编译原理/Compiler
Instruction Selection

周雅倩
复旦大学计算机科学技术学院
zhouyaqian@fudan.edu.cn
2018/12/13

References

- http://www.stanford.edu/class/cs143/
  - Partial contents are copied from the slides of Alex Aiken

Backend Optimizations

- **Instruction selection**
  - Translate low-level IR to assembly instructions
  - A machine instruction may model multiple IR instructions
  - Especially applicable to CISC architectures
- **Register Allocation(chapter 11)**
  - Place variables into registers
  - Avoid spilling variables on stack

Outline

- Tree Patterns
- Algorithms For Instruction Selection
- Low-level IR to IR Tree/DAG

TREE PATTERNS

Tree patterns

- **Each IR tree node expresses one operation.**
  - Most real architectures can perform several in one instruction as at right.
  - Instruction selection - determining machine instructions to implement an IR tree
  - Tree pattern - the IR tree fragment that corresponds to a valid instruction
- **Tiling - instruction selection to cover IR tree with minimal tree patterns**

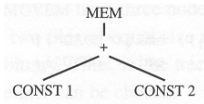
Jouette Architecture

Name	Effect	Trees
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	+
MUL	$r_i \leftarrow r_j \times r_k$	*
SUB	$r_i \leftarrow r_j - r_k$	-
DIV	$r_i \leftarrow r_j / r_k$	/
ADDI	$r_i \leftarrow r_j + c$	+ CONST
SUBI	$r_i \leftarrow r_j - c$	- CONST
LOAD	$r_i \leftarrow M[r_j + c]$	MEM + CONST
STORE	$M[r_j + c] \leftarrow r_i$	MOVE MEM + CONST CONST
MOVEM	$M[r_j] \leftarrow M[r_i]$	MOVE MEM MEM

FIGURE 9.1. Arithmetic and memory instructions. The notation $M[x]$ denotes the memory word at address x.

Example

- Find a tiling that cover the IR tree at right.
- Give the effect of the tilings.
- Are other tilings possible?

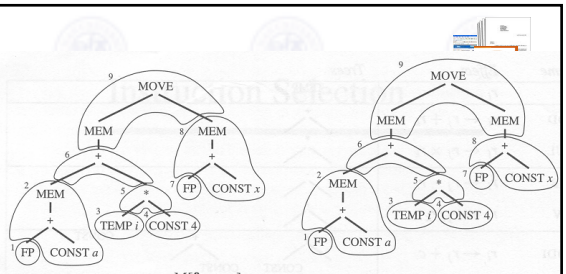



```
1. addi r1←r0+1
2. load r2←M[r1+2]
```

```
1. addi r1←r0+1
2. addi r2←r0+2
3. add r3←r1+r2
4. load r4←M[r3+0]
```

Example

- a[i] = x
  - where *i* is register, *a* and *x* are frame-resident
  - *a* is frame offset to reference to an array



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	LOAD	$r_2 \leftarrow M[\mathbf{fp} + x]$
9	STORE	$M[r_1 + 0] \leftarrow r_2$

(a)

2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	ADDI	$r_2 \leftarrow \mathbf{fp} + x$
9	MOVEM	$M[r_1] \leftarrow M[r_2]$

(b)

FIGURE 9.2. A tree tiled in two ways.

One tile covers one node

ADDI	$r_1 \leftarrow r_0 + a$
ADD	$r_1 \leftarrow \mathbf{fp} + r_1$
LOAD	$r_1 \leftarrow M[r_1 + 0]$
ADDI	$r_2 \leftarrow r_0 + 4$
MUL	$r_2 \leftarrow r_i \times r_2$
ADD	$r_1 \leftarrow r_1 + r_2$
ADDI	$r_2 \leftarrow r_0 + x$
ADD	$r_2 \leftarrow \mathbf{fp} + r_2$
LOAD	$r_2 \leftarrow M[r_2 + 0]$
STORE	$M[r_1 + 0] \leftarrow r_2$

Optimal and Optimum Tilings

- least cost - fewest instructions, least time, etc.
- Optimum（最优） - tiling whose cost sum to lowest possible value
- Optimal（最佳） - no two adjacent tiles can be combined into a single tile of lower total cost.

Optimal is not necessarily optimum

- Suppose all instructions have cost of 1 except MOVEM which has cost of *m*.
- *m>1* then (a) optimum, (b) optimal:
  - (a) = 6
  - (b) > 6
- *m<1* then (a) optimal, (b) optimum:
  - (a) = 6
  - (b) < 6
- *m=1* then (a) and (b) are optimum:
  - (a) = (b)

ALGORITHMS FOR INSTRUCTION SELECTION

Algorithms For Instruction Selection

- CISC - Complex Instruction Set Computers often accomplish multiple operations per instruction, can have large tilings
- RISC - Reduced Instruction Set Computers generally accomplish few operations per instruction, have smaller tilings
- Optimum/Optimal differences larger on CISC than on RISC

Maximal Munch - Optimal tiling

- Algorithm
  - Start at root
  - Find largest tile that fits (covers the most nodes) leaving subtrees
  - Generate instruction for tiling
  - Repeat on each subtree

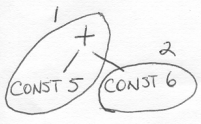
Implementation

- Two recursive functions
  - munchStm for statements
    - One clause for each statement
  - munchExp for expressions
    - each clause will match one tile
    - clauses ordered by tile preference (biggest to smallest)
- Key is finding target instruction that covers the largest tiling
- Order into cases where largest tiling matches are checked first.
- *Jouette* STORE and MOVEM same size tiling, choose either
- Top-down, depth-first recursion over IR tree, emit instructions *after* visiting subtrees
- Only instruction is emitted, registers allocated later

Exercise 2

- Munch the following IR into Jouette instructions using PROGRAM 9.3

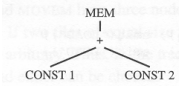
MEM(BINOP(PLUS, e1, CONST(2)))
MOVE(TEMP(t0), CONST(3))



Dynamic Programming

- Assigns a *cost* to every node in tree, the sum of the cost of subtree nodes
- Bottom-up, summing cost of node *n* subtrees first
- Matches *n* with each tile kind, minimum cost match selected
- *cost (n)* = sum(subtree cost) + cost (tiling *n*)
- *leaves* - places where subtrees can be attached
- Each tile has 0 or more *leaves*

Example



Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
+	ADD	1	1+1	3
+ CONST	ADDI	1	1	2
+ CONST	ADDI	1	1	2

Example - Cost is number of tilings

- cost(ADD) = 3 since total 3 the number of tilings required to cover ADD and subtrees
  - ADDI $r_2 \leftarrow r_0 + 2$
  - ADDI $r_1 \leftarrow r_0 + 1$
  - ADD $r_3 \leftarrow r_1 + r_2$
- cost(ADI) = 2 since total 2 the number of tilings required to cover ADDI and subtrees
  - ADDI $r_2 \leftarrow r_0 + 2$
  - ADDI $r_3 \leftarrow r_2 + 1$

Using minimum cost of subtree

Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
MEM	LOAD	1	2	3
MEM + CONST	LOAD	1	1	2
MEM + CONST	LOAD	1	1	2

ADDI $r_1 \leftarrow r_0 + 1$
LOAD $r_1 \leftarrow M[r_1 + 2]$

- cost(MEM) = 3
  - ADDI $r_2 \leftarrow r_0 + 2$
    ADDI $r_3 \leftarrow r_2 + 1$
    LOAD $r_4 \leftarrow M[r_3 + 0]$
- cost(MEM) = 2
  - ADDI $r_2 \leftarrow r_0 + 2$
    LOAD $r_4 \leftarrow M[r_2 + 1]$
- cost(MEM) = 2
  - ADDI $r_1 \leftarrow r_0 + 1$
    LOAD $r_4 \leftarrow M[r_1 + 2]$

•Note that other costs are possible:
  •Number of memory accesses
  •Number of registers used

Tree Grammars

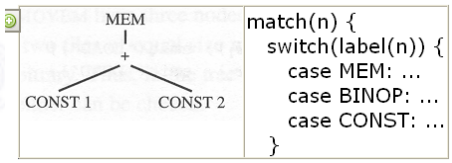
- Use a context-free grammar to describe the tiles
  - nonterminals s(statements), a(expressions)
- Skip for now

Fast Matching

- Maximal munch and dynamic-programming algorithms examine all tiles that match each node.
- Tile match if each non-leaf node of tile is labeled with same operator as node of tree (MEM, CONST, etc.)
- More efficient to examine only tiles with matching root label

Example

- The MEM root has only 4 possible subtrees, examine only those rather than recursively examining every possible expression again.
- In practice, *visitor* pattern yields similar performance; discussed at end of notes.



Efficiency of Tiling Algorithms

- Linear runtime of both Maximal Munch and Dynamic Programming
- Dependent on number of nodes in input tree
- In practice
  - Instruction selection is very fast
  - Even lexical analysis takes more time to run

MM vs DP

- Maximal Munch
  - Good for RISC
  - Tile size is small and of uniform cost
  - Usually no difference between optimum and optimal tiling
- Dynamic Programming
  - Good for CISC
  - Tile size is large and of widely varying cost
  - Small, but noticeable differences between optimum and optimal tiling

CISC&RISC

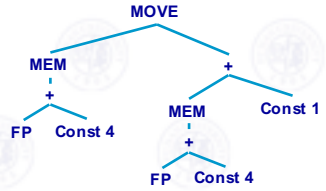
CISC&RISC

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

Complex Instruction Set Computers Reduced Instruction Set Computers

Example

x=x+1




```
load r1←M[fp+4]
addi r2←r1+1
store M[fp+4] ←r2
```

```
CISC
add[fp+4], $1
```

LOW-LEVEL IR TO IR TREE/DAG

Instruction Selection

- Different sets of instructions in low-level IR and in the target machine
- Instruction selection = translate low-level IR to assembly instructions on the target machine
- Straightforward solution: translate each low-level IR instruction to a sequence of machine instructions
- Example:

x = y + z

```
mov y, r1
mov z, r2
add r2, r1
mov r1, x
```

Instruction Selection

- Problem: straightforward translation is inefficient
  - One machine instruction may perform the computation in multiple low-level IR instructions
  - Excessive memory traffic

Example

- Consider a machine that includes the following instructions:

```
add r2, r1      r1 ← r1+r2
mulc c, r1      r1 ← r1*c
load r2, r1     r1 ← *r2
store r2, r1        *r1 ← r2
movem r2, r1        *r1 ← *r2
movex r3, r2, r1 *r1 ← *(r2+r3)
```

Example

- Consider the computation:
  a[i+1] = b[j]
- Assume a,b, i, j are global variables
  - register ra holds address of a
  - register rb holds address of b
  - register ri holds value of i
  - register rj holds value of j

Low-level IR:

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

Possible Translation

- Address of b[j]: mulc 4, rj
  add rj, rb
- Load value b[j]: load rb, r1
- Address of a[i+1]: add 1, ri
  mulc 4, ri
  add ri, ra
- Store into a[i+1]: store r1, ra

Low-level IR:
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3

Another Translation

- Address of b[j]: mulc 4, rj
  add rj, rb
- Address of a[i+1]: add 1, ri
  mulc 4, ri
  add ri, ra
- Store into a[i+1]: movem rb, ra

Low-level IR:
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3

Yet Another Translation

- Index of b[j]: mulc 4, rj
- Address of a[i+1]: add 1, ri
  mulc 4, ri
  add ri, ra
- Store into a[i+1]: movex rj, rb, ra

Low-level IR:
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3

Issue: Instruction Costs

- Different machine instructions have different costs
  - Time cost: how fast instructions are executed
  - Space cost: how much space instructions take
- Example: cost = number of cycles
  - add r2, r1 cost=1
  - mulc c, r1 cost=10
  - load r2, r1 cost=3
  - store r2, r1 cost=3
  - movem r2, r1 cost=4
  - movex r3, r2, r1 cost=5
- Goal: find translation with smallest cost

How to Solve the Problem?

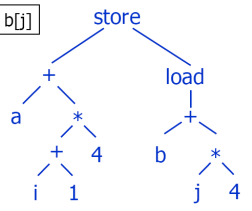
- Difficulty: low-level IR instruction matched by a machine instructions may not be adjacent
- Example: movem rb, ra
- Idea: use tree-like representation!
  - Easier to detect matching instructions

Low-level IR:
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3

Tree Representation

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

a[i+1] = b[j]

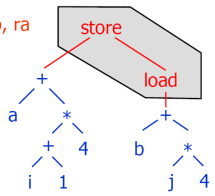


Low-level IR:
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3

Tiles

- Tile = tree patterns (subtrees) corresponding to machine instructions

movem rb, ra

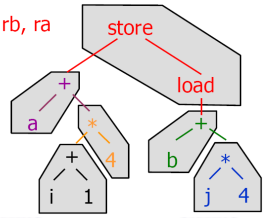


Low-level IR:
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3

Tiling

- Tiling = cover the tree with disjoint tiles

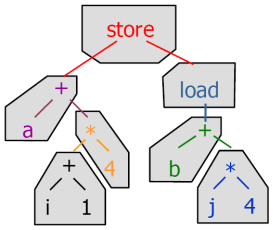
movem rb, ra



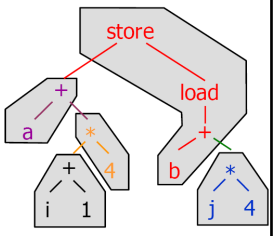
Low-level IR:
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3

Tiling

store rb, ra



movex rj, rb, ra

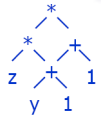


Directed Acyclic Graphs

- Tree representation: appropriate for instruction selection
  - Tiles = subtrees → machine instructions
- DAG = more general structure for representing instructions
  - Common sub-expressions represented by the same node
  - Tile the expression DAG

Example

```
t = y+1
y = z*t
t = t+1
z = t*y
```



Big Picture

- What the compiler has to do:
  1. Translate low-level IR code into DAG representation
  2. Then find a good tiling of the DAG
  - Maximal munch algorithm
  - Dynamic programming algorithm

DAG Construction

- Input: a sequence of low IR instructions in a basic block
- Output: an expression DAG for the block
- Idea:
  - Label each DAG node with variable holding that value
  - Build DAG bottom-up
  - A variable may have multiple values in a block
- Use different variable indices for different values of the variable: t0, t1, t2, etc

Algorithm

```
index[v] = 0 for each variable v
For each instruction I (in the order they appear)
    For each variable v that I directly uses, with
    n=index[v]
        if node vn doesn't exist
                create node vn , with label vn
    Create expression node for instruction I, with
    children
        { vn | v ∈use[I] }
    For each v∈def[I]
        index[v] = index[v] + 1
    If I is of the form x = … and n = index[x]
        label the new node with xn
```

Issues

- Function/method calls
  - May update global variables or object fields
  - def[I] = set of globals/fields
- •Store instructions
  - May update any variable
  - If stack addresses are not taken (e.g., Java), def[I] = set of heap objects