CSE 401 – Compilers

Two Cool Algorithms: Instruction Selection and Register Allocation

Hal Perkins
Winter 2009
Agenda

- We’ve seen how minijava handles code gen
- This lecture
  - Instruction selection by tree pattern matching
  - Register allocation by graph coloring
A Simple Low-Level IR (1)

- We want a low-level similar to Minijava’s IL. But much simpler here for the examples.

- Expressions:
  - CONST(i) – integer constant i
  - TEMP(t) – temporary t (i.e., register)
  - BINOP(op,e₁,e₂) – application of op to e₁,e₂
  - MEM(e) – contents of memory at address e
    - Means value when used in an expression
    - Means address when used on left side of assignment
  - CALL(f,args) – application of function f to argument list args
Simple Low-Level IR (2)

- **Statements**
  - MOVE(TEMP t, e) – evaluate e and store in temporary t
  - MOVE(MEM(e1), e2) – evaluate e1 to yield address a; evaluate e2 and store at a
  - EXP(e) – evaluate expressions e and discard result
  - SEQ(s1, s2) – execute s1 followed by s2
  - NAME(n) – assembly language label n
  - JUMP(e) – jump to e, which can be a NAME label, or more complex (e.g., switch)
  - CJUMP(op, e1, e2, t, f) – evaluate e1 op e2; if true jump to label t, otherwise jump to f
  - LABEL(n) – defines location of label n in the code
For a local variable at a known offset $k$ from the frame pointer $fp$

- Linear
  \[
  \text{MEM(BINOP(PLUS, TEMP $fp$, CONST $k$))}
  \]

- Tree
  \[
  \begin{aligned}
  &\text{MEM} \\
  &\downarrow \\
  &+ \\
  &\text{TEMP $fp$} \quad \text{CONST $k$}
  \end{aligned}
  \]
For an array element $e[k]$, where each element takes up $w$ storage locations.
Instruction Selection Issues

- Given the low-level IR, there are many possible code sequences that implement it correctly
  - e.g. to set eax to 0 on x86
    - `mov eax,0`  `xor eax,eax`
    - `sub eax,eax`  `imul eax,0`
  - Many machine instructions do several things at once – e.g., register arithmetic and effective address calculation
Implementation

- Problem: We need some representation of the target machine instruction set that facilitates code generation
- Idea: Describe machine instructions using same low-level IR used for program
- Use pattern matching techniques to pick machine instructions that match fragments of the program IR tree
  - Want this to run quickly
  - Would like to automate as much as possible
Matching: How?

- **Tree IR** – pattern match on trees
  - Tree patterns as input
  - Each pattern maps to target machine instruction (or sequence)
  - Use dynamic programming or bottom-up rewrite system (BURS)

- **Linear IR** – some sort of string matching
  - Strings as input
  - Each string maps to target machine instruction sequence
  - Use text matching or peephole matching

- Both work well in practice; actual algorithms are quite different
An Example Target Machine (1)

- Arithmetic Instructions
  - (unnamed) ri
  - ADD ri <- rj + rk
  - MUL ri <- rj * rk
  - SUB and DIV are similar
An Example Target Machine (2)

- Immediate Instructions

  - ADDI \( r_i \leftarrow r_j + c \)
    
    \[
    \begin{array}{c}
    + \\
    + \\
    \end{array}
    \]
    
    \[
    \begin{array}{c}
    \text{CONST} \\
    \text{CONST} \\
    \text{CONST}
    \end{array}
    \]

  - SUBI \( r_i \leftarrow r_j - c \)
    
    \[
    -
    \]
    
    \[
    \text{CONST}
    \]
An Example Target Machine (3)

- **Load**
  - \( \text{LOAD } ri \leftarrow M[rj + c] \)

```
MEM
  |  MEM
  +  |
  +  |
CONST  CONST
```

`N-12`
An Example Target Machine (4)

- Store
  - STORE $M[r_j + c] <- r_i$

```
        MOVE
        /   \
       /     \
MEM     MEM
|      /  |
|     +   |
|       |
+-------

CONST
```
Tree Pattern Matching (1)

- **Goal:** Tile the low-level tree with operation (instruction) trees

- A *tiling* is a collection of `<node, op>` pairs
  - *node* is a node in the tree
  - *op* is an operation tree
  - `<node, op>` means that *op* could implement the subtree at *node*
Tree Pattern Matching (2)

- A tiling “implements” a tree if it covers every node in the tree and the overlap between any two tiles (trees) is limited to a single node.
  - If <node,op> is in the tiling, then node is also covered by a leaf in another operation tree in the tiling – unless it is the root.
  - Where two operation trees meet, they must be compatible (i.e., expect the same value in the same location).
Generating Code

- Two ways to get good tilings
  - Maximal munch: walk the tree top-down. At each node find the largest node that fits (covers the largest subtree at that point).
  - Dynamic programming:
    - Assign a cost to each node in the tree = \( \Sigma \) cost of that node + subtrees
    - Try all possible combinations bottom-up and pick minimal cost at each subtree
Example

- Codegen for $a[i] = x$, where $i$ is a register variable, and $a$ and $x$ are memory resident
Register Allocation by Graph Coloring

- How to convert the infinite sequence of temporary data references, t1, t2, ... into finite assignment register numbers $8, 9, ..., 25$
- Goal: Use available registers with minimum spilling
- Problem: Minimizing the number of registers is NP-complete ... it is equivalent to chromatic number--minimum colors to color nodes of graph so no edge connects same color
Begin With Data Flow Graph

- procedure-wide register allocation
- only *live* variables require register storage

**dataflow analysis**: a variable is *live* at node N if the *value* it holds is used on some path further down the control-flow graph; otherwise it is *dead*

- two variables(values) interfere when their live ranges overlap
a := read();
b := read();
c := read();
d := a + b*c;
if (d < 10 ) then
    e := c+8;
    print(c);
else
    f := 10;
    e := f + d;
    print(f);
fi
print(e);
a := read();
b := read();
c := read();
d := a + b*c;
d < 10

e := c+8;
print(c);
f := 10;
e := f + d;
print(f);
print(e);
Graph Coloring

- NP complete problem
- Heuristic: color easy nodes last
  - find node $N$ with lowest degree
  - remove $N$ from the graph
  - color the simplified graph
  - set color of $N$ to the first color that is not used by any of $N$’s neighbors
- Basics due to Chaitin (1982)
Apply Heuristic
Apply Heuristic

Diagram:

- a connected to b
- c connected to d
- e connected to f

Network configuration:
- Three configurations are shown, each with different connections between the nodes.
Apply Heuristic

Diagram showing the application of a heuristic process with nodes labeled a, b, c, d, e, and f.
Continued

a → b

b → a

c → d

d → c

e → f

f → e
Continued
Continued
Continued

Diagram showing the relationship between nodes a, b, c, d, e, and f. The diagram is split into two parts, with nodes a, b, c, d, e, and f connected in a specific pattern. The transformation from one part to the other is indicated by an arrow.
Continued
Continued
a := read();
b := read();
c := read();
d := a + b*c;
if (d < 10 ) then
  e := c+8;
  print(c);
  else
  f := 10;
  e := f + d;
  print(f);
fi
print(e);
Some Graph Coloring Issues

- May run out of registers
  - Solution: insert spill code and reallocate
- Special-purpose and dedicated registers
  - Examples: function return register, function argument registers, registers required for particular instructions
  - Solution: “pre-color” some nodes to force allocation to a particular register
Exercise

{ int tmp_2ab = 2*a*b;
  int tmp_aa = a*a;
  int tmp_bb = b*b;
  x := tmp_aa + tmp_2ab + tmp_bb;
  y := tmp_aa - tmp_2ab + tmp_bb;
}

given that a and b are live on entry and dead on exit, and that x and y are live on exit:

(a) construct the register interference graph
(b) color the graph; how many registers are needed?
4 Registers Needed