Performance Evaluation

CS 0447

Tale of the Two Multipliers

- Slow shift-adder multiplier
  - **Suppose, 3 distinct steps**:
    (1) add, (2) shift left, & (3) shift right
  - **N=32 bits**, and each operand is 32 bits
    - Thus, result of multiply is 64 bits
    - And, the adder (ALU) is 64 bits

- Suppose **64-bit addition takes 10ns**
  - Each distinct step will take this same time

Tale of the Two Multipliers

- Total time to do multiplication

- Slow shift add multiplier:

  32 bits × steps each = **96 steps**

  96 steps × 10 ns per step = 960 ns

Tale of the Two Multipliers

- Fast shift-add multiplier improves
  1. Combines some registers
  2. Inherently does the 3 steps simultaneously
  3. Needs only a 32 bit adder

- Assuming linear relationship between bits and adder speed, then:
  - **32-bit add latency = 10 ns 64-bit add / 2 = 5 ns**

Tale of the Two Multipliers

- Total time to do multiplication

- Fast shift add multiplier:

  32 bits × 1 step each (combined) = **32 steps**

  32 steps × 5 ns per step = 160 ns

Tale of the Two Multipliers

- How much faster is the multiply hardware?
  - Compute "***speedup***" ratio: The factor by which the new version is faster than the old one
  - Speedup = Slow multiply time / Fast multiply time
  - Speedup = 960 ns / 160 ns = 6 times faster!
- ***Will a "real" program see a 6x improvement?***
  - Depends on how much the multiply is used
  - Never used: No speedup!

A Simple Program

```
        li    $1,100
L0:     lw    $2,A[i]       ; pseudo-code to load A[i]
        lw    $3,B[i]       ; pseudo-code to load B[i]
        mult  $3,$2
        mflo  $4
        sw    $4,C[i]       ; pseudo-code to store C[i]
        addi  $1,$1,-1
        bne   $1,$0,L0
```

How many times does this loop execute? **100**

How many instructions are executed?

1 (li) + 100 * 6 (non-multiply) + 100 * 1 (multiply) = 701 instructions

Execution Time

- Execution time is how long ("latency") it takes to execute the program.
- What's the ***time with the slow shift-multiply***?

time = 1 × 10ns + 6 × 100 × 10 ns + 1 × 100 × 960 ns
= 10 ns + 6,000 ns + 96,000 ns
= 102,010ns

- ***Time with the fast shift multiply***?

time = 1 × 10 ns + 6 × 100 × 10 ns + 1 × 100 × 160 ns
= 10 ns + 6,000 ns + 16,000 ns
= 22,010 ns

Execution Time

- Execution time is how long ("latency") it takes to execute the program.
- What's the ***time with the slow shift-multiply***?

time = 1 × 10ns + 6 × 100 × 10 ns + 1 × 100 × 960 ns

= 10 ns + 6,000 ns + 96,000 ns

= 102,010ns

- ***Time with the fast shift multiply***?

time = 1 × 10 ns + 6 × 100 × 10 ns + 1 × 100 × 160 ns

= 10 ns + 6,000 ns + 16,000 ns

= 22,010 ns

Speedup of Multipliers

- What's the speedup of the program with the faster multiplier over the slower one?

Speedup = time of slow / time of fast

= 102,010 ns / 22,010 ns

= ***4.63 speedup***

It's not a 6x speedup. Why?

Hint: Is the multiplier always used by the program?

Speedup of Multipliers

- Indeed, what happens as we decrease proportion of execution time on multiply?

Suppose 101 instructions: 100 non-multiply, 1 multiply

Time of slow: 100 × 10 ns + 960 ns = 1960 ns

Time of fast: 100 × 10 ns + 160 ns = 1160ns

Speedup: 1960 ns / 1160 ns = ***1.7***

Suppose 1001 instructions: 1000 non-multiply, 1 multiply

Time of slow: 1000 × 10 ns + 960 ns = 10960ns

Time of fast: 1000 × 10 ns + 160 ns = 10160 ns

Speedup = 10960 ns / 10160 ns = ***1.08***

Hmmm....

- **6x ➔ 4.63x ➔ 1.7x ➔ 1.08x** Interesting.
- What happened?
  - Proportion of time on multiply wasn't enough to realize the gains from it
- Thus, we must be careful. ***Strike a balance.***
  - Look for "bottleneck" for the common case
  - Improve it to a point
  - There's a diminishing return! Be careful.

Now, let's consider...

- How much faster in practice is the pipelined version of MIPS than the multi-cycle one?
- As before, we can compute speedup:

  Speedup = Time on slow / Time on fast

- But how do we compute "time"?
  - Concept of **CPU time**: Execution time of a program when run on the processor.

CPU Time

- ***CPU clock cycles = Instructions for a program × Average CPI***
- CPI = Clock **C**ycles **p**er **I**nstruction for an average instruction

- **Classic CPU Performance Equation:**

  CPU time = Instruction count × CPI × Clock cycle time

  = IC × CPI × CC

  or, rewritten:

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock speed}}$$

Average Instruction (CPI)

- What's an "average instruction" (***CPI***)?
- Given a program, how many ***cycles*** does an instruction ***typically*** take?
  - Depends on how many instructions, what types
  - E.g., all adds vs. all loads for multi-cycle impl.
  - It is just an average cycle count per instruction

Instruction Mix

- **Instruction mix**: % total instruction count (IC) corresponding to each instruction class
- **Program A**: 100 adds, 100 subtracts, 50 loads, 25 stores, 50 branches, and 10 jumps

  Instruction Count = 100 + 100 + 50 + 25 + 50 + 10 = 335

- Thus, the ***mix*** (% of each class):

Arithmetic	(100+100) / 335 =	0.597 =	59.7%
Load	50 / 335 =	0.149 =	14.9%
Store	25 / 335 =	0.075 =	7.5%
Branch	50 / 335 =	0.149 =	14.9%
Jump	10 / 335 =	0.03 =	3.0%

Cycles Per Instruction

Average Cycles Per Instruction (CPI)

Computed as weighted average

CPI = sum for all *class i* of *freq i* * *cycles i*

Class	**freq i**	**cycles i**
Arithmetic	59.7%	4
Load	14.9%	5
Store	7.5%	4
Branch	14.9%	3
Jump	3.0%	3

Multicycle Impl.

Cycles Per Instruction

Average Cycles Per Instruction (CPI)

Computed as weighted average

CPI = sum for all *class i* of *freq i* * *cycles i*

Class	**freq i**		**cycles i**		**contribution**
Arithmetic	59.7%	×	4	=	2.388
Load	14.9%	×	5	=	0.745
Store	7.5%	×	4	=	0.3
Branch	14.9%	×	3	=	0.447
Jump	3.0%	×	3	=	0.09

Cycles Per Instruction

Average Cycles Per Instruction (CPI)

Computed as weighted average

CPI = sum for all *class i* of *freq i* * *cycles i*

Class	**freq i**		**cycles i**		**contribution**
Arithmetic	59.7%	×	4	=	2.388
Load	14.9%	×	5	=	0.745
Store	7.5%	×	4	=	0.3
Branch	14.9%	×	3	=	0.447
Jump	3.0%	×	3	=	0.09
	100%				**3.97 CPI**

CPU Time

- **CPU time = IC ✕ CPI ✕ Cycle Length**
- Suppose Multicycle Cycle Length is 2 ns
- Then, the CPU time for program A is:

CPU time = IC ✕3.97 ✕ 2ns

= 7.94 ns ✕ IC

Or, IC=335: 335 ✕ 3.97 ✕ 2ns = 2,660 ns

An Example

- Suppose cycle time is **1 ns**
- Multi-cycle takes how many cycles for each?
  - Arithmetic **4**
  - Load **5**
  - Store **4**
  - Branch **3**

An Example (Cont)

```
.data
A:  .word 10,20,30,40,50,60,70,80,90
B:  .word  0, 0, 0, 0, 0, 0, 0, 0, 0
.text
        li      $t0,10          # 1 instruction
        la      $t1,A           # 2 instructions
loop:   lw      $t3,0($t1)# executed 10 times, 10 loads
        add     $t3,$t3,$t3
        add     $t3,$t3,$t3
        sw      $t3,40($t1)   # executed 10 times, 10 stores
        addi    $t1,$t1,4
        addi    $t0,$t0,-1# 4 adds/iteration * 10 = 40 adds
        bne     $t0,$0,loop   # executed 10 times, 10 branches
        li      $v0,10          # 1 instruction
        syscall                 # 1 instruction
```

An Example

- ***Now, for the multi-cycle example, answer the following:***

  What is the instruction mix?

  What is the CPI?

  What is the CPU execution time?

Multi-cycle Example

- Instruction mix: Determine IC, and then proportion of IC for each type

Class	Frequency	Cycles
arithmetic	45 / 75 = 0.6	4
load	10 / 75 = 0.13	5
stores	10 / 75 = 0.13	4
branch	10 / 75 = 0.14 (rounding)	3

- What is the CPI?

CPImulti $= 0.6 \times 4 + 0.13 \times 5 + 0.13 \times 4 + 0.14 \times 3$

$= 2.4 + 0.65 + 0.52 + 0.42$

$= 3.99$ cycles

Multi-cycle Example

- What is the CPU execution time?

CPU time multi = ICmulti × CPImulti × CCmulti

ICmulti = 75

CPImulti = 3.99 (computed previous slide)

CCmulti = 1ns (given)

Thus, CPU time multi = ICmulti × CPImulti × CCmulti

= 75 × 3.99 × 1ns

= 299.25ns

An Example: Pipeline Version!

- ***Consider the same program but execute it on a pipelined processor.***
- In the best case, what is the CPI?
- In the typical case, what is the CPI?
  - Say, we filled 20% of delay slots (no delay!)
  - 60% of loads have a load-use delay of 1 cycle
- Assuming each stage takes 1ns, what is the CPU execution time for pipelied?

Pipelined Speedup

- Instruction mix: Treat the load-use and branch delays as separate instruction classes

Class	Frequency	Cycles
arithmetic	45 / 75 = **0.6**	1
load – no delay	((100% - 60%) × 10) / 75 = **0.05**	1
load – delayed	(60% × 10) / 75 = **0.08**	2
stores	10 / 75 = **0.13**	1
branch – filled slot	(20% × 10) / 75 = **0.03**	1
branch – unfilled slot	(80% × 10) / 75 = **0.11**	2

Pipelined Speedup

Class	Frequency	Cycles
arithmetic	45 / 75 = **0.6**	1
load – no delay	((100% - 60%) × 10) / 75 = **0.05**	1
load – delayed	(60% × 10) / 75 = **0.08**	2
stores	10 / 75 = **0.13**	1
branch – filled slot	(20% × 10) / 75 = **0.03**	1
branch – unfilled slot	(80% × 10) / 75 = **0.11**	2

Using the mix, compute the CPI of the pipelined implementation:

CPIpipe = 0.6×1 + 0.05×1 + **0.08×2** + 0.13×1 + 0.03×1 + **0.11×2**

= 0.6 + 0.05 + 0.16 + 0.13 + 0.03 + 0.22

= 1.19

Pipelined Speedup

- Compute CPU execution time of pipelined implementation
- CPU time pipe = ICpipe × CPIpipe × CCpipe
- ICpipe is just 75, same instruction count as multicycle!
  - ***Property of the program*** – not of the hw implementation
- CPIpipe computed (previous slide) as 1.19
- Assuming the same cycle time (1ns) as multicycle:

CPU time pipe = 75 × 1.19 × 1 ns

= **89.25 ns**

Speedup of pipe = CPU time multi / CPU time pipe

= 299.25 ns / 89.25 ns

= **3.35x**

Comparison

- Speedup of pipelined implementation over the multicycle?

  Speedup = CPU time multi / CPU time pipe
  = (IC × CPImulti × CCmulti) / (IC × CPIpipe × CCpipe)
  = (75 × **3.99 × 1ns**) / (75 × **1.19 × 1ns**)
  = 3.99 / 1.19
  = 3.35x

- Speedup of pipelined implementation over single cycle?

  Speedup = CPU time single / CPU time pipe
  = (IC × CPIsingle × CCsingle) / (IC × CPIpipe × Ccpipe)
  = (75 × **1 × 5ns**) / (75 × 1.19 × 1ns)
  = 5 / 1.19
  =4.2x

Note single cycle CPI is always 1! ☺

The three implementations

CPU time = IC × CPI × CC

For same instruction set (IC same):

Single cycle: CPI = 1, long CC

Multi cycle: CPI>1, probably 3-4, short CC

Pipelined: CPI>1, probably 1.2-1.4, short CC

IC is affected by the program (what instructions executed)

CPI is affected by the program and the implementation

CC is affected by the implementation

Fundamentally, these are the “three knobs” that we can control when we design a processor.