CS412/CS413

Introduction to Compilers

Tim Teitelbaum

Lecture 31: Instruction Selection

13 Apr 07

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
  - Excessive memory traffic
- Consider a machine that 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 global 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

Low-level IR:

```
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)

Low-level IR:

```
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
- Address of a[i+1]: add 1, ri
  mulc 4, ri
  add ri, ra
- Store into a[i+1]: movem rb, ra

Low-level IR:

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
- Address of a[i+1]: add 1, ri
  mulc 4, ri
  add ri, ra
- Store into a[i+1]: movex rj, rb, ra

Low-level IR:

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

Issue: 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?

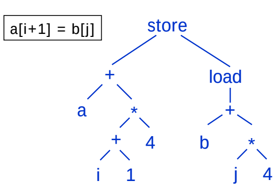
- Difficulty: low-level IR instruction matched by a machine instructions may not be adjacent
- Example: movem rb, ra
- Idea: use tree-like representation!
  - Easier to detect matching instructions

Low-level IR:

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 that correspond to machine instructions

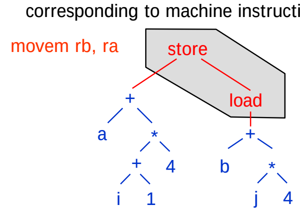


Low-level IR:

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

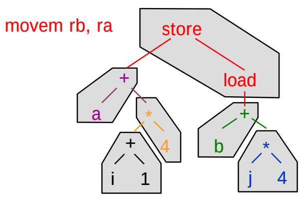


Low-level IR:

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

Tiling

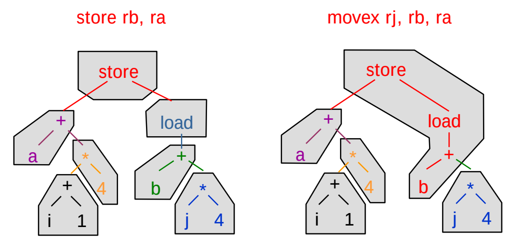
- Tiling = cover the tree with disjoint tiles



Assembly:

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

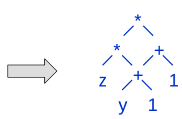
Tiling



Directed Acyclic Graphs

- Tree representation: appropriate for instruction selection
  - Tiles = subtrees → machine instructions
- DAG = more general structure for representing instructions
  - 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 low-level IR code into DAG representation
2. Then find a good tiling of the DAG
   - Maximal munch algorithm
   - Dynamic programming algorithm

DAG Construction

- 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 holding 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.

Algorithm

For each variable v do index[v] = 0

For each instruction I (in the order they appear)
    For each variable v that I directly uses, with n=index[v]
        if node v_n doesn't exist
            create node v_n , with label v_n
    Create expression 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

Example

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