Compiling Techniques

Lecture 12: Code Shapes
(EaC Chapter 7)

Christophe Dubach

17 November 2015

Coursework Demo: Friday 4th of December

- In order to comply with the school regulations, you will have to give a demonstration of your compiler.
- There is nothing for you to prepare; we will simply ask you to run your compiler and ask questions about your code to verify you are the one who actually wrote it.
- This demo will take place on Friday 4th of December between 1-5pm. We will organise a more detailed timetable using doodle poll in the following days.
- Attendance is mandatory; if we cannot see a demo, you will fail the course.

Table of contents

Boolean and Relational Values

How should the compiler represent them?

It depends on the target machine

Several approaches:

- Numerical representation
- Positional Encoding (*e.g.*, Java ByteCode)
- Conditional Move and Predication

Correct choice depends on both context and ISA (instruction set architecture)

Numerical Representation

- Assign values to true and false, usually 1 and 0
- Use comparison operator to get a value from a relational expression

Example

`x < y`	`cmp_LT rx, ry → r1`

`if (x < y)`
`  stmt1`
`else`
`  stmt2`
	`    cmp_LT rx, ry → r1`
	`    cbr    r1     → L1`
	`    stmt2`
	`    br            → Le`
	`L1: stmt1`
	`Le:`

Positional Encoding

What if the ISA does not provide comparison operators that returns a value?

- Must use conditional branch to interpret the result of a comparison
- Necessitates branches in the evaluation
- This is the case for Java ByteCode (if_cmp<cond>)

Example: x<y

```
      br_LT  rx, ry  →  L_T
      loadI  0       →  r2
      br             →  L_E
L_T:  loadI  1       →  r1
L_E:  ...
```

If the result is used to control an operation, then positional encoding is not that bad.

Example

```
if (x < y)
  a = c + d;
else
  a = e + f;
```

Corresponding assembly code

Boolean comparison	Positional encoding
cmp_LT rx, ry → r1 cbr r1 → L_T add re, rf → ra br → L_E L_T: add rc, rd → ra L_E: ...	br_LT rx, ry → L_T add re, rf → ra br → L_E L_T: add rc, rd → ra L_E: ...

Conditional Move and Predication

Conditional move and predication can simplify this code.

Example

```
if (x < y)
  a = c + d;
else
  a = e + f;
```

Corresponding assembly code

Conditional Move	Predicated Execution
cmp_LT rx, ry → r1	cmp_LT rx, ry → r1
add rc, rd → r2	(r1)? add rc, rd → ra
add re, rf → r3	(!r1)? add re, rf → ra
cmov r1, r2, r3 → ra	

Last word on boolean and relational values: consider the following code x = (a<b) & (c<d)

Corresponding assembly code

Positional encoding	Boolean Comparison
br_LT ra , rb → L_1	
br → L_2	
L_1: br_LT rc , rd → L_3	cmp_LT ra , rb → r1
L_2: loadI 0 → rx	cmp_LT rc , rd → r2
br → Le	and r1 , r2 → rx
L_3: loadI 1 → rx	
L_e: . . .	

Here the boolean comparison produces much better code.

Best choice depends on two things

- Context
- Hardware

Control-Flow

- If-then-else
- Loops (for, while, ...)
- Switch/case statements

If-then-else

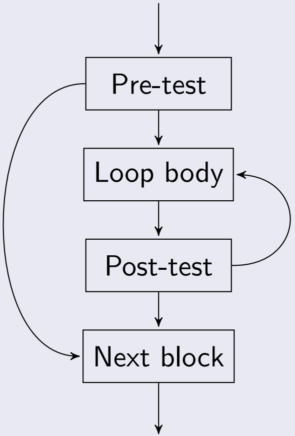
Follow the model for evaluating relational and boolean with branches.

Branching versus predication (*e.g.*, IA-64, ARM ISA) trade-off:

- Frequency of execution:
  uneven distribution, try to speedup common case
- Amount of code in each case:
  unequal amounts means predication might waste issue slots
- Nested control flow:
  any nested branches complicates the predicates and makes branching attractive

Loops

Basic pattern



- evaluate condition before the loop (if needed)
- evaluate condition after the loop
- branch back to the top (if needed)

while, for and do while loops all fit this basic model.

Example: for loop

```
for (i=1; i<100; i++) {
  body
}
next stmt
```

Corresponding assembly

```
     loadI  1      →  r1
     loadI  100    →  r2
     br_GT   r1, r2  →  L2
L1:  body
     addI  r1,1   →  r1
     br_LT  r1, r2  →  L1
L2:  next stmt
```

Exercise

Write the assembly code for the following while loop:

```
while (x >= y) {
  body
}
next stmt
```

Most modern programming languages include a break statements

- Exits from the innermost control-flow statement
  - Out of the innermost loop
  - Out of a case statement
- Solution:
  - use an unconditional branch to the next statement following the control-flow construct (loop or case statement).
  - skip or continue statement branch to the next iteration (start of the loop)

Case Statement (switch)

Case statement

```
switch (c) {
  case 'a': stmt1;
  case 'b': stmt2; break;
  case 'c': stmt3;
}
```

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

Part 2 is key.

Strategies:

- Linear search (nested if-then-else)
- Build a table of case expressions and use binary search on it
- Directly compute an address (requires dense case set)

Exercise

Knowing that the character 'a' corresponds to the decimal value 97 (ASCII table), write the assembly code for the example below using linear search.

```
char c;
...
switch (c) {
  case 'a': stmt1;
  case 'b': stmt2; break;
  case 'c': stmt3; break;
  case 'd': stmt4;
}
stmt5;
```

Instruction selection

- Peephole Matching
- Tree-pattern matching