CS412/413

Introduction to Compilers
Radu Rugina

Lecture 32: Instruction Selection
17 Apr 06

Instruction Selection

1. Translate three-address code into DAG structure
2. Then find a good tiling of the DAG
   - disjoint set of tiles that cover the DAG
   - Maximal munch algorithm
   - Dynamic programming algorithm

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

Pentium ISA

• Pentium: two-address CISC architecture
• 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
Tiles

- Tiles capture compiler's understanding of instruction set
- May require additional move instructions
- Tiling = cover the DAG with tiles
- Need tiles for all single-node trees to guarantee that every tree can be tiled

Examples

- $\text{mov t1, t2}$
- $\text{add c, t2}$

- $\text{mov t2, t1}$
- $\text{mov c,0(t1,t2)}$

- $\text{mov t2, t3}$
- $\text{add t1, t3}$

- $\text{mov t1, %eax}$
- $\text{mul t2}$
- $\text{mov %eax, t3}$

Conditional Branches

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

- $\text{test t1,t1}$
- $\text{jnz L}$

- $\text{cmp t1,t2}$
- $\text{jl L}$

Branches in RISC Machines

- $\text{tjump/fjump translate easily into RISC instructions}$
- $\text{MIPS: cmp computes the test, br performs the jump}$

- $\text{cmp t2, t3, t1}$
- $\text{br t1, L}$

Load Effective Address

- lea instruction: computes a memory address
- All forms of indirect memory accesses are supported

- $\text{lea (t1,t2), t3}$

- $\text{lea cl(t1,t2,c2), t3}$

Maximal Munch Algorithm

- Maximal Munch = a greedy algorithm
- Start from top of tree
- Find largest tile that matches top node
- Tile remaining the rest of the structure recursively
Example

\[ x = x + 1; \]

Better Tiling

\[ x = x + 1; \]

Matching Tiles

```java
abstract class IRStmt {
    Assembly munch();
}
class IRAssign extends IRStmt {
    Assembly munch() {
        if (src instanceof IRPlus &&
            ((IRPlus)src).lhs.equals(dat) &&
            !isKeyMem32(dat)) {
            Assembly e = ((IRPlus)src).rhs.munch();
            return e.append(new AddInst(dat, e.target()));
        } else if ... } }
```

Implementation

- Maximal Munch algorithm starts from a root node
- Find largest tile matching root
- Involve recursively on all children of the tile
- Generate code for this tile
- Code for children will have been generated already during the recursive calls

Improving Instruction Selection

- Because it is greedy, Maximal Munch does not necessarily generate the optimal tiling
- Dynamic Programming approach: for every node, find the optimal tiling for that node and the sub-graph rooted at that node
  - Once we have computed the optimal tiling of all nodes in the sub-graph, the best tiling of the node by trying out all possible tiles matching the node
  - Start from leaves, work upward to the root
Recursive Implementation

- Dynamic programming algorithm uses memoization
- For each node, record best tile for node
- Start at the root:
  - First, check the best tile for this node, if available
  - If not computed, try each matching tile to see which one has lowest cost
  - Store the best tile and return this tile
- Finally, use entries in table to emit code

Dynamic Programming

class IRAssign extends IRStmt {
  INExpr src, dst;
  Assembly best = null;
  int optTileCost() {
    if (best != null) return best.cost();
    if (src instanceof INPlus && ((INPlus)src).lha.equals(dst) &&
        isBagHex2(dst)) {
      best = new Add(dst, src, e.target);
    }
    // consider all other tiles */
    return best.cost();
  }
}

Automating Instruction Selection

- Code generator generators
  - Start with a specification for the tiles (with costs)
  - Explicitly create data structures representing each tile
  - Tiling is then performed by a generic tree-matching and code generation procedure
  - For RISC instruction sets, over-engineering

Modern Processors

- Modern processors have various forms of parallelism
  - Execution time not sum of tile times
  - Instruction order matters
    - Processors pipeline instructions and execute different pieces of instructions in parallel
    - Bad ordering (e.g., too many memory operations in sequence) stalls processor pipeline
    - Processor can execute some instructions in parallel (super-scalar)
  - Cost is merely an approximation
  - Instruction scheduling needed