



COMP 181

Lecture 17


Instruction selection

November 21, 2005

MICRO report



- Major themes
 - How to utilize more transistors
 - Frequency "wall" – problem is energy
 - Speculation
 - "Helper" threads, run-ahead execution
 - Branch prediction
 - Out-of-order execution
 - *Keynote* – Norm Jouppi (HP Labs)
 - Processors need to get simpler
 - New themes
 - Manageability
 - Availability
 - Security
 - Languages – Java, C#



2


Prelude

- Bose "noise cancelling" headphones
 - How do they work?
 - Measure noise, generate a wave with same frequency, but inverted amplitude
 - Two waves cancel out
- We may already have this ability
 - *Obvious*: nerves go from ear to brain
 - *Not as obvious*: nerves go from brain to ear
 - Actively change frequency response of the cochlea
- I'm on a grant to help study this phenomenon

3


Back end



4

Back end

- Essential tasks
 - Instruction selection
 - Map low-level IR to actual machine instructions
 - Not necessarily 1-1 mapping
 - CISC architectures, addressing modes
 - Register allocation
 - Low-level IR assumes unlimited registers
 - Map to actual resources of machines
 - Goal: maximize use of registers
- Optimizations
 - General optimizations (from last time)
 - Instruction scheduling
 - "Peep-hole" optimizations




5

Instruction Selection

- Low-level IR different from machine ISA
 - Why?
 - Allow different back ends
 - Abstraction – to make optimization easier
- Differences between IR and ISA
 - IR: simple, uniform set of operations
 - ISA: many specialized instructions
- Often a single instruction does work of several operations in the IR

Instruction Set Architecture



6

Instruction Selection

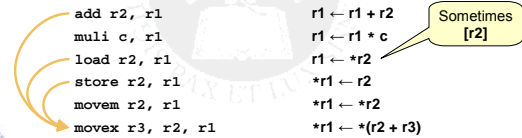
- Easy solution
 - Map each IR operation to a single instruction
 - May need to include memory operations



- **Problem:** inefficient use of ISA

Instruction Selection

- Instruction sets
 - ISA often has many ways to do the same thing
 - **Idiom:**
A single instruction that represents a common pattern or sequence of operations
- Consider a machine with the following instructions:



Example

- Generate code for:

```
a[i+1] = b[j]
```

- Simplifying assumptions
 - All variables are globals
(No stack offset computation)
 - All variables are in registers
(Ignore load/store of variables)

```

IR
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
    
```

Possible Translation

	IR	Assembly
• Address of b[j]:	t1 = j*4 t2 = b+t1	muli 4, rj add rj, rb
• Load value b[j]:	t3 = *t2	load rb, r1
• Address of a[i+1]:	t4 = i+1 t5 = t4*4 t6 = a+t5	addi 1, ri muli 4, ri add ri, ra
• Store into a[i+1]:	*t6 = t3	store r1, ra

Another Translation

- Address of b[j]:
- (no load)
- Address of a[i+1]:
- Store into a[i+1]:

IR	Assembly
t1 = j*4 t2 = b+t1 t3 = *t2	muli 4, rj add rj, rb
t4 = i+1 t5 = t4*4 t6 = a+t5 *t6 = t3	addi 1, ri muli 4, ri add ri, ra movem rb, ra

Direct memory-to-memory operation

Yet Another Translation

- Index of b[j]:
- (no load)
- Address of a[i+1]:
- Store into a[i+1]:

IR	Assembly
t1 = j*4 t2 = b+t1 t3 = *t2	muli 4, rj
t4 = i+1 t5 = t4*4 t6 = a+t5 *t6 = t3	addi 1, ri muli 4, ri add ri, ra movex rj,rb,ra

Compute the address of b[j] in the memory move operation

```
movex rj, rb, ra  *ra ← *(rj + rb)
```

Different translations

- Why is last translation preferable?
 - Fewer instructions
 - Instructions have different costs
 - Space cost: size of each instruction
 - Time cost: number of cycles to complete

- Example

Idioms are cheaper than constituent parts

```

add r2, r1      cost = 1 cycle
multi c, r1    cost = 10 cycles
load r2, r1    cost = 3 cycles
store r2, r1   cost = 3 cycles
movem r2, r1   cost = 4 cycles
movex r3, r2, r1 cost = 5 cycles
    
```



Minimizing cost

- Goal:
 - Find instructions with low overall cost
- Difficulty
 - How to find these patterns?
 - Machine idioms may subsume IR operations that are not adjacent
- Idea: back to tree representation
 - Convert computation into a tree
 - Match parts of the tree

IR

```

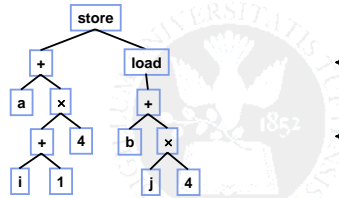
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t4
    
```

movem rb, ra



Tree Representation

- Build a tree: `a[i+1] = b[j]`



IR

```

t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
    
```

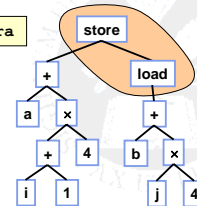
- Goal: find parts of the tree that correspond to machine instructions



Tiles

- Idea: a *tile* is contiguous piece of the tree that corresponds to a machine instruction

movem rb, ra



IR

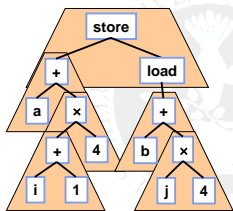
```

t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
    
```



Tiling

- Tiling: cover the tree with tiles



Assembly

```

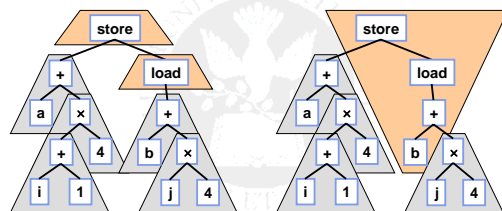
multi 4, rj
addi rj, rb
addi 1, ri
multi 4, ri
addi ri, ra
movem rb, ra
    
```



Tiling

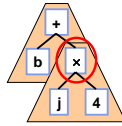
load rb, r1
store r1, ra

movex rj, rb, ra



Generating code

- Given a tiling of a tree
 - A tiling *implements* a tree if:
 - It covers all nodes in the tree
 - The overlap between tiles is exactly one node
- Post-order tree walk
 - Emit machine instructions for each tile
 - Tie boundaries together with registers
 - Note: order of children matters



Tiling

- What's hard about this?
 - Define system of tiles in the compiler
 - Finding a tiling that implements the tree
(Covers all nodes in the tree)
 - Finding a "good" tiling
- Different approaches
 - Ad-hoc pattern matching
 - Automated tools



Algorithms

- Goal: find a tiling with the fewest tiles
- Ad-hoc top-down algorithm
 - Start at top of the tree
 - Find largest tile matches top node
 - Tile remaining subtrees recursively

```
Tile(n) {
  if ((op(n) == PLUS) &&
      (left(n).isNum()))
  {
    Code c = Tile(right(n));
    c.append(ADDI left(n) right(n))
  }
}
```



Ad-hoc algorithm

- Problem: what does tile size mean?
 - Not necessarily the best fastest code
(Example: multiply vs add)
 - How to include cost?
- Idea:
 - Total cost of a tiling is sum of costs of each tile
- Goal: find a minimum cost tiling



Including cost

- Algorithm:
 - For each node, find minimum total cost tiling for that node and the subtrees below
- Key:
 - Once we have a minimum cost for subtree, can find minimum cost tiling for a node by trying out all possible tiles matching the node
- Use dynamic programming



Dynamic programming

- Idea
 - For problems with *optimal substructure*
 - Compute optimal solutions to sub-problems
 - Combine into an optimal overall solution
- How does this help?
 - Use *memoization*:
Save previously computed solutions to sub-problems
 - Sub-problems recur many times
 - Can work top-down or bottom-up

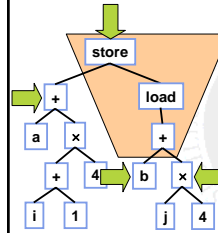


Recursive algorithm

- Memoization
 - For each subtree, record best tiling in a table
 - (Note: need a quick way to find out if we've seen a subtree before – some systems use DAGs instead of trees)
- At each node
 - First check table for optimal tiling for this node
 - If none, try all possible tiles, remember lowest cost
 - Record lowest cost tile in table
 - Greedy, top-down algorithm
- We can emit code from table



Pseudocode



```

Tile(n) {
  if (best(n)) return best(n)
  // -- Check all tiles
  if ((op(n) == STORE) &&
      (op(right(n)) == LOAD) &&
      (op(child(right(n))) == PLUS)) {
    Code c = Tile(left(n))
    c.add(Tile(left(child(right(n))))
    c.add(Tile(right(child(right(n))))
    c.append(MOVEX . . .)
    if (cost(c) < cost(best(n)))
      best(n) = c
  }
  // . . . and all other tiles . . .
  return best(n)
}
    
```



Ad-hoc algorithm

- Problem:
 - Hard-codes the tiles in the code generator
- Alternative:
 - Define tiles in a separate specification
 - Use a generic tree pattern matching algorithm to compute tiling
 - Tools: *code generator generators*
 - Probably overkill for RISC



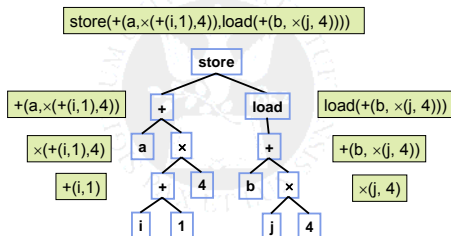
Code generator generators

- Tree description language
 - Represent IR tree as text
- Specification
 - IR tree patterns
 - Code generation actions
- Generator
 - Takes the specification
 - Produces a code generator



Tree notation

- Use prefix notation to avoid confusion



Rewrite rules

- Rule
 - Pattern to match and replacement
 - Cost
 - Code generation template
 - May include actions – e.g., generate register name

Pattern, replacement	Cost	Template
$+(reg_1, reg_2) \rightarrow reg_2$	1	add r1, r2
$store(reg_1, load(reg_2)) \rightarrow done$	5	movem r2, r1



Rewrite rules

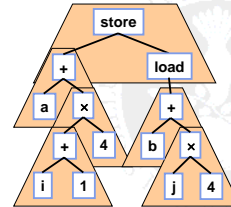
- Example rules:

#	Pattern, replacement	Cost	Template
1	$+(reg_1, reg_2) \rightarrow reg_2$	1	add r1, r2
2	$\times(reg_1, reg_2) \rightarrow reg_2$	10	mul r1, r2
3	$+(num, reg_1) \rightarrow reg_2$	1	addi num, r1
4	$\times(num, reg_1) \rightarrow reg_2$	10	muli num, r1
5	$store(reg_1, load(reg_2)) \rightarrow done$	5	movem r2, r1

- What kinds of optimizations can we do?
 - Strength reduction: multiply to shift or add



Example



Assembly

```
muli 4, rj
add rj, rb
addi 1, ri
muli 4, ri
add ri, ra
movem rb, ra
```



Rewriting process

4	store(+ (ra, x(+ (ri, 1), 4)), load(+ (rb, x(rj, 4))))	
1	store(+ (ra, x(+ (ri, 1), 4)), load(+ (rb, rj)))	muli 4, rj
3	store(+ (ra, x(+ (ri, 1), 4)), load(rb))	add rj, rb
4	store(+ (ra, x(ri, 4)), load(rb))	addi 1, ri
1	store(+ (ra, ri), load(rb))	muli 4, ri
5	store(ra, load(rb))	add ri, ra
	done	movem rb, ra



Implementation

- What does this remind you of?
- Similar to parsing
 - Implement as an automaton
 - Use cost to choose from competing productions
- Provides linear time optimal code generation
 - BURS (bottom-up rewrite system)
 - burg, Twig, BEG



Summary

Ad-hoc pattern matchers	Probably reasonable for RISC machines
Encode matching as automaton	Fast, optimal code generation – requires separate tool
Use parsers	Can lead to highly ambiguous grammars



Modern processors

- Execution time not sum of tile times
- Instruction order matters
 - Pipelining: parts of different instructions overlap
 - Bad ordering stalls the pipeline – e.g., too many operations of one type
 - Superscalar: some operations executed in parallel
- Cost is an approximation
- Instruction scheduling helps



Next time...

- Happy Thanksgiving!
- Monday: register allocation
- Only four more lectures after this one



Top-down algorithm

```
Tile(n)
Label(n) ← ∅
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) ∈ Label(left(n)) and
       right(r) ∈ Label(right(n)))
      then Label(n) ← Label(n) ∪ {r}
else if n has one child
  Tile(child of n)
  for each rule r that implements n
    if (left(r) ∈ Label(child(n)))
      then Label(n) ← Label(n) ∪ {r}
else /* n is a leaf */
  Label(n) ← {all rules that implement n}
```

Match binary nodes against binary rules

Match unary nodes against unary rules



```
Tile(n)
Label(n) ← ∅
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) ∈ Label(left(n)) and
       right(r) ∈ Label(right(n)))
      then Label(n) ← Label(n) ∪ {r}
else if n has one child
  Tile(child of n)
  for each rule r that implements n
    if (left(r) ∈ Label(child(n)))
      then Label(n) ← Label(n) ∪ {r}
else /* n is a leaf */
  Label(n) ← {all rules that implement n}
```

This algorithm

- Finds all matches in rule set
- Labels node n with that set
- Can keep lowest cost match at each point
- Leads to a notion of local optimality — lowest cost at each point
- Spends its time in the two matching loops



Homework

- Due on Monday
- Define code generation schemes for the project:
 - Procedure call
 - Evaluate and assign parameters
 - Retrieve the result
 - Case (switch) statement
 - Use form "if (val == caseval) jump to case code"
 - Use two passes
 - For loop
 - Note differences between Pascal and C
 - Handle both "to" and "downto" loops

Use generate(), emit(), new_temp(), and new_label() from Lectures 14, 15.

