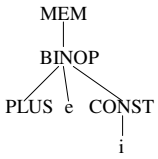
COMPILERS
Instruction Selection

hussein suleman
uct csc305w 2004

Introduction

- IR expresses only one operation in each node.
- MC performs several instructions in an instruction.
  - e.g., fetch and add



Preliminaries

- Express each machine instruction as a fragment of an IR tree – "tree pattern".
- Instruction selection is then equivalent to tiling the tree with a minimal set of tree patterns.

Jouette Architecture

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

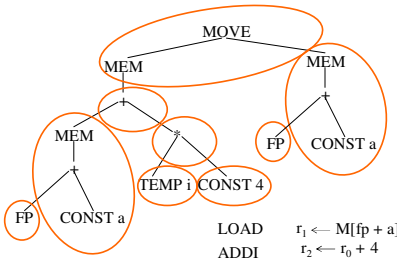
Jouette Architecture

Name	Effect	Trees
STORE	$M[r_j + c] \leftarrow r_i$	MOVE MOVE MOVE MOVE MEM MEM MEM MEM + + CONST CONST CONST
MOVEM	$M[r_j] \leftarrow M[r_i]$	MOVE MEM MEM

Instruction Selection

- The concept of instruction selection is tiling.
- Tiles are the set of tree patterns corresponding to legal machine instructions.
- We want to cover the tree with non-overlapping tiles.
- Note: We wont worry about which registers to use - yet.

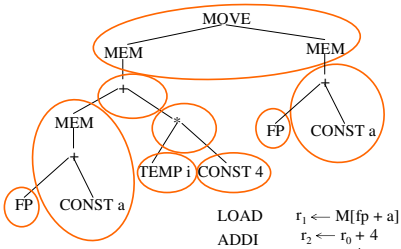
Tiled Tree 1



Operation: a[i] = x

LOAD	$r_1 \leftarrow M[fp + a]$
ADDI	$r_2 \leftarrow r_0 + 4$
MUL	$r_2 \leftarrow r_0 * r_2$
ADD	$r_1 \leftarrow r_1 + r_2$
LOAD	$r_2 \leftarrow M[fp + x]$
STORE	$M[r_1 + 0] \leftarrow r_2$

Tiled Tree 2



Operation: a[i] = x

LOAD	$r_1 \leftarrow M[fp + a]$
ADDI	$r_2 \leftarrow r_0 + 4$
MUL	$r_2 \leftarrow r_0 * r_2$
ADD	$r_1 \leftarrow r_1 + r_2$
ADDI	$r_2 \leftarrow fp + x$
MOVEM	$M[r_1] \leftarrow M[r_2]$

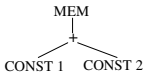
Optimum and Optimal Tilings

- Best tiling corresponds to least cost instruction sequence.
- Each instructions is costed (somehow).
- Optimum tiling
  - tiles sum to lowest possible value
- Optimal tiling
  - no two adjacent tiles can be combined to a tile of lower cost
- Note: Optimum tiling is Optimal, but not vice versa!

Maximal Munch Algorithm

- Start at the root.
- Find the largest tile that fits.
- Cover the root and possibly several other nodes with this tile.
- Repeat for each subtree.
- Generates instructions in reverse order.
- If two tiles of equal size match the current node, choose either.

Maximal Munch Example



MEM is matched by LOAD

CONST is matched by ADDI

Instructions emitted (in reverse order) are:

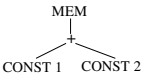
ADDI $r_1 \leftarrow r_0 + 2$

LOAD $r_1 \leftarrow M[r_1 + 1]$

Dynamic Programming Algorithm

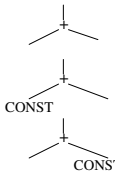
- Assign a cost to every node.
  - Sum of instruction costs of the best instruction sequence that can tile that subtree.
- For each node n, proceeding bottom-up:
  - For each tile t of cost c that matches at n there will be zero or more subtrees, si, that correspond to the leaves (bottom edges) of the tile.
    - Cost of matching t is cost of t + sum of costs of all child trees of t
  - Assign tile with minimum cost to n.
- Walk tree from root and emit instructions for assigned tiles.

Dynamic Programming Example 1/2



CONST is only matched by an ADDI instruction with cost 1

The + node can be matched by



ADD cost 1 leaves 2 Total 3

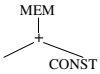
ADDI cost 1 leaves 1 Total 2

ADDI cost 1 leaves 1 Total 2

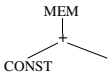
Dynamic Programming Example 2/2

The MEM node can be matched by

LOAD cost 1 leaves 2 Total 3



LOAD cost 1 leaves 1 Total 2



LOAD cost 1 leaves 1 Total 2

Instructions emitted (in reverse order, in second pass) are:

ADDI $r_1 \leftarrow r_0 + 1$

LOAD $r_1 \leftarrow M[r_1 + 2]$

Efficiency of Algorithms

- Assume (on average):
  - T tiles
  - K non-leaf nodes in matching tile
  - Kp is largest number of nodes to check to find matching tile
  - Tp no of different tiles matching at each node
  - N nodes in tree
- Cost of MM: O((Kp + Tp)N/K)
- Cost of DP: O((Kp + Tp)N)
- In both cases, with Kp, Tp, K constant
  - O(N)

Handling CISC Machine Code

- Fewer registers:
  - E.g., Pentium has only 6 general registers
  - Allocate TEMPs and solve problem later!
- Register use is restricted:
  - E.g., MUL on Pentium requires use of eax
  - Introduce additional LOAD/MOVE instructions to copy values.
- Complex addressing modes:
  - E.g., Pentium allows ADD [ebp-8],ecx
  - Simple code generation still works, but is not as size-efficient, and can trash registers.

Implementation Issues

- If registers are allocated after instruction selection, generated code must have "holes".
  - Assembly code template: LOAD d0,s0
  - List of source registers
  - List of destination registers
    - Including registers trashed by instruction (e.g., return address and return value registers for CALLs)
- Register allocation will then fill in the holes, by (simplistically) matching source and destination registers and eliminating redundancy.