CS42/413
Introduction to Compilers
Radu Rugina

Lecture 31: More Instruction Selection
16 Apr 04

Instruction Selection
1. Translate low-level IR code into DAG representation
2. Then find a good tiling of the DAG
   - disjoint set of tiles that cover the DAG
   - Maximal munch algorithm
   - Dynamic programming algorithm

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/local variables stored in registers
  – Parameters/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 => many possible tiles and tilings
  – Example: multiple addressing modes (CISC) versus load/store architectures (RISC)

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

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:
  mov %ebp, t0
  sub $20, t0
  mov 0(t0), t1
  add t1, t

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:
  mov %ebp, t0
  sub $20, t0
  mov 0(t0), t1
  add t1, t
Tiles

- Tiles capture compiler’s understanding of instruction set
- Each tile: sequence of machine instructions that match a subgraph of the DAG
- May need additional move instructions
- Tiling = cover the DAG with tiles

Some Tiles

- mov t1, t2
- add $1, t2

- mov t2, t1
- mov $10, 0(t1,t2)
- mov t2, t3
- add t1, t3
- mov t1, %eax
- mul t2
- mov %eax, t3

Conditional Branches

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

- test t1, t1
- jnz L

- cmp t1, t2
- je L

Maximal Munch Algorithm

- Maximal Munch = find largest tiles (greedy algorithm)
- Start from top of tree
- Find largest tile that matches top node
- Tile remaining subtrees recursively

DAG Representation

- DAG: a node may have multiple parents
- Algorithm: same, but nodes with multiple parents occur inside tiles only if all parents are in the tile

Example

- x = x + 1;
Example

\[
x = x + 1;
\]

Alternate (CISC) Tiling

\[
x = x + 1;
\]

\[
\text{mov } 8(\%ebp), t1
\]

\[
\text{mov t1, t2}
\]

\[
\text{add } 1, t2
\]

\[
\text{mov t2, 8(\%ebp)}
\]

ADD Expression Tiles

\[
\text{mov } t2, t1
\]

\[
\text{add } r/m32, t1
\]

\[
\text{mov } t2, t1
\]

\[
\text{add } \text{imm}32, t1
\]

ADD Statement Tiles

Intel Architecture

\[
\text{add } \text{imm}32, %eax
\]

\[
\text{add } \text{imm}32, r/m32
\]

\[
\text{add } \text{imm}8, r/m32
\]

\[
\text{add } r32, r/m32
\]

\[
\text{add } r/m32, r32
\]

Designing Tiles

- Only add tiles that are useful to compiler
- Many instructions will be too hard to use effectively or will offer no advantage
- Need tiles for all single-node trees to guarantee that every tree can be tiled, e.g.

\[
\text{mov } t2, t1
\]

\[
\text{add } t3, t1
\]

More Handy Tiles

le a instruction computes a memory address

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

\[
\text{lea c1(t1,t2,c2), t3}
\]
Matching Jump for RISC

- As defined in lecture, have
  \[ \text{tjump}(\text{cond}, \text{destination}) \]
  \[ \text{fjump}(\text{cond}, \text{destination}) \]
- Our tjump/fjump translates easily to RISC ISAs that have explicit comparison result

Fixed-register instructions

```
mul r/m32
   Multiply value in register eax
   Result: low 32 bits in eax, high 32 bits in edx
jcc x L
   Jump to label L if ecx is zero
add r/m32, %eax
   Add to eax
```
- No fixed registers in low IR except frame pointer
- Need extra move instructions

Matching Tiles

```
abstract class LIR_Stmt {
   Assembly munch();
}
class LIR_Assign extends LIR_Stmt {
   LIR_Expr src, dst;
   Assembly munch() {
      if (src instanceof IR_Plus &
          ((IR_Plus)src).lhs.equals(dst) &
          is_regemnt32(dst) {
         Assembly e = ((IR_Plus)src).rhs.munch();
         return e.append(new AddInsn(dst,
                                     e.target()));
      } else if ...
   }
```

Condition Code ISA

- Pentium: condition encoded in jump instruction
- cmp: compare operands and set flags
- jcc: conditional jump according to flags

Implementation

- Maximal Munch: start from top node
- Find largest tile matching top node and all of the children nodes
- Invoke recursively on all children of tile
- Generate code for this tile
- Code for children will have been generated already in recursive calls
- How to find matching tiles?

Tile Specifications

- Previous approach simple, efficient, but hard-codes tiles and their priorities
- Another option: explicitly create data structures representing each tile in instruction set
  - Tiling performed by a generic tree-matching and code generation procedure
  - Can generate from instruction set description: code generator generators
  - For RISC instruction sets, over-engineering
How Good Is It?

- Very rough approximation on modern pipelined architectures: execution time is number of tiles
- Maximal munch finds an optimal but not necessarily optimum tiling
- Metric used: tile size

Improving Instruction Selection

- Because greedy, Maximal Munch does not necessarily generate best code
  - Always selects largest tile, but not necessarily the fastest instruction
  - May pull nodes up into tiles inappropriately – it may be better to leave below (use smaller tiles)
- Can do better using dynamic programming algorithm

Timing Cost Model

- Idea: associate cost with each tile (proportional to number of cycles to execute)
  - may not be a good metric on modern architectures
- Total execution time is sum of costs of all tiles

Finding optimum tiling

- Goal: find minimum total cost tiling of DAG
- Algorithm: for every node, find minimum total cost tiling of that node and sub-graph
- Lemma: once minimum cost tiling of all nodes in subgraph, can find minimum cost tiling of the node by trying out all possible tiles matching the node
- Therefore: start from leaves, work upward to top node

Dynamic Programming: a[i]

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

Recursive Implementation

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

class IR_Move extends IRStmt {
    IR_Expr src, dst;
    Assembly best; // initialized to null
    int optTileCost() {
        if (best != null) return best.cost();
        if (src instanceof IR_Pile &&
            ((IR_Pile)src).equals(dst) && is_regmem32(dst)) {
            int src_cost = ((IR_Pile)src).optTileCost();
            int cost = src_cost + CISC_ADD_COST;
            if (cost < best.cost())
                best = new AddInstr(dst, e.target);
        }...
        return best.cost();
    }
}

Problems with Model

- Modern processors:
  - 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

Summary

- Can specify code generation process as a set of tiles that relate low IR trees (DAGs) to instruction sequences
- Instructions using fixed registers problematic but can be handled using extra temporaries
- Maximal Munch algorithm implemented simply as recursive traversal
- Dynamic programming algorithm generates better code, can be implemented recursively using memoization
- Real optimization will also require instruction scheduling