Instruction Selection

A.Pnueli

Honors Compilers, NYU, Spring, 2007

Lecture 13: Instruction Selection

Instruction Selection by Tree Matching

- Work on AST or on DAG for basic block
- Use patterns to describe semantics of machine
- Cover free with smallest/cheapest set of patterns
- Cover tree with dynamic programming to find optimal covering
- Use greedy strategy to find good covering, or
- Generalize patterns automatically from machine description
- Express commutativity, and note that an addition can effect the load of a constant
- Asssume frame pointer in register, and $r_0 = 0$ always.

\[ (c + d).N = \\overline{.r} \]

4 sub-trees.

\[ \text{LOAD} \]

\[ c + \overline{.r} = \\overline{.r} \]

\[ \text{ADD} \]

\[ \text{ADDI} \]

\[ \text{LOAD} \]

\[ r_i = M[r_j + c] \]

4 sub-trees.

Algorithm is fast but does not guarantee optimality

Greedy Strategy: The Maximal Munch Approach

- Traverse free top-down and emit instructions (in reverse order)
- Proceed top-down, and try larger patterns first, to minimize total number of instructions.
- Construct free patterns for each machine instruction
- Cover (tile) the AST with matching tree patterns.

- Construct free patterns automatically from machine description
- Use patterns to describe semantics of machine
- Cover free with smallest/cheapest set of patterns
- Cover tree with dynamic programming to find optimal covering
- Use greedy strategy to find good covering, or
Example: Tiling for $a[i] := x$

Naïve Tiling: 10 Instructions

- ADDI $r1 := r0 + a$
- ADD $r1 := fp + r1$
- LOAD $r1 := M[r1 + 0]$
- ADDI $r2 := r1 + r2$
- ADD $r2 := r0 + r2$
- LOAD $r2 := M[r2 + 0]$
- ADD $r1 := r1 + r2$
- ADDI $r2 := r0 + x$
- LOAD $f[fp + x]$
- STORE $M[r1 + 0] := r2$

Better Tiling: Two Memory Instructions

- LOAD $r1 := M[fp + a]$
- ADDI $r2 := r0 + 4$
- MUL $r2 := r1 * r2$
- ADD $r1 := r1 + r2$
- ADDI $r2 := fp + x$
- LOAD $f[fp + x]$
- MOVEM $M[r1 + 0] := M[r2]$

Best Tiling: Two Memory Instructions

- LOAD $r1 := M[fp + a]$
- ADDI $r2 := r0 + 4$
- MUL $r2 := r1 * r2$
- ADD $r1 := r1 + r2$
- ADDI $r2 := r0 + x$
- STORE $M[r1 + 0] := r2$

Honors Compilers, NYU, Spring 2007
A.Pnueli

The Maximal Munch Algorithm

**Example: Matching an Expression Sub-tree**

```
else if (op = "+" && kind(leftOperand) == Constant)
    Munch(leftOperand);
else if (op = "+" && kind(rightOperand) == Constant)
    Munch(rightOperand);
else
    Emit(ADDItarget, leftOperand, rightOperand);
```

---

**Bottom-up Rewriting System (BURS)**

- Load \( R_x \):
  \[ \text{cost}=1 \]
- Load \( R_x \) from memory:
  \[ \text{cost}=3 \]
- \( R_x + \) memory:
  \[ \text{cost}=3 \]
- \( R_x \) from memory:
  \[ \text{cost}=1 \]
- Load-
  \[ \text{cost}=1 \]

---

**All Operations Have Costs**

- \( R_x + R_y \):
  \[ \text{cost}=2 \]
- \( R_x + \) memory:
  \[ \text{cost}=3 \]

---

**Different Procedures for Statements and for Expressions.**

- Tree matches if operator matches and its descendants.
- Order pattern trees by size, largest first.
- Top-down procedure.
Three-Pass Algorithm

- Instruction-collecting: bottom-up tree matching, using cost information.
- Instruction-selection: top-down pass to select single candidates.
- Instruction generation: bottom-up traversal to emit instructions in proper order.
- Interpretation: for node, instruction interpretation.
- Interpretation (like accept state):
  - Item with dot at root represents full match of operand.
  - Extend match by moving dot.
  - Label free nodes with sets of items.
  - Introduce items: patterns with dot, indicating partial match.
- Similar approach to LR parsing.

Bottom-up Pattern Matching

To annotate node N with a set of possible matches:

1. Compute cost of (L1, L2) and add label to N.
2. If matches (P, L1, L2) then:
   - For each label of right operand L2:
     - Match right operand
   - Match left operand

(Labels):

Tree for a

```
{ load-mem R2
  mem ! R1
  load-mem ! R2
  Mult mem R1 R2
  mem !
}
```

Honors Compilers, NYU, Spring, 2007
Lecture 13: Instruction Selection

A. Pnueli

Automating the Process

Describes semantics of instruction set in a machine

Generates tree patterns automatically

Pattern-matching code is machine independent

Generates an equivalent program $P'_0$ such that:

1. $P'_0$ is of no higher cost than $P$.
2. $P'_0$ uses no more registers than $P$.
3. $P'_0$ evaluates $T_p$ contiguously.

For the machine model considered here, we can show that we can find an equivalent program $P'$, evaluating the expression $T'$, such that $P$ evaluates $T$ contiguously by a contiguous program.

This implies that every expression tree can be evaluated optimally by a contiguous program.

Example: Some Patterns for LCC on X86

Add %c, %d 
1 // cost 1

Move %c, %d

Reg: ADDI (reg, mrc1) (pattern on X86)

Describe addition with one operand in memory

mrcl: rc %0 
1 // load, cost 1

mrcl: mem %0 
1 // load, cost 1

Describe a memory location:

Describe some patterns for LCC on X86

Honors Compilers, NYU, Spring, 2007
The Dynamic Programming Algorithm

The algorithm proceeds in three phases (assuming \( r \) available registers):

1. **Compute** bottom-up for each node \( n \) in \( T \): an array \( C \) of costs, in which \( C[i] \) is the optimal cost of computing the subtree \( S \) rooted at \( n \) into a register, assuming \( i \) available registers.
2. **Traverse** \( T \), using the cost vectors to determine which available registers \( S \) subtrees must be computed into memory.
3. **Compute** bottom-up for each node \( n \) in \( T \): an array \( C \) of costs (assuming \( r \) available registers). The algorithm can be evaluated optimally by the following program.

### Example: An Expression Tree

Consider the following expression tree:

```
(3) (2) +
(0) (1) +
(3) (2) (5) (4)
(0) (1) a
(0) (1) b
(0) (1) d
(0) (1) e
```

Using 2 registers, this expression can be evaluated by a program of 7 instructions:

```
LDR0, c // R0 = c
LDR1, d // R1 = d
DIVR1, R1, e // R1 = R1/e
MULR0, R0, R1 // R0 = R0 * R1
LDR1, a // R1 = a
SUBR1, R1, b // R1 = R1 - b
ADDR1, R1, R0 // R1 = R1 + R0
```

### References

- Fraser & Hanson: A Retargetable C Compiler
- Maximal Munch: Honors Compilers, NYU, Spring, 2007
- BURS: Brune, Bal, et al: Modern Compiler Design
- Modern Compiler Implementation in X (for various values of X)
- Andrew Appel: Modern Compiler Implementation in LCC
- Proebsting: TOPLAS 17(3) May 1995
- Original paper: Fraser & Hanson: A Retargetable C Compiler