

COMPILERS

Basic Blocks and Traces

hussein suleman
uct csc305w 2004

Evaluation Order

- Its useful to evaluate the subexpressions of an expression in any order.
- Some IR trees can contain side effects.
- ESEQ and CALL can contain side effects
 - assignment
 - I/O
- If there were no side effects in these statements then the order of evaluation would not matter.

IR/MC mismatches

- CJUMP jumps to one of two labels not one label and next instruction.
- ESEQ nodes within expressions make order of evaluation significant.
- CALL nodes within expressions similar.
- CALL nodes within the argument of other CALL nodes make allocation of formal-parameter registers difficult.

Canonical Trees

- 1: No SEQ or ESEQ
- 2: CALL can only be subtree of EXP(. .) or MOVE(TEMP t, . .)
- Transformations:
 - lift ESEQs up tree until they can become SEQs
 - turn SEQs into linear list

Simplification Rules

- $ESEQ(s_1, ESEQ(s_2, e)) \Rightarrow$
 - $ESEQ(SEQ(s_1, s_2), e)$
- $BINOP(op, ESEQ(s, e_1), e_2) \Rightarrow$
 - $ESEQ(s, BINOP(op, e_1, e_2))$
- $MEM(ESEQ(s, e_1)) \Rightarrow$
 - $ESEQ(s, MEM(e_1))$
- $JUMP(ESEQ(s, e_1)) \Rightarrow$
 - $SEQ(s, JUMP(e_1))$
- $CJUMP(op, ESEQ(s, e_1), e_2, l_1, l_2) \Rightarrow$
 - $SEQ(s, CJUMP(op, e_1, e_2, l_1, l_2))$
- $MOVE(ESEQ(s, e_1), e_2) \Rightarrow$
 - $SEQ(s, MOVE(e_1, e_2))$
- $BINOP(op, e_1, ESEQ(s, e_2)) \Rightarrow$
 - $ESEQ(MOVE(TEMP t, e_1), ESEQ(s, BINOP(op, TEMP t, e_2)))$
- $CJUMP(op, e_1, ESEQ(s, e_2), l_1, l_2) \Rightarrow$
 - $SEQ(MOVE(TEMP t, e_1), SEQ(s, CJUMP(op, TEMP t, e_2, l_1, l_2)))$
- $CALL(f, a) =$
 - $ESEQ(MOVE(TEMP t, CALL(f, a)), TEMP(t))$

General Technique

- For subexpressions of a node, $e_1..e_n$,
 - $[e_1, e_2, \dots, ESEQ(s, e_i), \dots, e_{n-1}, e_n]$
 - if s commutes with $e_1..e_{i-1}$ (independent),
 - $(s; [e_1, e_2, \dots, e_i, \dots, e_{n-1}, e_n])$
 - otherwise,
 - $SEQ(MOVE(TEMP t_1, e_1),$
 - $SEQ(MOVE(TEMP t_2, e_2),$
 - $\dots SEQ(MOVE(TEMP t_{i-1}, e_{i-1})\dots)$
 - $[TEMP t_1, TEMP t_2, \dots, TEMP t_{i-1}, e_i, \dots, e_{n-1}, e_n]$
- In general, extract children, reorder and then reinsert children

Basic Blocks

- Divide linear sequence of nodes in each subprogram into basic blocks, where:
 - execution always starts at top and stops at bottom
 - first statement is a LABEL
 - last statement is a JUMP or CJUMP
 - no intervening LABELs, JUMPs or CJUMPs
- Basic blocks are easier to work with for future optimisations since they can be rearranged, while maintaining logic.

Basic Blocks Algorithm

- Scan sequence of statements from start to end
 - If LABEL, start new block
 - If JUMP or CJUMP, end block
- If a block does not start with a LABEL
 - Create new LABEL
- If a block does not end with JUMP/CJUMP
 - Create new JUMP to next LABEL
- Add terminal "JUMP done" for end of subprogram.

Traces

- We want to rearrange basic blocks to optimise the number and nature of jumps.
- A trace is a sequence of statements that can be consecutively executed during the program execution (e.g., b1, b3, b6 below)
 - block b1: LABEL a ... JUMP b
 - block b3: LABEL b ... JUMP c
 - block b6: LABEL c ... CJUMP ?,a
- Every program has many overlapping traces – we want a single set that covers all the instructions.

Trace Generation

- Put all basic blocks into a list Q
- while Q is not empty
 - Start a new (empty) trace T
 - Remove an element b from Q
 - while b is not marked
 - Mark b
 - Append b to T
 - Check successors if b for unmarked node and make this the new b
 - End the trace T

JUMP considerations

- We prefer CJUMP followed by its false label, since this translates to MC conditional jump.
- If CJUMP followed by its true label,
 - switch true and false labels, and negate conditional
- If CJUMP (cond, a, b, It, If) followed by some other label, replace with:
 - CJUMP (cond, a, b, It, Ifprime)
 - LABEL Ifprime
 - JUMP (NAME If)
- Remove all JUMPs followed by their target LABELs.