CS 4120 Introduction to Compilers

Ross Tate
Cornell University

Lecture 16: Basic blocks, CFGs, traces

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

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.
- $L[s] = s_1 \ldots s_n$
- $L[e] = s_1 \ldots s_n ; e'$
  - 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 CJUMP nodes : two-way branches
- Real machines : fall-through branches (*e.g.* **JZ, JNZ**)

```
CJUMP(e, t, f)          evaluate e
...                     JZ f
LABEL(t)                if-true code
if-true code          f:
LABEL(f)
```

Simple Solution

- Translate CJUMP into conditional branch followed by unconditional branch

```
CJUMP(TEMP(t1)==TEMP(t2), t, f)     CMP t1,t2
                                     JZ t
                                     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 JUMP, CJUMP, or RETURN statement, or just before a LABEL statement
  - contains no other JUMP or CJUMP statement
  - contains no interior LABEL used as a jump target
- No point to breaking up a basic block during reordering

```
LABEL(l)
...
CJUMP(e, l1, l2)
```

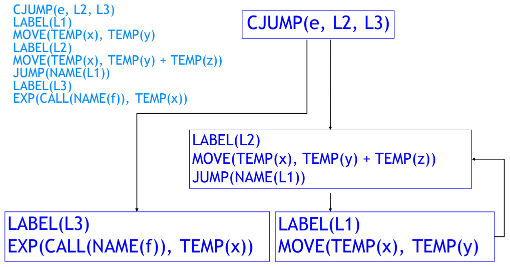
Basic block example

```
CJUMP(e, L2, L3)
LABEL(L1)
MOVE(TEMP(x), TEMP(y)
LABEL(L2)
MOVE(TEMP(x), TEMP(y) + TEMP(z))
JUMP(NAME(L1))
LABEL(L3)
EXP(CALL(NAME(f)), TEMP(x))
```

Control-flow graph

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

```
CJUMP(e, L2, L3)
LABEL(L1)
MOVE(TEMP(x), TEMP(y)
LABEL(L2)
MOVE(TEMP(x), TEMP(y) + TEMP(z))
JUMP(NAME(L1))
LABEL(L3)
EXP(CALL(NAME(f)), TEMP(x))
```



```
CJUMP(e, L2, L3)
```

```
LABEL(L2)
MOVE(TEMP(x), TEMP(y) + TEMP(z))
JUMP(NAME(L1))
```

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

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

Fixing conditional jumps

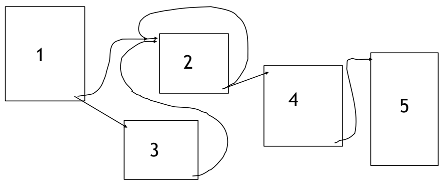
- Reorder basic blocks so that (if possible)
  - the “false” direction of two-way jumps goes to the very next block
  - JUMPs go to the next block (are deleted)
- What if not satisfied?
  - For CJUMP add another JUMP immediately after to go to the right basic block
- How to find such an ordering of the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

Traces

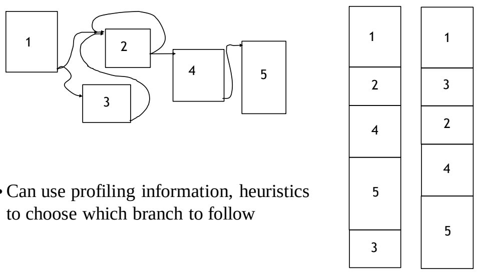
- Idea: order blocks according to a possible *trace:* a sequence of blocks that might (naively) be executed in sequence, never visiting a block more than once
- Algorithm:
  - pick an unmarked block (begin w/ start block)
  - run a trace until no more unmarked blocks can be visited, marking each block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

Example

- Possible traces?

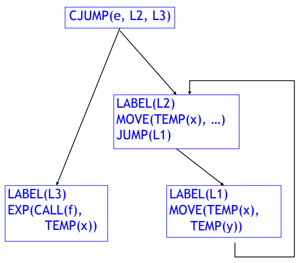


Arranging by traces



- 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))
JUMP(L1)

LABEL(L1)
MOVE(TEMP(x), TEMP(y)
JUMP(L2)

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

Reversing sense of jumps

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

LABEL(L2)
MOVE(TEMP(x), TEMP(y) + TEMP(z))
JUMP(L1)

LABEL(L1)
MOVE(TEMP(x), TEMP(y)
JUMP(L2)

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

⇒

```
CJUMP(NOT(e), L3, [L2])

LABEL(L2)
MOVE(TEMP(x), TEMP(y) + TEMP(z))
JUMP(L1)

LABEL(L1)
MOVE(TEMP(x), TEMP(y)
JUMP(L2)

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