

# Measuring and Reasoning About Performance

Readings: 1.4-1.5

# Goals for this Class

- Understand how CPUs run programs
  - How do we express the computation the CPU?
  - How does the CPU execute it?
  - How does the CPU support other system components (e.g., the OS)?
  - What techniques and technologies are involved and how do they work?
- Understand why CPU performance varies
  - How does CPU design impact performance?
  - What trade-offs are involved in designing a CPU?
  - How can we meaningfully measure and compare computer performance?
- Understand why program performance varies
  - How do program characteristics affect performance?
  - How can we improve a programs performance by considering the CPU running it?
  - How do other system components impact program performance?

# Goals

- Understand and distinguish between computer performance metrics
  - Latency
  - Bandwidth
  - Various kinds of efficiency
  - Composite metrics
- Understand and apply the CPU performance equation
- Understand how applications and the compiler impact performance
- Understand and apply Amdahl's Law

# What do you want in a computer?

- Quietness (dB)
- Speed
- Preceived speed
  - Responsiveness
- Batter life
- Good lookin'
- Volume
- Dimensions
- Portability
  - Weight
  - Size
- Flexibility
- Reliability
- Expandability/Upgradability
- Workmanship
- Memory bandwidth
- Power consumption
- Good support
- Popularity/Facebook likes
- Thermal performance
- Display quality
- Is it a mac?
- Ergonomics
- FPS
  - Crysis metric
  - But at what Res?
- Sound quality
- Network speed
- Connectivity
  - USB 3.0
  - Thunderbolt
  - HDMI
  - Ethernet
  - Bluetooth
  - Floppy
- Warranty
- Storage capacity
- Storage speed
- Peripherals
  - quality
- bells and whistles
- Price!!!
- Awesome
  - Beiber

# What do you want in a computer?

- Power efficiency
- speed
  - Instruction throughput
  - Latency
  - FLOPS
- Reliability
- Security
- Memory capacity
- Fast memory
- Storage capacity
- Connectivity
- Easy-to-use
- Fully function keyboard
- Cooling capacity
- Heating
- User interface
- Blue lights
- Cool gadgets
- Frame rate
  - Crysis metric
- Weight
- Size
- Battery life
- dB
- Awesomeness
  - Beiber
- Coolness
  - Gaganess
- Extenpandability
- Software compatibility
- Cost

# Metrics

# Basic Metrics

- Latency or delay (Lower is better)
  - Complete a task as soon as possible
  - Measured in seconds, us, ns, clock cycles, etc.
- Throughput (Higher is better)
  - Complete as many tasks per time as possible
  - Measured in bytes/s, instructions/s, instructions/cycle
- Cost (Lower is better)
  - Complete tasks for as little money as possible
  - Measured in dollars, yen, etc.
- Power (Lower is better)
  - Complete tasks while dissipating as few joules/sec as possible
  - Measured in Watts (joules/sec)
- Energy (Lower is better)
  - Complete tasks using as few joules as possible
  - Measured in Joules, Joules/instruction, Joules/execution
- Reliability (Higher is better)
  - Complete tasks with low probability of failure
  - Measured in “Mean time to failure” MTTF -- the average time until a failure occurs.

# Example: Latency

- Latency is the most common metric in architecture
  - $\text{Speed} = 1/\text{Latency}$
  - Latency = Run time
  - “Performance” usually, but not always, means latency
- A measured latency is for some particular task
  - A CPU doesn't have a latency
  - An application has a latency on a particular CPU



# Where latency matters

- Application responsiveness
  - Any time a person is waiting.
  - GUIs
  - Games
  - Internet services (from the users perspective)
- “Real-time” applications
  - Tight constraints enforced by the real world
  - Anti-lock braking systems -- “hard” real time
  - Multi-media applications -- “soft” real time

# Ratios of Measurements

- We often want to compare measurements of two systems
  - e.g., the speedup of CPU A vs CPU B
  - e.g., the battery life of laptop X vs Laptop Y
- The terminology around these comparisons can be confusing.
- For this class, these are equivalent
  - $V_{\text{new}} = 2.5 * V_{\text{old}}$
  - A metric increased by 2.5 times (sometimes written 2.5x, “2.5 ex”)
  - A metric increased by 150% ( $x\%$  increase  $\Rightarrow 0.01 * x + 1$  times increase)
- And these
  - $V_{\text{new}} = V_{\text{old}} / 2.5$
  - A metric decreased by 2.5x (Deprecated. It’s confusing)
  - A metric decreased by 60% ( $x\%$  decrease  $\Rightarrow (1 - 0.01 * x)$  times increase)
  - A metric increased by 0.4 times
- For bigger-is-better metrics, “improved” means “increase”; for smaller-is-better metrics, “improved” means “decrease”. Likewise, for “worsened,” “was degraded,” etc.
  - e.g., Latency improved by 2x, means latency decreased by 2x (i.e., dropped by 50%)
  - e.g., Battery life worsened by 50%, means battery life decrease by 50%.

# Example: Speedup

- Speedup is the ratio of two latencies
  - $\text{Speedup} = \text{Latency}_{\text{old}} / \text{Latency}_{\text{new}}$
  - $\text{Speedup} > 1$  means performance increased
  - $\text{Speedup} < 1$  means performance decreased
- If machine A is 2x faster than machine B
  - $\text{Latency}_A = \text{Latency}_B / 2$
  - The speedup of B relative to A is  $1/2x$  or  $0.5x$ .
- Speedup (and other ratios of metrics) allows the comparison of two systems without reference to an absolute unit
  - We can say “doubling the clock speed will give 2x speedup” without knowing anything about a concrete latency.
  - It’s much easier than saying “If the program’s latency was 1,254 seconds, doubling the clock rate would reduce the latency to 627 seconds.”

# Derived metrics

- Often we care about multiple metrics at once.
- Examples (Bigger is better)
  - Bandwidth per dollar (e.g., in networking (GB/s)/\$)
  - BW/Watt (e.g., in memory systems (GB/s)/W)
  - Work/Joule (e.g., instructions/joule)
  - In general: Multiply by big-is-better metrics, divide by smaller-is-better
- Examples (Smaller is better)
  - Cycles/Instruction (i.e., Time per work)
  - Latency \* Energy -- “Energy Delay Product”
  - In general: Multiply by smaller-is-better metrics, divide by bigger-is-better

# Example: Energy-Delay

- Mobile systems must balance latency (delay) and battery (energy) usage for computation.
- The energy-delay product (EDP) is a “smaller is better” metric
  - Base units: Delay in seconds; Energy in Joules;
  - EDP units: Joules\*seconds

# Example: Energy-Delay

- If we use EDP to evaluate design alternatives, the following designs are equally good
- One that reduces battery life by half and reduces delay by half
  - $E_{\text{new}} = 2 * E_{\text{base}}$
  - $D_{\text{new}} = 0.5 * D_{\text{base}}$
  - $D_{\text{new}} * E_{\text{new}} = 1 * D_{\text{old}} * E_{\text{old}}$
- One that increases delay by 100%, but doubles battery life.
  - $E_{\text{new}} = 0.5 * E_{\text{base}}$
  - $D_{\text{new}} = 2 * D_{\text{base}}$
  - $D_{\text{new}} * E_{\text{new}} = 1 * D_{\text{new}} * E_{\text{new}}$
- One that reduces delay by 25%, but increases energy consumption by 33%
  - $E_{\text{new}} = 1.33 * E_{\text{base}}$
  - $D_{\text{new}} = 0.75 * D_{\text{base}}$
  - $D_{\text{new}} * E_{\text{new}} = 1 * D_{\text{new}} * E_{\text{new}}$

# Example: Energy-Delay<sup>2</sup>

- Or we might care more about performance than energy
  - Multiply by delay twice:  $E * D^2$
- If we use  $ED^2$  to evaluate systems, the following are equally good
  - One that reduces battery life by half and reduces delay by 29%
    - $E_{\text{new}} = 2 * E_{\text{base}}$
    - $D_{\text{new}} = 0.71 * D_{\text{base}}$
    - $D_{\text{new}}^2 * E_{\text{new}} = 1 * D_{\text{new}}^2 * E_{\text{new}}$
  - One that increases delay by 100%, but quadruples battery life.
    - $E_{\text{new}} = 0.25 * E_{\text{base}}$
    - $D_{\text{new}} = 2 * D_{\text{base}}$
    - $D_{\text{new}}^2 * E_{\text{new}} = 1 * D_{\text{new}}^2 * E_{\text{new}}$
- You would like to reduce energy consumption by 1/2, without reducing  $ED^2$ . By what factor can delay increase?
  - $ED^2 = 0.5 * E * (x * D)^2$ ; Solve for  $x$
  - $x = \text{sqrt}(2)$

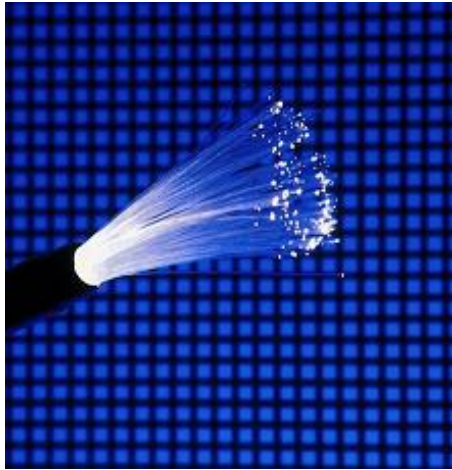
# What's the Right Metric?

- There is not universally correct metric
- You can use *any* metric you like to evaluate computer systems
  - Latency for gcc
  - Frames per second on Crysis
  - (Database transactions/second)/\$
  - $(\text{Power} * \text{CaseVolume}) / (\text{System weight} * \$)$
- The right metric depends on the situation.
  - What does the computer need to accomplish?
  - What constraints is it under?
- Usually some, relatively simple, combination of metrics on the “basic metrics” slide.
- We will mostly focus on performance (latency and/or bandwidth)



# The Internet “Land”-Speed Record

Fiber-optic cable



State of the art  
networking  
medium  
(sent 585 GB)

Latency (s)

1800

BW (GB/s)

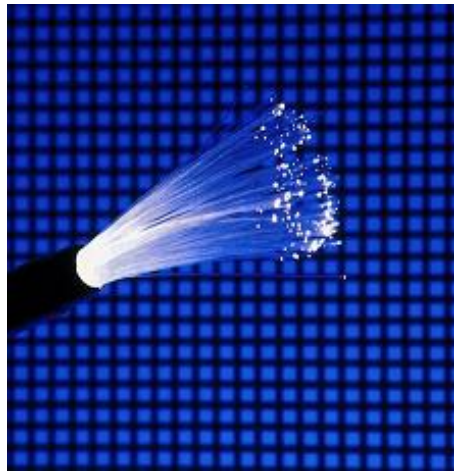
1.13

Tb-m/s

272,400

# The Internet “Land”-Speed Record

Fiber-optic cable



State of the art  
networking  
medium  
(sent 585 GB)

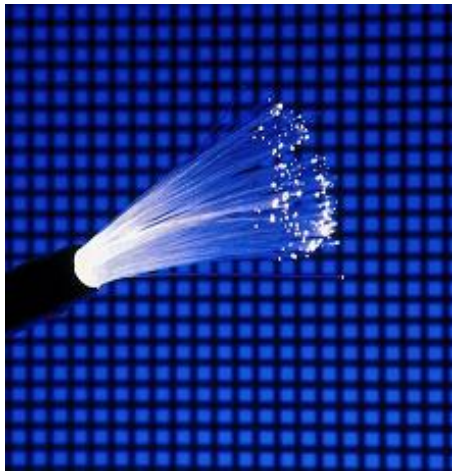
"Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway."

-- Andrew S. Tanenbaum

|             |         |
|-------------|---------|
| Latency (s) | 1800    |
| BW (GB/s)   | 1.13    |
| Tb-m/s      | 272,400 |

# The Internet “Land”-Speed Record

Fiber-optic cable



State of the art  
networking  
medium  
(sent 585 GB)



3.5 in Hard drive  
3 TB  
0.68 kg

Latency (s)

1800

BW (GB/s)

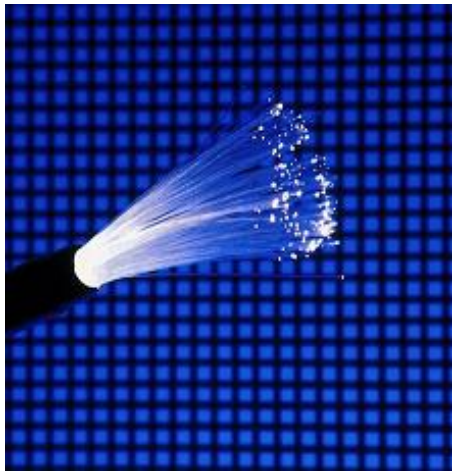
1.13

Tb-m/s

272,400

# The Internet “Land”-Speed Record

Fiber-optic cable



State of the art  
networking  
medium  
(sent 585 GB)

Cargo

Speed

Latency (s)

BW (GB/s)

Tb-m/s

1800

1.13

272,400

Subaru Outback

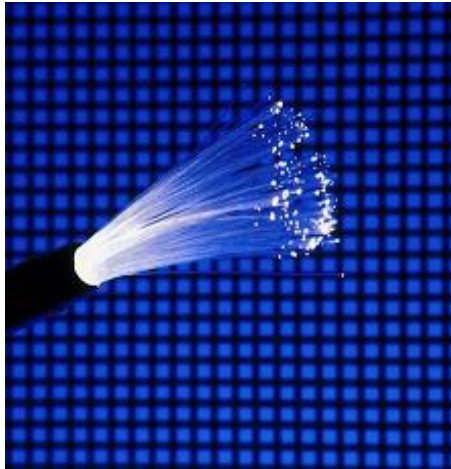



Sensible station  
wagon

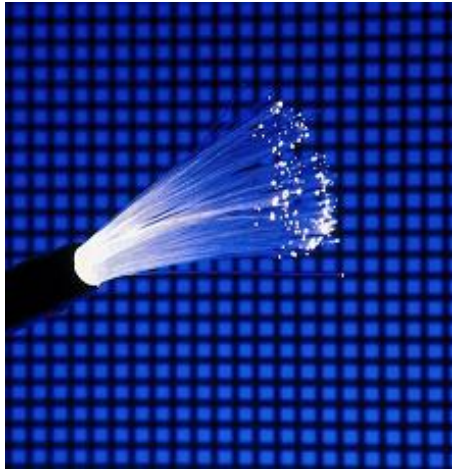

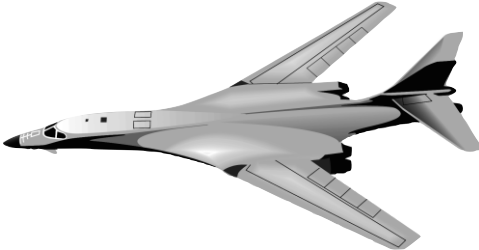
183 kg

119 MPH

# The Internet “Land”-Speed Record

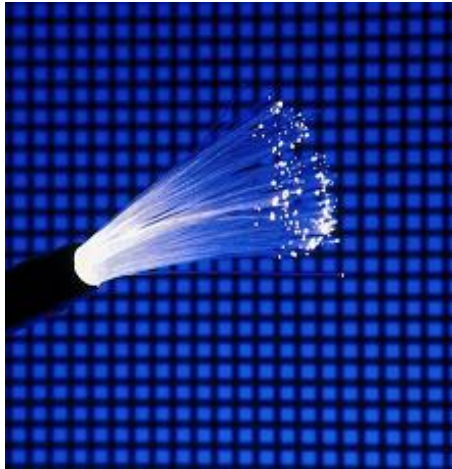

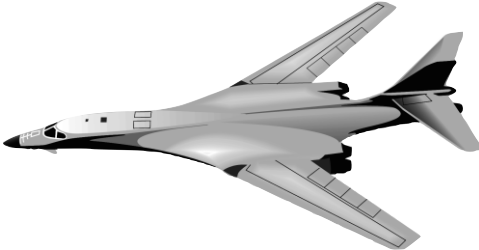
|             | Fiber-optic cable  | Subaru Outback  |
|-------------|--|---|
|             |  |  |
|             | State of the art networking medium<br>(sent 585 GB)                                | Sensible station wagon  |
| Cargo       |  | 183 kg  |
| Speed       |  | 119 MPH   |
| Latency (s) | 1800   | 563,984   |
| BW (GB/s)   | 1.13   | 0.0014  |
| Tb-m/s      | 272,400  | 344,690   |

# The Internet “Land”-Speed Record

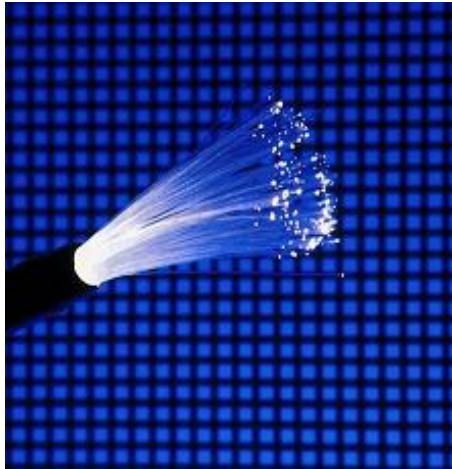

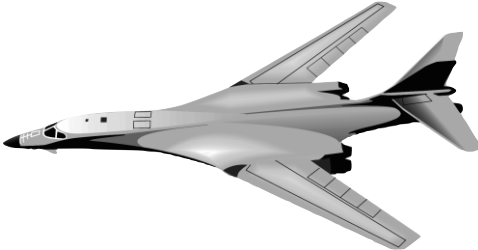

|             | Fiber-optic cable  | Subaru Outback  | B1-B   |
|-------------|--|---|--|
|             |  <p>State of the art networking medium<br/>(sent 585 GB)</p> |  <p>Sensible station wagon</p> |  <p>Supersonic bomber</p> |
| Cargo       |  | 183 kg  | 25,515 kg  |
| Speed       |  | 119 MPH   | 950 MPH  |
| Latency (s) | 1800   | 563,984   |  |
| BW (GB/s)   | 1.13   | 0.0014  |  |
| Tb-m/s      | 272,400  | 344,690   |  |



# The Internet “Land”-Speed Record

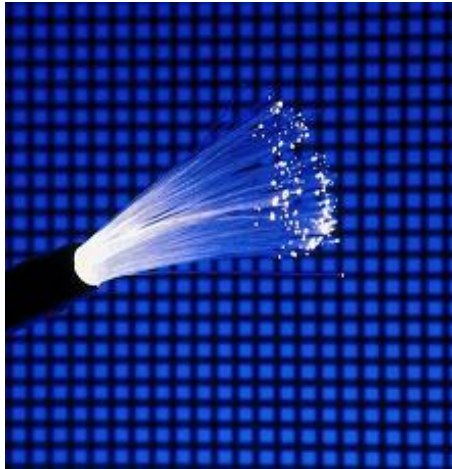

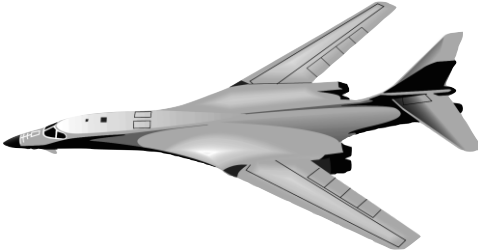

|             | Fiber-optic cable  | Subaru Outback  | B1-B  |
|-------------|--|---|---|
|             |  |  |  |
|             | State of the art networking medium<br>(sent 585 GB)                                | Sensible station wagon  | Supersonic bomber   |
| Cargo       |  | 183 kg  | 25,515 kg   |
| Speed       |  | 119 MPH   | 950 MPH   |
| Latency (s) | 1800   | 563,984   | 70,646  |
| BW (GB/s)   | 1.13   | 0.0014  | 1.6   |
| Tb-m/s      | 272,400  | 344,690   | 382,409,815   |

# The Internet “Land”-Speed Record

|             | Fiber-optic cable  | Subaru Outback  | B1-B   | Hellespont Alhambra  |
|-------------|--|---|--|--|
|             |  <p>State of the art networking medium (sent 585 GB)</p> |  <p>Sensible station wagon</p> |  <p>Supersonic bomber</p> |  <p>World's largest supertanker</p> |
| Cargo       |  | 183 kg  | 25,515 kg  | 400,975,655 kg   |
| Speed       |  | 119 MPH   | 950 MPH  | 18.9 MPH   |
| Latency (s) | 1800   | 563,984   | 70,646   |  |
| BW (GB/s)   | 1.13   | 0.0014  | 1.6  |  |
| Tb-m/s      | 272,400  | 344,690   | 382,409,815  |  |



# The Internet “Land”-Speed Record

|             | Fiber-optic cable  | Subaru Outback  | B1-B  | Hellespont Alhambra   |
|-------------|--|---|---|---|
|             |  |  |  |  |
|             | State of the art networking medium (sent 585 GB)                                   | Sensible station wagon  | Supersonic bomber   | World's largest supertanker   |
| Cargo       |  | 183 kg  | 25,515 kg   | 400,975,655 kg  |
| Speed       |  | 119 MPH   | 950 MPH   | 18.9 MPH  |
| Latency (s) | 1800   | 563,984   | 70,646  | 1,587,301   |
| BW (GB/s)   | 1.13   | 0.0014  | 1.6   | 1114.5  |
| Tb-m/s      | 272,400  | 344,690   | 382,409,815   | 267,000,000,000   |

# Benchmarks

# Benchmarks: Making Comparable Measurements

- A benchmark suite is a set of programs that are representative of a class of problems.
  - Desktop computing (many available online)
  - Server computing (SPECINT)
  - Scientific computing (SPECFP)
  - Embedded systems (EEMBC)
- There is no “best” benchmark suite.
  - Unless you are interested only in the applications in the suite, they are flawed
  - The applications in a suite can be selected for all kinds of reasons.
- To make broad comparisons possible, benchmarks usually are;
  - “Easy” to set up
  - Portable
  - Well-understood
  - Stand-alone
  - Run under standardized conditions
- Real software is none of these things.

# Classes of benchmarks

- Microbenchmarks measure one feature of system
  - e.g. memory accesses or communication speed
- Kernels – most compute-intensive part of applications
  - Amdahl's Law tells us that this is fine for some applications.
  - e.g. Linpack and NAS kernel benchmarks
- Full application:
  - SpecInt / SpecFP (for servers)
  - Other suites for databases, web servers, graphics,...

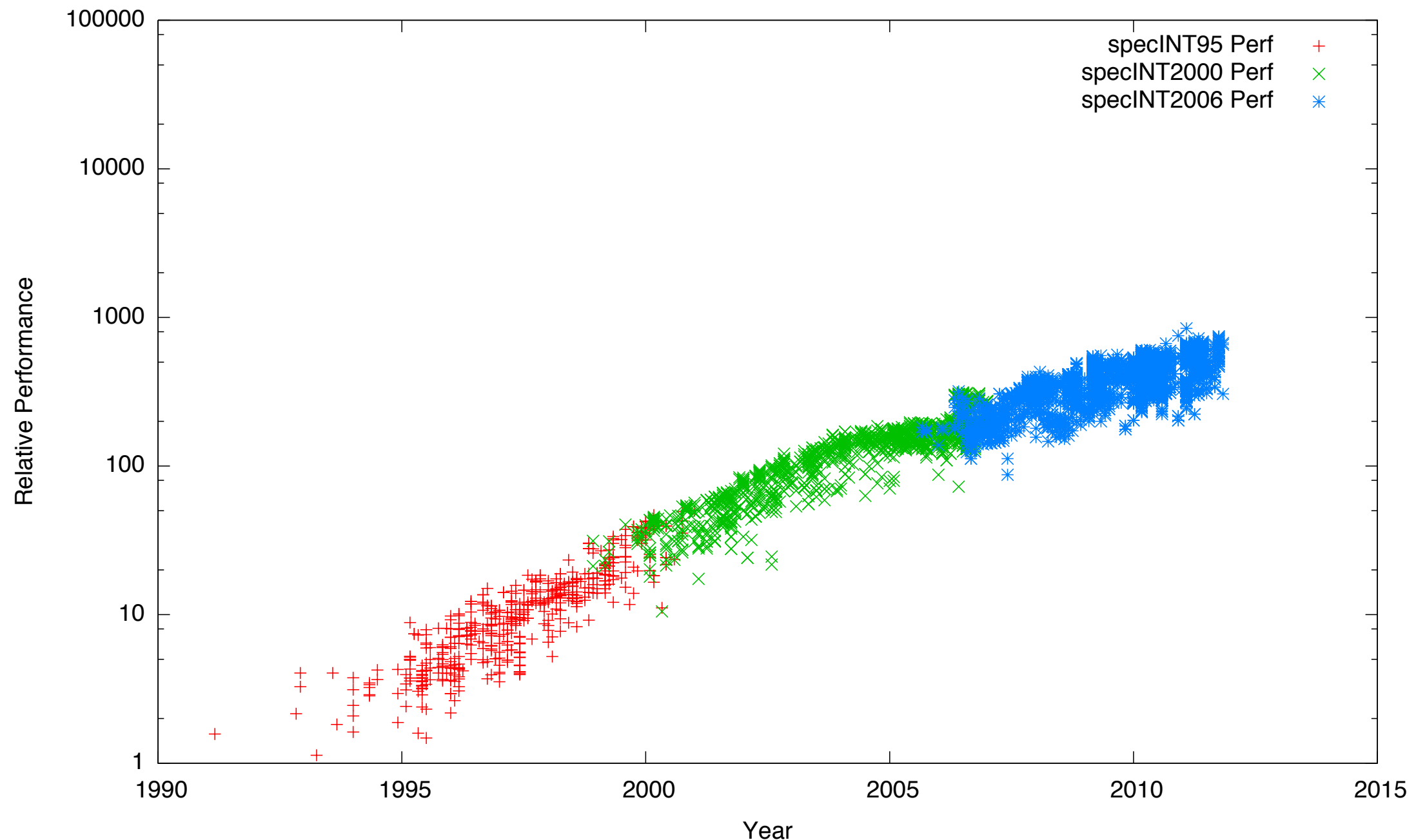
# SPECINT 2006

- In what ways are these not representative?

| Application    | Language | Description                |
|----------------|----------|----------------------------|
| 400.perlbench  | C        | PERL Programming Language  |
| 401.bzip2      | C        | Compression                |
| 403.gcc        | C        | C Compiler                 |
| 429.mcf        | C        | Combinatorial Optimization |
| 445.gobmk      | C        | AI: go                     |
| 456.hmmer      | C        | Search Gene Sequence       |
| 458.sjeng      | C        | AI: chess                  |
| 462.libquantum | C        | Quantum Computing          |
| 464.h264ref    | C        | Video Compression          |
| 471.omnetpp    | C++      | Discrete Event Simulation  |
| 473.astar      | C++      | Path-finding Algorithms    |
| 483.xalancbmk  | C++      | XML Processing             |

# SPECINT 2006

- Despite all that, benchmarks are quite useful.
  - e.g., they allow long-term performance comparisons



Question 1: Multiple Answer

Average Score 0.19355 points

Which of the following are characteristics of benchmark suites.

Correct Answers

|  | Percent Correct | Percent Incorrect |
|--|-----------------|-------------------|
| <input checked="" type="checkbox"/> They are collections of well-defined, stand-alone programs.  | 93.548%         | 6.452%            |
| <input checked="" type="checkbox"/> They always provide an accurate representation of the applications that will run on a computer system. | 12.903%         | 87.097%           |
| It is likely that a properly chosen set of benchmarks could be used as universal measure of computer performance.                          | 50%             | 50%               |
| Benchmarks are selected via an inherently fair and balanced process.   | 69.355%         | 30.645%           |
| Benchmarks are generally not very useful.  | 93.548%         | 6.452%            |

## Question 2: True/False

Average Score 0.69355 points

If I am concerned with energy consumption and performance, energy-delay product is always the ideal metric to use in evaluating computer systems.

| Correct                             | Answers           | Percent Answered |
|-------------------------------------|-------------------|------------------|
|                                     | True              | 30.645%          |
| <input checked="" type="checkbox"/> | False             | 69.355%          |
|                                     | <i>Unanswered</i> | 0%               |



### Question 3: True/False

Average Score 0.69355 points

The speedup of A relative to B is equal to Latency-of-B/Latency-of-A.

| Correct                             | Answers           | Percent Answered |
|-------------------------------------|-------------------|------------------|
| <input checked="" type="checkbox"/> | True              | 69.355%          |
| <input type="checkbox"/>            | False             | 30.645%          |
|                                     | <i>Unanswered</i> | 0%               |

#### Question 4: Multiple Answer

Average Score 1.09677 points

Consider the metric  $\text{MIPS}^2/\text{W}$ . MIPS is "millions of instructions per second." Assuming that you decide to evaluate processors based on this metric, which of the following are true?

| Correct | Answers  | Percent Correct | Percent Incorrect |
|---------|--|-----------------|-------------------|
|         | You prefer machines that consume more power to those that consume less power.                        | 90.323%         | 9.677%            |
| ✓       | You prefer machines that are "faster" to those that are "slower".                                    | 77.419%         | 22.581%           |
| ✓       | You would like the values for this metric to be higher rather than lower.                            | 91.935%         | 8.065%            |
|         | You believe that minimizing power consumption is more important than improving performance.          | 67.742%         | 32.258%           |
|         | You will not need to establish a consistent benchmark on which to measure the values of this metric. | 95.161%         | 4.839%            |

### Question 5: Calculated Numeric

Average Score 0.51613 points

Consider two machines that you want to compare using the  $\text{MIPS}^2/\text{W}$  from the previous question.

A -- Has a cycle time of 10 nanoseconds ( $10 \times 10^{-9}$  seconds) and a CPI of 1. It consumes 12 W.

B -- Has a clock speed of 200Mhz, and a CPI of 0.5.

How many watts can B consume without be worse that A with respect to the metric?

Correct Answers

Percent Answered

This question doesn't count.

### Question 6: Multiple Answer

Average Score 0.72581 points

Which of the following are equivalent to "The latency of processor A running a program is 4.2 times longer than that of processor B".

| Correct | Answers   | Percent Correct | Percent Incorrect |
|---------|---|-----------------|-------------------|
| ✓       | The latency of Machine B is 76% lower than Machine A. | 50%             | 50%               |
|         | The latency of machine A is 420% longer than B.       | 56.452%         | 43.548%           |
| ✓       | The latency of machine A is 320% longer than B.       | 41.935%         | 58.065%           |
| ✓       | The speedup of B relative to A is 4.2x.               | 70.968%         | 29.032%           |

- $L_a = 4.2 * L_b$
- Latency of machine B 76% Lower than machine A
  - $x\% \text{ decrease} == (1 - 0.01*x) \text{ times increase}$
  - $1 - 0.01*76 = .24 \text{ x increase}$
  - Speed of A compared to B:  $L_a/L_b = 1/4.2 = .24$
  - ->yes
- The Latency of A is 420% longer than B
  - $x\% \text{ increase} == 0.01*x + 1 \text{ times increase}$
  - $4.2 = 0.01*x + 1$
  - $x = 320$
  - -> No
- The Latency of A is 420% longer than B -- Yes
- $L_b/L_a = 4.2*L_a/L_a = 4.2$  -- yes

# Goals for this Class

- Understand how CPUs run programs
  - How do we express the computation the CPU?
  - How does the CPU execute it?
  - How does the CPU support other system components (e.g., the OS)?
  - What techniques and technologies are involved and how do they work?
- Understand why CPU performance varies
  - How does CPU design impact performance?
  - What trade-offs are involved in designing a CPU?
  - How can we meaningfully measure and compare computer performance?
- Understand why program performance varies
  - How do program characteristics affect performance?
  - How can we improve a programs performance by considering the CPU running it?
  - How do other system components impact program performance?

# Goals

- Understand and distinguish between computer performance metrics
  - Latency
  - Bandwidth
  - Various kinds of efficiency
  - Composite metrics
- Understand and apply the CPU performance equation
- Understand how applications and the compiler impact performance
- Understand and apply Amdahl's Law

# 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:

$$\text{Latency} = \text{Instruction Count} * \text{Cycles/Instruction} * \text{Seconds/Cycle}$$
$$L = IC * CPI * CT$$



# The PE as Mathematical Model

$$\text{Latency} = \text{Instructions} * \text{Cycles/Instruction} * \text{Seconds/Cycle}$$

- 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.)
  - $x \text{ Hz} \Rightarrow 1/x \text{ seconds per cycle}$
  - $2.5\text{GHz} \Rightarrow 1/2.5 \times 10^9 \text{ seconds (0.4ns) per cycle}$

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  $\Rightarrow 1/(1-0.2) = 1.25x$  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:
```

*file: cpi-noopt.s*

# Example: Reducing IC

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

- Same computation
- Variables in registers
- Just 1 store
- 9 static insts
- 63 dynamic insts
  - Instruction count reduced by 44%
  - Speedup projected by the PE: 1.8x.

```
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)
```

*file: cpi-opt.s*

# 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.



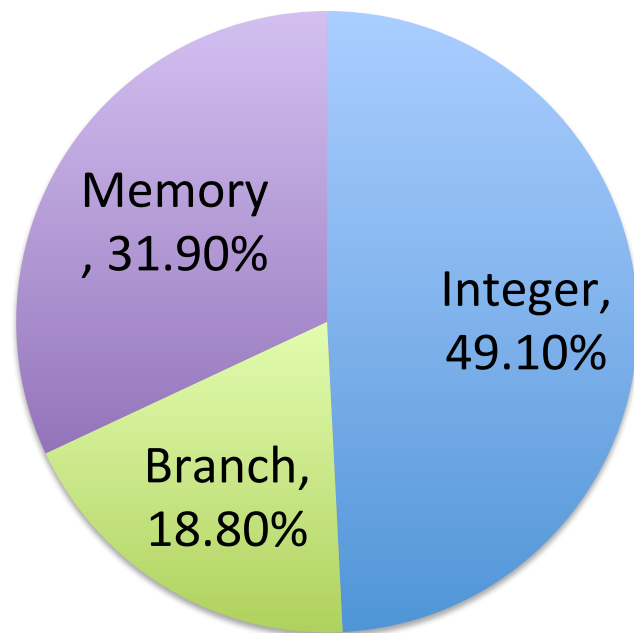
# Cycles Per Instruction

- 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.
- I find  $1/\text{CPI}$  (Instructions Per Cycle; IPC) to be more intuitive, because it emphasizes that it is an average.

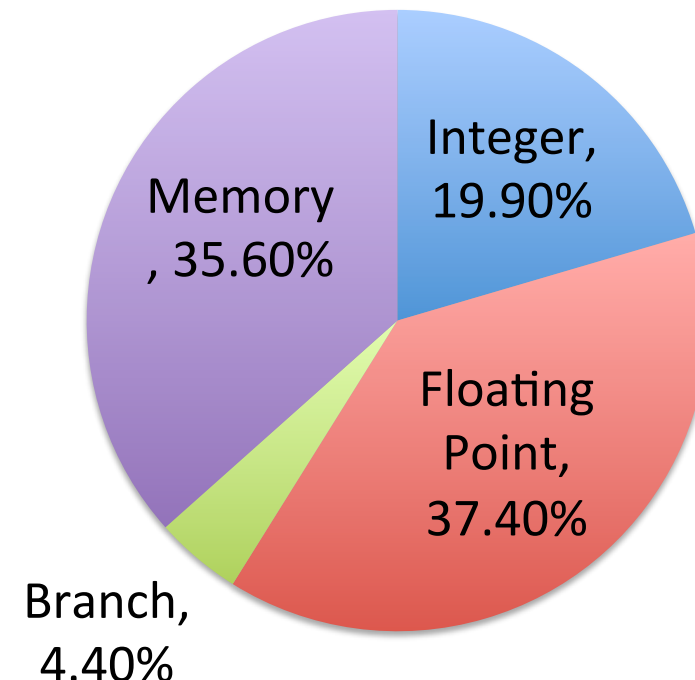
# Instruction Mix and CPI

- Different programs need different kinds of instructions
  - e.g., “Integer apps” don’t do much floating point math.
- The compiler also has some flexibility in which instructions it uses.
- As a result the combination and ratio of instruction types that programs execute (their *instruction mix*) varies.

Spec INT 2006



Spec FP 2006



Spec INT and Spec FP are popular benchmark suites

# Instruction Mix and CPI

- Instruction selections (and, therefore, instruction selection) impacts CPI because some instructions require extra cycles to execute
- All these 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

# Example: Reducing CPI

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

| Type  | CPI | Static # | Dyn# |
|-------|-----|----------|------|
| mem   | 5   | 6        | 42   |
| int   | 1   | 5        | 50   |
| br    | 1   | 2        | 20   |
| Total | 2.5 | 13       | 112  |

Average CPI:

$$(5 \cdot 42 + 1 \cdot 50 + 1 \cdot 20) / 112 = 2.5$$

```
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:
```

*file: cpi-noopt.s*

# Example: 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        | 42   |
| br    | 1    | 2        | 20   |
| Total | 1.06 | 9        | 63   |

Average CPI:

$$(5*1 + 1*42 + 1*20)/63 = 1.06$$

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

```
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)
```

*file: cpi-opt.s*

# 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:
```

```
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)
```

# 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

```
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

# 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

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

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



# 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

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

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

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

```

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

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

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

# 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

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

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

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

# 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

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

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

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

```

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

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

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

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.

# Comparing Similar Systems

$$\text{Latency} = \text{Instructions} * \text{Cycles/Instruction} * \text{Seconds/Cycle}$$

- Often, we will comparing systems that are partly the same
  - e.g., Two CPUs running the same program
  - e.g., One CPU running two programs
- In these cases, many terms of the equation are not relevant
  - e.g., If the CPU doesn't change, neither does CT, so performance can be measured in cycles:  
 $\text{Instructions} * \text{Cycles/Instruction} == \text{Cycles}.$
  - e.g., If the workload is fixed, IC doesn't change, so performance can be measured in Instructions/Second:  
 $1/(\text{Cycles/Instruction} * \text{Seconds/Cycle})$
  - e.g., If the workload *and* clock rate are fixed, the latency is equivalent to CPI (smaller-is-better). Alternately, performance is equivalent to Instructions per Cycle (IPC; bigger-is-better).

You can only ignore terms in the PE, if they are *identical* across the two systems

# Dropping Terms From the PE

- The PE is built to make it easy to focus on aspects of latency by dropping terms
- Example:  $\text{CPI} * \text{CT}$ 
  - Seconds/Instruction = IS (instruction latency)
  - $1/\text{IS} = \text{Inst/Sec}$  or M(ega)IPS, FLOPS
  - Could also be called “raw speed”
  - CPI is still in terms of some particular application or instruction mix.
- Example:  $\text{IC} * \text{CPI}$ 
  - Clock-speed independent latency (cycle count)

# Treating PE Terms Differently

- The PE also allows us to apply “rules of thumb” and/or make projections.
- Example: “CPI is modern processors is between 1 and 2”
  - $L = IC * CPI_{\text{guess}} * CT$
  - In this case, IC corresponds to a particular application, but  $CPI_{\text{guess}}$  is an estimate.
- Example: This new processor will reduce CPI by 50% and reduce CT by 50%.
  - $L = IC * 0.5CPI * CT/2$
  - Now CPI and CT are both estimates, and the resulting L is also an estimate. IC may not be an estimate.

# Abusing the PE

- Be ware of Guaranteed Not To Exceed (GTNE) metrics
- Example: “Processor X has a speed of 10 GOPS (giga insts/sec)”
  - This is equivalent to saying that the average instruction latency is 0.1ns.
  - No workload is given!
  - Does this means that  $L = IC * 0.1ns$ ? Probably not!
- The above claim (probably) means that the processor is capable of 10 GOPS under perfect conditions
  - The vendor promises it will never go faster.
  - That’s very different that saying how fast it will go in practice.
- It may also mean they get 10 GOPS on an industry standard benchmark
  - All the hazards of benchmarks apply.
  - Does your workload behave the same as the industry standard benchmark?



# The Top 500 List

- What's the fastest computer in the world?
  - <http://www.top500.org> will tell you.
  - It's a list of the fastest 500 machines in the world.
- They report floating point operations per second (FLOPS)
  - They use the LINPACK benchmark suite (dense matrix algebra)
  - They constrain the algorithm the system uses.
- Top machine
  - The "K Computer" at RIKEN Advanced Institute for Computational Science (AICS) (Japan)
  - 10.51 PFLOPS ( $10.5 \times 10^{15}$ ), GTNE: 11.2 PFLOPS
  - 705,024 cores, 1.4PB of DRAM
  - 12.7MW of power
- Is this fair? Is it meaningful?
  - Yes, but there's a new list, [www.graph500.org](http://www.graph500.org), that uses a different workload.

# 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!

# 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.

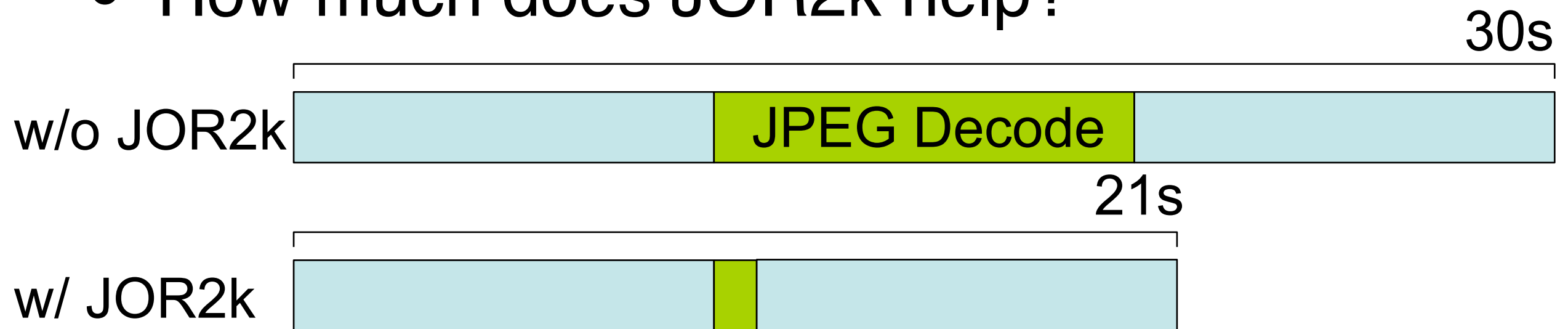
# Amdahl's Law in Action

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

...anna. ...will not, on grounds of principle, ...  
...dy Gaga maybe transposed, and meat dresses may be  
...US export control laws or the Geneva convention,  
...e of dog. Increases processor cost by 45%. Objects  
...er than they are. Or is it farther? Either way, watch  
...e cake will not be a lie. All your base are belong to  
...ingeing is allowed, but only in countries where “win

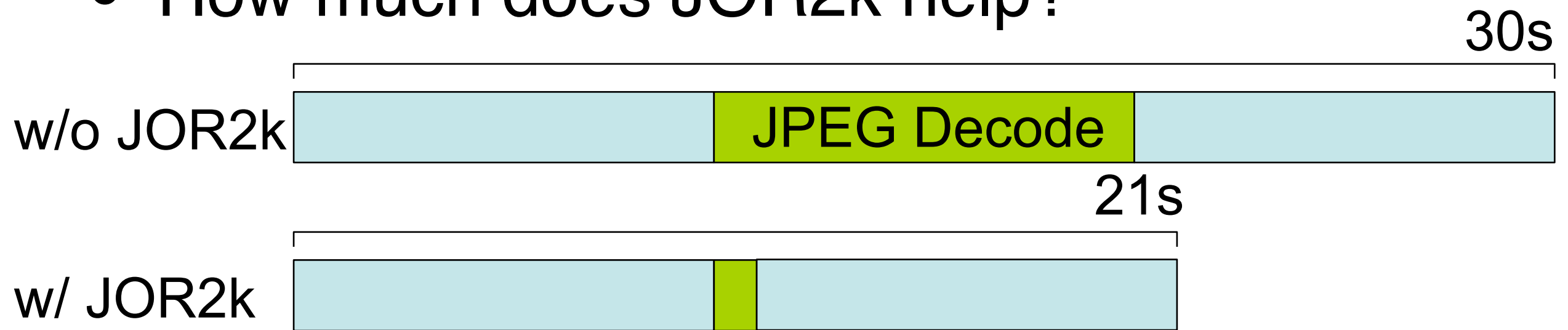
# Amdahl's Law in Action

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



# 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?

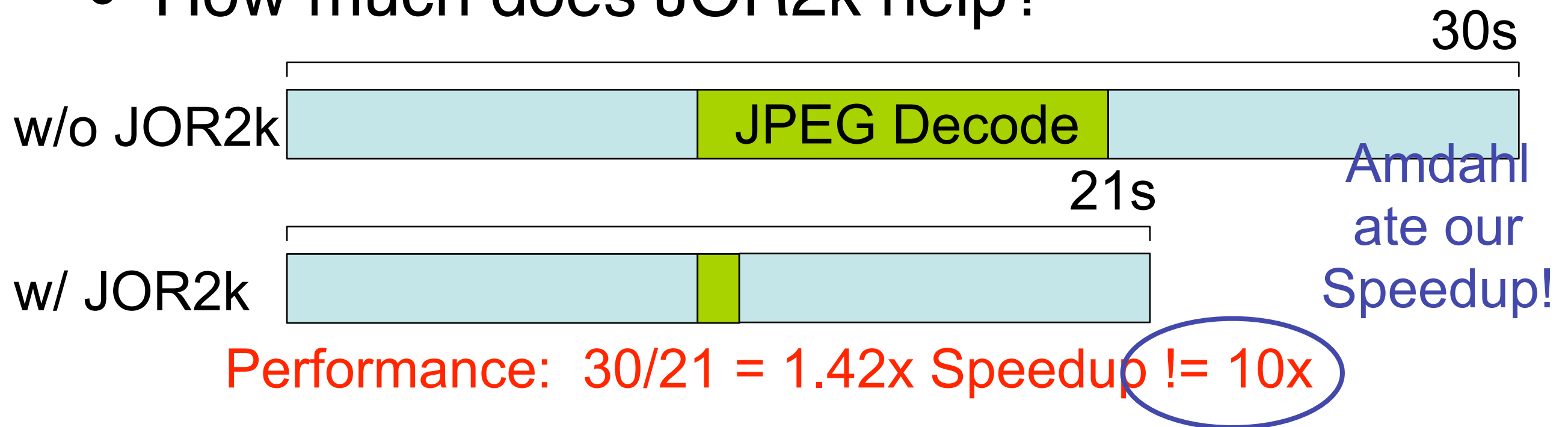


Performance:  $30/21 = 1.42x$  Speedup  $\neq 10x$



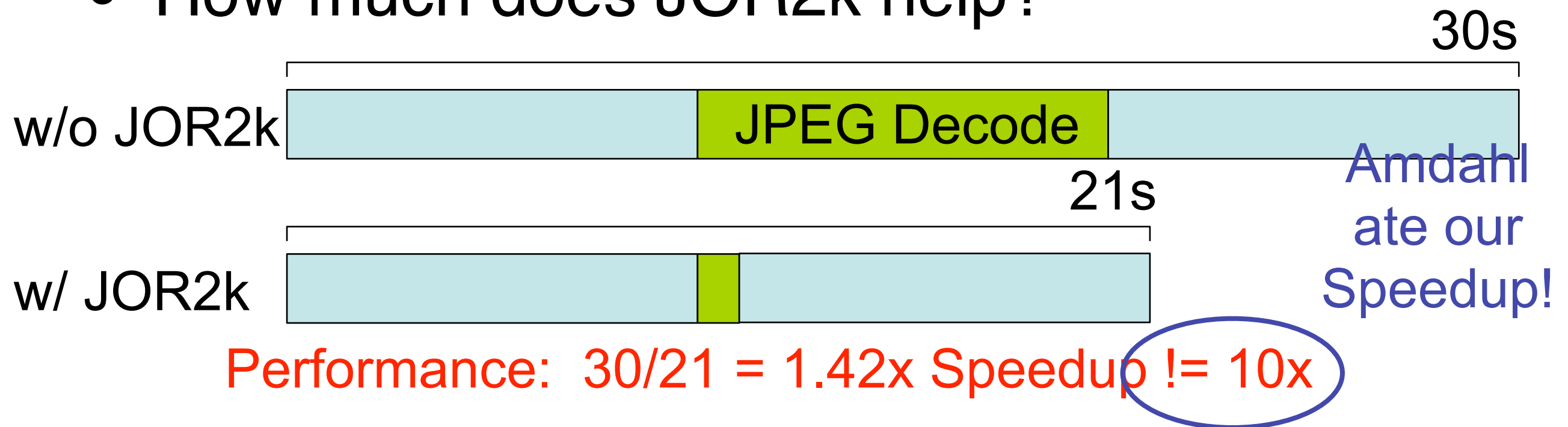
# Amdahl's Law in Action

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



# Amdahl's Law in Action

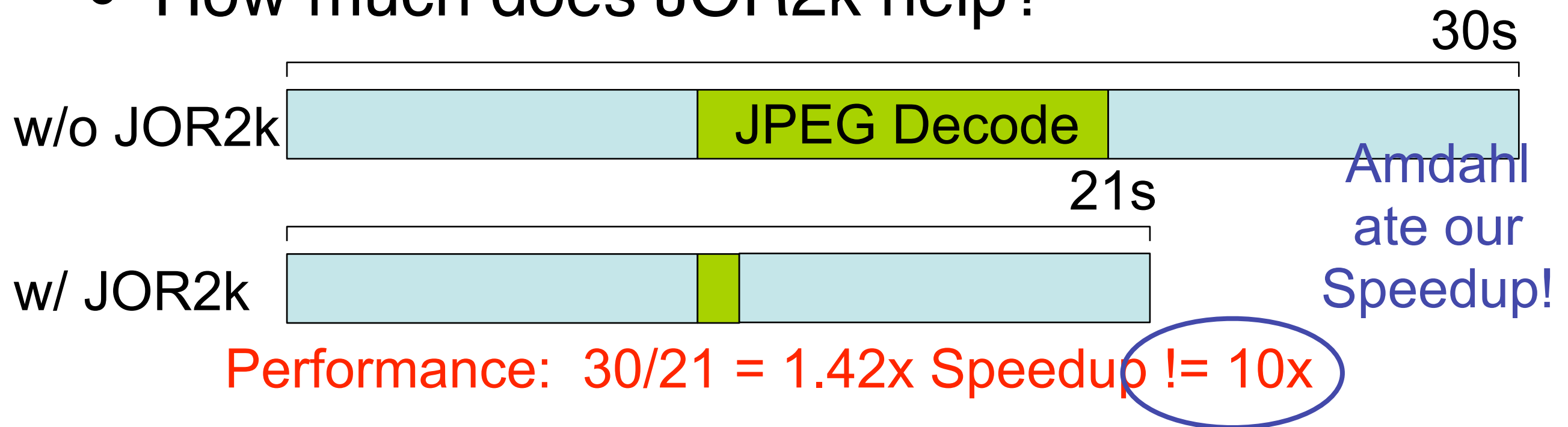
- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of it's time doing JPEG decode
- How much does JOR2k help?



Is this worth the  
45% increase in  
cost?

# Amdahl's Law in Action

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

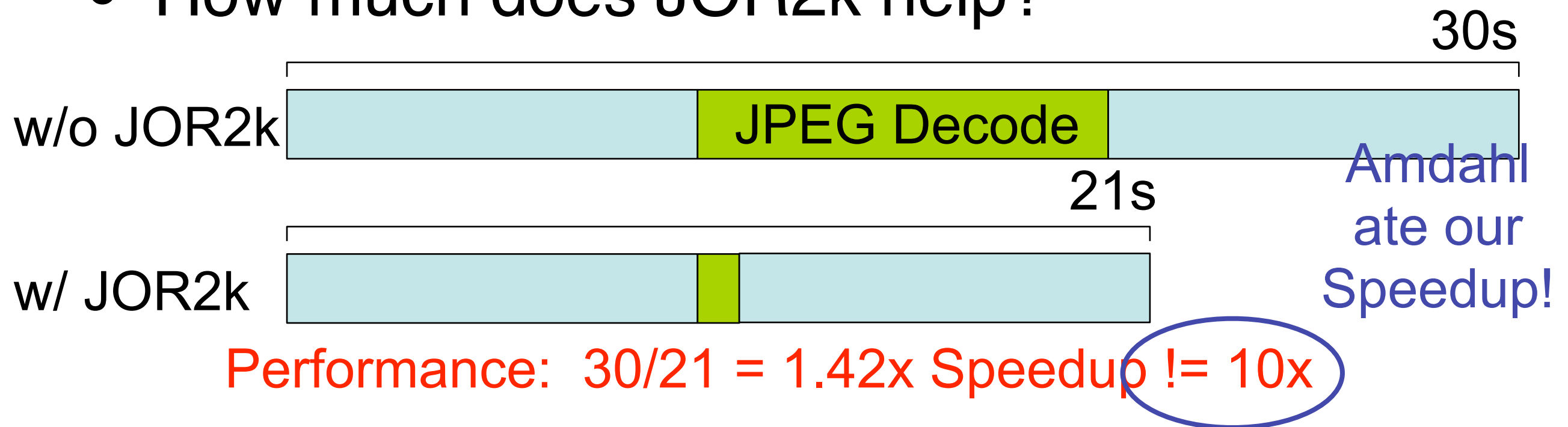


Is this worth the  
45% increase in  
cost?

Metric = Latency \* Cost =>

# 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?



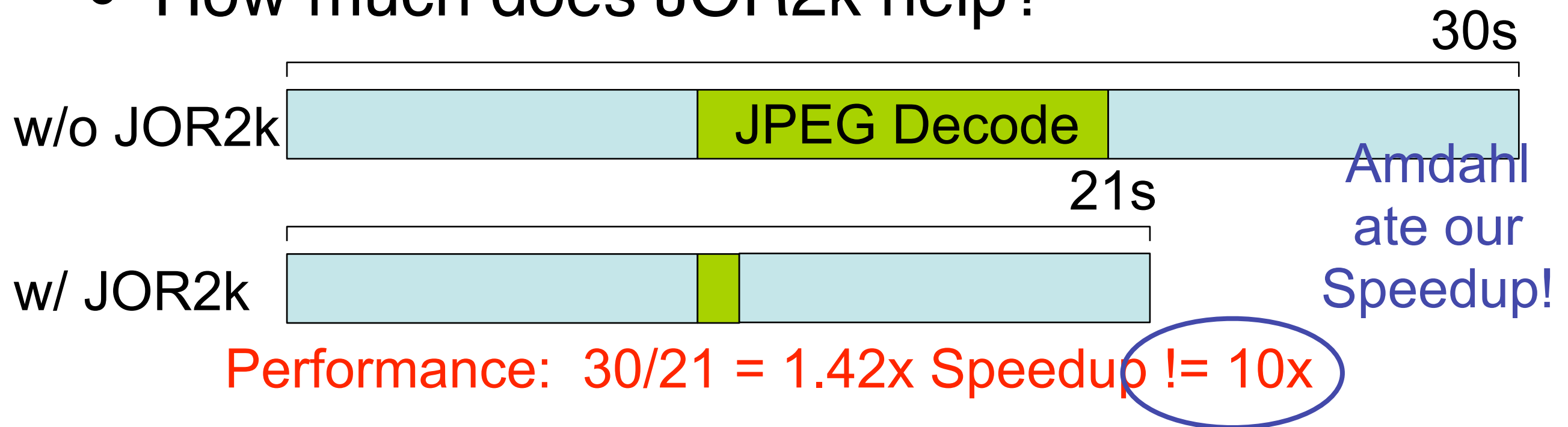
Is this worth the  
45% increase in  
cost?

Metric = Latency \* Cost =>

No

# 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?

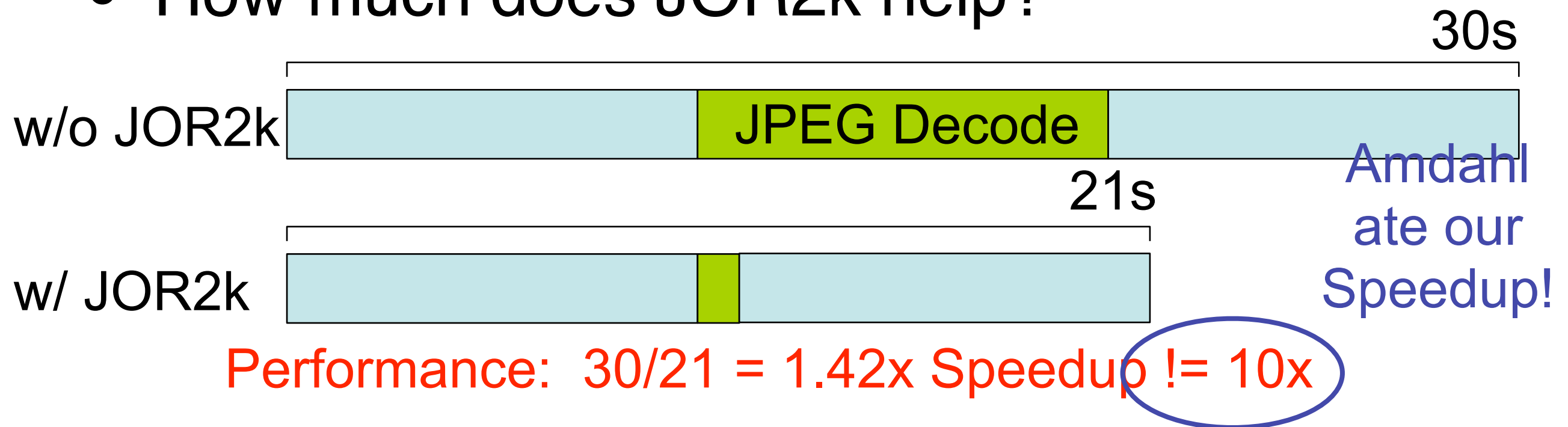
Metric = Latency \* Cost =>

No

Metric = Latency<sup>2</sup> \* Cost =>

# 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?

Metric = Latency \* Cost =>

No

Metric = Latency<sup>2</sup> \* Cost =>

Yes

# Explanation

- Latency\*Cost and Latency<sup>2</sup>\*Cost are smaller-is-better metrics.
- Old System: No JOR2k
  - Latency = 30s
  - Cost = C (we don't know exactly, so we assume a constant, C)
- New System: With JOR2k
  - Latency = 21s
  - Cost = 1.45 \* C
- Latency\*Cost
  - Old: 30\*C
  - New: 21\*1.45\*C
  - New/Old = 21\*1.45\*C/30\*C = 1.015
  - New is bigger (worse) than old by 1.015x
- Latency<sup>2</sup>\*Cost
  - Old: 30<sup>2</sup>\*C
  - New: 21<sup>2</sup>\*1.45\*C
  - New/Old = 21<sup>2</sup>\*1.45\*C/30<sup>2</sup>\*C = 0.71
  - New is smaller (better) than old by 0.71x
- In general, you can make C = 1, and just leave it out.

# 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,  $S_{tot}$

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



# 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 Corollary #1

- Maximum possible speedup  $S_{\max}$ , if we are targeting  $x$  of the program.

$$S = \textit{infinity}$$

$$S_{\max} = \frac{1}{(1-x)}$$

# 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 10 hours faster?
  - How much faster must you make the integer unit to make the code run 50 hours faster?

A) 1.1

B) 1.25

C) 1.75

D) 1.31

E) 10.0

F) 50.0

G) 1 million times

H) Other

# Explanation

- 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 10 hours faster?
- Solution:
  - $S_{\text{tot}} = 200/190 = 1.05$
  - $x = 0.2$  (or 20%)
  - $S_{\text{tot}} = 1/(0.2/S + (1-0.2))$
  - $1.05 = 1/(0.2/S + (1-0.2)) = 1/(0.2/S + 0.8)$
  - $1/1.05 = 0.952 = 0.2/S + 0.8$
  - Solve for  $S \Rightarrow S = 1.3125$

# Explanation

- 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?
- Solution:
  - $S_{\text{tot}} = 200/150 = 1.33$
  - $x = 0.2$  (or 20%)
  - $S_{\text{tot}} = 1/(0.2/S + (1-0.2))$
  - $1.33 = 1/(0.2/S + (1-0.2)) = 1/(0.2/S + 0.8)$
  - $1/1.33 = 0.75 = 0.2/S + 0.8$
  - Solve for  $S \Rightarrow S = -4$  !!! Negative speedups are not possible

# Explanation, Take 2

- 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?
- Solution:
  - Corollary #1. What's the max speedup given that  $x = 0.2$ ?
  - $S_{\max} = 1/(1-x) = 1/0.8 = 1.25$
  - Target speed up = old/new =  $200/150 = 1.33 > 1.25$
  - The target is not achievable.

# 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
  - $x = 0.2$
  - $S = 1/(1-0.25) = 1.33$
  - $S_{\text{int}} = 1/(0.2/1.33 + 0.8) = 1.052$
- Speed up IO
  - $x = 0.35$
  - $S = 6\mu\text{s}/5\mu\text{s} = 1.2$
  - $S_{\text{io}} = 1/(\cdot 35/1.2 + 0.65) = 1.062$
- Speeding up IO is better



# Amdahl's Corollary #2

- 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

Common case



- In the end, there is no common case!
- Options:
  - Global optimizations (faster clock, better compiler)
  - Divide the program up differently
    - e.g. Focus on classes of instructions (maybe memory or FP?), rather than functions.
    - e.g. Focus on function call overheads (which are everywhere).
  - War of attrition
  - Total redesign (You are probably well-prepared for this)

# Amdahl's Corollary #2: Example

Common case



■  $7x \Rightarrow 1.4x$



- In the end, there is no common case!
- Options:
  - Global optimizations (faster clock, better compiler)
  - Divide the program up differently
    - e.g. Focus on classes of instructions (maybe memory or FP?), rather than functions.
    - e.g. Focus on function call overheads (which are everywhere).
  - War of attrition
  - Total redesign (You are probably well-prepared for this)

# Amdahl's Corollary #2: Example

Common case



■  $7x \Rightarrow 1.4x$



■  $4x \Rightarrow 1.3x$



- In the end, there is no common case!
- Options:
  - Global optimizations (faster clock, better compiler)
  - Divide the program up differently
    - e.g. Focus on classes of instructions (maybe memory or FP?), rather than functions.
    - e.g. Focus on function call overheads (which are everywhere).
  - War of attrition
  - Total redesign (You are probably well-prepared for this)

# Amdahl's Corollary #2: Example

Common case



■  $7x \Rightarrow 1.4x$   
■  $4x \Rightarrow 1.3x$   
■  $1.3x \Rightarrow 1.1x$

Total =  $20/10 = 2x$

- In the end, there is no common case!
- Options:
  - Global optimizations (faster clock, better compiler)
  - Divide the program up differently
    - e.g. Focus on classes of instructions (maybe memory or FP?), rather than functions.
    - e.g. Focus on function call overheads (which are everywhere).
  - War of attrition
  - Total redesign (You are probably well-prepared for this)

# Amdahl's Corollary #3

- Benefits of parallel processing
- $p$  processors
- $x$  of the program is  $p$ -way parallizable
- Maximum speedup,  $S_{par}$

$$S_{par} = \frac{1}{(x/p + (1-x))}$$

- A key challenge in parallel programming is increasing  $x$  for large  $p$ .
  - $x$  is pretty small for desktop applications, even for  $p = 2$
  - This is a big part of why multi-processors are of limited usefulness.

# Example #3

- Recent advances in process technology have quadruple the number transistors you can fit on your die.
- Currently, your key customer can use up to 4 processors for 40% of their application.
- You have two choices:
  - Increase the number of processors from 1 to 4
  - Use 2 processors but add features that will allow the application to use 2 processors for 80% of execution.
- Which will you choose?

# Amdahl's Corollary #4

- Amdahl's law for latency (L)
- By definition
  - $\text{Speedup} = \text{oldLatency} / \text{newLatency}$
  - $\text{newLatency} = \text{oldLatency} * 1 / \text{Speedup}$
- By Amdahl's law:
  - $\text{newLatency} = \text{old Latency} * (x/S + (1-x))$
  - $\text{newLatency} = x * \text{oldLatency} / S + \text{oldLatency} * (1-x)$
- Amdahl's law for latency
  - $\text{newLatency} = x * \text{oldLatency} / S + \text{oldLatency} * (1-x)$



# Amdahl's Non-Corollary

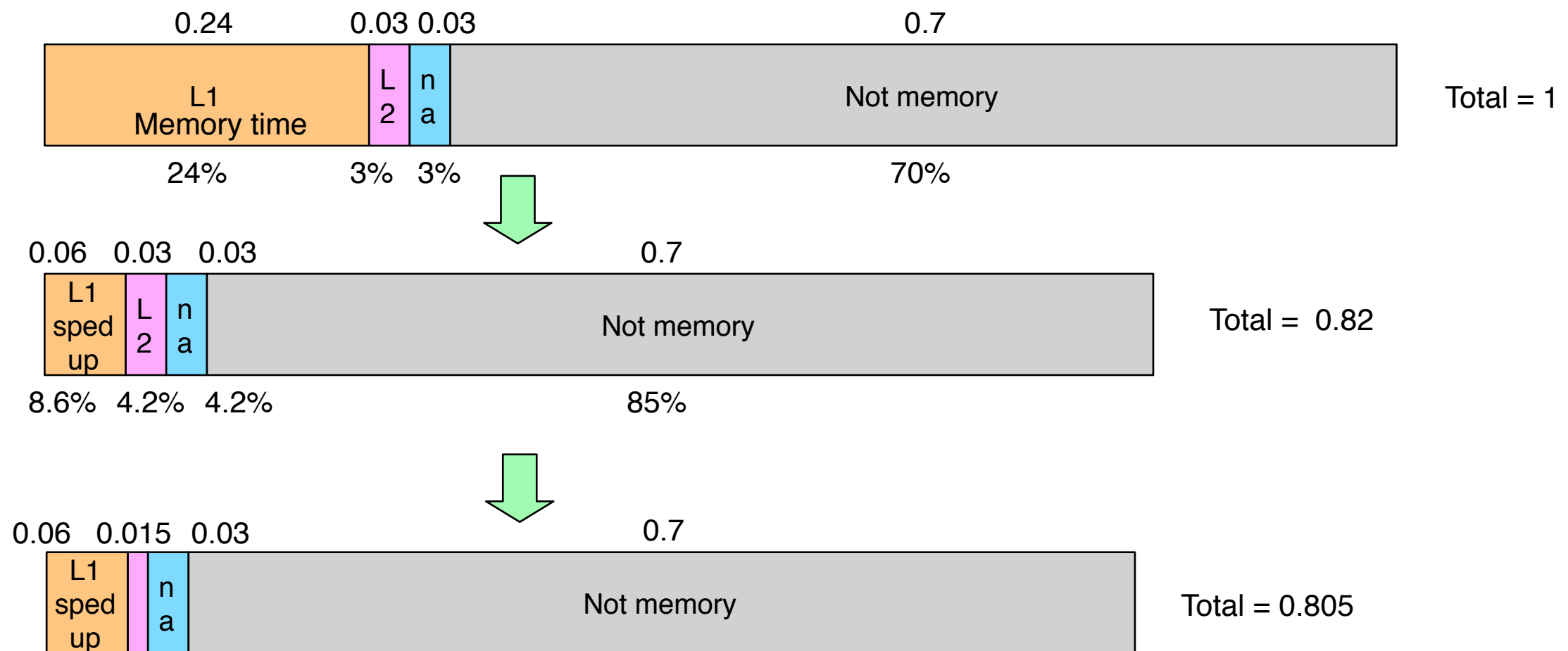
- Amdahl's law does not bound slowdown
  - $\text{newLatency} = x \cdot \text{oldLatency} / S + \text{oldLatency} \cdot (1 - x)$
  - newLatency is linear in  $1/S$
- Example:  $x = 0.01$  of execution,  $\text{oldLat} = 1$ 
  - $S = 0.001$ ;
    - $\text{Newlat} = 1000 \cdot \text{Oldlat} \cdot 0.01 + \text{Oldlat} \cdot (0.99) = \sim 10 \cdot \text{Oldlat}$
  - $S = 0.000001$ ;
    - $\text{Newlat} = 100000 \cdot \text{Oldlat} \cdot 0.01 + \text{Oldlat} \cdot (0.99) = \sim 1000 \cdot \text{Oldlat}$
- Things can only get so fast, but they can get arbitrarily slow.
  - Do not hurt the non-common case too much!

# Amdahl's Example #4

This one is tricky

- Memory operations currently take 30% of execution time.
- A new widget called a “cache” speeds up 80% of memory operations by a factor of 4
- A second new widget called a “L2 cache” speeds up 1/2 the remaining 20% by a factor of 2.
- What is the total speed up?

# Answer in Pictures



Speed up = 1.242

# Amdahl's Pitfall: This is wrong!

- You cannot trivially apply optimizations one at a time with Amdahl's law.
- Apply the L1 cache first
  - $S_1 = 4$
  - $x_1 = .8 \cdot .3$
  - $S_{\text{totL1}} = 1/(x_1/S_1 + (1-x_1))$
  - $S_{\text{totL1}} = 1/(0.8 \cdot 0.3/4 + (1-(0.8 \cdot 0.3))) = 1/(0.06 + 0.76) = 1.2195$  times
- Then, apply the L2 cache
  - $S_{L2} = 2$
  - $x_{L2} = 0.3 \cdot (1 - 0.8)/2 = 0.03$
  - $S_{\text{totL2}} = 1/(0.03/2 + (1-0.03)) = 1/(.015 + .97) = 1.015$  times
- Combine
  - $S_{\text{totL2}} = S_{\text{totL2}}' \cdot S_{\text{totL1}} = 1.02 \cdot 1.21 = 1.237$
- What's wrong? -- after we do the L1 cache, the execution time changes, so the fraction of execution that the L2 effects actually grows

# Amdahl's Pitfall: This is wrong!

- You cannot trivially apply optimizations one at a time with Amdahl's law.
- Apply the L1 cache first
  - $S_1 = 4$
  - $x_1 = .8 * .3$
  - $S_{\text{totL1}} = 1/(x_1/S_1 + (1-x_1))$
  - $S_{\text{totL1}} = 1/(0.8*0.3/4 + (1-(0.8*0.3))) = 1/(0.06 + 0.76) = 1.2195$  times
- Then, apply the L2 cache
  - $S_{\text{L2}} = 2$
  - $x_{\text{L2}} = 0.3*(1 - 0.8)/2 = 0.03$
  - $S_{\text{totL2}} = 1/(0.03/2 + (1-0.03)) = 1/(.015 + .97) = 1.015$  times
- Combine
  - $S_{\text{totL2}} = S_{\text{totL2}}' * S_{\text{totL1}} = 1.02*1.21 = 1.237$

This is wrong

- What's wrong? -- after we do the L1 cache, the execution time changes, so the fraction of execution that the L2 effects actually grows

# Amdahl's Pitfall: This is wrong!

- You cannot trivially apply optimizations one at a time with Amdahl's law.
- Apply the L1 cache first
  - $S_1 = 4$
  - $x_1 = .8 \cdot .3$
  - $S_{\text{totL1}} = 1/(x_1/S_1 + (1-x_1))$
  - $S_{\text{totL1}} = 1/(0.8 \cdot 0.3/4 + (1-(0.8 \cdot 0.3))) = 1/(0.06 + 0.76) = 1.2195$  times

- Then, apply the L2 cache

- $S_{L2} = 2$
- $x_{L2} = 0.3 \cdot (1 - 0.8)/2 = 0.03$
- $S_{\text{totL2}} = 1/(0.03/2 + (1-0.03)) = 1/(.015 + .97) = 1.015$  times

- Combine

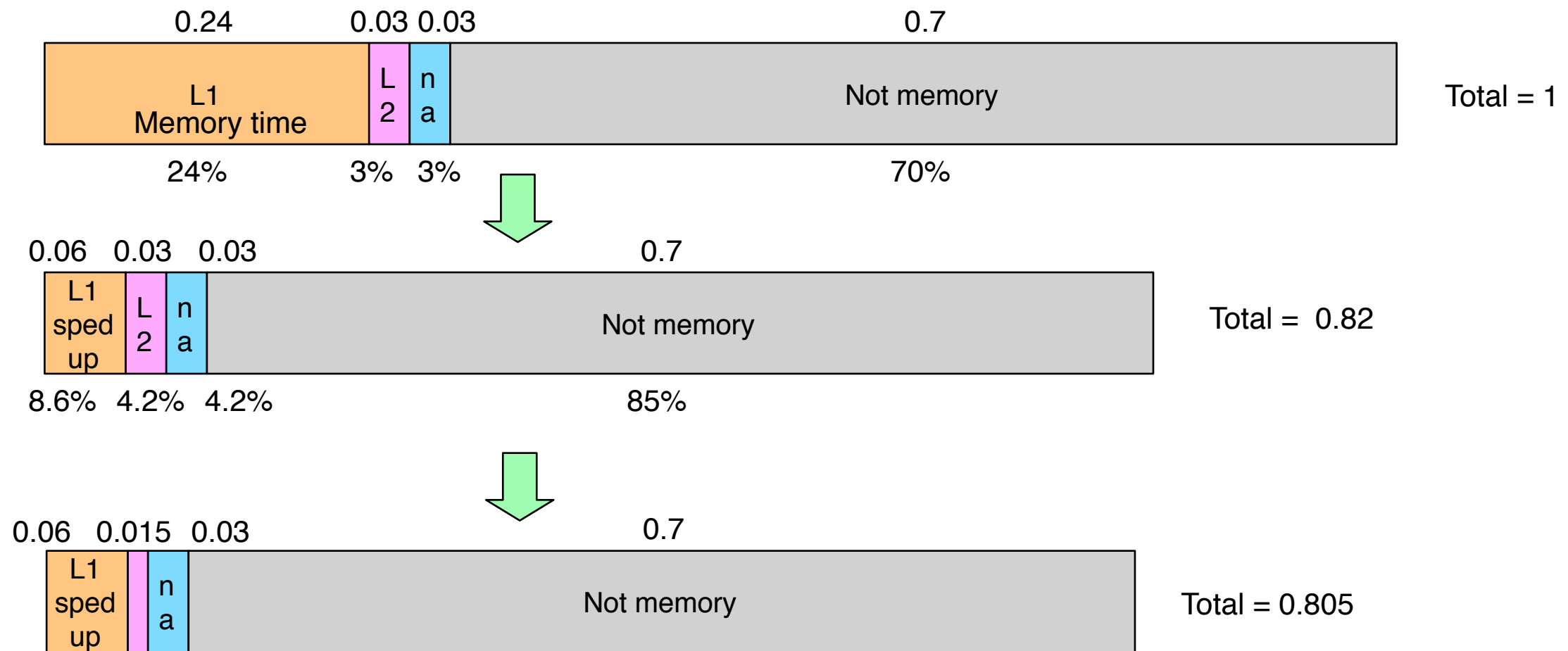
- $S_{\text{totL2}} = S_{\text{totL2}}' \cdot S_{\text{totL1}} = 1.02 \cdot 1.21 = 1.237$

This is wrong

So is this

- What's wrong? -- after we do the L1 cache, the execution time changes, so the fraction of execution that the L2 effects actually grows

# Answer in Pictures



Speed up = 1.242

# Multiple optimizations done right

- We can apply the law for multiple optimizations
- Optimization 1 speeds up  $x_1$  of the program by  $S_1$
- Optimization 2 speeds up  $x_2$  of the program by  $S_2$ 
  - $S_{tot} = 1/(x_1/S_1 + x_2/S_2 + (1-x_1-x_2))$
- Note that  $x_1$  and  $x_2$  must be disjoint!
  - i.e.,  $S_1$  and  $S_2$  must not apply to the same portion of execution.
- If not then, treat the overlap as a separate portion of execution and measure its speed up independently
  - ex: we have  $x_{1only}$ ,  $x_{2only}$ , and  $x_{1\&2}$  and  $S_{1only}$ ,  $S_{2only}$ , and  $S_{1\&2}$
  - Then  $S_{tot} = 1/(x_{1only}/S_{1only} + x_{2only}/S_{2only} + x_{1\&2}/S_{1\&2} + (1 - x_{1only} - x_{2only} - x_{1\&2}))$
  - You can estimate  $S_{1\&2}$  as  $S_{1only} * S_{2only}$ , but the real value could be higher or lower.



# Multiple Opt. Practice

- Combine both the L1 and the L2
  - memory operations are 30% of execution time
  - $S_{L1} = 4$
  - $x_{L1} = 0.3 * 0.8 = .24$
  - $S_{L2} = 2$
  - $x_{L2} = 0.3 * (1 - 0.8) / 2 = 0.03$
  - $S_{\text{totL2}} = 1 / (x_{L1} / S_{L1} + x_{L2} / S_{L2} + (1 - x_{L1} - x_{L2}))$
  - $S_{\text{totL2}} = 1 / (0.24 / 4 + 0.03 / 2 + (1 - .24 - 0.03))$   
 $= 1 / (0.06 + 0.015 + .73) = 1.24 \text{ times}$

# 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 minimize 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 also determines how much cooling you need. Big systems need 0.3-1 Watt of cooling for every watt of compute.

# The End

# Power in Processors

- $P = aCV^2f$ 
  - $a$  = activity factor (what fraction of the xtrs switch every cycles)
  - $C$  = total capacitance (i.e, how many xtrs there are on the chip)
  - $V$  = supply voltage
  - $f$  = clock frequency
- Generally,  $f$  is linear in  $V$ , so  $P$  is roughly  $f^3$
- Architects can improve
  - $a$  -- make the micro architecture more efficient.  
Less useless transistor switchings
  - $C$  -- smaller chips, with fewer transistors



# Metrics in the wild

- Millions of instructions per second (MIPS)
- Floating point operations per second (FLOPS)
- Giga-(integer)operations per second (GOPS)
- Why are these all bandwidth metric?
  - Peak bandwidth is workload independent, so these metrics describe a hardware capability
  - When you see these, they are generally GNTE (Guaranteed not to exceed) numbers.

# More Complex Metrics

- For instance, want low power and low latency
  - $\text{Power} * \text{Latency}$
- More concerned about Power?
  - $\text{Power}^2 * \text{Latency}$
- High bandwidth, low cost?
  - $(\text{MB/s})/\$$
- In general, put the good things in the numerator, the bad things in the denominator.
  - $\text{MIPS}^2/\text{W}$

# What affects Performance

- $\text{Latency} = \text{InstCount} * \text{CPI} * \text{CycleTime}$

|                | Inst Count | CPI | Cycle time |
|----------------|------------|-----|------------|
| Program        | x          |     |            |
| Compiler       | x          | (x) |            |
| Inst. Set.     | x          | x   | (x)        |
| Implementation |            | x   | x          |
| Technology     |            |     | x          |

# The Performance Equation

$$\text{Latency} = \text{Instructions} * \text{Cycles/Instruction} * \text{Seconds/Cycle}$$

- The units work out! Remember your dimensional analysis!
- Cycles/Instruction == CPI
- Seconds/Cycle == 1/hz
- Example:
  - 1GHz clock
  - 1 billion instructions
  - CPI = 4
  - What is the latency?

# The Compiler's Impact on CPI

- Compilers affect CPI...
  - Wise instruction selection
    - “Strength reduction”:  $x * 2^n \rightarrow x \ll n$
    - Use registers to eliminate loads and stores
  - More compact code  $\rightarrow$  less waiting for instructions
- ...and instruction count
  - Common sub-expression elimination
  - Use registers to eliminate loads and stores

# The Compiler's Impact on CPI

- Different instructions impact CPI differently because some require “extra” cycles to execute
  - All these values depend on the particular implementation, not the ISA
- Total CPI depends on the app's instruction mix -- how many of each instruction type executes
  - What program is running?
  - How was it compiled?

| Instruction Type             | Total Cycles | “Extra” cycles |
|------------------------------|--------------|----------------|
| Integer +, -,  , &, branches | 1            | 0              |

# Impacts on CPI

- Biggest contributor: Micro architectural implementation
  - More on this later.
- Other contributors
  - Program inputs
    - can change the cycles required for a particular dynamic instruction
  - Instruction mix
    - since different instructions take different numbers of cycles
    - Floating point divide always takes more cycles than an integer add.

# Stupid Compiler

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

| Type  | CPI | Static # | Dyn# |
|-------|-----|----------|------|
| mem   | 5   | 6        | 42   |
| int   | 1   | 3        | 30   |
| br    | 1   | 2        | 20   |
| Total | 2.8 | 11       | 92   |

$$(5*42 + 1*30 + 1*20)/92 = 2.8$$

```
sw 0($sp), $0 #sum =  
    0  
sw 4($sp), $0 #i = 0  
    loop:  
    lw $1, 4($sp)  
    sub $3, $1, 10  
    beq $3, $0, end  
    lw $2, 0($sp)  
    add $2, $2, $1  
    st 0($sp), $2  
    addi $1, $1, 1  
    st 4($sp), $1  
    b loop  
end:
```



# Smart Compiler

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

| Type  | CPI  | Static # | Dyn# |
|-------|------|----------|------|
| mem   | 5    | 1        | 1    |
| int   | 1    | 5        | 32   |
| br    | 1    | 2        | 20   |
| Total | 1.01 | 8        | 53   |

$$(5*1 + 1*32 + 1*20)/53 = 1.01$$

```
add    $1, $0, $0 # i  
add    $2, $0, $0 # sum  
loop:  
    sub $3, $1, 10  
    beq $3, $0, end  
    add $2, $2, $1  
    addi $1, $1, 1  
    b loop  
end:  
sw 0($sp), $2
```

# X86 Examples

- <http://cseweb.ucsd.edu/classes/wi11/cse141/x86/>