The CPU Performance Equation

The Performance Equation (PE)

- We would like to model how architecture impacts performance (latency)
- This means we need to quantify performance in terms of architectural parameters.
  - Instruction Count -- The number of instructions the CPU executes
  - Cycles per instructions -- The ratio of cycles for execution to the number of instructions executed.
  - Cycle time -- The length of a clock cycle in seconds
- The first fundamental theorem of computer architecture:

Latency = Instruction Count * Cycles/Instruction * Seconds/Cycle

L = IC * CPI * CT

The PE as Mathematical Model

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

- Good models give insight into the systems they model
  - Latency changes linearly with IC
  - Latency changes linearly with CPI
  - Latency changes linearly with CT
- It also suggests several ways to improve performance
  - Reduce CT (increase clock rate)
  - Reduce IC
  - Reduce CPI
- It also allows us to evaluate potential trade-offs
  - Reducing cycle time by 50% and increasing CPI by 1.5 is a net win.

Reducing Cycle Time

- Cycle time is a function of the processor’s design
  - If the design does less work during a clock cycle, it’s cycle time will be shorter.
  - More on this later, when we discuss pipelining.
- Cycle time is a function of process technology.
  - If we scale a fixed design to a more advanced process technology, it’s clock speed will go up.
  - However, clock rates aren’t increasing much, due to power problems.
- Cycle time is a function of manufacturing variation
  - Manufacturers “bin” individual CPUs by how fast they can run.
  - The more you pay, the faster your chip will run.

The Clock Speed Corollary

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

- We use clock speed more than second/cycle
- Clock speed is measured in Hz (e.g., MHz, GHz, etc.)
  - x Hz => 1/x seconds per cycle
  - 2.5GHz => $1/2.5 \times 10^9$ seconds (0.4ns) per cycle

Latency = (Instructions * Cycle/Insts)/(Clock speed in Hz)

A Note About Instruction Count

- The instruction count in the performance equation is the “dynamic” instruction count
- “Dynamic”
  - Having to do with the execution of the program or counted at run time
  - ex: When I ran that program it executed 1 million dynamic instructions.
- “Static”
  - Fixed at compile time or referring to the program as it was compiled
  - e.g.: The compiled version of that function contains 10 static instructions.

Reducing Instruction Count (IC)

- There are many ways to implement a particular computation
  - Algorithmic improvements (e.g., quicksort vs. bubble sort)
  - Compiler optimizations (e.g., pass -O4 to gcc)
- If one version requires executing fewer dynamic instructions, the PE predicts it will be faster
  - Assuming that the CPI and clock speed remain the same
  - A x% reduction in IC should give a speedup of
  - 1/(1-0.01*x) times
  - e.g., 20% reduction in IC => 1/(1-0.2) = 1.25x speedup

Example: Reducing IC

```
int i, sum = 0;
for(i=0;i<10;i++)
    sum += i;
```

- No optimizations
- All variables are on the stack.
- Lots of extra loads and stores
- 13 static insts
- 112 dynamic insts

```
    sw  0($sp), $zero#sum = 0
    sw  4($sp), $zero#i = 0
loop:
    lw  $s1, 4($sp)
    nop
    sub $s3, $s1, 10
    beq $s3, $s0, end
    lw  $s2, 0($sp)
    nop
    add  $s2, $s2, $s1
    st  0($sp), $s2
    addi $s1, $s1, 1
    b    loop
    st  4($sp), $s1 #br delay
end:
```

Example: Reducing IC

```
int i, sum = 0;
for(i=0;i<10;i++)
    sum += i;
```

```
        ori  $t1, $zero, 0 # i
        ori  $t2, $zero, 0 # sum
loop:
        sub  $t3, $t1, 10
        beq  $t3, $t0, end
        nop
        add  $t2, $t2, $t1
        b    loop
        addi $t1, $t1, 1
end:
        sw   $t2, 0($sp)
```

- Same computation
- Variables in registers
- Just 1 store
- 9 static insts

What's the speedup of B vs A?

```
    sw    0($sp), $zero#sum = 0
    sw    4($sp), $zero#i = 0
loop:
    lw    $s1, 4($sp)
    nop
    sub   $s3, $s1, 10
    beq   $s3, $s0, end
    lw    $s2, 0($sp)
    nop
    add   $s2, $s2, $s1
    st    0($sp), $s2
    addi  $s1, $s1, 1
    b      loop
    st    4($sp), $s1 #br delay
end:
```

A

```
int i, sum = 0;
for(i=0;i<10;i++)
    sum += i;
```

```
    ori   $t1, $zero, 0 # i
    ori   $t2, $zero, 0 # sum
loop:
    subi  $t3, $t1, 10
    beq   $t3, $t0, end
    nop
    add   $t2, $t2, $t1
    b      loop
    addi  $t1, $t1, 1
end:
    sw    $t2, 0($sp)
```

B

13 static insts

112 dynamic insts

Speed of B vs A

$\frac{112}{63} = \frac{L_A}{L_B} = 1.8$

	B dyn. Insts	Speedup
A	9	1.4
B	9	12.4
C	60	0.22
D	63	1.8
E	9	1.8

Other Impacts on Instruction Count

- Different programs do different amounts of work
  - e.g., Playing a DVD vs writing a word document
- The same program may do different amounts of work depending on its input
  - e.g., Compiling a 1000-line program vs compiling a 100-line program
- The same program may require a different number of instructions on different ISAs
  - We will see this later with MIPS vs. x86
- To make a meaningful comparison between two computer systems, they must be doing the same work.
  - They may execute a different number of instructions (e.g., because they use different ISAs or a different compilers)
  - But the task they accomplish should be exactly the same.

Cycles Per Instruction

- CPI is the most complex term in the PE, since many aspects of processor design impact it
  - The compiler
  - The program’s inputs
  - The processor’s design (more on this later)
  - The memory system (more on this later)
- It is not the cycles required to execute one instruction
- It is the ratio of the cycles required to execute a program and the IC for that program. It is an average.

Instruction Mix and CPI

- Instruction selections (and, therefore, instruction selection) impacts CPI because some instructions require extra cycles to execute
- All theses values depend on the particular implementation, not the ISA.

Instruction Type	Cycles
Integer +, -, |, &, branches	1
Integer multiply	3-5
integer divide	11-100
Floating point +, -, *, etc.	3-5
Floating point /, sqrt	7-27
Loads and Stores	1-100s

These values are for Intel’s Nehalem processor

Practice: Reducing CPI

```
int i, sum = 0;
for(i=0;i<10;i++)
    sum += i;
```

Type	CPI	Static #	Dyn#
mem	5	6	44
int	1	5	52
br	1	2	21
Total	2.5	13	117

Average CPI:

(5*44+ 1*52+ 1*21)/117= 2.504

```
    sw  0($sp), $zero#sum = 0
    sw  4($sp), $zero#i = 0
loop:
    lw  $s1, 4($sp)
    nop
    addi $s3, $s1,-10
    beq $s3, $zero, end
    lw  $s2, 0($sp)
    nop
    add  $s2, $s2, $s1
    st  0($sp), $s2
    addi $s1, $s1, 1
    b    loop
    st  4($sp), $s1
end:
```

Practice: Reducing CPI

```
int i, sum = 0;
for(i=0;i<10;i++)
    sum += i;
```

Type	CPI	Static #	Dyn#
mem	5	1	1
int	1	6	44
br	1	2	21
Total	???	9	66

Previous CPI = 2.5

```
    ori  $t1, $zero, 0 # i
    ori  $t2, $zero, 0 # sum
loop:
    addi $t3, $t1, -10
    beq  $t3, $zero, end
    nop
    add  $t2, $t2, $t1
    b    loop
    addi $t1, $t1, 1
end:
    sw   $t2, 0($sp)
```

	New CPI	Speedup
A	1.44	1.74
B	1.06	0.42
C	2.33	1.07
D	1.44	0.58
E	1.06	2.36

Example: Reducing CPI

```
int i, sum = 0;
for(i=0;i<10;i++)
    sum += i;
```

```
    ori  $t1, $zero, 0 # i
    ori  $t2, $zero, 0 # sum
loop:
    sub  $t3, $t1, 10
    beq  $t3, $t0, end
    nop
    add  $t2, $t2, $t1
    b    loop
    addi $t1, $t1, 1
end:
    sw   $t2, 0($sp)
```

Type	CPI	Static #	Dyn#
mem	5	1	1
int	1	6	44
br	1	2	21
Total	1.06	9	66

Average CPI:

(5*1 + 1*42 + 1*20)/66= 1.06

- Average CPI reduced by 57.6%
- Speedup projected by the PE: 2.36x.

Reducing CPI & IC Together

```
        sw      0($sp), $zero#sum = 0
        sw      4($sp), $zero#i = 0
loop:
        lw      $s1, 4($sp)
        nop
        sub     $s3, $s1, 10
        beq     $s3, $s0, end
        lw      $s2, 0($sp)
        nop
        add     $s2, $s2, $s1
        st      0($sp), $s2
        addi    $s1, $s1, 1
        b        loop
        st      4($sp), $s1 #br delay
end:
```

Unoptimized Code (UC)

IC: 112

CPI: 2.5

$L_{UC} = IC_{UC} * CPI_{UC} * CT_{UC}$

```
        ori   $t1, $zero, 0 # i
        ori   $t2, $zero, 0 # sum
loop:
        sub   $t3, $t1, 10
        beq   $t3, $t0, end
        nop
        add   $t2, $t2, $t1
        b     loop
        addi  $t1, $t1, 1
end:
        sw    $t2, 0($sp)
```

Optimized Code (OC)

IC: 63

CPI: 1.06

$L_{OC} = IC_{OC} * CPI_{OC} * CT_{OC}$

	Total speedup
A	3.56
B	4.19
C	4.14
D	1.78
E	Can't tell. Need to know the cycle time.

Reducing CPI & IC Together

```
        sw      0($sp), $zero#sum = 0
        sw      4($sp), $zero#i = 0
loop:
        lw      $s1, 4($sp)
        nop
        sub     $s3, $s1, 10
        beq     $s3, $s0, end
        lw      $s2, 0($sp)
        nop
        add     $s2, $s2, $s1
        st      0($sp), $s2
        addi    $s1, $s1, 1
        b        loop
        st      4($sp), $s1 #br delay
end:
```

Unoptimized Code (UC)
IC: 112
CPI: 2.5

$L_{UC} = IC_{UC} * CPI_{UC} * CT_{UC}$

$L_{UC} = 112 * 2.5 * CT_{UC}$

```
        ori   $t1, $zero, 0 # i
        ori   $t2, $zero, 0 # sum
loop:
        sub   $t3, $t1, 10
        beq   $t3, $t0, end
        nop
        add   $t2, $t2, $t1
        b     loop
        addi  $t1, $t1, 1
end:
        sw    $t2, 0($sp)
```

Optimized Code (OC)
IC: 63
CPI: 1.06

$L_{OC} = IC_{OC} * CPI_{OC} * CT_{OC}$

$L_{OC} = 63 * 1.06 * CT_{OC}$

$$\text{Speed up} = \frac{112 * 2.5 * CT_{UC}}{63 * 1.06 * CT_{OC}} = 4.19x = \frac{112}{63} * \frac{2.5}{1.06}$$

Since hardware is unchanged, CT is the same and cancels

Program Inputs and CPI

- Different inputs make programs behave differently
  - They execute different functions
  - They branches will go in different directions
  - These all affect the instruction mix (and instruction count) of the program.

Amdahl’s Law

Amdahl's Law

- The fundamental theorem of performance optimization
- Made by Amdahl!
  - One of the designers of the IBM 360
  - Gave "FUD" it's modern meaning

- Optimizations do not (generally) uniformly affect the entire program
  - The more widely applicable a technique is, the more valuable it is
  - Conversely, limited applicability can (drastically) reduce the impact of an optimization.

Always heed Amdahl's Law!!!

It is central to many many optimization problems

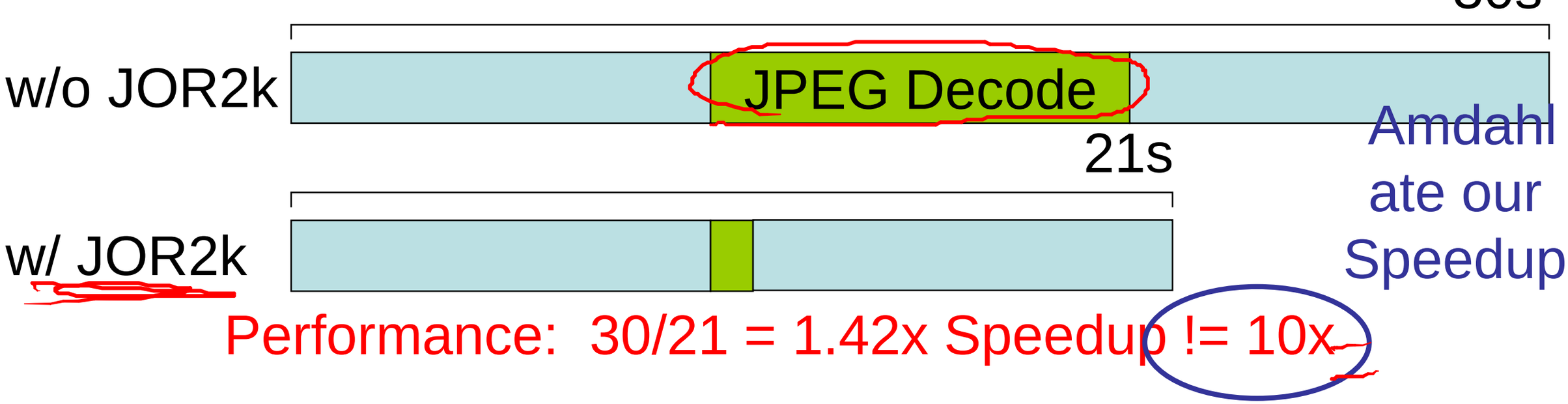
Amdahl’s Law in Action

- SuperJPEG-O-Rama2010 ISA extensions **
  - —Speeds up JPEG decode by 10x!!!
  - —Act now! While Supplies Last!

** SuperJPEG-O-Rama Inc. makes no claims about the usefulness of this software for any purpose whatsoever. It may not even build. It may cause fatigue, blindness, lethargy, malaise, and irritability. Debugging maybe hazardous. It will almost certainly cause ennui. Do not taunt SuperJPEG-O-Rama. Will not, on grounds of principle, decode images of Justin Beiber. Images of Lady Gaga maybe transposed, and meat dresses may be rendered as tofu. Not covered by US export control laws or the Geneva convention, although it probably should be. Beware of dog. Increases processor cost by 45%. Objects in the rear view mirror may appear closer than they are. Or is it farther? Either way, watch out! If you use SuperJPEG-O-Rama, the cake will not be a lie. All your base are belong to 141L. No whining or complaining. Wingeing is allowed, but only in countries where “wingeing” is a word.

Amdahl’s Law in Action

- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of it’s time doing JPEG decode
- How much does JOR2k help?



Is this worth the 45% increase in cost?

Amdahl’s Law

- The second fundamental theorem of computer architecture.
- If we can speed up *x* of the program by S times
- Amdahl’s Law gives the total speed up, S_{tot}

$$S_{tot} = \frac{1}{(x/S + (1-x))}$$

Sanity check:

$$x = 1 \Rightarrow S_{tot} = \frac{1}{(1/S + (1-1))} = \frac{1}{1/S} = S$$

Amdahl’s Law

- Protein String Matching Code
  - It runs for 200 hours on the current machine, and spends 20% of time doing integer instructions
  - How much faster must you make the integer unit to make the code run 1.1 times faster?

A) 1.13
B) 1.975
C) 0.022
D) 1.31
E) None of the above

$1.1 \quad S_{tot} = \dfrac{1}{\frac{x}{5} + (1 - \frac{x}{5})}$

$S_{tot} = 1.1$

$x = .2$

Amdahl’s Law Example #1

- Protein String Matching Code
  - It runs for 200 hours on the current machine, and spends 20% of time doing integer instructions
  - How much faster must you make the integer unit to make the code run 50 hours faster?

A) 10.0
B) 50.0
C) 1 million times
D) None of the above

Practice

Amdahl’s Law Example #2

- Protein String Matching Code
  - 4 days execution time on current machine
    - 20% of time doing integer instructions
    - 35% percent of time doing I/O
  - Which is the better tradeoff?
    - Compiler optimization that reduces number of integer instructions by 25% (assume each integer instruction takes the same amount of time)
    - Hardware optimization that reduces the latency of each IO operations from 6us to 5us.

Explanation

- Speed up integer ops
  - x = 0.2
  - S = 1/(1-0.25) = 1.33
  - S_{int} = 1/(0.2/1.33 + 0.8) = 1.052
- Speed up IO
  - x = 0.35
  - S = 6us/5us = 1.2
  - S_{io} = 1/(.35/1.2 + 0.65) = 1.062
- Speeding up IO is better

Amdahl’s Corollary

- Make the common case fast (i.e., x should be large)!
  - Common == “most time consuming” not necessarily “most frequent”
  - The uncommon case doesn’t make much difference
  - Be sure of what the common case is
  - The common case can change based on inputs, compiler options, optimizations you’ve applied, etc.
- Repeat…
  - With optimization, the common becomes uncommon.
  - An uncommon case will (hopefully) become the new common case.
  - Now you have a new target for optimization.

Amdahl’s Corollary #2: Example

Colors represent C functions

Common case

- 7x => 1.4x
- 4x => 1.3x
- 1.3x => 1.1x

Total = 20/10 = 2x

- In the end, there is no common case!
- What now?:
  - Global optimizations (faster clock, better compiler)
  - Divide the program up differently
    - e.g. Focus on classes of instructions (maybe memory or FP?), rather than code.
    - e.g. Focus on function call over heads (which are everywhere).

Amdahl’s Revenge

- Amdahl’s law does not bound slowdown
- Rewrite it for latency
  - newLatency = x*oldLatency/S + oldLatency*(1-x)
  - newLatency is linear in 1/S
- Example: x = 0.01 of execution, oldLat = 1
  - S = 0.001;
    - Newlat = 1000*Oldlat *0.01 + Oldlat *(0.99) = ~ 10*Oldlat
  - S = 0.00001;
    - Newlat = 100000*Oldlat *0.01 + Oldlat *(0.99) = ~ 1000*Oldlat
- Amdahl says: “Things can’t get that much better, but they sure can get worse.”

Bandwidth and Other Metrics

Bandwidth

- The amount of work (or data) per time
  - MB/s, GB/s -- network BW, disk BW, etc.
  - Frames per second -- Games, video transcoding
- Also called “throughput”

Latency-BW Trade-offs

- Often, increasing latency for one task can lead to increased BW for many tasks.
- Ex: Waiting in line for one of 4 bank tellers
  - If the line is empty, your latency is low, but utilization is low
  - If there is always a line, you wait longer (your latency goes up), but utilization is better (there is always work available for tellers)
  - Which is better for the bank? Which is better for you?
- Much of computer performance is about scheduling work onto resources
  - Network links.
  - Memory ports.
  - Processors, functional units, etc.
  - IO channels.
  - Increasing contention (i.e., utilization) for these resources generally increases throughput but hurts latency.

Reliability Metrics

- Mean time to failure (MTTF)
  - Average time before a system stops working
  - Very complicated to calculate for complex systems
- Why would a processor fail?
  - Electromigration
  - High-energy particle strikes
  - cracks due to heat/cooling
- It used to be that processors would last longer than their useful life time. This is becoming less true.

Power/Energy Metrics

- Energy == joules
  - You buy electricity in joules.
  - Battery capacity is in joules
  - To minimizes operating costs, minimize energy
  - You can also think of this as the amount of work that computer must actually do
- Power == joules/sec
  - Power is how fast your machine uses joules
  - It determines battery life
  - It is also determines how much cooling you need. Big systems need 0.3-1 Watt of cooling for every watt of compute.

The End