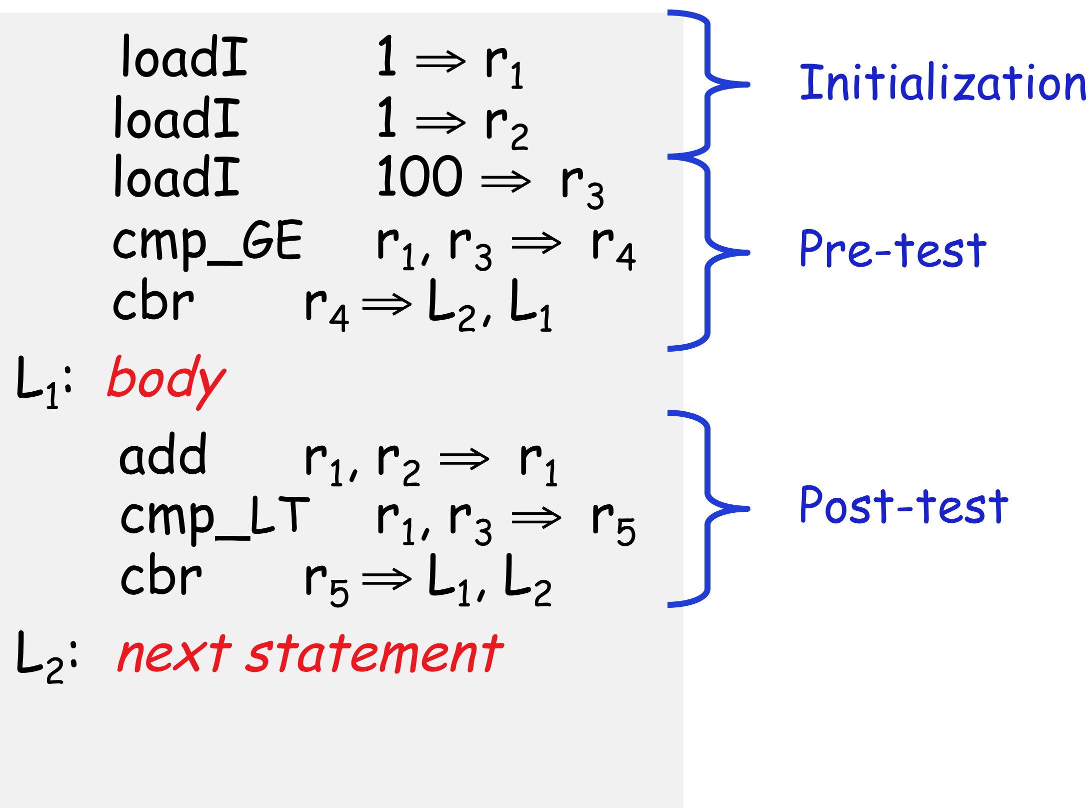
Merges test with last block of loop body



while, for, do, & until all fit this basic model

Loop Implementation Code

for (i = 1; i< 100; i++) { *body* }
next statement



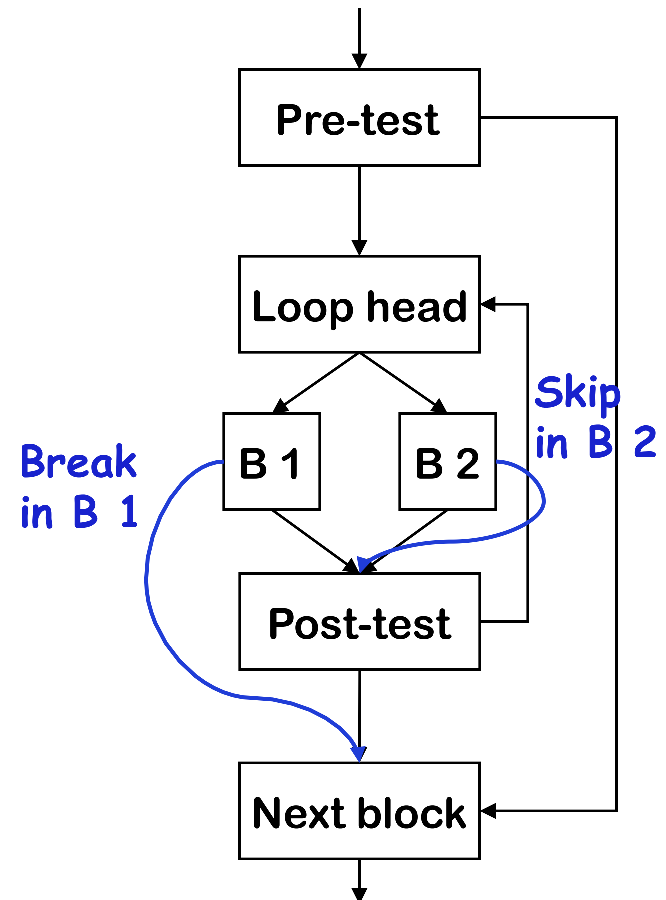
Break statements

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



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

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 (*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)

Next class

Global Register Allocation

Read ALSU: Chapter 8.8