

# CSE 582 – Compilers

## Instruction Selection

Hal Perkins  
Autumn 2002

11/19/2002 © 2002 Hal Perkins & UW CSE N-1

## Agenda

- n Compiler back-end organization
- n Low-level intermediate representations
  - n Trees
  - n Linear
- n Instruction selection algorithms
  - n Tree pattern matching
  - n Peephole matching
- n Credits: Much of this material is adapted from slides by Keith Cooper (Rice) and material in Appel's *Modern Compiler Implementation in Java*

11/19/2002 © 2002 Hal Perkins & UW CSE N-2

## Compiler Organization

```

graph LR
    subgraph Front_End [front end]
        scan[scan]
        parse[parse]
        semantics[semantics]
    end
    subgraph Middle [middle]
        opt1[opt1]
        opt2[opt2]
        optn[optn]
    end
    subgraph Back_End [back end]
        instr_select[instr. select]
        instr_sched[instr. sched]
        reg_alloc[reg. alloc]
    end
    Front_End --> Middle
    Middle --> Back_End
    infra[infrastructure - symbol tables, trees, graphs, etc.]
  
```

11/19/2002 © 2002 Hal Perkins & UW CSE N-3

## Big Picture

- n Compiler consists of lots of fast stuff followed by hard problems
  - n Scanner:  $O(n)$
  - n Parser:  $O(n)$
  - n Analysis & Optimization:  $\sim O(n \log n)$
  - n Instruction selection: fast or NP-Complete
  - n Instruction scheduling: NP-Complete
  - n Register allocation: NP-Complete

11/19/2002 © 2002 Hal Perkins & UW CSE N-4

## Intermediate Representations

- n Tree or linear?
- n Closer to source language or machine?
  - n Source language: more context for high-level optimizations
  - n Machine: exposes opportunities for low-level optimizations and easier to map to actual code
- n Common strategy
  - n Initial IR is AST, close to source
  - n After some optimizations, transform to lower-level IR, either tree or linear; use this to optimize further and generate code

11/19/2002 © 2002 Hal Perkins & UW CSE N-5

## IR for Code Generation

- n Assume a low-level RISC-like IR
  - n 3 address, register-register instructions + load/store
    - n  $r1 \leftarrow r2 \text{ op } r3$
  - n Could be tree structure or linear
  - n Expose as much detail as possible
- n Assume "enough" registers
  - n Invent new temporaries for intermediate results
  - n Map to actual registers later

11/19/2002 © 2002 Hal Perkins & UW CSE N-6

## Overview

### Instruction Selection

- Map IR into assembly code
- Assume known storage layout and code shape
  - i.e., the optimization phases have already done their thing
- Combine low-level IR operations into machine instructions (addressing modes, etc.)

11/19/2002

© 2002 Hal Perkins & UW CSE

N-7

## Overview

### Instruction Scheduling

- Reorder operations to hide latencies – processor function units; memory/cache
  - Originally invented for supercomputers (1960s)
  - Now important for consumer machines
- Assume fixed program

11/19/2002

© 2002 Hal Perkins & UW CSE

N-8

## Overview

### Register Allocation

- Map values to actual registers
  - Previous phases change need for registers
- Add code to spill values to temporaries as needed, etc.

11/19/2002

© 2002 Hal Perkins & UW CSE

N-9

## How Hard?

- Instruction selection
  - Can make locally optimal choices
  - Global is undoubtedly NP-Complete
- Instruction scheduling
  - Single basic block – quick heuristics
  - General problem – NP Complete
- Register allocation
  - Single basic block, no spilling, interchangeable registers – linear
  - General – NP Complete

11/19/2002

© 2002 Hal Perkins & UW CSE

N-10

## Conventional Wisdom

- We probably lose little by solving these independently
- Instruction selection
  - Use some form of pattern matching
  - Assume “enough” registers
- Instruction scheduling
  - Within a block, list scheduling is close to optimal
  - Across blocks: build framework to apply list scheduling
- Register allocation
  - Start with virtual registers and map “enough” to K
  - Targeting, use good priority heuristic

11/19/2002

© 2002 Hal Perkins & UW CSE

N-11

## An Simple Low-Level IR (1)

Source: Appel, Modern Compiler Implementation

- Details not important for our purposes; point is to get a feeling for the level of detail involved
- 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 right side of assignment
  - CALL(f,args) – application of function f to argument list args
  - NAME(n) – assembly language label n

11/19/2002

© 2002 Hal Perkins & UW CSE

N-12

## Simple Low-Level IR (2)

- Statements
  - MOVE(TEMP  $t$ ,  $e$ ) – evaluate  $e$  and store in temporary  $t$
  - MOVE(MEM( $e_1$ ),  $e_2$ ) – evaluate  $e_1$  to yield address  $a$ ; evaluate  $e_2$  and store at  $a$
  - EXP( $e$ ) – evaluate expressions  $e$  and discard result
  - JUMP( $e$ ) – jump to  $e$ , which can be a NAME label, or more complex (e.g., switch)
  - CJUMP( $op, e_1, e_2, t, f$ ) – evaluate  $e_1$  op  $e_2$ ; if true jump to label  $t$ , otherwise jump to  $f$
  - SEQ( $s_1, s_2$ ) – execute  $s_1$  followed by  $s_2$
  - LABEL( $n$ ) – defines location of label  $n$  in the code

11/19/2002

© 2002 Hal Perkins & UW CSE

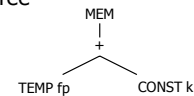
N-13

## Low-Level IR Example (1)

- For a local variable at a known offset  $k$  from the frame pointer  $fp$ 
  - Linear

MEM(BINOP(PLUS, TEMP  $fp$ , CONST  $k$ ))

- Tree



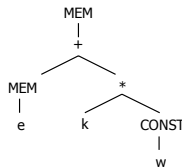
11/19/2002

© 2002 Hal Perkins & UW CSE

N-14

## Low-Level IR Example (2)

- For an array element  $e(k)$ , where each element takes up  $w$  storage locations



11/19/2002

© 2002 Hal Perkins & UW CSE

N-15

## Generating Low-Level IR

- Assuming initial IR is an AST, a simple treewalk can be used to generate the low-level IR
  - Can be done before, during, or after optimizations in the middle part of the compiler
- Create registers (temporaries) for values and intermediate results
  - Value can be safely allocated in a register when only 1 name can reference it
    - Trouble: pointers, arrays, reference parameters
  - Assign a virtual register to anything that can go into one
  - Generate loads/stores for other values

11/19/2002

© 2002 Hal Perkins & UW CSE

N-16

## 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

11/19/2002

© 2002 Hal Perkins & UW CSE

N-17

## Instruction Selection Criteria

- Several possibilities
  - Fastest
  - Smallest
  - Minimize power consumption
- Sometimes not obvious
  - e.g., if one of the function units in the processor is idle and we can select an instruction that uses that unit, it effectively executes for free, even if that instruction wouldn't be chosen normally

11/19/2002

© 2002 Hal Perkins & UW CSE

N-18

## Implementation

- Problem: We need some representation of the target machine instruction set that facilitates code generation
- Idea: Describe machine instructions in some 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

11/19/2002 © 2002 Hal Perkins & UW CSE N-19

## 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

11/19/2002 © 2002 Hal Perkins & UW CSE N-20

## An Example Target Machine (1)

Also from Appel

- Arithmetic Instructions
  - (unnamed) `ri`      TEMP
  - ADD `ri <- rj + rk`
  - MUL `ri <- rj * rk`
  - SUB and DIV are similar

11/19/2002 © 2002 Hal Perkins & UW CSE N-21

## An Example Target Machine (2)

- Immediate Instructions
  - ADDI `ri <- rj + c`
  - SUBI `ri <- rj - c`

11/19/2002 © 2002 Hal Perkins & UW CSE N-22

## An Example Target Machine (3)

- Load
  - LOAD `ri <- M[rj + c]`

11/19/2002 © 2002 Hal Perkins & UW CSE N-23

## An Example Target Machine (4)

- Store
  - STORE `M[rj + c] <- ri`

11/19/2002 © 2002 Hal Perkins & UW CSE N-24

## Tree Pattern Matching (1)

- Goal: Tile the low-level tree with operation trees
- A *tiling* is a collection of  $\langle \text{node}, \text{op} \rangle$  pairs
  - node is a node in the tree
  - op is an operation tree
  - $\langle \text{node}, \text{op} \rangle$  means that op could implement the subtree at node

11/19/2002

© 2002 Hal Perkins & UW CSE

N-25

## 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  $\langle \text{node}, \text{op} \rangle$  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)

11/19/2002

© 2002 Hal Perkins & UW CSE

N-26

## Example

- A tiling for  $a[i] = x;$

11/19/2002

© 2002 Hal Perkins & UW CSE

N-27

## Generating Code

- Given a tiled tree, to generate code
  - Postorder treewalk; node-dependant order for children
  - Emit code sequences corresponding to tiles in order
  - Connect tiles by using same register name to tie boundaries together

11/19/2002

© 2002 Hal Perkins & UW CSE

N-28

## Example

- Generating code for  $a[i] = x;$

11/19/2002

© 2002 Hal Perkins & UW CSE

N-29

## Tiling Algorithm

- There may be many tiles that could match at a particular node
- Idea: Walk the tree and accumulate the set of all possible tiles that could match at that point –  $\text{Tiles}(n)$ 
  - Later: can keep lowest cost match at each point
  - Generates local optimality – lowest cost match at each point

11/19/2002

© 2002 Hal Perkins & UW CSE

N-30

## Tile(Node n)

```

Tiles(n) <- empty;
if n has two children then
  Tile(left child of n)
  Tile(right child of n)
  for each rule r that implements n
    if (left(r) is in Tiles(left(n)) and right(r) is in Tiles(right(n)))
      Tiles(n) <- Tiles(n) + r
else if n has one child then
  Tile(child of n)
  for each rule r that implements n
    if (left(r) is in Tiles(child(n)))
      Tiles(n) <- Tiles(n) + r
else /* n is a leaf */
  Tiles(n) <- { all rules that implement n }

```

11/19/2002 © 2002 Hal Perkins & UW CSE N-31

## Peephole Matching

- n A code generator/improvement strategy for linear representations
- n Basic idea
  - n Look at small sequences of adjacent operations
  - n Compiler moves a sliding window ("peephole") over the code and looks for improvements

11/19/2002 © 2002 Hal Perkins & UW CSE N-32

## Peephole Optimizations (1)

- n Classic example: store followed by a load, or push followed by a pop

original	improved
mov [ebp-8],eax	mov [ebp-8],eax
mov eax,[ebp-8]	
push eax	---
pop eax	

11/19/2002 © 2002 Hal Perkins & UW CSE N-33

## Peephole Optimizations (2)

- n Simple algebraic identities

original	improved
add eax,0	---
add eax,1	inc eax
mul eax,2	add eax,eax

11/19/2002 © 2002 Hal Perkins & UW CSE N-34

## Peephole Optimizations (3)

- n Jump to a Jump

original	improved
jmp here	jmp there
here: jmp there	

11/19/2002 © 2002 Hal Perkins & UW CSE N-35

## Implementing Peephole Matching

- n Early versions
  - n Limited set of hand-coded pattern
  - n Modest window size to ensure speed
- n Modern
  - n Break problem in to expander, simplifier, matcher
  - n Apply symbolic interpretation and simplification systematically

11/19/2002 © 2002 Hal Perkins & UW CSE N-36

## Expander

- Turn IR code into very low-level IR (LLIR)
- Template-driven rewriting
- LLIR includes all direct effects of instructions, e.g., setting condition codes
- Big, although constant size expansion

11/19/2002

© 2002 Hal Perkins & UW CSE

N-37

## Simplifier

- Look at LLIR through window and rewrite using
  - Forward substitution
  - Algebraic simplification
  - Local constant propagation
  - Eliminate dead code
- This is the heart of the processing

11/19/2002

© 2002 Hal Perkins & UW CSE

N-38

## Matcher

- Compare simplified LLIR against library of patterns
- Pick low-cost pattern that captures effects
- Must preserve LLIR effects, can add new ones (condition codes, etc.)
- Generates assembly code output

11/19/2002

© 2002 Hal Perkins & UW CSE

N-39

## Peephole Optimization Considered

- LLIR is largely machine independent (RTL)
- Target machine description is LLIR -> ASM patterns
- Pattern matching
  - Use hand-coded matcher (gcc)
  - Turn patterns into grammar and use LR parser
- Used in several important compilers
- Seems to produce good portable instruction selectors

11/19/2002

© 2002 Hal Perkins & UW CSE

N-40

## Coming Attractions

- Instruction Scheduling
- Register Allocation
- Survey of Optimization
- Survey of "new" technologies
  - Memory management & garbage collection
  - Virtual machines, portability, and security

11/19/2002

© 2002 Hal Perkins & UW CSE

N-41