Administration

- Prelim 1 Wednesday
  - topics covered: regular expressions, tokenizing, context-free grammars, LL & LR parsers, static semantics, intermediate code generation
- Prelim 1 review session Monday 7-9PM

Where we are

- abstract syntax tree
- syntax-directed translation (IR generation)
- Intermediate code
- syntax-directed translation (flattening)
- Reordering with traces
- Canonical intermediate code
- Instruction selection
- Abstract assembly code
- Register allocation
- Assembly code

Conditional jumps

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

\[
\text{CJUMP}(e, t, f) \\
\text{...} \\
\text{LABEL}(t) \\
\text{if-true code} \\
\text{LABEL}(f) \\
\text{if-false code} \\
\text{f:}
\]

Simple Solution

- Translate CJUMP into conditional branch followed by unconditional branch

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

- 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 LABEL(n) statement (or beginning of all statements)
  - ends with a JMP or CJUMP statement, or just before a LABEL statement
  - contains no other JMP or CJUMP statement
  - contains no interior LABEL that is used as the target for a JMP or CJUMP from elsewhere
- No point to breaking up a basic block during reordering
Basic block example

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

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

Abstract Assembly
- Abstract assembly = assembly code w/ infinite register set
- Canonical intermediate code = abstract assembly code – except for expression trees
  - MOVE(e₁, e₂) ⇒ mov e₁, e₂
  - JUMP(e) ⇒ jmp e
  - CJUMP(e, l₁, l₂) ⇒ cmp e₁, e₂, [jne|je|jgt|…] l₁
  - CALL(e, e₁, …) ⇒ push e₁; …; call e
  - LABEL(l) ⇒ l:

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

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?

Pentium instructions
- Need to map actual machine instructions to IR tree – need to know how instructions work
- Pentium has two-address CISC
- Typical instruction has
  - opcode (mov, add, sub, shl, shr, mul, div, jmp, jcc, &c.)
  - destination (r, [r], [k], [r+k], [r1+r2], [r1+w*r2], [r1+w*r2+k])
  - source (any legal destination, or a constant)
- mov eax,1  add ebx,ecx
- sub esi,[ebp]  add [ecx+16*esi],edi
- je label1  jmp [fp+4]
Tiling

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

\[
\text{mov } t2, \text{ ebp} \\
\text{add } t2, 4 \\
\text{mov } t3, [t2] \\
\text{add } t1, t3
\]

- Tiles are connected by new temporary registers \(t2, t3\)

Some tiles

\[
\text{mov } t1, t2 \\
\text{mov } t2, t1 \\
\text{mov } [t1+t2], i
\]

Problem

- How to pick tiles that cover IR statement tree with minimum execution time?
- Need a good selection of tiles
  - small tiles to make sure we can tile every tree
  - large tiles for efficiency
- Usually want to pick large tiles: fewer instructions
- Pentium: RISC core instructions take 1 cycle, other instructions take more

\[
\text{add } [\text{ecx}+4], \text{ eax} \\
\text{mov edx, } [\text{ecx}+4] \\
\text{mov } [\text{ecx}+4], \text{ eax}
\]

An annoying instruction

- Pentium mul instruction multiples single operand by \(\text{eax}\), puts result in \(\text{eax}\) (low 32 bits), \(\text{edx}\) (high 32 bits)
- Solution: add extra \text{mov} instructions, let register allocation deal with \(\text{edx}\) overwrite

\[
\text{mov } \text{eax}, t1 \\
\text{mul } t2 \\
\text{mov } t2, \text{ eax}
\]

Branches

- How to tile a conditional jump?
- Fold comparison operator into tile

\[
\text{test } t1 \\
\text{jnz } l1 \\
\text{cmp } t1, t2 \\
\text{je } l1
\]

More handy tiles

\[
\text{lea} \text{ instruction computes a memory address but doesn't actually load from memory}
\]

\[
\text{lea } t2, [t1+t2] \\
\text{lea } t2, [t1+k*t2] \quad (k \text{ one of } 2, 4, 8, 16)
\]
Maximal Munch Algorithm

- Assume larger tiles = better
- Greedy algorithm: start from top of tree and use largest tile that matches tree
- Tile remaining subtrees recursively

```
MOVE
MEM
ADD
MEM
ADD
FP

ADD
MEM
ADD
FP
```

Implementing tiles

- Explicitly building every possible tile: tedious
- Easier to write subroutines for matching Pentium source, destination operands
- Reuse matching for all opcodes

```
MOVE
dest
source
ADD
source
MEM
ADD
CONST(i)
```

How good is it?

- Very rough approximation on modern pipelined architectures: execution time is number of tiles
- Maximal munch finds an optimal but not necessarily optimum tiling: cannot combine two tiles into a lower-cost tile
- We can find the optimum tiling using dynamic programming!