IR lowering

- We lower the IR to a canonical form in which code is a sequence of statements, each containing a single side effect.
- Done by transformations that lift side-effecting statements to the top of the IR tree.
  - \( L[s] = s_1...s_n \)
  - \( L[e] = s_1...s_n; e' \)  
    - Side effects of \( e \) in \( s_i \). Value of \( e \) computed by side-effect-free \( e' \)

Conditional jumps

- IR is now just a linear list of statements with one side effect per statement
- Still contains \( \text{CJUMP} \) nodes: two-way branches
- Real machines: fall-through branches (e.g. \( \text{JZ} \), \( \text{JNZ} \))

\[
\begin{align*}
\text{CJUMP}(e, t, f) &\quad \text{evaluate } e \\
\ldots &\quad \text{LABEL}(t) \quad \text{if-true code} \\
\text{LABEL}(f) &\quad \text{LABEL}(f) \quad \text{if-true code} \\
&\quad f:
\end{align*}
\]
Simple Solution

- Translate CJUMP into conditional branch followed by unconditional branch

\[
\text{CJUMP}(\text{TEMP}(t1)==\text{TEMP}(t2), t, f) \quad \text{CMP } t1, t2 \\
\quad \text{JZ } t \\
\quad \text{JMP } f
\]

- JMP is usually gratuitous
- Code can be reordered so jump goes to next statement

Basic blocks

- Unit of reordering is a basic block
- A sequence of statements that is always begun at its start and always exits at the end:
  - starts with a \textit{LABEL}(\pi) statement (or beginning of all statements)
  - ends with a \textit{JUMP}, \textit{CJUMP}, or \textit{RETURN} statement, or just before a \textit{LABEL} statement
  - contains no other \textit{JUMP} or \textit{CJUMP} statement
  - contains no interior \textit{LABEL} used as a jump target
- No point to breaking up a basic block during reordering

Basic block example

\[
\text{CJUMP}(e, L2, L3) \\
\text{LABEL}(L1) \\
\text{MOVE}(\text{TEMP}(x), \text{TEMP}(y)) \\
\text{LABEL}(L2) \\
\text{MOVE}(\text{TEMP}(x), \text{TEMP}(y) + \text{TEMP}(z)) \\
\text{JUMP}(\text{NAME}(L1)) \\
\text{LABEL}(L3) \\
\text{EXP}(\text{CALL}(\text{NAME}(f)), \text{TEMP}(x))
\]

Control flow graph

- Control flow graph has basic blocks as nodes
- Edges show control flow between basic blocks
Fixing conditional jumps

- Reorder basic blocks so that (if possible)
  - the “false” direction of two-way jumps goes to
    the very next block
  - JUMPs go to the next block (are deleted)
- What if not satisfied?
  - For CJUMP add another JUMP immediately after to go to the right basic block
- How to find such an ordering of the basic blocks?

Traces

- Idea: order blocks according to a possible trace: a sequence of blocks that might (naively) be executed in sequence, never visiting a block more than once
- Algorithm:
  - pick an unmarked block (begin w/ start block)
  - run a trace until no more unmarked blocks can be visited, marking each block on arrival
  - repeat until no more unmarked blocks

Example

- Possible traces?

Arranging by traces

- Can use profiling information, heuristics to choose which branch to follow
Reordered code

CJUMP(e, L2, L3)

LABEL(L2)
MOVE(TEMP(x), TEMP(y) + TEMP(z))

LABEL(L1)
MOVE(TEMP(x), TEMP(y))

LABEL(L3)
EXP(CALL(NAME(f)), TEMP(x))

Reversing sense of jumps

CJUMP(e, L2, [L3])
JUMP(L3)

LABEL(L2)
MOVE(TEMP(x), TEMP(y) + TEMP(z))

LABEL(L1)
MOVE(TEMP(x), TEMP(y))

LABEL(L3)
EXP(CALL(NAME(f)), TEMP(x))

Abstract Assembly

- Abstract assembly = assembly code w/ infinite register set
- Canonical intermediate code = abstract assembly code – except for expression trees
  - MOVE(e₁, e₂) ⇒ mov e₂, e₁
  - JUMP(e) ⇒ jmp e
  - CJUMP(e, l) ⇒ cmp e₁, e₂
    [jne|je|jgt|... ] l
  - CALL(e, e₁,...) ⇒ push e₁; ... ; push eₙ;
    call e
  - LABEL(l) ⇒ l:
Instruction selection

• Conversion to abstract assembly is problem of instruction selection for a single IR statement node
• Full abstract assembly code: glue translated instructions from each of the statements
• Problem: more than one way to translate a given statement. How to choose?

Example

MOVE(TEMP(t1), TEMP(t1) + MEM(TEMP(FP)+4))

ADD t2, t3
add t3, t1

MOVE TEMP(t1) ADD MEM TEMP(t1) ADD TEMP(fp) 4

Pentium ISA

• Need to map IR tree to actual machine instructions – need to know how instructions work
• Pentium is two-address CISC architecture
• Typical instruction has

  opcode (mov, add, sub, shl, shr, imul, idiv, jmp, jcc, push, pop, cmp, test &c.)
  - some opcodes have width suffix: b=8, w=16, l=32, q=64
  – destination (r, (r), n, k(r), (r1,r2),
  (r1,r2,w), k(r1,r2,w))

  (may also be an operand)
  – source (any legal destination, or a constant $n)

  opcode src dest
  movq $1,%rax \ addq %r5x,%rcx
  subq [%rbp],%rsi \ addq %rdi,[%rcx+16*edi],edi
  je label1 \ jmp [fp+4]

Tiling

• Idea: each Pentium instruction performs computation for a piece of the IR tree: a tile

  movq fp, t2
  addq $4, t2
  movq (t2), t3
  addq t3, t1

  ADD t2, t3
  add t3, t1
Some tiles

\[ \text{mov } t_1, t_2 \]

\[ \text{mov } t_f, t_1 \quad (t_f \text{ a fresh temporary)} \]

\[ \text{mov } [t_1+t_2], i \]