CSC 255/455
Software Analysis and Improvement

Instruction Scheduling, Register Allocation, Partial Redundancy Removal

Instructor: Chen Ding

COMP 412
FALL 2008

Local Instruction Scheduling
— A Primer for Lab 3 —
Comp 412

What Makes Code Run Fast?

- Many operations have non-zero latencies
- Modern machines can issue several operations per cycle
- Execution time is *order-dependent* *(and has been since the 60's)*

Assumed latencies *(conservative)*

Operation	Cycles
load	3
store	3
loadI	1
add	1
mult	2
fadd	1
fmult	2
shift	1
branch	0 to 8

- Loads & stores may or may not block
  - > Non-blocking ⇒fill those issue slots
- Branch costs vary with path taken
- Branches typically have delay slots
  - > Fill slots with unrelated operations
  - > Percolates branch upward
- Scheduler should hide the latencies

Lab 3 will build a local scheduler

Example

$w \leftarrow w * 2 * x * y * z$

Simple schedule

```
 1  loadAI   r0,@w   ⇒ r1
 4  add      r1,r1   ⇒ r1
 5  loadAI   r0,@x   ⇒ r2
 8  mult     r1,r2   ⇒ r1
 9  loadAI   r0,@y   ⇒ r2
12  mult     r1,r2   ⇒ r1
13  loadAI   r0,@z   ⇒ r2
16  mult     r1,r2   ⇒ r1
18  storeAI  r1      ⇒ r0,@w
21  r1 is free
```

2 registers, 20 cycles

Schedule loads early

```
 1  loadAI   r0,@w   ⇒ r1
 2  loadAI   r0,@x   ⇒ r2
 3  loadAI   r0,@y   ⇒ r3
 4  add      r1,r1   ⇒ r1
 5  mult     r1,r2   ⇒ r1
 6  loadAI   r0,@z   ⇒ r2
 7  mult     r1,r3   ⇒ r1
 9  mult     r1,r2   ⇒ r1
11  storeAI  r1      ⇒ r0,@w
14  r1 is free
```

3 registers, 13 cycles

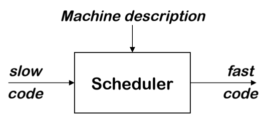
Reordering operations for speed is called *instruction scheduling*

Instruction Scheduling (Engineer's View)

The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time

The Concept



The Task

- Produce correct code
- Minimize wasted cycles
- Avoid spilling registers
- Operate efficiently

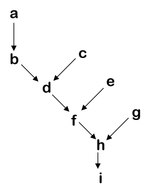
Instruction Scheduling (The Abstract View)

To capture properties of the code, build a <u>precedence graph</u> *G*

- Nodes $n \in G$ are operations with *type(n)* and *delay(n)*
- An edge $e = (n_1,n_2) \in G$ if & only if n_2 uses the result of n_1

```
a:  loadAI   r0,@w   ⇒ r1
b:  add      r1,r1   ⇒ r1
c:  loadAI   r0,@x   ⇒ r2
d:  mult     r1,r2   ⇒ r1
e:  loadAI   r0,@y   ⇒ r2
f:  mult     r1,r2   ⇒ r1
g:  loadAI   r0,@z   ⇒ r2
h:  mult     r1,r2   ⇒ r1
i:  storeAI  r1      ⇒ r0,@w
```

The Code



The Precedence Graph

Instruction Scheduling (Definitions)

A *correct schedule* S maps each $n \in N$ into a non-negative integer representing its cycle number, and

1. $S(n) \geq 0$, for all $n \in N$, obviously
2. If $(n_1,n_2) \in E$, $S(n_1) + delay(n_1) \leq S(n_2)$
3. For each type t, there are no more operations of type t in any cycle than the target machine can issue

The *length* of a schedule S, denoted $L(S)$, is
$L(S) = max_{n \in N} (S(n) + delay(n))$

The goal is to find the shortest possible correct schedule.
S is *time-optimal* if $L(S) \leq L(S_1)$, for all other schedules S_1
A schedule might also be optimal in terms of registers, power, or space….

Instruction Scheduling (What's so difficult?)

Critical Points
- All operands must be available
- Multiple operations can be *ready*
- Moving operations can lengthen register lifetimes
- Placing uses near definitions can shorten register lifetimes
- Operands can have multiple predecessors

Together, these issues make scheduling *hard* (NP-Complete)

Local scheduling is the simple case
- Restricted to straight-line code
- Consistent and predictable latencies

Instruction Scheduling: The Big Picture

1. Build a precedence graph, *P*
2. Compute a *priority function* over the nodes in *P*
3. Use list scheduling to construct a schedule, one cycle at a time
   a. Use a queue of operations that are ready
   b. At each cycle
      I. Choose the highest priority ready operation and schedule it
      II. Update the ready queue

Local list scheduling
- The dominant algorithm for twenty years
- A greedy, heuristic, local technique

Local List Scheduling

```
Cycle ← 1
Ready ← leaves of P
Active ← Ø

while (Ready ∪ Active ≠ Ø)
   if (Ready ≠ Ø) then
      remove an op from Ready
      S(op) ← Cycle
      Active ¬ Active ∪ op

   Cycle ← Cycle + 1

   for each op ∈ Active
      if (S(op) + delay(op) ≤ Cycle) then
         remove op from Active
         for each successor s of op in P
            if (s is ready) then
               Ready ← Ready ∪ s
```

Removal in priority order

op has completed execution

If successor's operands are ready, put it on Ready

Scheduling Example

1. Build the precedence graph

```
a:  loadAI   r0,@w   ⇒ r1
b:  add      r1,r1   ⇒ r1
c:  loadAI   r0,@x   ⇒ r2
d:  mult     r1,r2   ⇒ r1
e:  loadAI   r0,@y   ⇒ r2
f:  mult     r1,r2   ⇒ r1
g:  loadAI   r0,@z   ⇒ r2
h:  mult     r1,r2   ⇒ r1
i:  storeAI  r1      ⇒ r0,@w
```

The Code

$$\begin{tikzcd}
a \arrow[d] & & & & \\
b \arrow[dr] & & c \arrow[dl] & & \\
& d \arrow[dr] & & e \arrow[dl] & \\
& & f \arrow[dr] & & g \arrow[dl] \\
& & & h \arrow[d] & \\
& & & i &
\end{tikzcd}$$

The Precedence Graph

Scheduling Example

1. Build the precedence graph
2. Determine priorities: longest latency-weighted path

```
a:  loadAI   r0,@w   ⇒ r1
b:  add      r1,r1   ⇒ r1
c:  loadAI   r0,@x   ⇒ r2
d:  mult     r1,r2   ⇒ r1
e:  loadAI   r0,@y   ⇒ r2
f:  mult     r1,r2   ⇒ r1
g:  loadAI   r0,@z   ⇒ r2
h:  mult     r1,r2   ⇒ r1
i:  storeAI  r1      ⇒ r0,@w
```

The Code

$$\begin{tikzcd}
a\ 13 \arrow[d] & & & & \\
10\ b \arrow[dr] & & c\ 12 \arrow[dl] & & \\
& 9\ d \arrow[dr] & & e\ 10 \arrow[dl] & \\
& & 7\ f\ 5 \arrow[dr] & & g\ 8 \arrow[dl] \\
& & & h \arrow[d] & \\
& & & i\ 3 &
\end{tikzcd}$$

The Precedence Graph

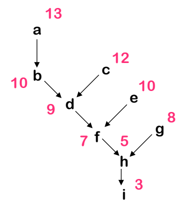
Scheduling Example

1. Build the precedence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling

New register name used

```
1)  a:  loadAI   r0,@w   ⇒ r1
2)  c:  loadAI   r0,@x   ⇒ r2
3)  e:  loadAI   r0,@y   ⇒ r3
4)  b:  add      r1,r1   ⇒ r1
5)  d:  mult     r1,r2   ⇒ r1
6)  g:  loadAI   r0,@z   ⇒ r2
7)  f:  mult     r1,r3   ⇒ r1
9)  h:  mult     r1,r2   ⇒ r1
11) i:  storeAI  r1      ⇒ r0,@w
```

The Code



The Precedence Graph

More List Scheduling

List scheduling breaks down into two distinct classes

Forward list scheduling	Backward list scheduling
• Start with available operations	• Start with no successors
• Work forward in time	• Work backward in time
• Ready ⇒ all operands available	• Ready ⇒ latency covers uses

Variations on list scheduling

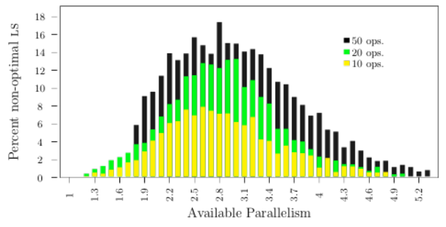
- Prioritize critical path(s)
- Schedule last use as soon as possible
- Depth first in precedence graph (minimize registers)
- Breadth first in precedence graph (minimize interlocks)
- Prefer operation with most successors

3 compute months of work

Schielke's Scheduling Study

Non-optimal list schedules (%) versus available parallelism

1 functional unit, randomly generated blocks of 10, 20, 50 ops

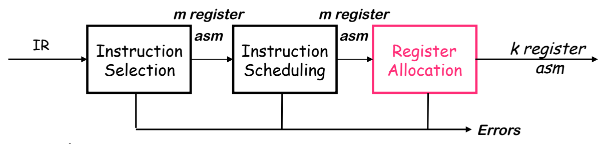


- 85,000 randomly generated blocks
- RBF found optimal schedules for > 80%
- Peak difficulty (for RBF) is around 2.8

Tie-breaking matters because it affects the choice when queue has > 1 element

Register Allocation

Part of the compiler's back end



Critical properties

- Produce <u>correct</u> code that uses *k* (or fewer) registers
- Minimize added loads and stores
- Minimize space used to hold *spilled values*
- Operate efficiently

  O(n), O(n $\log_2$n), maybe O(n^2), but not O(2^n)

Code to Interference Graph

RUTGERS An Example : Bottom-up

➢ Live Ranges

```
1| loadI   1028    ⇒ r1
2| load    r1      ⇒ r2
3| mult    r1, r2  ⇒ r3
4| loadI   5       ⇒ r4
5| sub     r4, r2  ⇒ r5
6| loadI   8       ⇒ r6
7| mult    r5, r6  ⇒ r7
8| sub     r7, r3  ⇒ r8
9| store   r8      ⇒ r1
```

NOTE: live sets on exit of each instruction

An Example : Bottom-up

- Live Ranges

```
1 loadI   1028    ⇒ r1    // r1
2 load    r1      ⇒ r2    // r1 r2
3 mult    r1, r2  ⇒ r3    // r1 r2 r3
4 loadI   5       ⇒ r4    // r1 r2 r3 r4
5 sub     r4, r2  ⇒ r5    // r1    r3    r5
6 loadI   8       ⇒ r6    // r1    r3    r5 r6
7 mult    r5, r6  ⇒ r7    // r1    r3          r7
8 sub     r7, r3  ⇒ r8    // r1                   r8
9 store   r8      ⇒ r1    //
```

NOTE: live sets on exit of each instruction

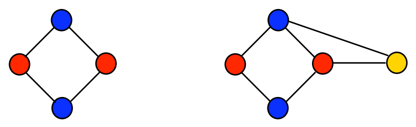
Interference Graph to Coloring

Graph Coloring (A Background Digression)

The problem

A graph *G* is said to be *k-colorable* iff the nodes can be labeled with integers 1… k so that no edge in G connects two nodes with the same label

Examples



2-colorable 3-colorable

Each color can be mapped to a distinct physical register

Global Register Allocation

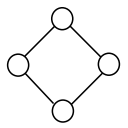
Taking a global approach
- Abandon the distinction between local & global
- Make systematic use of registers or memory
- Adopt a general scheme to approximate a good allocation

Graph coloring paradigm (*Lavrov & (later) Chaitin*)
1. Build an interference graph G_I for the procedure
   - Computing LIVE is harder than in the local case
   - G_I is not an interval graph
2. (try to) construct a *k*-coloring
   - Minimal coloring is NP-Complete
   - Spill placement becomes a critical issue
3. Map colors onto physical registers

Improvement in Coloring Scheme

Optimistic Coloring (*Briggs, Cooper, Kennedy, and Torczon*)
- If Chaitin's algorithm reaches a state where every node has *k* or more neighbors, it chooses a node to spill.
- Briggs said, take that same node and push it on the stack
  - When you pop it off, a color might be available for it!

2 Registers:



Chaitin's algorithm immediately spills one of these nodes

  - For example, a node *n* might have *k+2* neighbors, but those neighbors might only use 3 (<*k*) colors
    - → Degree is a *loose upper bound* on colorability

Lazy Code Motion

(Partial Redundancy Elimination)

Partially Redundant Expression

An expression is partially redundant at *p* if it is redundant along some, but not all, paths reaching *p*

Example

$$\begin{tikzcd}
b \leftarrow b + 1 \arrow[dr] & & a \leftarrow b + c \arrow[dl] \\
& a \leftarrow b + c &
\end{tikzcd}$$

Partially Redundant Expression

An expression is partially redundant at *p* if it is redundant along some, but not all, paths reaching *p*

Example

$$\begin{tikzcd}
b \leftarrow b + 1 \arrow[dr] & & a \leftarrow b + c \arrow[dl] & & \begin{matrix} b \leftarrow b + 1 \\ a \leftarrow b + c \end{matrix} \arrow[dr] & & a \leftarrow b + c \arrow[dl] \\
& a \leftarrow b + c & & & & a \leftarrow b + c &
\end{tikzcd}$$

Inserting a copy of "a ← b + c" after the definition of b can make it redundant ← fully redundant?

Lazy Code Motion

- Sources
  - Knoop, Ruthing, Steffen, PLDI 1992
  - Dreschsler and Stadel, SIGPLAN 1993
  - Chapter 10, Engineering a compiler
- Intuition
  - computer available and anticipable expressions
  - compute the earliest placement for each expression
  - push expressions down the CFG to the last point with no redundancy
- Solving a set of data flow equations
  - availability, anticipable, earliest placement, later placement, insert, delete