Key Points

- Pick up homeworks at the end of class

- Amdahl's law and how to apply it in a variety of situations
- It's role in guiding optimization of a system
- It's role in determining the impact of localized changes on the entire system

Limits on Speedup: Amdahl's Law

- "The fundamental theorem of performance optimization"
- Coined by Gene 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

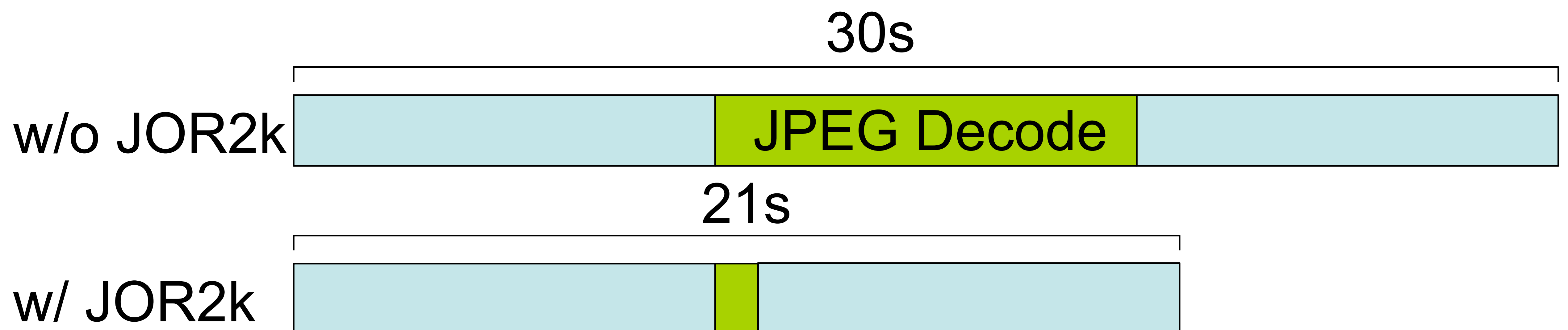
Amdahl's Law in Action

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

** Increases processor cost by 45%

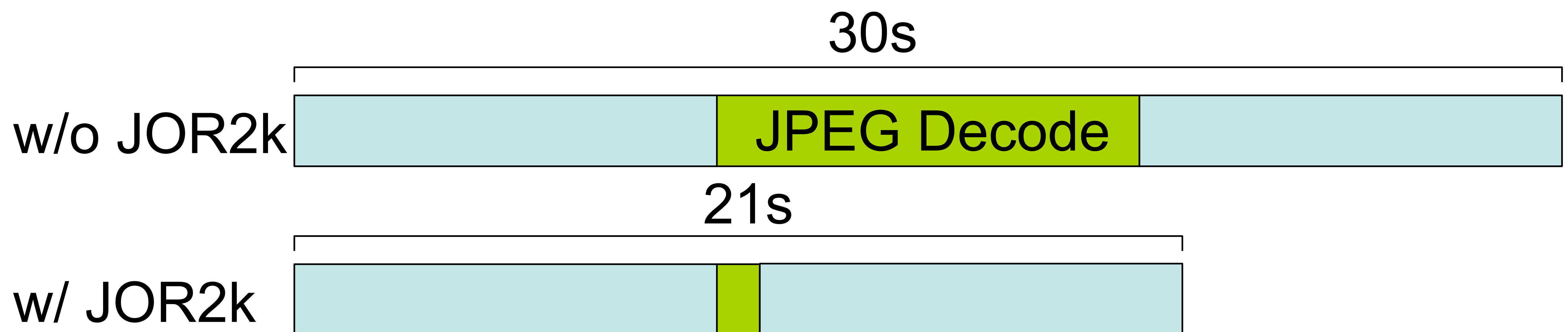
Amdahl's Law in Action

- 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

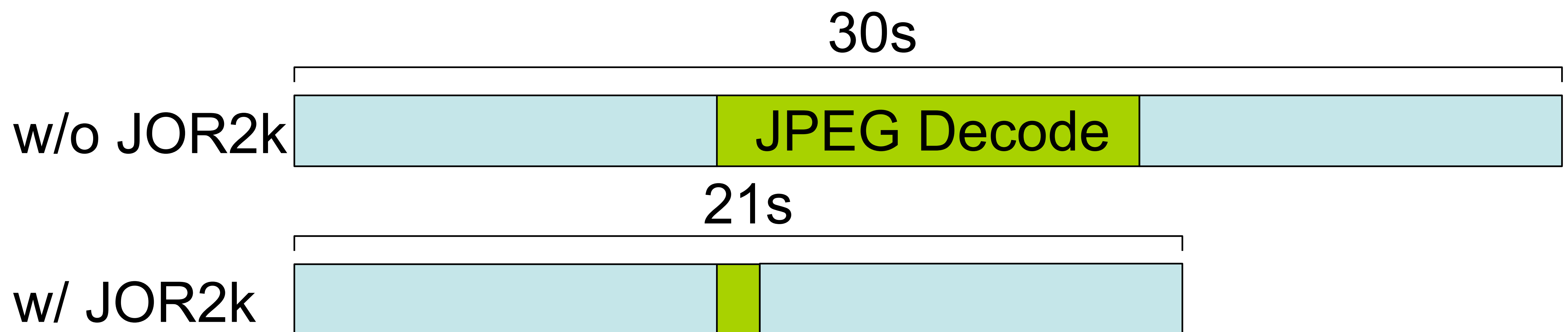
- SuperJPEG-O-Rama2010 in the wild
- PictoBench spends 33% of it's time doing JPEG decode
- How much does JOR2k help?



Performance: 30/21 = 1.4x 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?

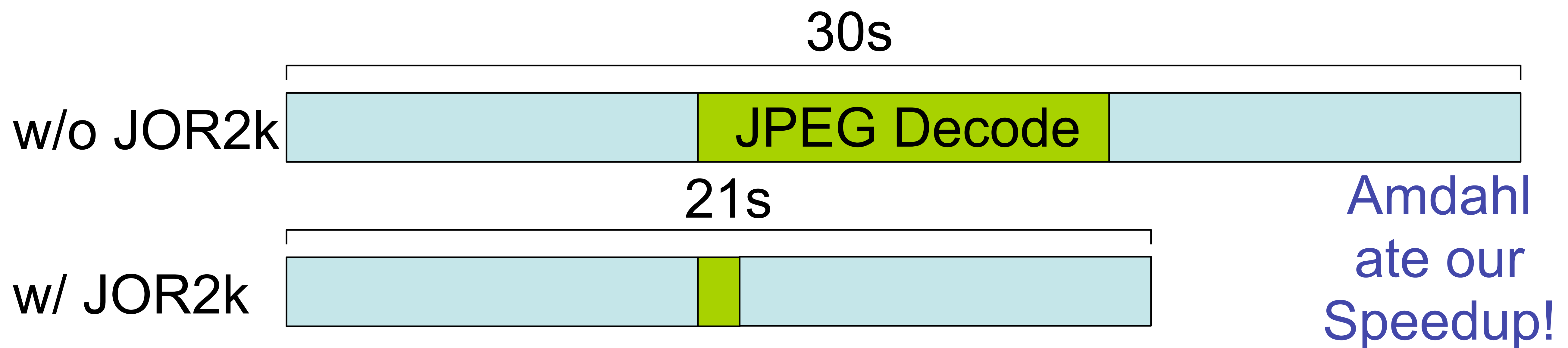


Performance: 30/21 = 1.4x Speedup != 10x

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?



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}

$$S = \textit{infinity}$$

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

Amdahl's Law Example #1

- Protein String Matching Code
  - 200 hours to run on current machine, 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?

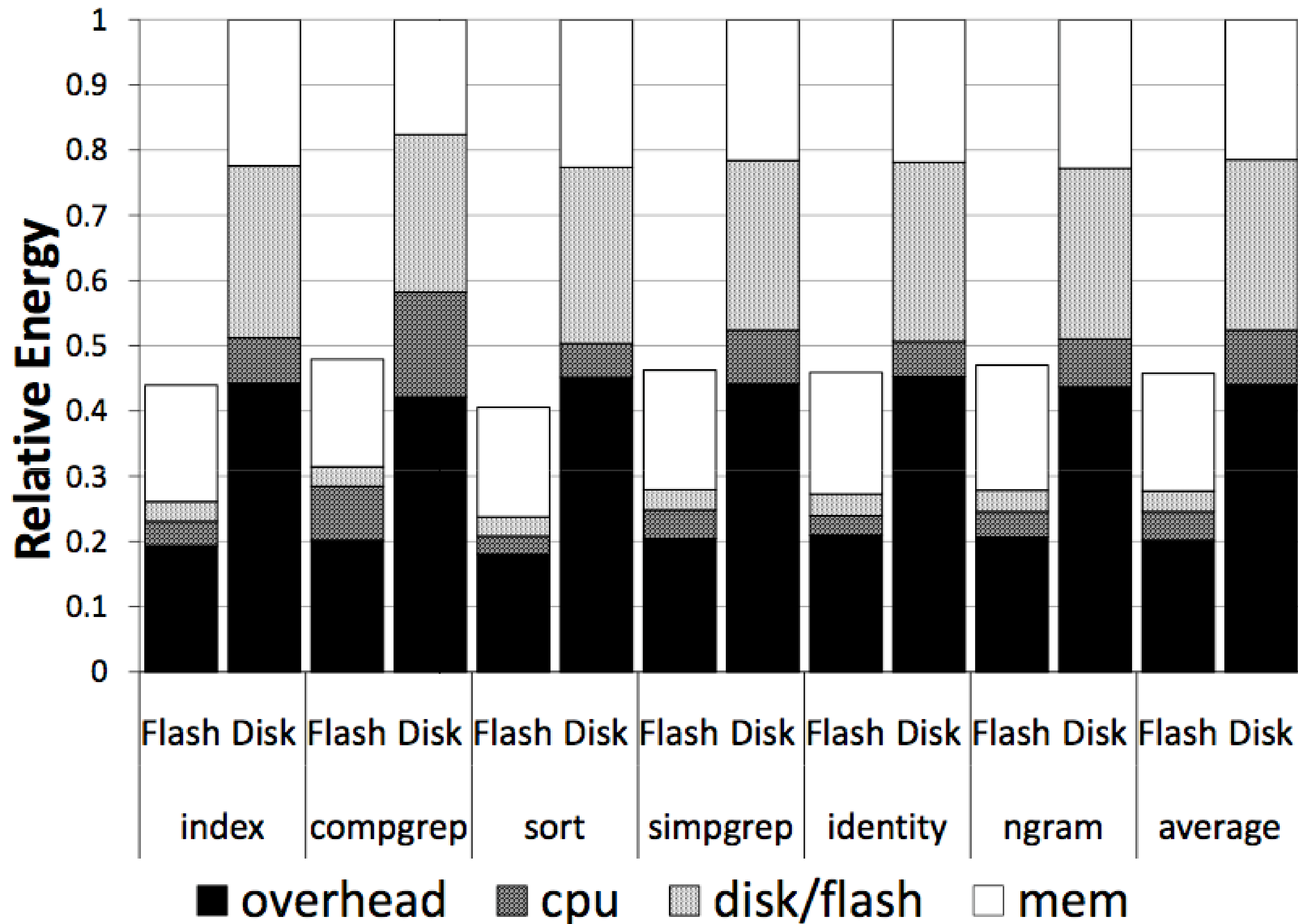
A) 1.1
B) 1.25
C) 1.75
D) 1.33

E) 10.0
F) 50.0
G) 1 million times
H) Other

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 inst takes the same amount of time)
    - Hardware optimization that reduces the latency of each IO operations from 6us to 5us.

Amdahl's Law Applies All Over



- SSDs use 10x less power than HDs
- But they only save you ~50% overall.

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 changes.
- Repeat…
  - With optimization, the common becomes uncommon and vice versa.

Amdahl's Corollary #2: Example

Common case

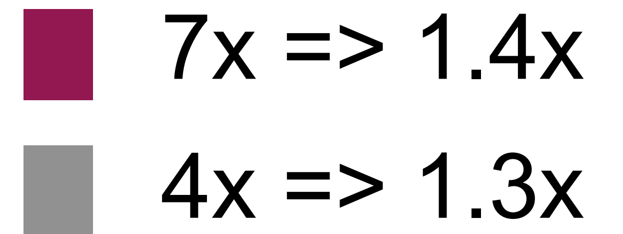
Amdahl's Corollary #2: Example

Common case

7x => 1.4x

Amdahl's Corollary #2: Example

Common case



Amdahl’s Corollary #2: Example

Common case

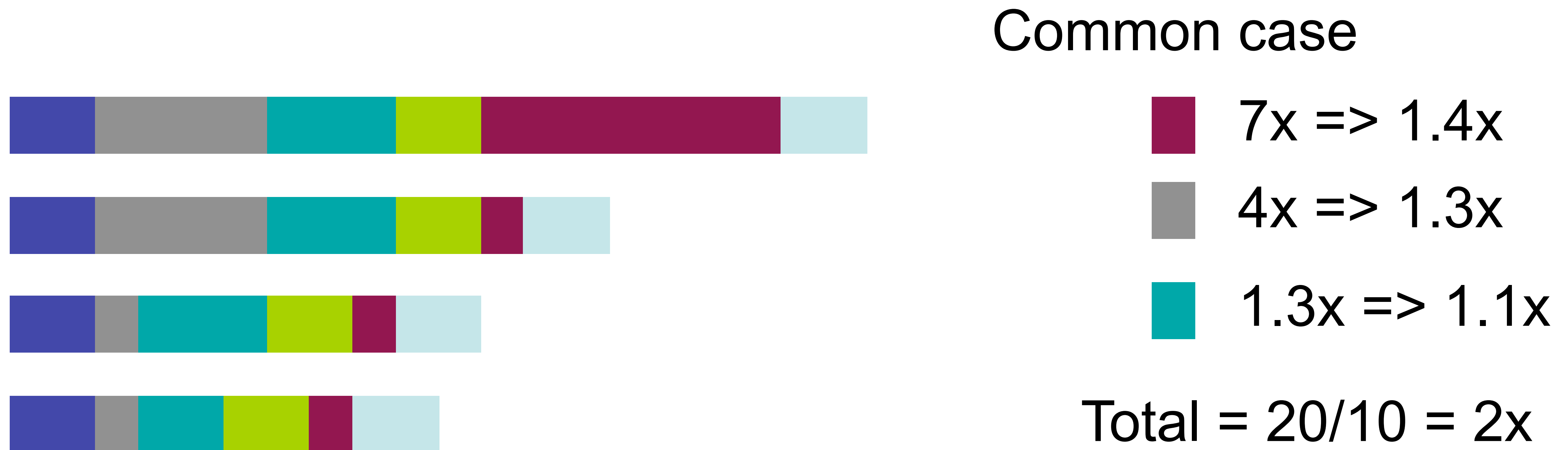
7x => 1.4x

4x => 1.3x

1.3x => 1.1x

Total = 20/10 = 2x

Amdahl's Corollary #2: Example



- In the end, there is no common case!
- Options:
  - Global optimizations (faster clock, better compiler)
  - Find something common to work on (i.e. memory latency)
  - War of attrition
  - Total redesign (You are probably well-prepared for this)

Amdahl's Corollary #3

- Benefits of parallel processing
- *p* processors
- *x*% is *p*-way parallizable
- maximum speedup, S_{par}

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

Amdahl's Corollary #3

- Benefits of parallel processing
- p processors
- x% is p-way parallizable
- maximum speedup, S_{par}

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

x is pretty small for desktop applications, even for p = 2

Amdahl's Corollary #3

- Benefits of parallel processing
- *p* processors
- *x*% is *p*-way parallizable
- maximum speedup, S_{par}

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

x is pretty small for desktop applications, even for p = 2

Does Intel's 80-core processor make much sense?

Example #3

- Recent advances in process technology have double the number transistors you can fit on your die.
- Currently, your key customer can use up to 16 processors for 50% of their application.
- You have two choices:
  - Double the number of processors from 2 to 4
  - Stick with 2 processors but add features that will allow the applications to use two cores for 70% of execution.
- Which will you choose?

Amdahl's Corollary #4

- Amdahl's law for latency (L)

$$L_{new} = L_{base} * 1/Speedup$$

$$L_{new} = L_{base} * (x/S + (1-x))$$

$$L_{new} = (L_{base}/S)*x + L_{base}*(1-x)$$

Amdahl's Non-Corollary

- Amdahl's law does not bound slowdown

  $L_{new} = (L_{base}/S)*x + L_{base}*(1-x)$

- L_{new} is linear in 1/S
- Example: x = 0.01 of execution, $L_{base} = 1$
  - S = 0.001;
    - $E_{new} = 1000*L_{base}*0.01 + L_{base}*(0.99) \sim 10*L_{base}$
  - S = 0.00001;
    - $E_{new} = 100000*L_{base}*0.01 + L_{base}*(0.99) \sim 1000*L_{base}$
- Things can only get so fast, but they can get arbitrarily slow.
  - Do not hurt the non-common case too much!

Multiple optimizations

- 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

$$S_{tot} = 1/(x_1/S_1 + x_2/S_2 + (1-x_1-x_2))$$

Note that x_1 and x_2 must be disjoint!

If not then, treat the overlap as a separate portion of execution and measure it's speed up independently

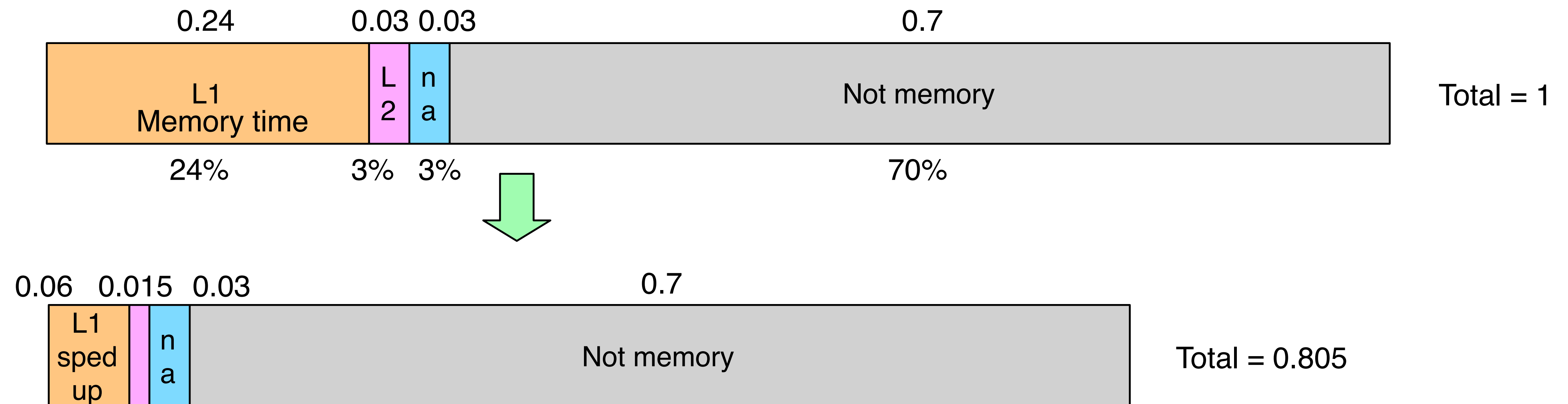
ex: we have x_{1only}, x_{2only}, and $x_{1\&2}$ and S_{1only}, S_{2only}, and $S_{1\&2}$, Then

$$S_{tot} = 1/(x_{1only}/S_{1only} + x_{2only}/S_{2only} + x_{1\&2}/S_{1\&2} + (1-x_{1only}-x_{2only}+x_{1\&2}))$$

Amdahl's Example #4

- 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 or 2.
- What is the total speed up?

Answer in Pictures



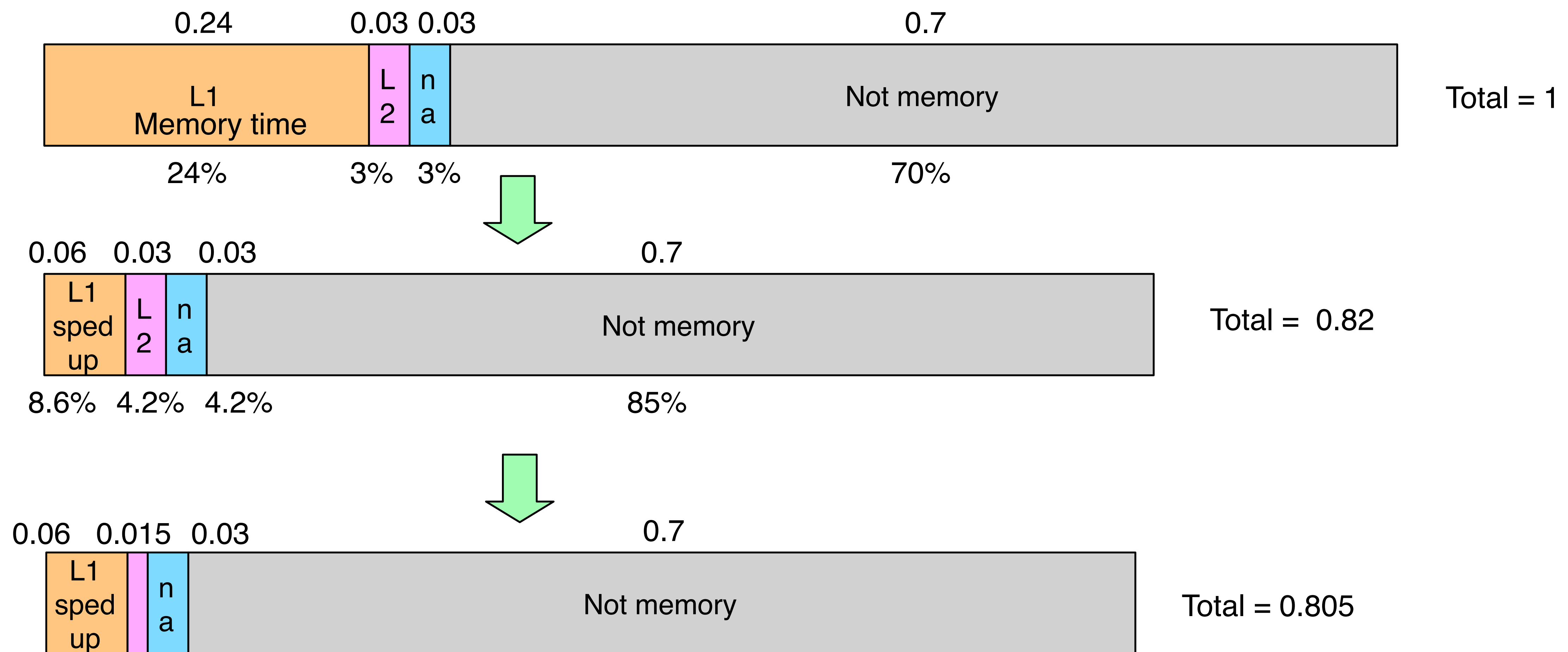
Speed up = 1.242

Amdahl's Pitfall

- You cannot trivially apply optimizations one at a time with Amdahl's law.
- Just the L1 cache
  - $S_1 = 4$
  - $x_1 = .8*.3$
  - $S_{totL1} = 1/(x_1/S_1 + (1-x_1))$
  - $S_{totL1} = 1/(0.8*0.3/4 + (1-(0.8*0.3))) = 1/(0.06 + 0.76) = 1.2195$ times
- Add the L2 cache
  - $S_{totL2}' = 1/(0.2*0.5*0.3/2 + (1-0.2*0.5*0.3)) = 1/(.015 + .97) =$ 1.02times
  - $S_{totL2} = S_{totL2'} * S_{totL1} = 1.04*1.21 = 1.234$

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

What went wrong



Amdahl's Practice

- Add the L2 cache separately and correctly
  - $S_{totL2}' = 1/(0.042/2 + (1-0.042)) = 1/(.04 + .92) = 1.02$ times
  - $S_{totL2} = S_{totL2'} * S_{totL1} = 1.02*1.21 = 1.24$

- Combine both the L1 and the L2
  - $S_2 = 2$
  - $x_2 = .1$
  - $S_{totL2} = 1/(x_1/S_1 + x_2/S_2 + (1 - x_1 - x_2))$
  - $S_{totL2} = 1/(0.8*0.3/4 + 0.1*0.3/2 + (1-(0.8*0.3)-(0.1*0.3)))$
    $= 1/(0.06+0.015+.73)) = 1.24$ times

- Remember: Amdahl's law is about the fraction of *time* spent in the optimized and un-optimized portions.

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”

Measuring Bandwidth

- Measure how much work is done
- Measure latency
- Divide

Latency-BW Trade-offs

- Often, increasing latency for one task and increase BW for many tasks.
  - Think of waiting in line for one of 4 bank tellers
  - If the line is empty, your response time is minimized, but throughput is low because utilization is low.
  - If there is always a line, you wait longer (your latency goes up), but there is always work available for tellers.
- Much of computer performance is about scheduling work onto resources
  - Network links.
  - Memory ports.
  - Processors, functional units, etc.
  - IO channels.
  - Increasing contention for these resources generally increases throughput but hurts latency.

Reliability Metrics

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

Power/Energy Metrics

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

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 xtr switchings
  - C -- smaller chips, with fewer xtrs

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
  - Power * Latency
- More concerned about Power?
  - $Power^2$ * Latency
- High bandwidth, low cost?
  - (MB/s)/$
- In general, put the good things in the numerator, the bad things in the denominator.
  - $MIPS^2/W$

Stationwagon Digression

- IPv6 Internet 2: 272,400 terabit-meters per second
  - 585GB in 30 minutes over 30,000 Km
  - 9.08 Gb/s

- Subaru outback wagon
  - Max load = 408Kg
  - 21Mpg
- MHX2 BT 300 Laptop drive
  - 300GB/Drive
  - 0.135Kg
- 906TB
- Legal speed: 75MPH (33.3 m/s)
- BW = 8.2 Gb/s
- Latency = 10 days
- 241,535 terabit-meters per second

Prius Digression

- IPv6 Internet 2: 272,400 terabit-meters per second
  - 585GB in 30 minutes over 30,000 Km
  - 9.08 Gb/s

- My Toyota Prius
  - Max load = 374Kg
  - 44Mpg (2x power efficiency)
- MHX2 BT 300
  - 300GB/Drive
  - 0.135Kg
- 831TB
- Legal speed: 75MPH (33.3 m/s)
- BW = 7.5 Gb/s
- Latency = 10 days
- 221,407 terabit-meters per second (13% performance hit)

Benchmarks: Standard Candles for Performance

- It's hard to convince manufacturers to run *your* program (unless you're a BIG customer)
- A benchmark is a set of programs that are representative of a class of problems.
- To increase predictability, collections of benchmark applications, called *benchmark suites*, are popular
  - "Easy" to set up
  - Portable
  - Well-understood
  - Stand-alone
  - Standardized conditions
  - These are all things that real software is not.

Classes of benchmarks

- Microbenchmark – measure one feature of system
  - e.g. memory accesses or communication speed
- Kernels – most compute-intensive part of applications
  - e.g. Linpack and NAS kernel b'marks (for supercomputers)
- Full application:
  - SpecInt / SpecFP (int and float) (for Unix workstations)
  - Other suites for databases, web servers, graphics,...