CSCI-564 Advanced Computer Architecture

Lecture 3: Amdahl's Law

Ismet Dagli

Credit to: Dr. Bo Wu

Colorado School of Mines

Remainder: Performance Equation (PE)

- 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

Wheel time!!!

Assume that we are running the *same application (not necessarily the same executable)* with the *same inputs* on two different systems. **What is the *speedup* of System B relative to System A?**

- System A:
  - Instruction count: 3 million
  - Cycles per instruction: 4.7
  - Seconds per cycle: 1ns
- System B:
  - Instruction count: 4 million
  - Cycles per instruction: 2.3
  - Seconds per cycle: 1ns

Solution

Assume that we are running the *same application (not necessarily the same executable)* with the *same inputs* on two different systems. **What is the *speedup* of System B relative to System A?**

- System A:
  - Instruction count: 3 million
  - Cycles per instruction: 4.7
  - Seconds per cycle: 1ns
- System B:
  - Instruction count: 4 million
  - Cycles per instruction: 2.3
  - Seconds per cycle: 1ns

System A is the "old" system, so it goes on top of the speedup equation.

$$\text{Speedup} = \frac{\text{Latency}_{\text{old}}}{\text{Latency}_{\text{new}}} = \frac{3\cancel{\text{million}} \times 4.7 \times 1\cancel{ns}}{4\cancel{\text{million}} \times 2.3 \times 1\cancel{ns}} = \boxed{1.5326}$$

Practice Questions-1

Assume that we are running the *same program* with the *same inputs* on two different systems.
What is the *speedup* of System B relative to System A?

- System A:
  - Instructions per cycle: 0.4
  - Clock speed: 3.7 GHz
- System B:
  - Instructions per cycle: 0.3
  - Clock speed: 4.0 GHz

Practice Questions-1

Assume that we are running the *same program* with the *same inputs* on two different systems.
What is the *speedup* of System B relative to System A?

- System A:
  - Instructions per cycle: 0.4
  - Clock speed: 3.7 GHz
- System B:
  - Instructions per cycle: 0.3
  - Clock speed: 4.0 GHz

Same program means same IC

$$\text{Speedup} = \frac{\text{Latency}_{\text{old}}}{\text{Latency}_{\text{new}}} = \frac{\cancel{IC} \times \frac{1}{0.4} \times \frac{1}{3.7GHz}}{\cancel{IC} \times \frac{1}{0.3} \times \frac{1}{4.0GHz}} = \boxed{0.8108}$$

Practice Questions-2

You have a processor that runs at 4.9 GHz with a CPI of 1.4.

You can either spend $10,000 to hire a CS@Mines graduate for two weeks to optimize your algorithm so that it requires 37% less instructions to execute as before (assume same CPI).

Or, you can spend $1500 on a new CPU that runs at 5.3 GHz (with the same CPI).

Which option gives you the biggest speedup per dollar spent?

Practice Questions-2

You have a processor that runs at 4.9 GHz with a CPI of 1.4.

You can either spend \$10,000 to hire a CS@Mines graduate for two weeks to optimize your algorithm so that it requires 37% less instructions to execute as before (assume same CPI).

Or, you can spend \$1500 on a new CPU that runs at 5.3 GHz (with the same CPI).

Which option gives you the biggest speedup per dollar spent?

Speedup to dollar spent ratio for Mines grad:

$$\text{Speedup} = \frac{\text{Latency}_{\text{old}}}{\text{Latency}_{\text{new}}} = \frac{\cancel{IC} \times \cancel{1.4} \times \frac{1}{\cancel{4.9GHz}}}{(1-0.37)\cancel{IC} \times \cancel{1.4} \times \frac{1}{\cancel{4.9GHz}}} = 1.5873$$

$$\text{Speedup to dollar spent ratio} = \frac{1.5873}{\$10,000} = 0.00015873$$

Speedup of new CPU

$$\text{Speedup} = \frac{\text{Latency}_{\text{old}}}{\text{Latency}_{\text{new}}} = \frac{\cancel{IC} \times \cancel{1.4} \times \frac{1}{4.9GHz}}{\cancel{IC} \times \cancel{1.4} \times \frac{1}{5.3GHz}} = 1.0816$$

$$\text{Speedup to dollar spent ratio} = \frac{1.0816}{\$1,500} = 0.0007211$$

Getting a new CPU is probably the better investment unless you *really* need to get every bit of possible performance.

Amdahl's Law

- The fundamental theorem of performance optimization
- Made by Amdahl!
  - One of the designers of the IBM 360

- 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.

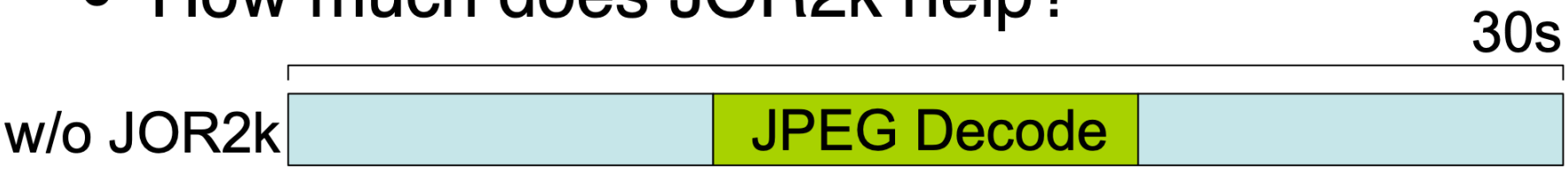
Amdahl's Law in Action

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

rama. Will not, on grounds of principle, decode
dy Gaga maybe transposed, and meat dresses may b
JS 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
he cake will not be a lie. All your base are belong to
ingeing is allowed, but only in countries where "wir

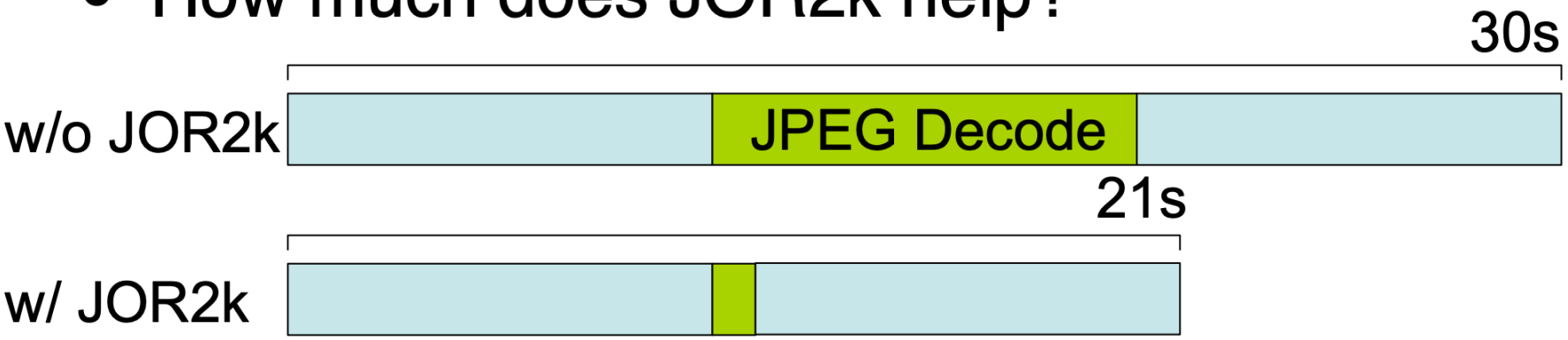
Amdahl's Law in Action

- 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

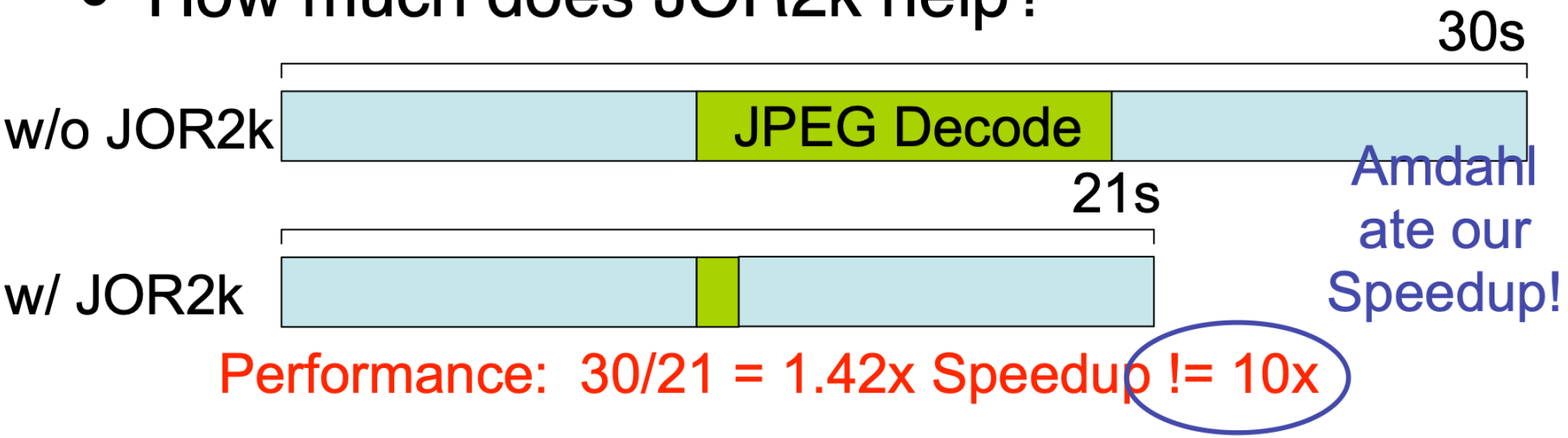
- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of it's time doing JPEG decode
- How much does JOR2k help?



Performance: 30/21 = 1.42x Speedup != 10x

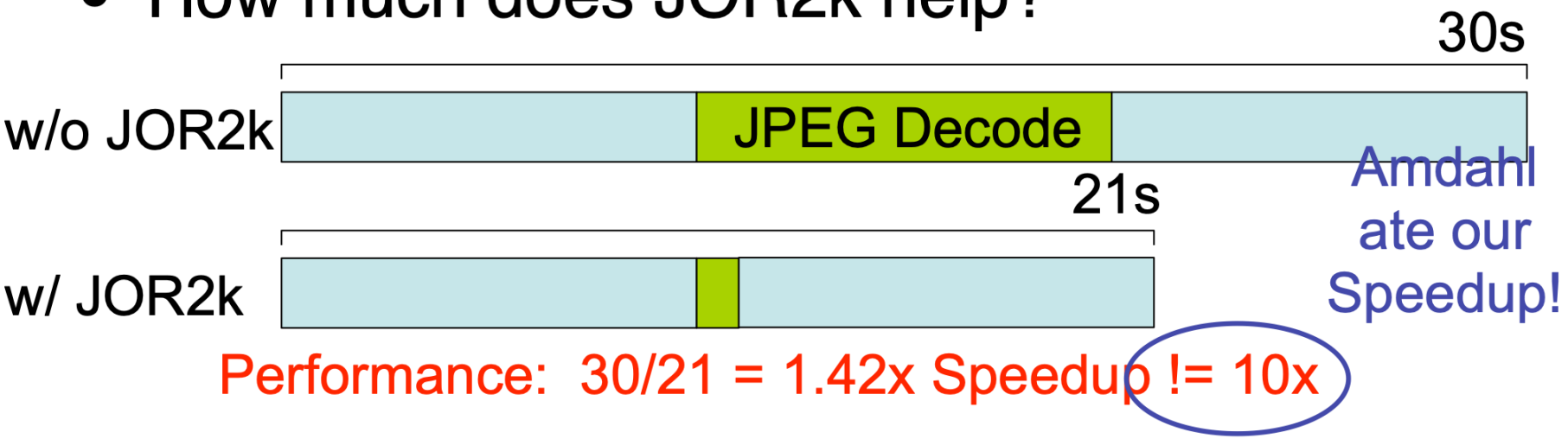
Amdahl's Law in Action

- 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

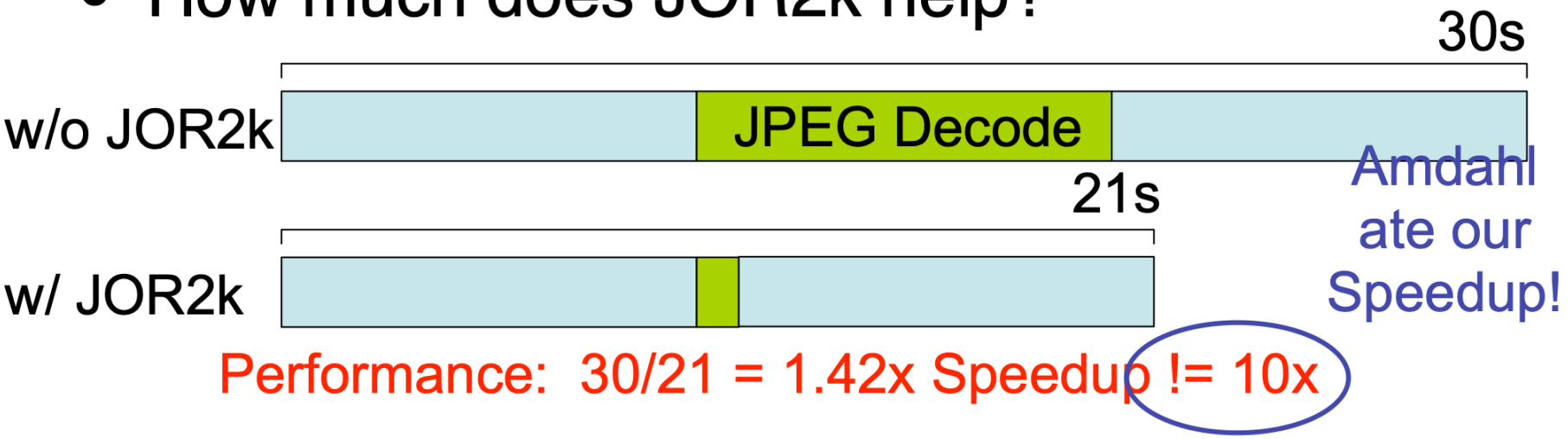
- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of it's time doing JPEG decode
- How much does JOR2k help?



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?

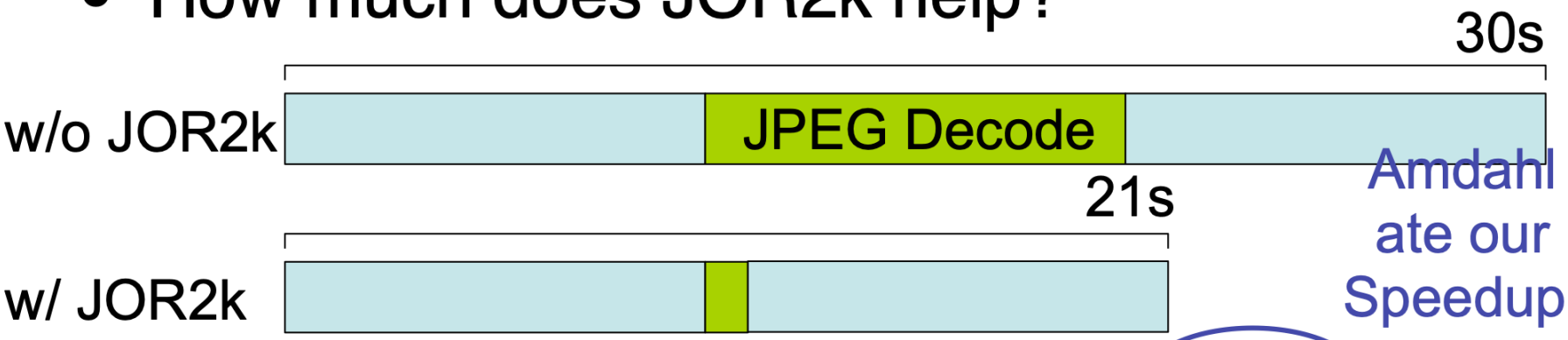


Is this worth the 45% increase in cost?

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?



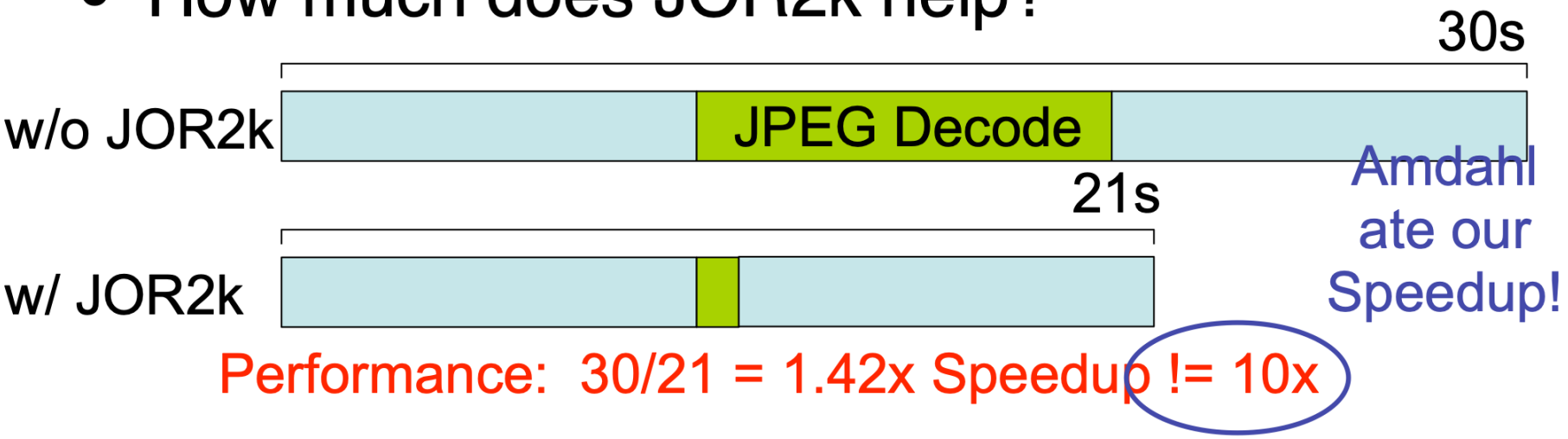
Performance: 30/21 = 1.42x Speedup != 10x

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?



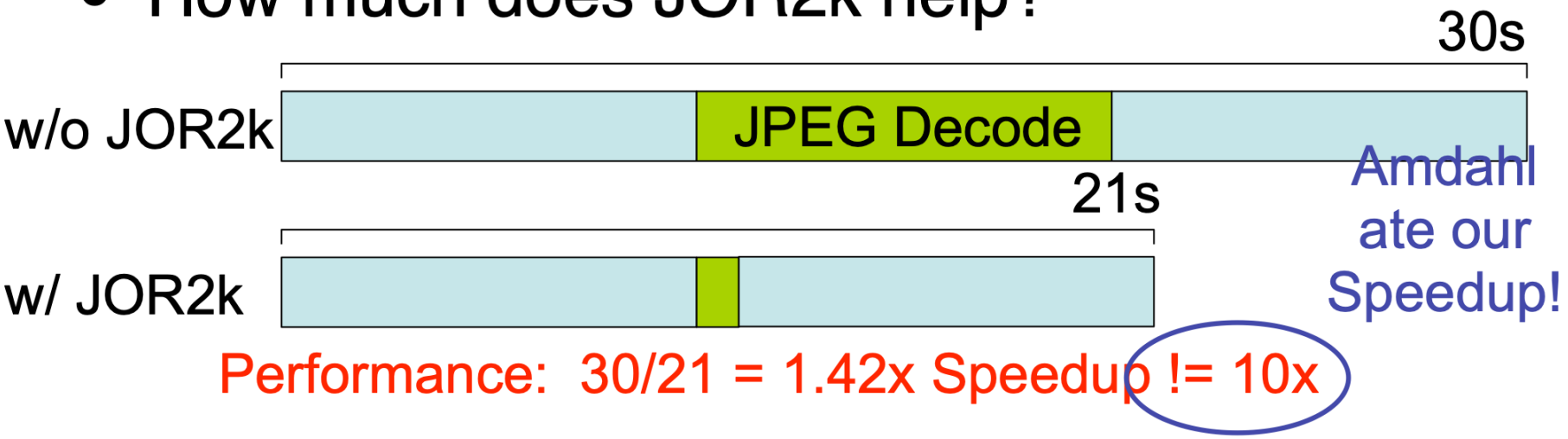
Is this worth the 45% increase in cost?

Metric = Latency * Cost => No

Metric = Latency^2 * 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^2 * Cost => Yes

Explanation

- Latency*Cost and Latency^2*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^2*Cost
  - Old: 30^2 *C
  - New: 21^2 *1.45*C
  - New/Old = 21^2*1.45*C/30^2*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.

$$S = \textit{infinity}$$

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

$$\lim_{S\to\infty} S_{tot} = \lim_{S\to\infty} \frac{1}{\left(\frac{1}{S} + (1-x)\right)} = \frac{1}{(0 + (1-x))} = \frac{1}{1-x}$$

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_{tot} = 200/190 = 1.05
  - x = 0.2 (or 20%)
  - S_{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 => 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_{tot} = 200/150 = 1.33
  - x = 0.2 (or 20%)
  - S_{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 => 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_{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 #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

Amdahl's Corollary #2: Example

Common case

7x => 1.4x

Amdahl's Corollary #2: Example

Common case

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

Amdahl's Corollary #2: Example

Common case

7x => 1.4x

4x => 1.3x

1.3x => 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 over heads (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, Spar

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

- 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
  - Speedup = oldLatency/newLatency
  - newLatency = oldLatency * 1/Speedup
- By Amdahl's law:
  - newLatency = old Latency * (x/S + (1-x))
  - newLatency = x*oldLatency/S + oldLatency*(1-x)
- Amdahl's law for latency
  - newLatency = x*oldLatency/S + oldLatency*(1-x)

Amdahl's Non-Corollary

- Amdahl's law does not bound slowdown
  - 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
- 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?

Explanation

- Apply the L1 cache first
  - $S_1 = 4$
  - $x_1 = .8*.3$
  - $S_{totL1} = 1/(x_1/S_1 + (1-x_1))$
  - $S_{totL1} = 1/(0.8*0.3/4 + (1-(0.8*0.3))) = 1/(0.06 + 0.76) = 1.2195$ times

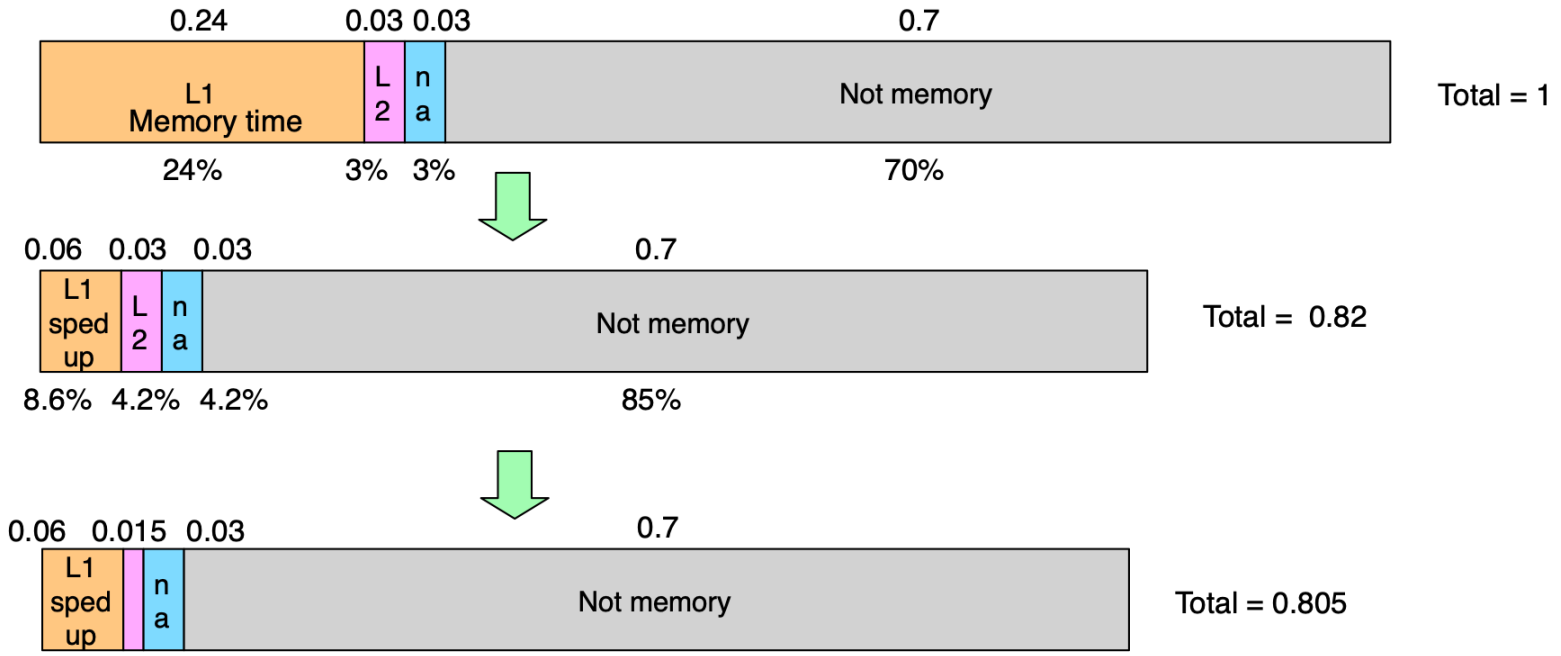
Explanation

- Apply the L1 cache first
  - $S_1 = 4$
  - $x_1 = .8*.3$
  - $S_{totL1} = 1/(x_1/S_1 + (1-x_1))$
  - $S_{totL1} = 1/(0.8*0.3/4 + (1-(0.8*0.3))) = 1/(0.06 + 0.76) = 1.2195$ times
- Then, apply the L2 cache
  - $S_{L2} = 2$
  - $x_{L2} = 0.3*(1 - 0.8)/2 = 0.03$
  - $S_{totL2} = 1/(0.03/2 + (1-0.03)) = 1/(.015 + .97) = 1.015$ times

Explanation

- Apply the L1 cache first
  - $S_1 = 4$
  - $x_1 = .8*.3$
  - $S_{totL1} = 1/(x_1/S_1 + (1-x_1))$
  - $S_{totL1} = 1/(0.8*0.3/4 + (1-(0.8*0.3))) = 1/(0.06 + 0.76) = 1.2195$ times
- Then, apply the L2 cache
  - $S_{L2} = 2$
  - $x_{L2} = 0.3*(1 - 0.8)/2 = 0.03$
  - $S_{totL2} = 1/(0.03/2 + (1-0.03)) = 1/(.015 + .97) = 1.015$ times
- Combine
  - $S_{totL2} = S_{totL2'} * S_{totL1} = 1.02*1.21 = 1.237$

Answer in Pictures



OOPS: 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
  - $S_1 = 4$
  - $x_1 = .8*.3$
  - $S_{totL1} = 1/(x_1/S_1 + (1-x_1))$
  - $S_{totL1} = 1/(0.8*0.3/4 + (1-(0.8*0.3))) = 1/(0.06 + 0.76) = 1.2195$ times
- Then, apply the L2 cache
  - $S_{L2} = 2$
  - $x_{L2} = 0.3*(1 - 0.8)/2 = 0.03$
  - $S_{totL2} = 1/(0.03/2 + (1-0.03)) = 1/(.015 + .97) = 1.015$ times
- Combine
  - $S_{totL2} = S_{totL2'} * S_{totL1} = 1.02*1.21 = 1.237$

- 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
  - $S_1 = 4$
  - $x_1 = .8*.3$
  - $S_{totL1} = 1/(x_1/S_1 + (1-x_1))$
  - $S_{totL1} = 1/(0.8*0.3/4 + (1-(0.8*0.3))) = 1/(0.06 + 0.76) = 1.2195$ times
- Then, apply the L2 cache
  - $S_{L2} = 2$
  - $x_{L2} = 0.3*(1 - 0.8)/2 = 0.03$
  - $S_{totL2} = 1/(0.03/2 + (1-0.03)) = 1/(.015 + .97) = 1.015$ times
- Combine
  - $S_{totL2} = S_{totL2'} * S_{totL1} = 1.02*1.21 = 1.237$

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!

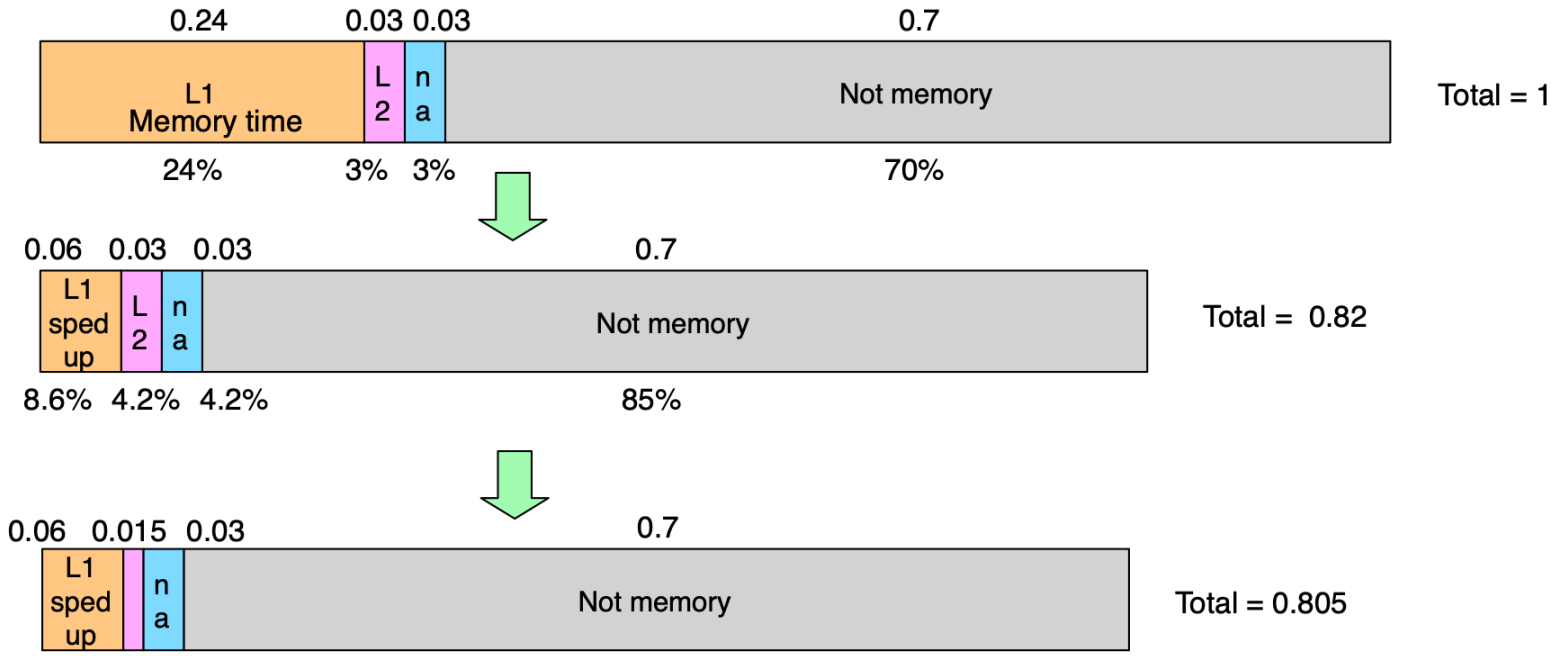
- You cannot trivially apply optimizations one at a time with Amdahl's law.
- Apply the L1 cache first
  - $S_1 = 4$
  - $x_1 = .8*.3$
  - $S_{totL1} = 1/(x_1/S_1 + (1-x_1))$
  - $S_{totL1} = 1/(0.8*0.3/4 + (1-(0.8*0.3))) = 1/(0.06 + 0.76) = 1.2195$ times
- Then, apply the L2 cache
  - $S_{L2} = 2$
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

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 x1 of the program by S1
- Optimization 2 speeds up x2 of the program by S2
  - $Stot = 1/(x_1/S_1 + x_2/S_2 + (1-x_1-x_2))$
- Note that x1 and x2 must be disjoint!
  - i.e., S1 and S2 must not apply to the same portion of execution.
- If not then, treat the overlap as a separate portion of execution and measure it's 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 optimizations done right

Amdahl's Law for Multiple Disjoint Optimizations

Given n **disjoint** optimizations, each of which speed up x_i of the program by S_i, the total speedup is given by

$$S_{\text{tot}} = \frac{1}{\frac{x_1}{S_1} + \frac{x_2}{S_2} + \cdots + \frac{x_i}{S_i} + (1 - x_1 - x_2 - \cdots - x_i)}$$

It is important that the optimizations are disjoint, meaning that for every (x_i, x_j) pair, S_i and S_j must *not* apply to the same portion of the execution.

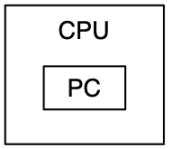
Multiple Opt. Practice

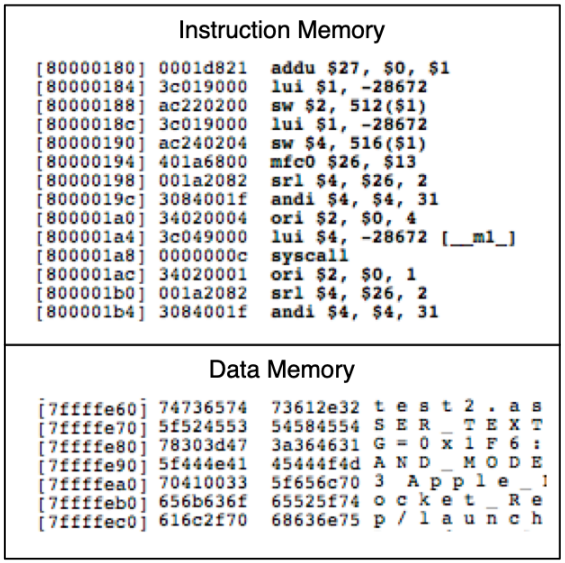
- Combine both the L1 and the L2
  - memory operations are 30% of execution time
  - $S_{L1} = 4$
  - $x_{L1} = 0.3*0.8 = .24$
  - $S_{L2} = 2$
  - $x_{L2} = 0.3*(1 - 0.8)/2 = 0.03$
  - $S_{totL2} = 1/(x_{L1}/S_{LI} + x_{L2}/S_{L2} + (1 - x_{L1} - x_{L2}))$
  - $S_{totL2} = 1/(0.24/4 + 0.03/2 + (1-.24-0.03))$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

The Idea of the CPU

The Stored Program Computer

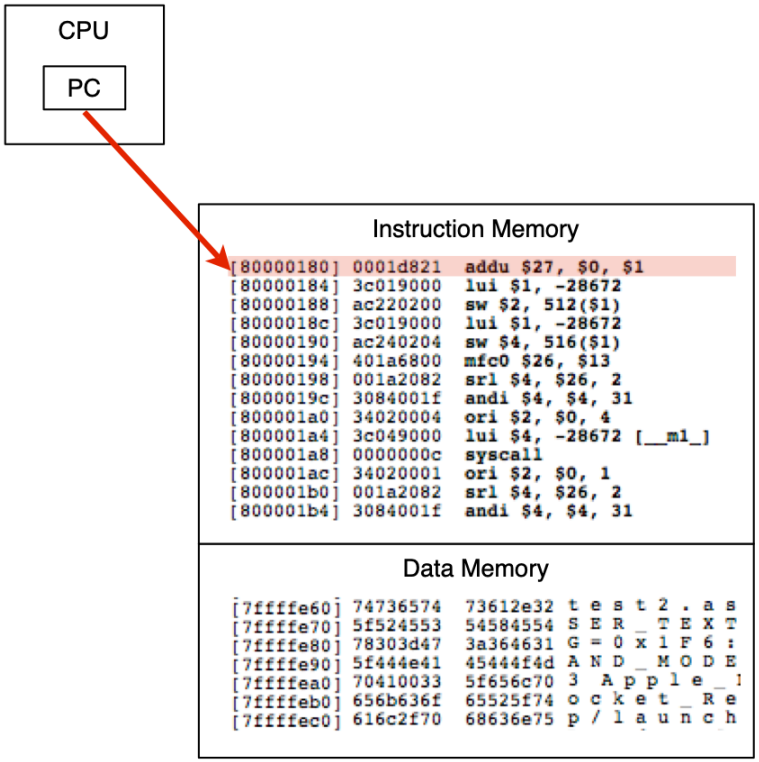
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program





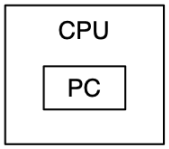
The Stored Program Computer

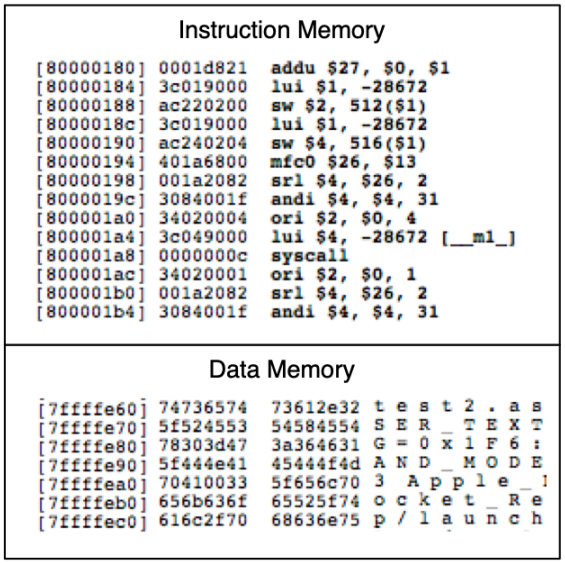
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program



The Stored Program Computer

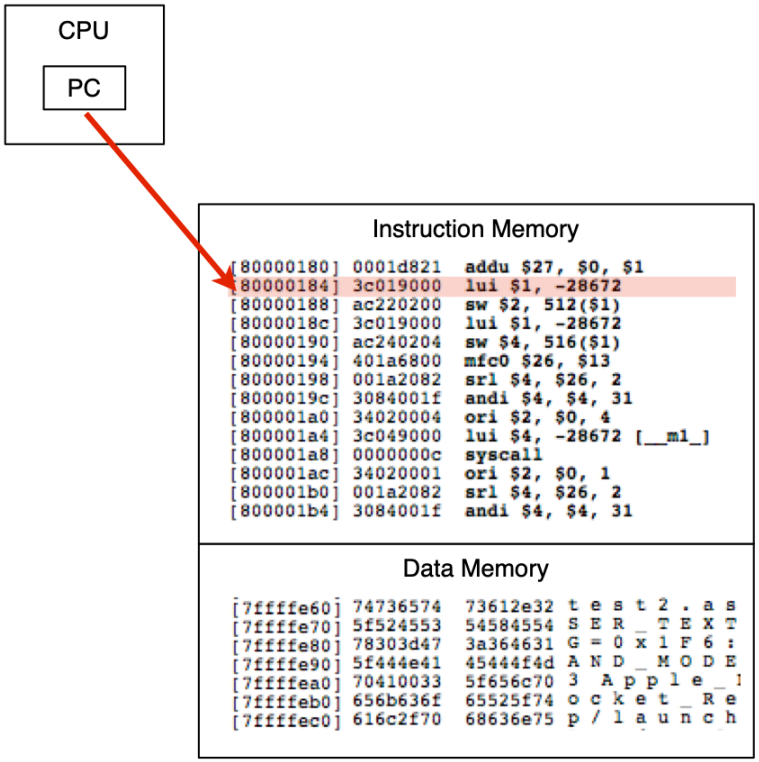
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program





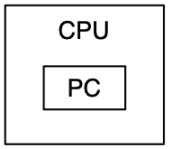
The Stored Program Computer

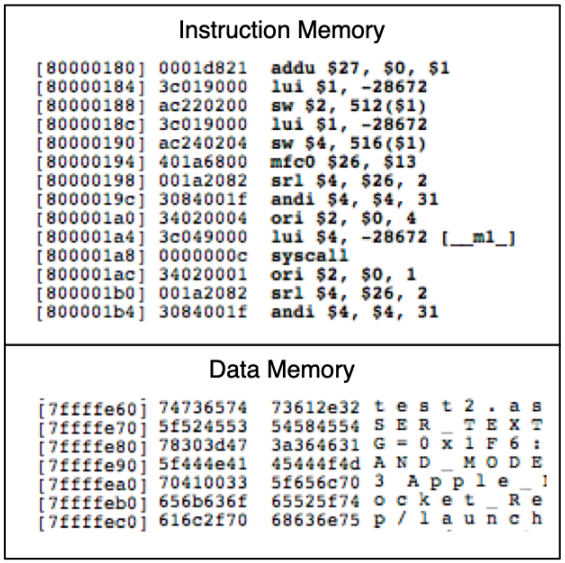
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program



The Stored Program Computer

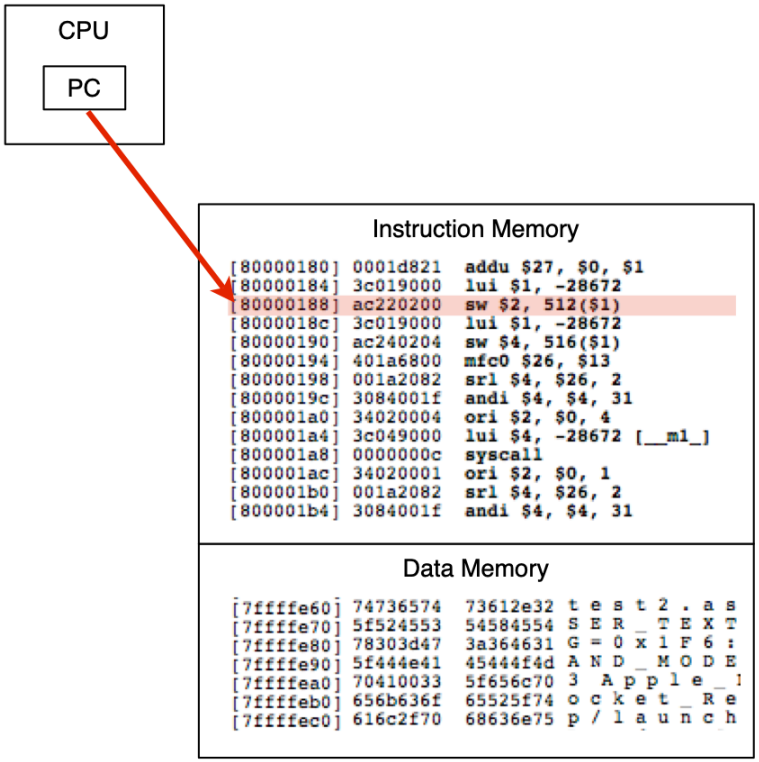
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program





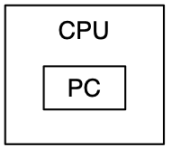
The Stored Program Computer

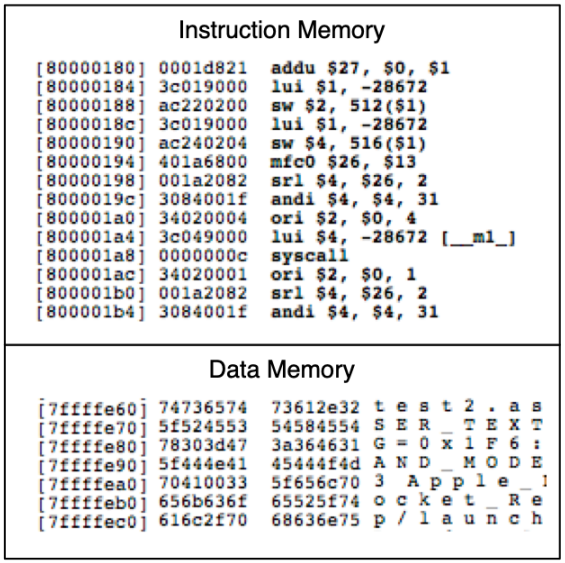
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program



The Stored Program Computer

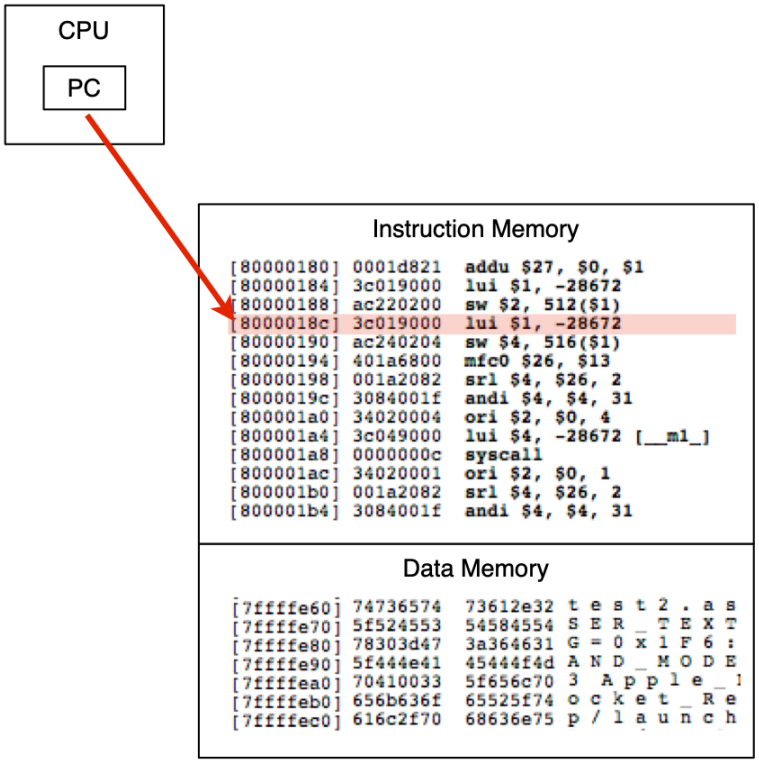
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program





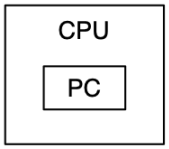
The Stored Program Computer

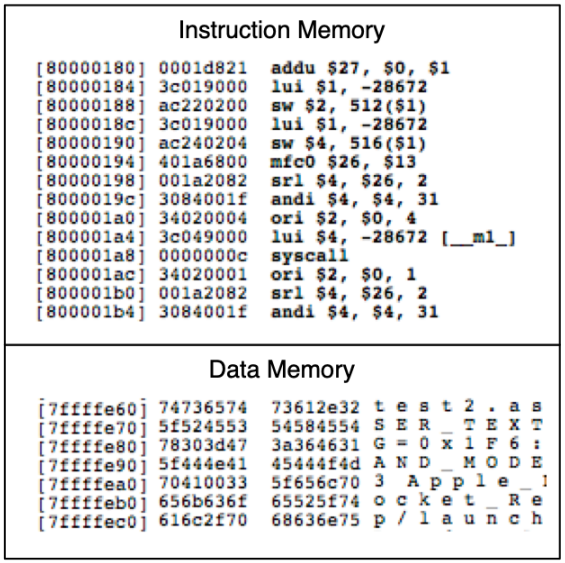
- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program



The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program





The Instruction Set Architecture (ISA)

- The ISA is the set of instructions a computer can execute
- All programs are combinations of these instructions