CS412/413

Introduction to Compilers

Radu Rugina

Lecture 31: Instruction Selection

14 Apr 06

Backend Optimizations

- Instruction selection
  - translate low-level IR to assembly instructions
  - A machine instruction may model multiple IR instructions
  - Especially applicable to CISC architectures
- Register Allocation
  - Place variables into registers
  - Avoid spilling variables on stack

Instruction Selection

- Different sets of instructions in low-level IR and in the target machine
- Instruction selection = translate low-level IR to assembly instructions on the target machine
- Straightforward solution: translate each low-level IR instruction to a sequence of machine instructions
- Example:

```
x = y + z   ⇒   mov y, r1
                mov z, r2
                add r2, r1
                mov r1, x
```


Instruction Selection

- Problem: straightforward translation is inefficient
  - One machine instruction may perform the computation in multiple low-level IR instructions
- Consider a machine with includes the following instructions:

`add r2, r1`	r1 ← r1+r2
`mulc c, r1`	r1 ← r1*c
`load r2, r1`	r1 ← *r2
`store r2, r1`	*r1 ← r2
`movem r2, r1`	*r1 ← *r2
`movex r3, r2, r1`	*r1 ← *(r2+r3)

Example

- Consider the computation: `a[i+1] = b[j]`
- Assume `a`,`b`,`i`,`j` are variables
  - register `ra` holds address of `a`
  - register `rb` holds address of `b`
  - register `ri` holds value of `i`
  - register `rj` holds value of `j`

IR code:

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```


Possible Translation

- Address of `b[j]`: `mulc 4, rj` / `add rj, rb` → { `t1 = j*4`, `t2 = b+t1` }
- Load value `b[j]`: `load rb, r1` → `t3 = *t2`
- Address of `a[i+1]`: `add 1, ri` / `mulc 4, ri` / `add ri, ra` → { `t4 = i+1`, `t5 = t4*4`, `t6 = a+t5` }
- Store into `a[i+1]`: `store r1, ra` → `*t6 = t3`

IR code:

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

Another Translation

- Address of `b[j]`: `mulc 4, rj` / `add rj, rb` → `t1 = j*4`, `t2 = b+t1`
- Address of `a[i+1]`: `add 1, ri` / `mulc 4, ri` / `add ri, ra` → `t4 = i+1`, `t5 = t4*4`, `t6 = a+t5`
- Store into `a[i+1]`: `movem rb, ra`

IR code:

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```


Yet Another Translation

- Index of `b[j]`: `mulc 4, rj` → `t1 = j*4`
- Address of `a[i+1]`: `add 1, ri` / `mulc 4, ri` / `add ri, ra` → `t4 = i+1`, `t5 = t4*4`, `t6 = a+t5`
- Store into `a[i+1]`: `movex rj, rb, ra`

IR code:

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```


Instruction Costs

- Different machine instructions have different costs
  - Time cost: how fast instructions are executed
  - Space cost: how much space instructions take
- Example: cost = number of cycles

```
add r2, r1          cost=1
mulc c, r1          cost=10
load r2, r1         cost=3
store r2, r1        cost=3
movem r2, r1        cost=4
movex r3, r2, r1    cost=5
```

- Goal: find translation with smallest cost

How to Solve the Problem?

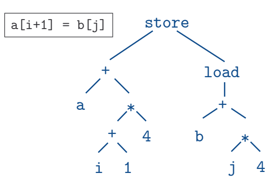
- Difficulty: low-level IR instruction matched by a machine instructions may not be adjacent
- Example: `movem rb, ra` → `t3 = *t2`, `*t6 = t3`
- Idea: use tree representation
  - Easier to detect matching instructions

IR code:

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```


Tree Representation

- Goal: determine parts of the tree which correspond to machine instructions

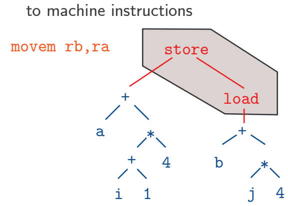


IR code:

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```


Tiles

- Tile = tree patterns (subtrees) corresponding to machine instructions

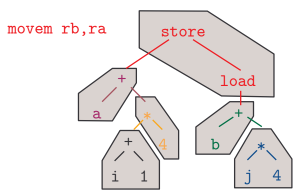


IR code:

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

Tiling

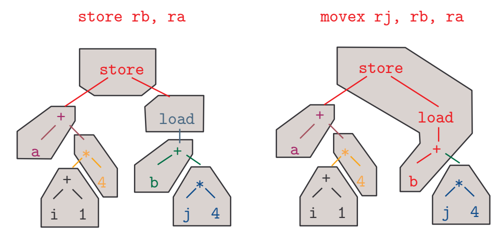
- Tiling = cover the tree with disjoint tiles



Assembly:

```
mulc 4, rj
add rj, rb
add 1, ri
mulc 4, ri
add ri, ra
movem rb,ra
```

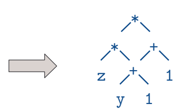

Different Tilings



Directed Acyclic Graphs

- Tree representation: appropriate for instruction selection
  - Tiles = subtrees → machine instructions
- DAG construction (aka Value Numbering)
  - Common sub-expressions represented by the same node
  - Tile the expression DAG
- Example:

```
t = y+1
y = z*t
t = t+1
z = t*y
```



Big Picture

- What the compiler has to do:
  1. Translate three-address code into a DAG representation
  2. Then find a good tiling of the DAG
     - Maximal munch algorithm
     - Dynamic programming algorithm

Value Numbering

- Input: a sequence of low IR instructions in a basic block
- Output: an expression DAG for the block
- Idea:
  - Label each DAG node with variable which holds that value
  - Build DAG bottom-up
- A variable may have multiple values in a block
- Use different variable indices for different values of the variable: t_0, t_1, t_2, etc.

Value Numbering Algorithm

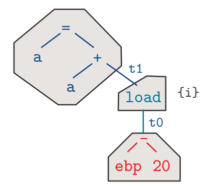
```
index[v] = 0 for each variable v

For each instruction I (in the order they appear)
    For each v ∈use[I], with n=index[v]
        if node v_n doesn't exist
            create node v_n
    Create node for instruction I, with children
        { v_n | v ∈use[I] }
    For each v∈def[I]
        index[v] = index[v] + 1
    If I is of the form x = … and n = index[x]
        label the new node with x_n
```

Next: Tiling

- Goal: find a good covering of DAG with tiles
- Issue: need to know what variables are in registers
- Assume abstract assembly:
  - Machine with infinite number of registers
  - Temporary/local variables stored in registers
  - Parameters/heap variables: use memory accesses

Example Tiling

- Consider the instruction `a = a + i`
  `a` = local variable
  `i` = parameter
- Need new temporary registers between tiles (unless operand node is labeled with temporary)
- Result code:




```
mov %ebp, t0
sub $20, t0
mov (t0), t1
add t1, a
```


Problems

- Classes of registers
  - Registers may have specific purposes
  - Example: Pentium multiply instruction
    - multiply register eax by contents of another register
    - store result in eax (low 32 bits) and edx (high 32 bits)
    - need extra instructions to move values into eax
- Two-address machine instructions
  - Three-address low-level code
  - Need multiple machine instructions for a single tile
- CISC versus RISC
  - Complex instruction sets: multiple possible tilings