CS412/413

Introduction to Compilers
Radu Rugina

Lecture 27: More Instruction Selection
03 Apr 02

Outline

• Tiles: review
• Maximal munch algorithm
• Some tricky tiles
  – conditional jumps
  – instructions with fixed registers
• Dynamic programming algorithm

Instruction Selection

• Current step: converting low-level intermediate code into abstract assembly
• Implement each IR instruction with a sequence of one or more assembly instructions
• DAG of IR instructions are broken into tiles associated with one or more assembly instructions

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

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
Another Example

\[ x = x + 1; \]

Example

\[ x = x + 1; \]

\begin{itemize}
  \item mov 8(%ebp), t1
  \item mov t1, t2
  \item add $1, t2
  \item mov t2, 8(%ebp)
\end{itemize}

Alternate (CISC) Tiling

\[ x = x + 1; \]

\begin{itemize}
  \item add $1, 8(%ebp)
\end{itemize}

ADD Expression Tiles

\begin{itemize}
  \item mov t2, t1
  \item add r/m32, t1
\end{itemize}

\begin{itemize}
  \item mov t2, t1
  \item add imm32, t1
\end{itemize}

ADD Statement Tiles

Intel Architecture

\begin{itemize}
  \item add imm32, %eax
  \item add imm32, r/m32
  \item add imm8, r/m32
  \item add r32, r/m32
  \item add r/m32, r32
\end{itemize}

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.

\begin{itemize}
  \item mov t2, t1
  \item add t3, t1
\end{itemize}
**More Handy Tiles**

lea instruction computes a memory address

![Diagram showing lea instruction](image1)

**Matching Jump for RISC**

- As defined in lecture, have
  - `jump(cond, destination)`
  - `jumpf(cond, destination)`

Our `jump/jumpf` translates easily to RISC ISAs that have explicit comparison result

![Diagram showing MIPS jump instructions](image2)

**Condition Code ISA**

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

![Diagram showing condition code jumps](image3)

**Fixed-register instructions**

- `mul r/m32`
  - Multiply value in register eax
  - Result: low 32 bits in eax, high 32 bits in edx

- `jeqz 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

**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?

**Matching Tiles**

abstract class LIRStmt {
    abstract fun munch():
}

class LIRAssign extends LIRStmt {
    fun LIR_AsmSrc dst: AssemblyMunch() {
        if (isinstance(IRPlus) && (IRPlus).isZero(dst)) {
            return dst.append((IRPlus).asZeroMunch())
        } else if ...
    }
}

---

3
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]
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 {
  IRS Ex src, dst;
  Assembly best; // initialized to null
  int optTileCost() {
    if (best != null) return best.cost();
    if (src instanceof IR_Plus &&
        ((IR_Plus)arg1).equals(dst) && is_regmem32(dst)) {
      int src_cost = ((IR_Plus)arg1).optTileCost();
      int cost = src_cost + CISC_ADD_COST;
      if (cost < best.cost())
        best = new AddIns(dst, e.target);
    }
    return best.cost();
  }
}

Problems with Model

- Modern processors:
  - execution time not sum of tile times
  - instruction order matters
    - Processors is pipelining instructions and executing
      different pieces of instructions in parallel
    - bad ordering (e.g. too many memory operations
      in sequence) stalls processor pipeline
    - process 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