Instruction Scheduling

Note by Baris Aktemur:
Our slides are adapted from Cooper and Torczon's slides that they prepared for COMP 412 at Rice.

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)*

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

Assumed latencies for example on next slide.

- Loads & stores may or may not block on issue
  - > 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

Example

a ← a * 2 * b * c * d

Simple schedule

Start	Operations
1	loadAI r_{arp},@a ⇒ r_1
4	add r_1,r_1 ⇒ r_1
5	loadAI r_{arp},@b ⇒ r_2
8	mult r_1,r_2 ⇒ r_1
10	loadAI r_{arp},@c ⇒ r_2
13	mult r_1,r_2 ⇒ r_1
15	loadAI r_{arp},@d ⇒ r_2
18	mult r_1,r_2 ⇒ r_1
20	storeAI r_1 ⇒ r_{arp},@a

Schedule loads early

Start	Operations
1	loadAI r_{arp},@a ⇒ r_1
2	loadAI r_{arp},@b ⇒ r_2
3	loadAI r_{arp},@c ⇒ r_3
4	add r_1,r_1 ⇒ r_1
5	mult r_1,r_2 ⇒ r_1
6	loadAI r_{arp},@d ⇒ r_2
7	mult r_1,r_3 ⇒ r_1
9	mult r_1,r_2 ⇒ r_1
11	storeAI r_1 ⇒ r_{arp},@a

Reordering operations for speed is called *instruction scheduling*

ALU Characteristics

PACE platform-aware compilation environment

This data is surprisingly hard to measure accurately

- Value-dependent behavior
- Context-dependent behavior
- Compiler behavior
  - Have seen gcc underallocate & inflate operation costs with memory references (spills)
  - Have seen commercial compiler generate 3 extra ops per divide raising effective cost by 3
- Difficult to reconcile measured reality with the data in the Manuals (e.g. integer divide on Nehalem)

Intel E5530 operation latencies	
Instruction	**Cost**
64 bit integer subtract	1
64 bit integer multiply	3
64 bit integer divide	41
Double precision add	3
Double precision subtract	3
Double precision multiply	5
Double precision divide	22
Single precision add	3
Single precision subtract	3
Single precision multiply	4
Single precision divide	14

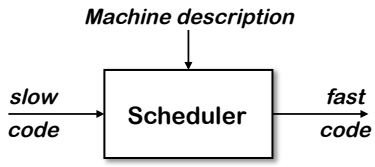
Xeon E5530 uses the Nehalem microarchitecture, as does I7

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

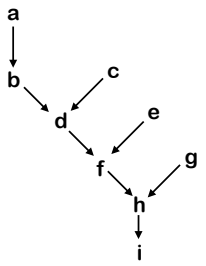
Instruction Scheduling (The Abstract View)

To capture properties of the code, build a precedence graph *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   r_arp,@a ⇒ r1
b:  add      r1,r1    ⇒ r1
c:  loadAI   r_arp,@b ⇒ r2
d:  mult     r1,r2    ⇒ r1
e:  loadAI   r_arp,@c ⇒ r3
f:  mult     r1,r2    ⇒ r1
g:  loadAI   r_arp,@d ⇒ r2
h:  mult     r1,r2    ⇒ r1
i:  storeAI  r1       ⇒ r_arp,@a
```

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, 1 cycle at a time
   a. Use a queue of operations that are ready
   b. At each cycle
      I. Choose the highest priority ready operation & schedule it
      II. Update the ready queue

Local list scheduling

- The dominant algorithm for thirty 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

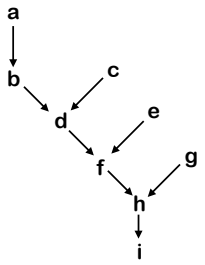
If successor's operands are "ready", add it to Ready

Scheduling Example

1. Build the precedence graph

```
a:  loadAI   r_arp,@a ⇒ r1
b:  add      r1,r1    ⇒ r1
c:  loadAI   r_arp,@b ⇒ r2
d:  mult     r1,r2    ⇒ r1
e:  loadAI   r_arp,@c ⇒ r3
f:  mult     r1,r2    ⇒ r1
g:  loadAI   r_arp,@d ⇒ r2
h:  mult     r1,r2    ⇒ r1
i:  storeAI  r1       ⇒ r_arp,@a
```

The Code



The Precedence Graph

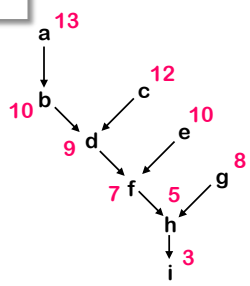
Scheduling Exa

1. Build the prece
2. Determine prior weighted path

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

```
a:  loadAI   r_arp,@a ⇒ r1
b:  add      r1,r1    ⇒ r1
c:  loadAI   r_arp,@b ⇒ r2
d:  mult     r1,r2    ⇒ r1
e:  loadAI   r_arp,@c ⇒ r3
f:  mult     r1,r2    ⇒ r1
g:  loadAI   r_arp,@d ⇒ r2
h:  mult     r1,r2    ⇒ r1
i:  storeAI  r1       ⇒ r_arp,@a
```

The Code



The Precedence Graph

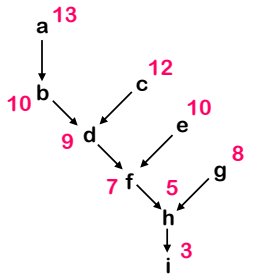
Scheduling Example

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

Used a new register name

```
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

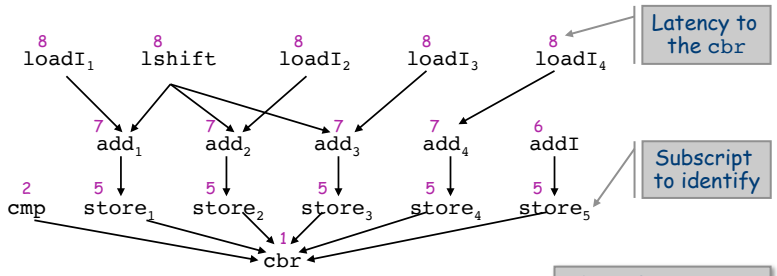
Forward list scheduling	Backward list scheduling
• Start with available operations	• Start with no successors
• Work forward in time	• Work backward in time
• Ready ⇒ all operands available	• Ready ⇒ latency covers uses

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

Local Scheduling

Forward and backward can produce different results



Block from SPEC benchmark "go"

Operation	load	loadI	add	addI	store	cmp
Latency	1	1	2	1	4	1

Local Scheduling

Forward Schedule

	Int	Int	Mem
1	$loadI_1$	lshift	
2	$loadI_2$	$loadI_3$	
3	$loadI_4$	add_1	
4	add_2	add_3	
5	add_4	addI	$store_1$
6	cmp		$store_2$
7			$store_3$
8			$store_4$
9			$store_5$
10			
11			
12			
13	cbr		

Backward Schedule

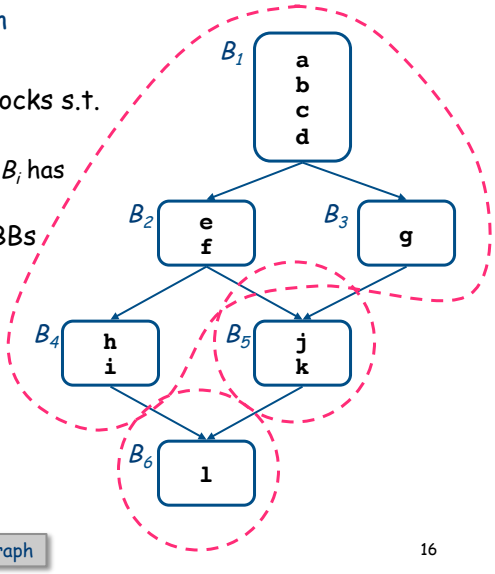
	Int	Int	Mem
1	$loadI_4$		
2	addI	lshift	
3	add_4	$loadI_3$	
4	add_3	$loadI_2$	$store_5$
5	add_2	$loadI_1$	$store_4$
6	add_1		$store_3$
7			$store_2$
8			$store_1$
9			
10			
11	cmp		
12	cbr		

Using "latency to root" as the priority function

Scheduling Larger Regions

One step beyond a block is an Extended Basic Block (EBB)

- EBB is a maximal set of blocks s.t.
  - Set has a single entry, B_i
  - Each block B_j other than B_i has exactly one predecessor
- Example CFG has three EBBs

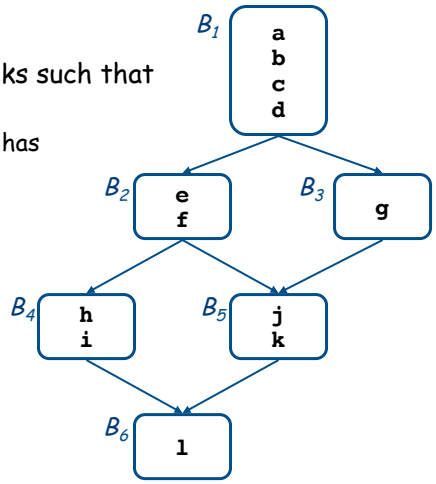


 CFG = Control Flow Graph

Scheduling Larger Regions

One step beyond a block is an Extended Basic Block (EBB)

- EBB is a maximal set of blocks such that
  - Set has a single entry, B_i
  - Each block B_j other than B_i has exactly one predecessor
- Example has three EBBs
  - Big EBB has two paths
  - $\{B_1,B_2,B_4\}$ & $\{B_1,B_3\}$
- Many optimizations operate on EBBs (including scheduling)



Scheduling Larger Regions

Superlocal Scheduling

- Schedule entire paths through EBBs
- Example has four EBB paths