CS42/413
Introduction to Compilers
Radu Rugina

Lecture 28: Instruction Selection
4 Apr 03

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
  mulc c, r1
  load r2, r1
  store r2, r1
  movem r2, r1
  movex r2, r2, r1
  - Consider a machine with includes the following instructions:
  - Consider a machine with includes the following instructions:
  - Consider a machine with includes the following instructions:
  - Consider a machine with includes the following instructions:
  - Consider a machine with includes the following instructions:
  - Consider a machine with includes the following instructions:

Example

- Consider the computation:
  - Consider the computation:
  - Consider the computation:
  - Consider the computation:
  - Consider the computation:
  - Consider the computation:
  - Consider the computation:

Possible Translation

- Address of b[j]:
  - Address of b[j]:
  - Address of b[j]:
  - Address of b[j]:
  - Address of b[j]:
  - Address of b[j]:

Another Translation

- Address of b[j]:
  - Address of b[j]:
  - Address of b[j]:
  - Address of b[j]:
  - Address of b[j]:
  - Address of b[j]:
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 j, rb, ra

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
  - movex 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

Tree Representation

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

Tiles

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

- movem rb, ra

Tiling

- Tiling = cover the tree with disjoint tiles

Assembly:

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

store rb, ra

movex rj, rb, ra

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 which holds that value
  - Build DAG bottom-up

Problem: a variable may have multiple values in a block
Solution: use different variable indices for different values of the variable: \( t_0, t_1, t_2, \ldots \)

Algorithm

index(v) = 0 for each variable v
For each instruction I (in the order they appear)
  For each v that I directly uses, with n=index[v]
    if node \( v \) doesn’t exist
      create node \( v \), with label \( v \)
    Create expression node for instruction I, with children
      \( ( v, | v \in \text{use}[I] ) \)
For each \( v \in \text{def}[I] \)
  index[v] = index[v] + 1
If I is of the form \( x = \ldots \) and \( n = \text{index}[x] \)
  label the new node with \( x_n \)

Issues

- Function/method calls
  - May update global variables or object fields
  - \( \text{def}[I] = \) set of globals/fields

- Store instructions
  - May update any variable
  - If stack addresses are not taken (e.g. Java),
    \( \text{def}[I] = \) set of heap objects
Local Variables in DAG

- Use stack pointers to access local variables
- Example: \( x = y + 1 \)

Next: DAG Tiling

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

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 \( \Rightarrow \) many possible tiles and tilings
  - Example: multiple addressing modes (CISC) versus load/store architectures (RISC)

Pentium ISA

- **Pentium:** two-address CISC architecture
- **General-purpose registers:** eax, ebx, ecx, edx, esi, edi
- **Stack registers:** ebp, esp
- **Typical instruction:**
  - Opcode (mov, add, sub, mul, div, jmp, etc)
  - Destination and source operands
- **Multiple addressing modes:** source operands may be
  - Immediate value: imm
  - Register: reg
  - Indirect address: [reg], [imm], [reg+imm],
  - Indexed address: [reg+reg'], [reg+imm+reg'],
  - [reg+imm*reg'+imm']
- **Destination operands = same, except immediate values**

Example Tiling

- Consider: \( t = t + i \)
  - \( t \) = temporary variable
  - \( i \) = parameter
- Need new temporary registers between tiles (unless operand node is labeled with temporary)
- Result code:
  - \( \text{mov} \ %\text{ebp}, t0 \)
  - \( \text{sub} \ 20, t0 \)
  - \( \text{mov} 0(t1), t1 \)
  - \( \text{add} t1, t \)
- **Note:** also compute \( i \), if it is live

Some Tiles

- \( \text{mov} t2, t1 \)
- \( \text{mov} 10, 0(t1,t2) \)
- \( \text{mov} t2, t3 \)
- \( \text{add} t1, t3 \)
- \( \text{mul} t2 \)
- \( \text{mov} \%\text{eax}, t1 \)
- \( \text{mov} \%\text{eax}, t3 \)
Conditional Branches

- How to tile a conditional jump?
- Fold comparison into tile

\[ \text{test } t1, t1 \]
\[ \text{jnz } L \]
\[ \text{cmp } t1, t2 \]
\[ \text{je } L \]

Load Effective Address

- Lea instruction computes a memory address
- Doesn't actually load the value from memory

\[ \text{lea } (t1, t2), t3 \]
\[ \text{lea } (t1, t2, 8), t3 \]