CS 4120
Introduction to Compilers

Andrew Myers

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)
...
LABEL(t)
if-true code
LABEL(f)
```

```
    evaluate e
    JZ f
    if-true code
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

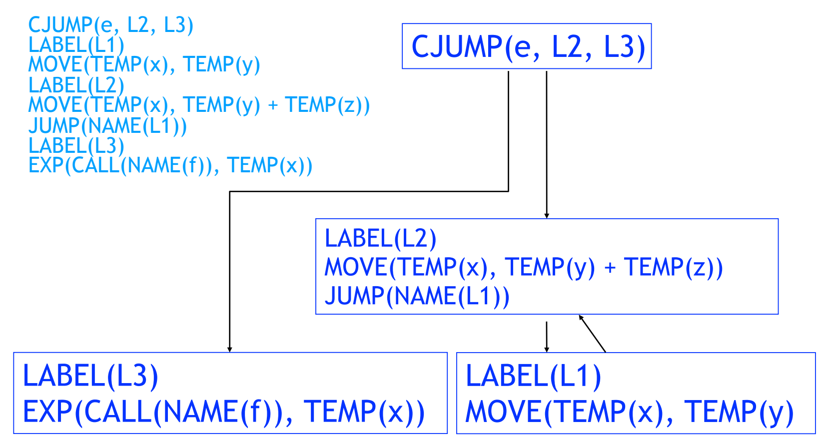
```
LABEL(l)
...
CJUMP(e, l1, l2)
```


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



Fixing conditional jumps

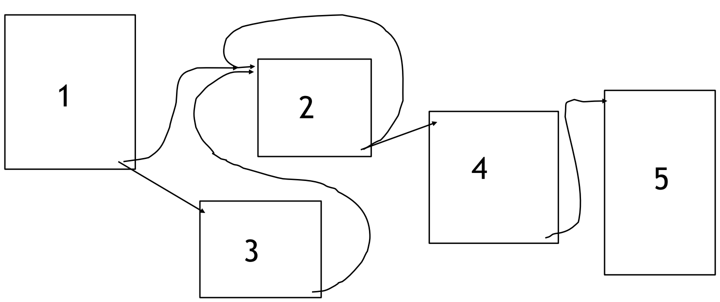
- Reorder basic blocks so that (if possible)
  - the “false” direction of two-way jumps goes to the very next block
  - JUMPs go to the next block (are deleted)
- What if not satisfied?
  - For CJUMP add another JUMP immediately after to go to the right basic block
- How to find such an ordering of the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

Traces

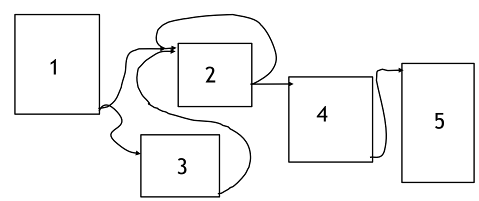
- Idea: order blocks according to a possible *trace:* a sequence of blocks that might (naively) be executed in sequence, never visiting a block more than once
- Algorithm:
  - pick an unmarked block (begin w/ start block)
  - run a trace until no more unmarked blocks can be visited, marking each block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

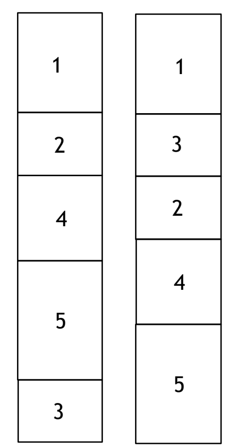
Example

- Possible traces?



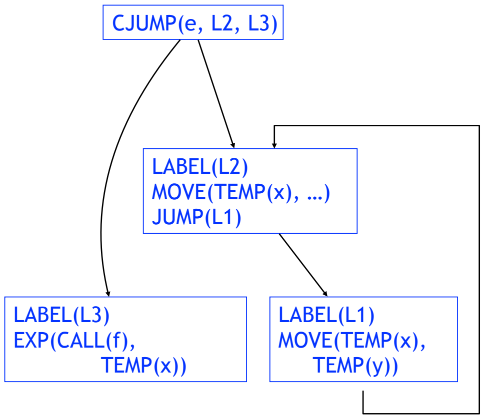
Arranging by traces





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

Reordered code




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

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

```
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))
```

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

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


Progress

abstract syntax tree

syntax-directed translation (IR generation)

intermediate code

syntax-directed translation (flattening)
reordering with traces

canonical intermediate code

instruction selection (tiling)

abstract assembly code

register allocation

assembly code

Abstract Assembly

- Abstract assembly = assembly code w/ infinite register set
- Canonical intermediate code = abstract assembly code – except for expression trees
- MOVE(e_1, e_2) ⇒ **mov** ***e2*, *e1***
- JUMP(e) ⇒ **jmp** ***e***
- CJUMP(e,l) ⇒ **cmp** ***e1*, *e2***
  [jne|je|jgt|…] ***l***

CALL(e, e_1,…) ⇒ **push** ***e_1*; … ; push *e_n*;**
call ***e***

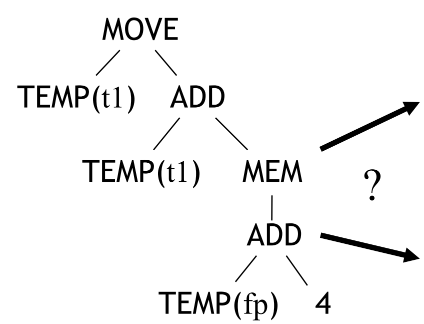
- LABEL(l) ⇒ **l:**

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))




```
mov fp, t2
add $4, t2
mov (t2), t3
add t3,t1
```

```
add t1,[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**)

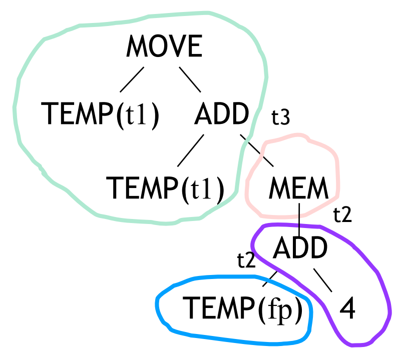
  – *source* (any legal destination, or a constant $n)

opcode *src* *dest*

```
movq $1,%rax              addq %rbx,%rcx
subq [%rbp],%rsi          addq %rdi,[%rcx+16*edi],edi
je label1           jmp [fp+4]
```


Tiling

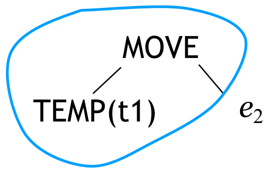
- Idea: each Pentium instruction performs computation for a piece of the IR tree: a *tile*



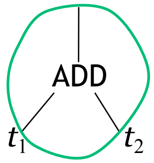

```
movq fp, t2
addq $4, t2
movq (t2), t3
addq t3,t1
```

- Tiles connected by new temporary registers (t2, t3) that hold result of tile

Some tiles

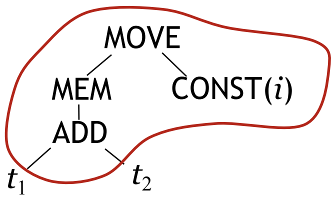


`mov t1, t`$_2$




```
mov tf, t1
add tf, t2
```

($\mathtt{t_f}$ a *fresh* temporary)



`mov [t`$_1$`+t`$_2$`],` i