Performance and Cost - Roadmap

- Performance metrics
- Benchmarks and benchmarking
- Averaging
- Iron law of performance
- Amdahl’s law
- Balance and bursty behavior

A is Faster than B means:

Machine A is **n times faster** than machine B iff:

$$\frac{Perf(A)}{Perf(B)} = \frac{\frac{1}{Time(A)}}{\frac{1}{Time(B)}} = \frac{Time(B)}{Time(A)} = n$$

Machine A is **X% faster** than machine B iff:

$$\frac{Perf(A)}{Perf(B)} = \frac{Time(B)}{Time(A)} = 1 + \frac{X}{100}$$

EXAMPLE: A 10 sec, B 15 sec

- 15/10 = 1.5 => A is 1.5 times faster than B

A is Faster than B cont.

BUT: There are two parameters TIME and TASK:

What is Time?

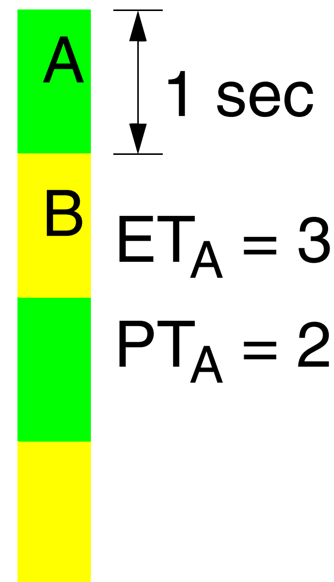
What is is the TASK we measure?

Performance Metrics: Latency vs. Bandwidth

"Computer A is FASTER than Computer B?

Time or **Latency:** *How long it takes to do something*

- Elapsed time: real time
- Processor time: computation component



Rate or **Bandwidth:** *How much work done per time.*

Rate = Work per time

Your goals dictate which one is the appropiate one for you.

Example: User vs. Data processing center.

Our Emphasis will be on Processor Time or Elapsed Time

A is Faster Than B? On What?

- Cars: Car A goes from 0 to 100 mph in 10 secs.

Task is important.

- How do we define meaningful tasks for comparing Computers?
- Let’s look at some unsuccessful attemps:

MIPS

MFLOPS

Frequency

- Mhz or Ghz:
- Hope is that:
  - — if GHz(A) > GHz (B) then Perf(A) > Perf(B)
  - — if Ghz(A) = a x GHz(B) then Perf(A) ~ a x Perf(B)

Meaningless, frequency has little to do with amount of work produced per unit of time

This is true even if the ISA is the same

MIPS and what's wrong with them

Million **I**nstructions **P**er **S**econd

$$\textbf{MIPS} = \frac{InstructionCount}{ExecutionTime \times 10^6} = \frac{ClockRate}{ClocksPerInstruction \times 10^6}$$

Intention: if $MIPS_A > MIPS_B$ then A faster/better than B

- Instruction sets are not equivalent: *add [bx+10], ax*
- Different programs use different instruction mix
- Instruction count is not a reliable indicator of work
  - some optimizations add/remove instructions
  - instructions may have varying work: *rep movs*

MFLOPS

$$\textbf{MFLOPS} = \frac{FloatingPointOPS}{Time \times 10^6}$$

- Program must be floating-point intensive
- Ignores other instructions (e.g., loads and stores)
- In the extreme, some programs have no FP ops

Safe interpretation of Peak MFLOPS:

What the manufacturer guarantees not to be able to exceed

Normalized MFLOPS

Normalized FP: assign a canonical # FP ops to a HLL program

Normalized MFLOPS = {# canonical FP ops / time} x 10^{-6}

Not all machines implement the same FP ops

- Cray does not implement divide
- Motorola has SQRT, SIN, and COS

Not all FP ops are same work

- adds usually faster than divide

Relative MIPS

relative MIPS = $(time_{ref} / time_{new}) \times MIPS_{ref}$

- e.g., VAX MIPS
- Somewhat better than absolute MIPS
- Sensitive to reference machine
    - amplifies programs where the ref. machine is weak
    - makes other programs less important
    - same applies to machine features

Compiler, ISA, OS have an impact

Still, maybe useful for same ISA, compiler, OS and workload

Benchmarks and Benchmarking

• In lack of a universal task pick some programs that represent common tasks

• Use these programs to compare performance of systems:

Compilers

Compression

Weather Simulation

CAUTION:

Comparisons are as good as the benchmarks are in representing your real workload.

Many parameters affect measured performance

Benchmark Types

Real programs

- representative of real workload
- best way to characterize performance
- requires considerable work

Kernels

- “representative” program fragments
- good for focussing on individual features - not big picture

Mixes

- instruction frequency

Benchmark Types

Toy benchmarks

- e.g., fibonacci, prime number, towers of Hanoi
- little value

Synthetic benchmarks

- programs intended to give specific mix
- worse than toy?
- Representative of what?
- Some value if carefully chosen

Benchmarking Process

1. Define workload:

    What applications to use

2. Extract benchmarks from applications:

    Convert into self-contained, non-interactive programs.

3. Choose metric:

    How to summarize performance

4. Execute programs, collect measurements and report performance

Source: J. Smith

SPEC95 CPU Benchmark

Integer

• go	plays a game of go
• m88ksim	motorola 88000 CPU simulator
• gcc	compiler
• compress	data compress/decompress
• li	lisp interpreter
• jpeg	graphics jpeg compression/decompression
• perl	perl language interpreter
• vortex	object-oriented database system

SPEC CPU is CPU Bound

- Assumption:

  program spends most of its time in user space

  does very little I/O

- Good for measuring CPU/memory system performance
- “Not good” for other application domains, e.g., databases, graphics

  Provides only a hint

- Integer & Floating Point benchmarks

SPEC95 Benchmark

Floating point

- tomcatv vectorized mesh generation
- swim shallow water model - finite differences
- su2cor quantum physics
- hydro2d galactic jets - navier stokes
- mgrid multigrid solver for 3d field
- applu partial differential equations
- turb3d simulation of turbulence in a cube
- apsi temperature and wind velocity
- fppp quantum chemistry
- wave5 n-body Maxwell's

SPEC CPU2000 Benchmark

NAME	REF Time	Description
164.gzip	1400	Data compression utility
175.vpr	1400	FPGA circuit placement and routing
176.gcc	1100	C compiler
181.mcf	1800	Minimum cost network flow solver
186.crafty	1000	Chess program
197.parser	1800	Natural language processing
252.eon	1300	Ray tracing
253.perlbmk	1800	Perl
254.gap	1100	Computational group theory
255.vortex	1900	Object Oriented Database
256.bzip2	1500	Data compression utility
300.twolf	3000	Place and route simulator

SPEC CPU2000 Benchmark

SpecCPU FP

168.wupwise	1600	Quantum chromodynamics
171.swim	3100	Shallow water modeling
172.mgrid	1800	Multi-grid solver in 3D potential field
173.applu	2100	Parabolic/elliptic partial differential equations
177.mesa	1400	3D Graphics library
178.galgel	2900	Fluid dynamics: analysis of oscillatory instability
179.art	2600	Neural network simulation; adaptive resonance theory
183.equake	1300	Finite element simulation; earthquake modeling
187.facerec	1900	Computer vision: recognizes faces
188.ammp	2200	Computational chemistry
189.lucas	2000	Number theory: primality testing
191.fma3d	2100	Finite element crash simulation
200.sixtrack	1100	Particle accelerator model
301.apsi	2600	Solves problems regarding temperature, wind, velocity and distribution of pollutants

CHECK WWW.SPEC.ORG for more info

SPEC CPU 2006

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

SPEC CPU 2006 contd.

410.bwaves	Fortran	Fluid Dynamics
416.gamess	Fortran	Quantum Chemistry
433.milc	C	Physics: Quantum Chromodynamics
434.zeusmp	Fortran	Physics / CFD
435.gromacs	C/Fortran	Biochemistry/Molecular Dynamics
436.cactusADM	C/Fortran	Physics / General Relativity
437.leslie3d	Fortran	Fluid Dynamics
444.namd	C++	Biology / Molecular Dynamics
447.dealII	C++	Finite Element Analysis
450.soplex	C++	Linear Programming, Optimization
453.povray	C++	Image Ray-tracing
454.calculix	C/Fortran	Structural Mechanics
459.GemsFDTD	Fortran	Computational Electromagnetics
465.tonto	Fortran	Quantum Chemistry
470.lbm	C	Fluid Dynamics
481.wrf	C/Fortran	Weather Prediction
482.sphinx3	C	Speech recognition

Why A New Version?

- Programs evolve
- Benchmarks become obsolete

  New Applications Appear

  Existing Applications may Scale

  Compilers/Architectures are tuned to existing ones

Other SPEC Benchmarks

- SPEC started with the CPU benchmarks
- Other benchmarks available

- Parallel Programs

- Graphics

- Filesystem

check www.spec.org

MediaBench

Developed at UCLA

Collection of Media-Oriented Applications

IJPEG	Image Compression/Decompression
MPEG	Movie Compression/Decompression
GSM	Audio Encoding/Decoding 8Khz 13-bit samples
ADPCM	Speech Encoding/Decoding
G.721	Guess....
PGP	Public Key-based Cryptography
PEGWIT	Ditto
Ghostscript	Postscript Interpreter
Mesa	3D Graphics Library (API)
SPEECH	Speech Processing Library
RASTA	Speech Recognition Components
EPIC	Image Compression

SPLASH II

- Multiprocessor benchmark
- Developed at Stanford
- Scientific applications and kernels
- Our focus is on uni-processor architecture
- Can be run in uniprocessor mode

Kernel Example

inner product

```
DO 3 L = 1, LP
    Q = 0.0
DO 3 K = 1,N
    Q = Q + Z(K)*X(K)
```

Synthetic Benchmark Example

Dhrystone, Whetstone

```
X = 1.0
Y = 1.0
Z = 1.0
DO 88 I = 1, N8, 1
   CALL P3(X,Y,Z)
```

```
SUBROUTINE P3(X,Y,Z)
COMMON T, T2
X1 = X
Y1 = Y
X1 = T * (X1 - Y1)
Y1 = T * (X1 + Y1)
Z  = (X1 + Y1)/T2
RETURN
```

Mix Example

Gibson Mix - developed in 1950's at IBM

• load/store	31%	branches	17%
• fixed add/sub	6%	compare	4%
• float add/sub	7%	float mult	4%
• float div	2%	fixed mul	1%
• fixed div	<1%	shifts	4%
• logical	2%		

generally speaking, these numbers are still valid today

Summarizing Performance

Consider:

	Computer A	Computer B	Computer C
Program P1	1	10	20
Program P2	1000	100	20
Program P3	1001	110	40

- Can answer: X is faster than Y for program Z
- But which is faster overall?

“Need” a way of summarizing performance

Total Execution Time

• Given $Time(X)_i$ the time it takes to run program i on computer X, measure:

$$\frac{Perf(A)}{Perf(B)} = \frac{\sum Time(B)_i}{\sum Time(A)_i}$$

In our previous example: B is 9.1 times faster than A

+ Consistent Summary Metric

if this your exact workload

- Longer running programs dominate

Over-emphasizes their importance

Arithmetic Mean

- Use (n is the number of benchmarks):

$$Time(A) = \frac{1}{n}\sum Time(A)_i$$

- In our previous example:

Time(A) = (1 + 1000 + 1001) / 3 = 677.33

Time(B) = (10 + 100 + 110) / 3 = 73.33

B is 9.1 times faster than A

Same as time ratio

Weighted Arithmetic Mean

- Assign Weight to each benchmark that better represents an unequal mix:

$$Time(A) = \sum Weight_i \times ActualTime(A)_i$$

- Could be used to give equal importance to each benchmark

- But really we are playing with numbers

Good only when we know the exact mix (embedded systems?)

How about Rates?

- What if we are given performance as a rate, e.g., IPC
- Can we use AM? Let's see. Consider speed:

  30 mph for first 10 miles

  90 mph for next 10 miles. average speed?

  Average speed = (30+90)/2 WRONG

  Average speed = total distance / total time

  - (20 / (10/30+10/90)) = 45 mph

This is the HARMONIC MEAN

Harmonic Mean

Harmonic mean of rates $= \dfrac{n}{\left\{\sum_{1}^{n} \dfrac{1}{rate(i)}\right\}}$

Use HM if forced to start and end with rates

Dealing with ratios

Performance is often reported normalized to a reference machine

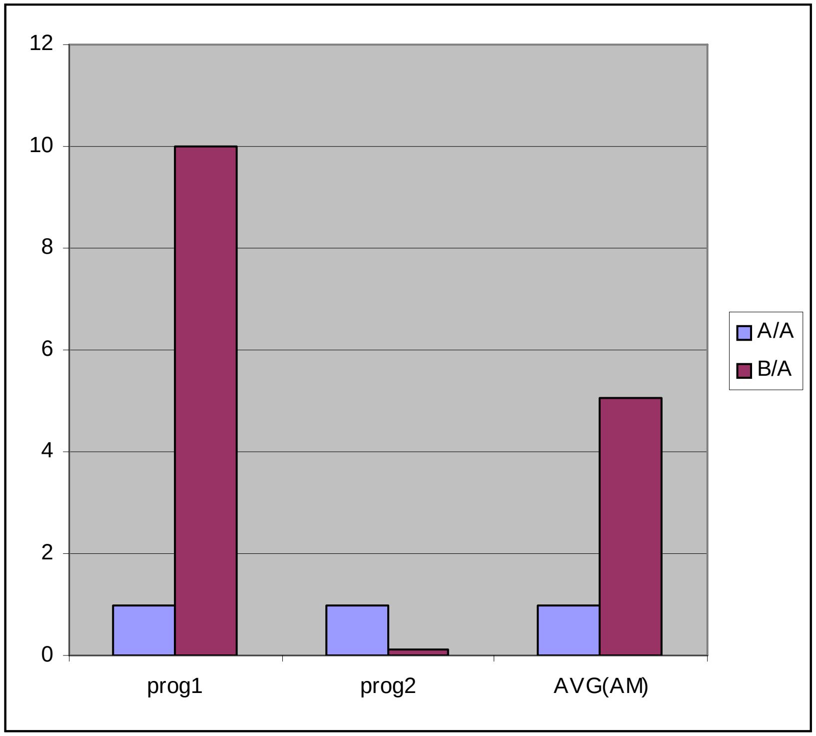
This is what SPEC does.

Can we use AM? NO. Example:

	Machine A			Machine B		
	TIME	/A	/B	TIME	/A	/B
Program 1	1	1	0.1	10	10	1
Program 2	1000	1	10	100	0.1	1
AM	**500.5**	**1**	**5.5**	**55**	**5.5**	**1**
Total Time	**1001**	**2**	**10.1**	**110**	**10.1**	**1.0**

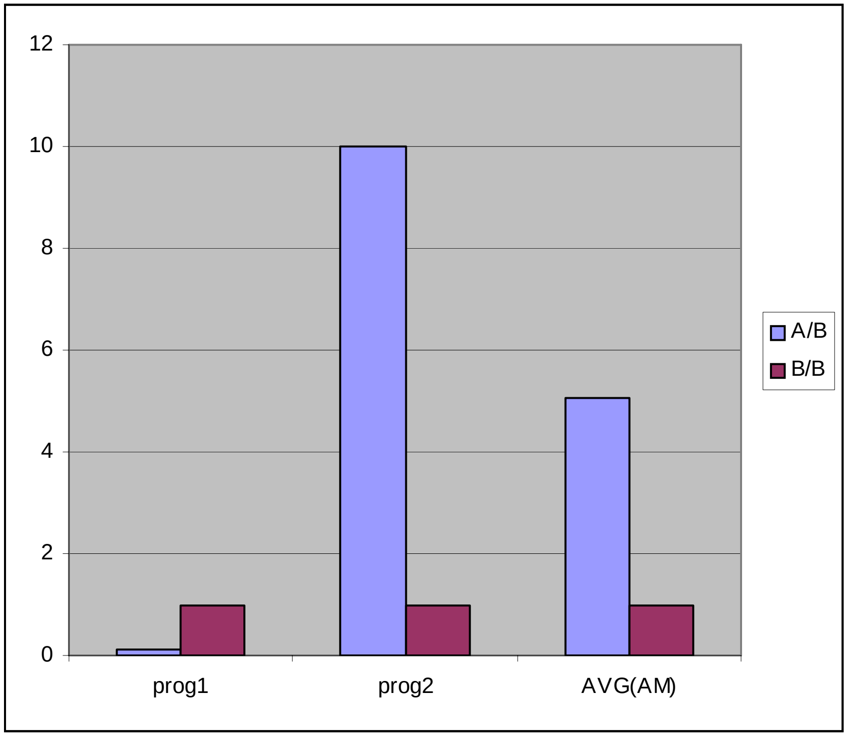
Dealing with ratios

Normalized over A



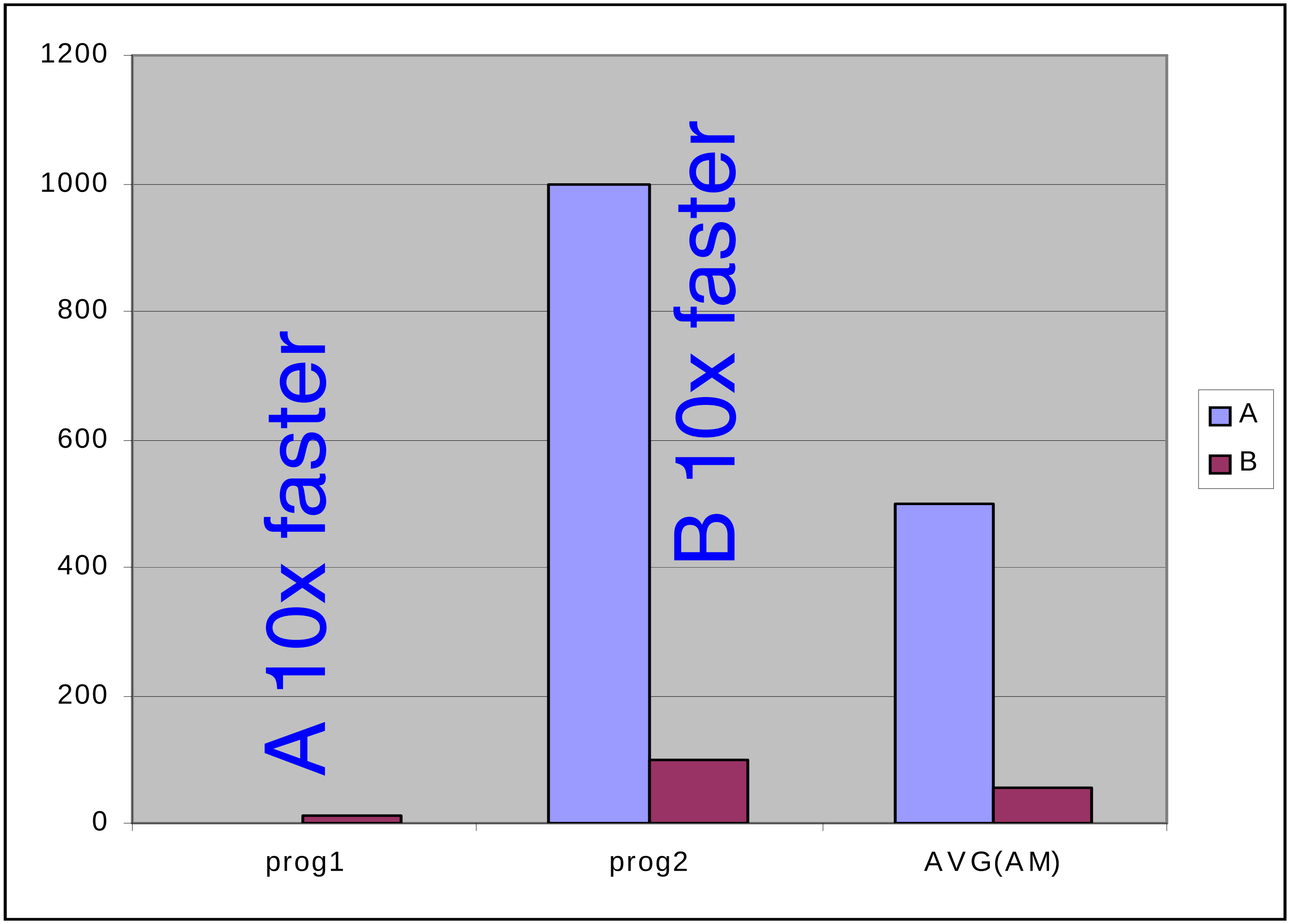
Conclusion A is better

Normalized over B



Conclusion B is better

Dealing with ratios



Spec Uses Geometric Mean

- Geometric Mean:

$$\sqrt[n]{\prod ExecutionTimeRatio_i}$$

- Independent of the particular running times.
- All benchmarks are equal
- But does not predict execution time

In our Example GM says A = B

- It over-emphasizes the easy cases

Generally, GM will mispredict for three or more machines

SPEC Benchmarking Process

steps:

- for each benchmark i, look up $T_{base, i}$
- foreach benchmark i, run target machine to get $T_{new, i}$
- compute geometric mean: $\sqrt[n]{\prod_{1}^{n} \frac{T_{base, i}}{T_{new, i}}}$

SPEC Benchmarking Process

Steps:

- extract bechmarks from applications
- choose performance metric
- execute benchmarks on candidate machines
- project performance in new machine

Means Compared

• Gmean gives equal reward for speeding up all benchmarks

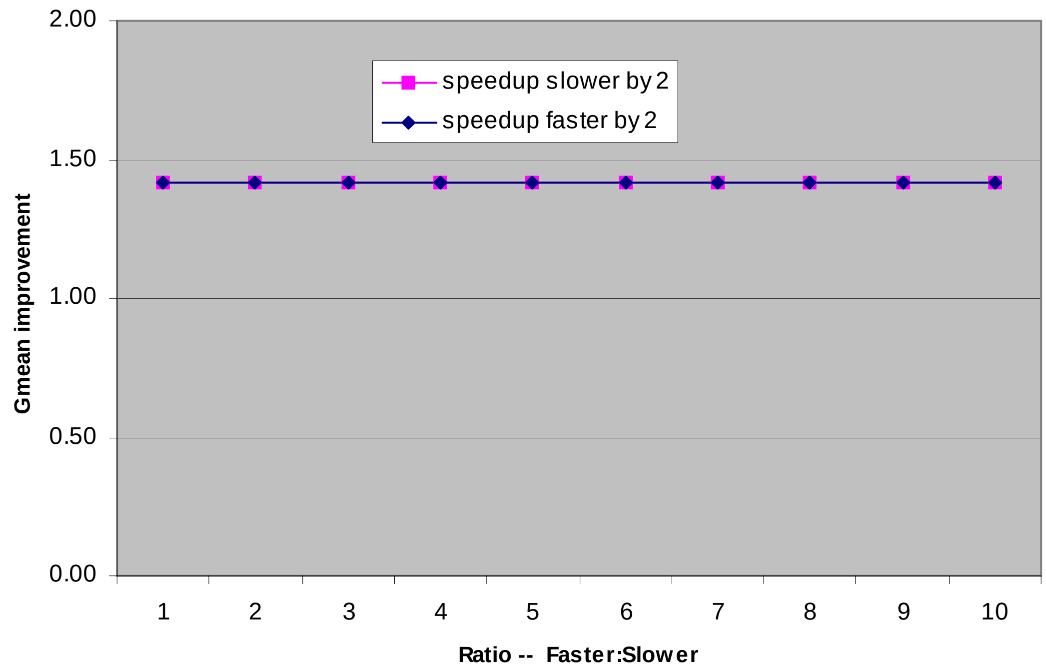
- the already fast programs get faster

• Hmean gives greater reward for speeding up the slow benchmarks

- Consistent with Amdahl's law

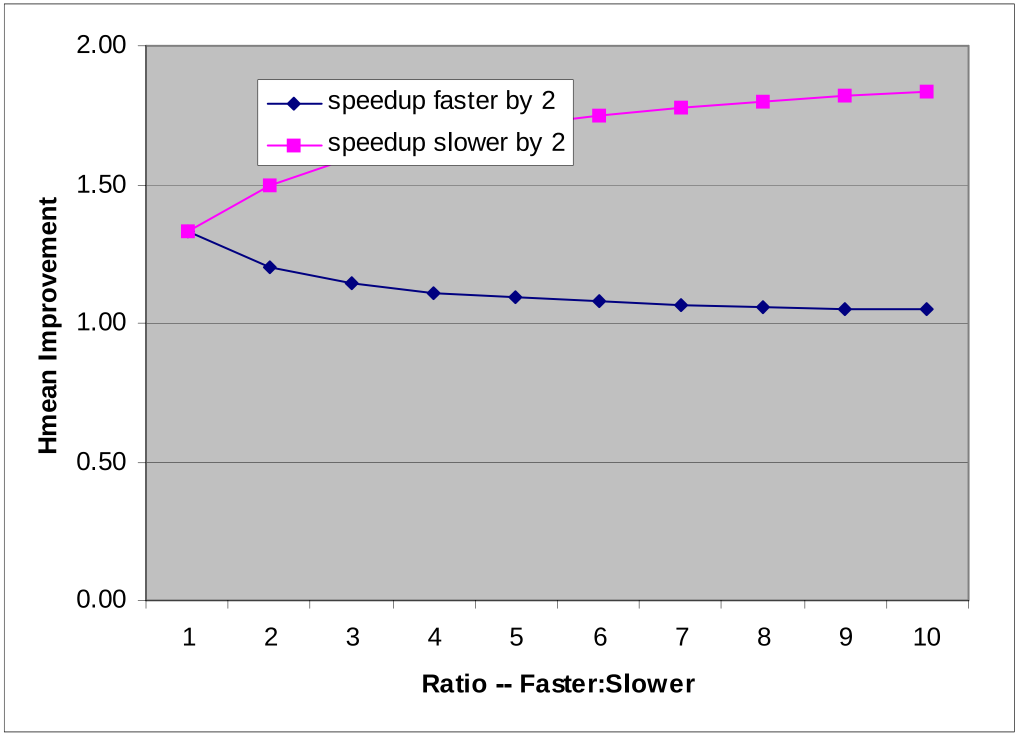
• Arithmetic mean gives greater reward for speeding up already fast benchmark

Reward for Speeding Up Slow Benchmark (Gmean)



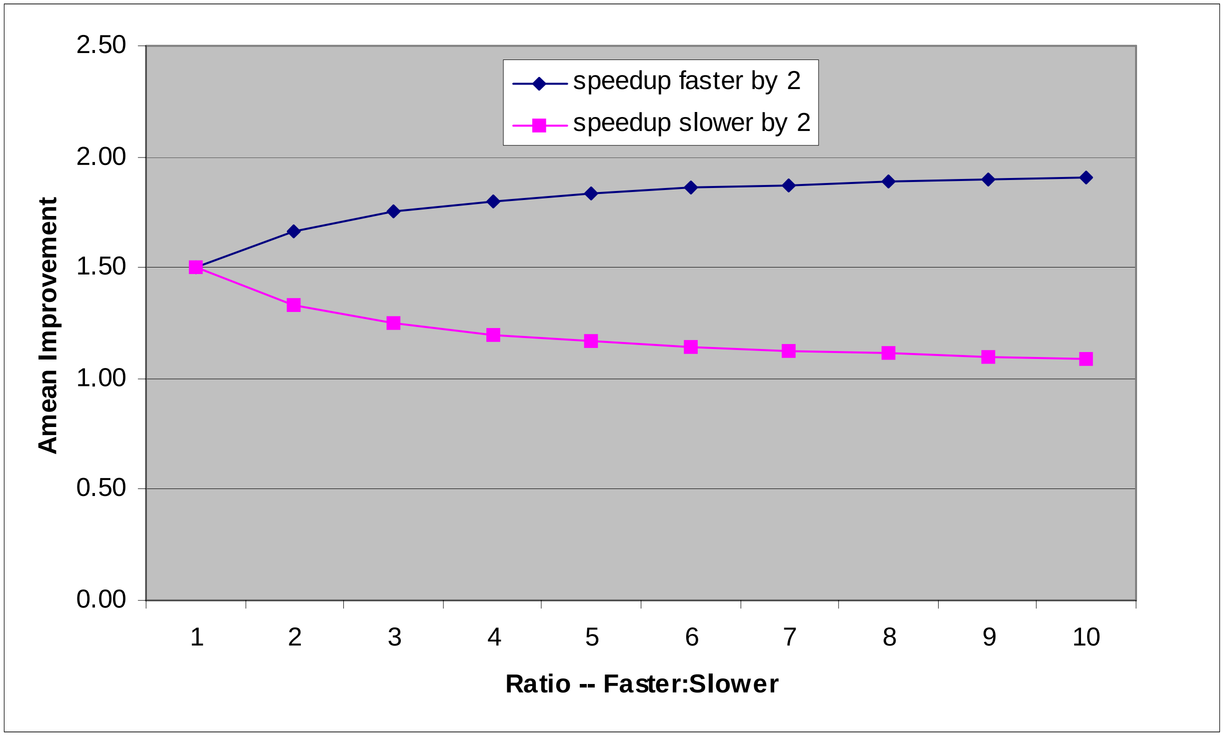
J. Smith

Reward for Speeding Up Slow Benchmark (Hmean)



J. Smith

Reward for Speeding Up Slow Benchmark (Amean)



J. Smith

Summary of Summarizing Peformance

- Absolute time: Use AM
- Ratios, e.g., IPC: Use HM
- Speedups/relative performance: Use GM

I suggest reporting detailed results so one can decide what is important for their target application

Pitfalls

Choosing benchmarks from the wrong application space

- e.g., for 3d gaming, choosing Microsoft Word

Choosing benchmarks from no application space

- e.g., synthetic workloads

Using toy benchmarks

- e.g., used to prove the value of RISC in early 80’s

Mismatch of benchmark properties with scale of features studied

- e.g., using SPEC for large cache studies

Pitfalls

Carelessly scaling benchmarks

- truncating benchmarks
- using only first few million instructions
- reducing program data size

Carelessly extracting or constructing benchmarks

- Ghostscipt in Mediabench
- Output is written in a file in ASCII (one char per bit)

Too many easy cases

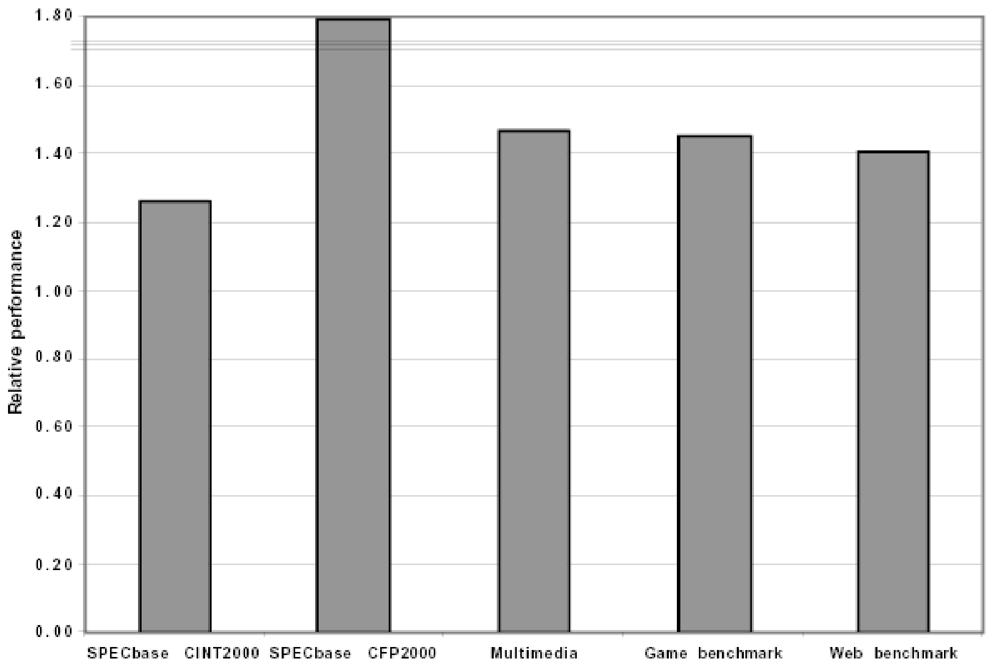
- may not show value of a feature

Too few easy cases

- may exaggerate importance of a feature

Fallacies

The relative performance of two processors with the same ISA can be judged by clock rate or by the performance of a single benchmark suite.



P4 1.7GHz performance over P3 1Ghz

- Not linear with respect to frequency

Fallacies

The best design for a computer is the one that optimizes the primary objective without considering implementation.

Time-to-market (completion). Probability of design errors.

Neglecting the cost of software in either evaluating a system or examining cost-performance.

Software can be a big part of the total cost.

Simulation

- Cannot afford to build every interesting configuration
- Often, mechanism implementation does not exist:
  - Q? of the type “what if we could do this...what performance we could expect?
- Simulate to **estimate** performance

Simulator Models

- Trace vs. Execution driven
- Functional vs. Timing
- Execution Driven:

- Must emulate system calls: map to host/use pre-recorded results

- Not absolute time: only cycles

- Results only as good as your model/benchmarks

- Validation should be done

Iron Law of Performance

$$\textbf{CPUtime} = IC \times CPI \times ClockCycleTime$$

IC = Instruction Count

- instrs executed NOT static code
- mostly determined by program, compiler, ISA

CPI = Clocks Per Instruction

- mostly determined by ISA and CPU organization
- overlap among instructions makes this smaller

ClockCycleTime:

- mostly determined by technology and CPU organization

CPU Performance contd.

$$\text{CPU Time} = ClockCycleTime \times \sum_i IC_i \times CPI_i$$

Where IC_i and CPI_i refer to specific instructions or categories of instructions

Example

Op	Frequency	Cycle count
ALU ops	43%	1
Loads	21%	1
Stores	12%	2
Branches	24%	2

Assume stores can execute in 1 cycle by slowing clock 15%

Should this be implemented?

Example, contd.

Old CPI = 0.43 + 0.21 + 0.12 x 2 + 0.24 x 2 = 1.36

New CPI = 0.43 + 0.21 + 0.12 + 0.24 x 2 = 1.24

Speedup = old time/new time

= {P x old CPI x T}/{P x new CPI x 1.15 T}

= 1.36 / (1.24 x 1.15) = 0.95

Answer: Don't make the change

Iron Law, contd.

- Clock Cycle Time: hardware technology and organization
- CPI: Organization and instruction set architecture
- Instruction Count: Instruction Set Architecture and Compiler

Amdahl's Law: Making the Common Case Fast

Performance impact of optimizing part of a program:

$$Speedup = \frac{OldTime}{NewTime} = \frac{NewRate}{OldRate}$$

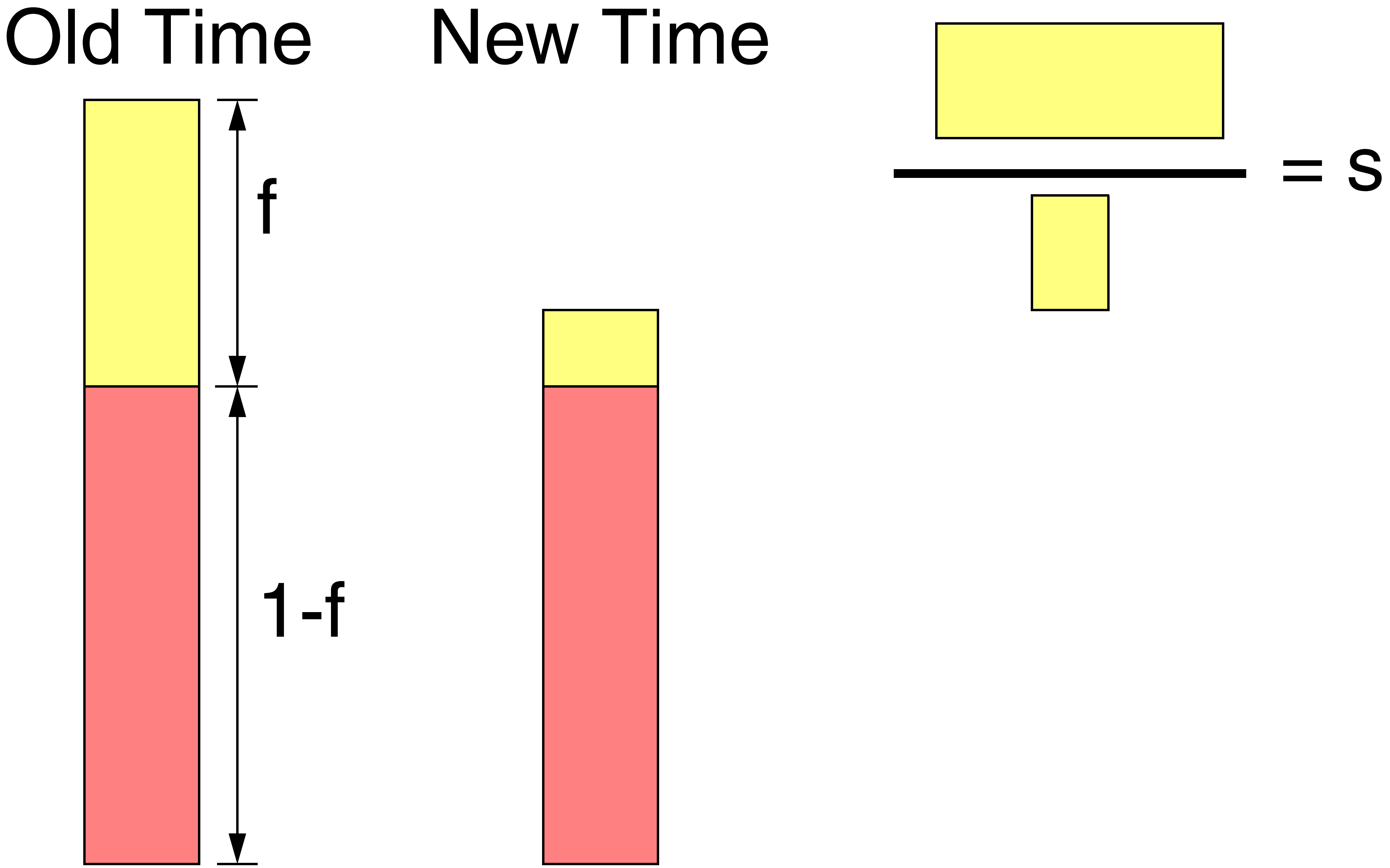
Let an optimization speed ***f*** fraction of time by a factor of **s**:

$$NewTime = OldTime \times \left[(1 - f) \times 1 + f \times \frac{1}{s}\right]$$

$$Speedup = \frac{OldTime}{OldTime \times \left[(1 - f) + \frac{f}{s}\right]} = \frac{1}{1 - f + \frac{f}{s}}$$

s > 1.0 for speedup, f <= 1.0 as it is a fraction :-)

Amdhal's Law



Amdahl's Law - Example

f = 95% and s = 1.10 - speedup common case

SPEEDUP = 1/((1-0.95) + (0.95/1.10)) = 1.094, or 9.4%

f = 5% and s = 10.00 - speedup uncommon case

SPEEDUP = 1/((1-0.05) + (0.05/10)) = 1.047, or 4.7%

f = 5% and s $\rightarrow$ ∞ - Limit of speeding up uncommon case

SPEEDUP = 1/((1-0.05) + (0.05/ ∞)) = 1.052, or 5.2%

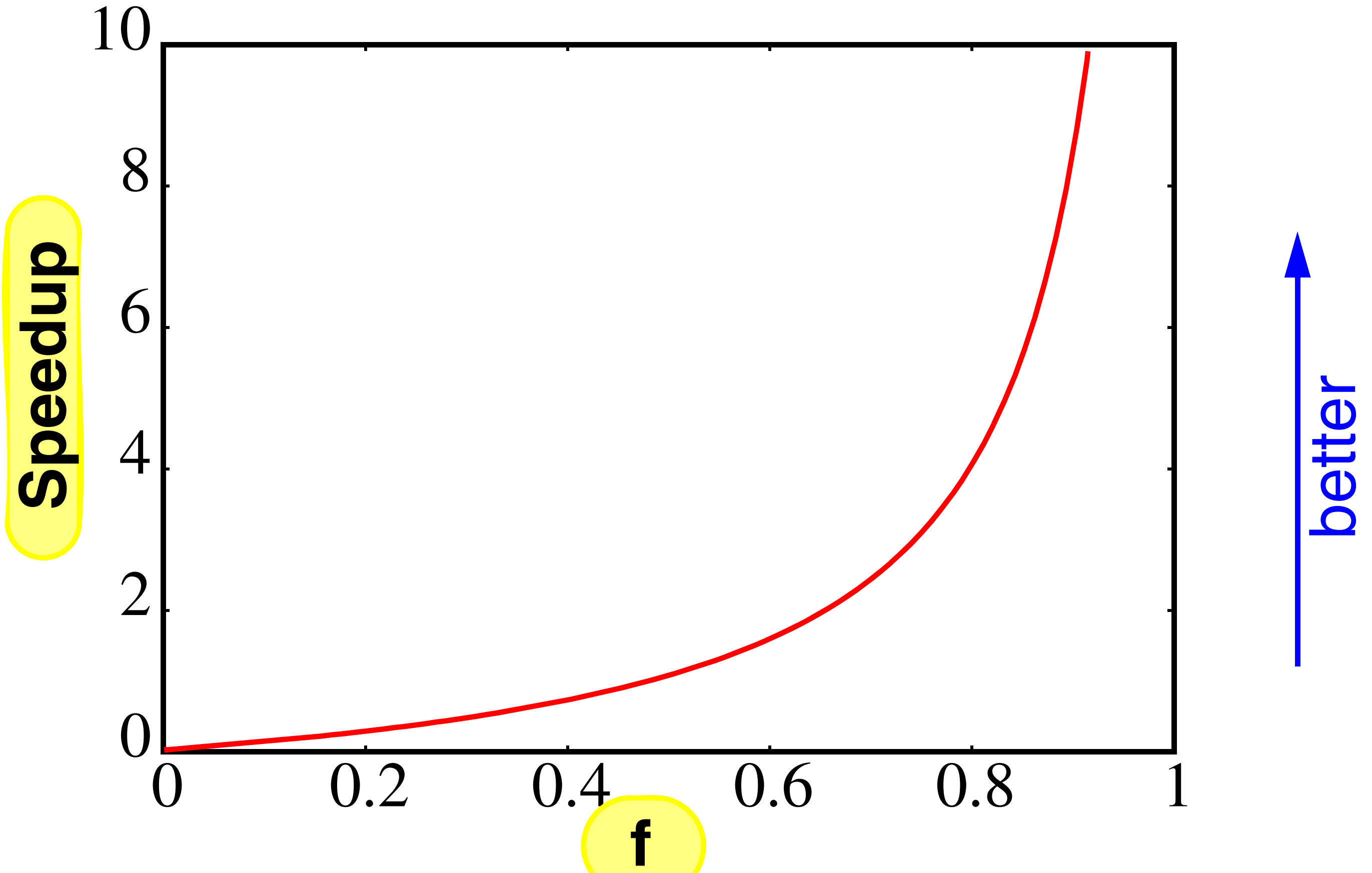
f = 95% and s $\rightarrow$ ∞ - Limit of speeding up common case

SPEEDUP = 1/((1-0.95) + (0.95/ ∞)) = 20, or 2000%

What should we go after? **Common** or **Uncommon** case?

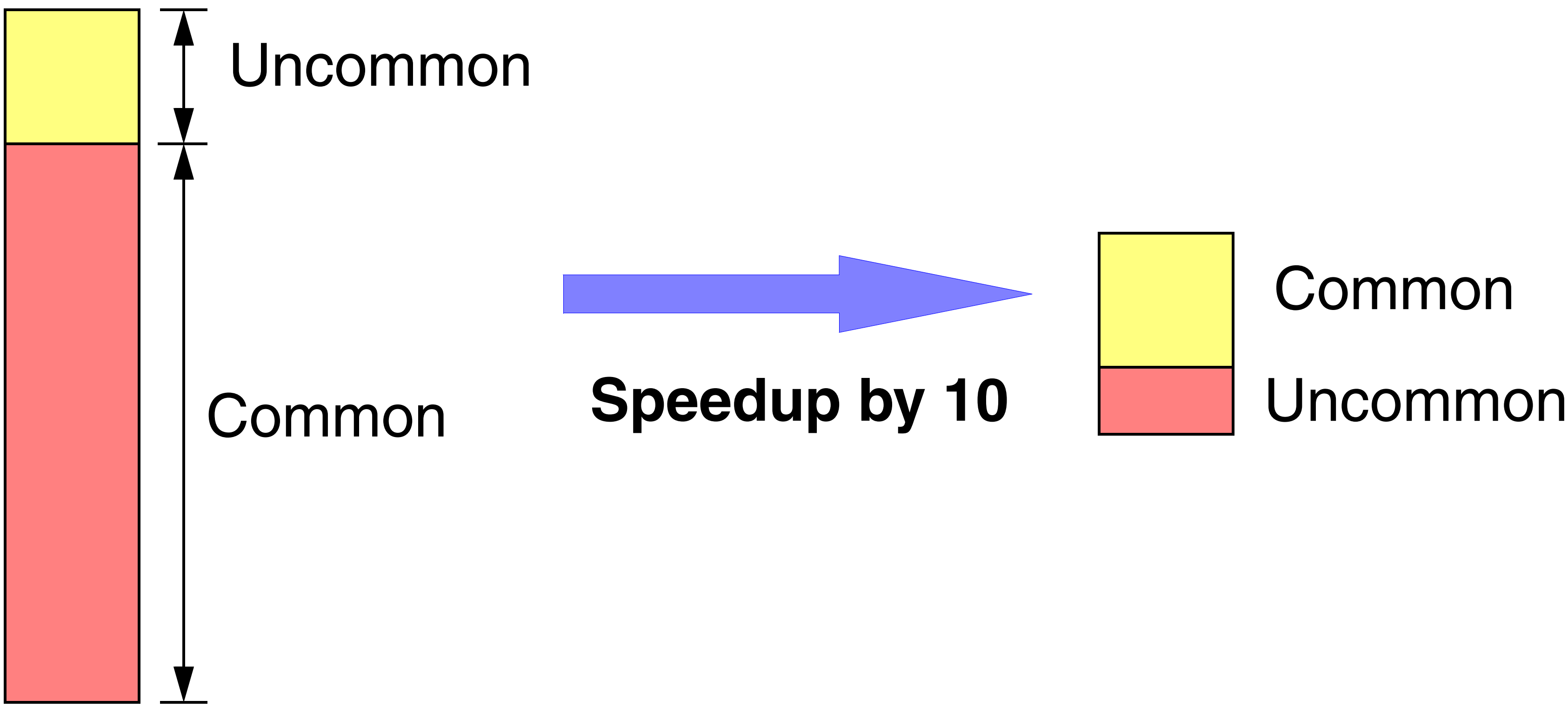
Amdahl's Law

$$\lim_{s \to \infty}\left(\frac{1}{1 - f + f/s}\right) = \frac{1}{1-f}$$ => **Make common case fast**



Amdahl's Law

Recall "COMMON" is relative and it MAY CHANGE once you optimize

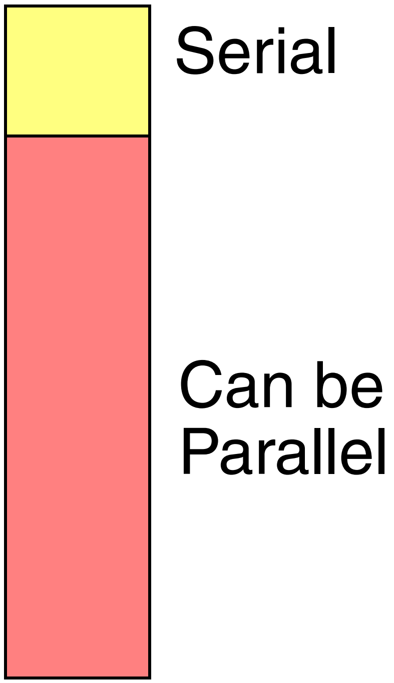


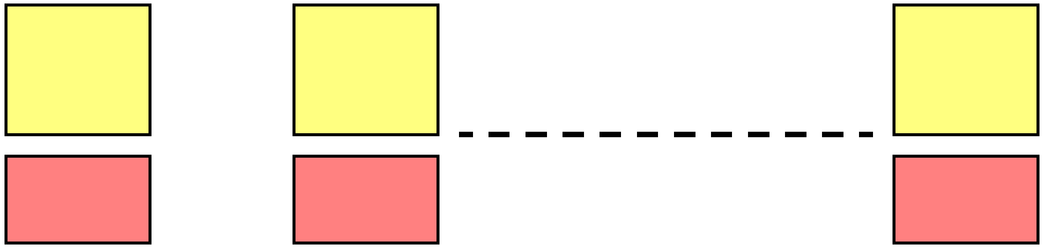
Example - Parallel Processing

Amdahl was talking about a parallel processor with large speedup.

At some point you have to pay attention to the serial part

Another example: Vector processing





Example Cont.

Assume f = 90%

S	Speedup
1	1.0
2	1.8
10	5.3
100	9.2
1000	9.9
10000	9.99

Instead of using the last 9000 processors we should have speedup the serial part

Amdahl's Law Example

Assume Fsqrt accounts for 20% of execution time in a GPU. FP account for 50% of overall execution time.

Option A: Improve Fsqrt performance by 10

Option B: Improve all FP ops by 1.6

Which is better?

$$Speedup_{Fsqrt}=(1 / [(1 - 0.2) + 0.2 / 10]) = 1.22$$

$$Speedup_{FPall}=(1 / [(1 - 0.5) + 0.5 / 1.6]) = 1.23$$

Improving all FP is better

Making the Common Case Fast

uniprocessor example: memory hierarchy

- keep recently referenced data/insts onchip (fast)
- exploit locality

Recall “must pay attention to technology”:

- on-chip faster than off-chip today
- SRAM faster than DRAM faster than disk

solution: memory hierarchy

Memory Hierarchy Specs

type	size	speed	bandwidth
reg	< 3k	500ps	64GB/s
L1	8k-64k	1ns	32GB/s
L2	128k-8M	18ns	48GB/s
main mem	4G	80ns	3.2GB/s
disk	120G	14ms	48MB/s- 23MB/s

Data for reg/L1 ignores multiporting in the register file and assumes single port for L1.

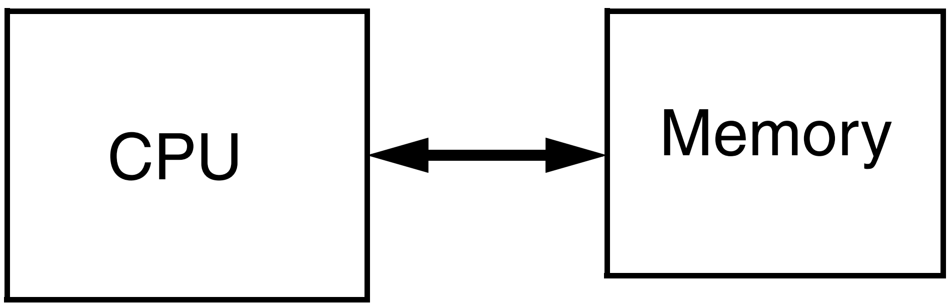
L1 may have 2 ports and a register file may have 12

Balance

At a system level, bandwidths and capacities should be balanced

Each level capable of demanding/supplying bandwidths

Refer to memory hierarchy figure



Memory Should be able to provide data in the rate req. by the CPU

CPU should be able to consume as much data as Memory can provide

Balance: Example

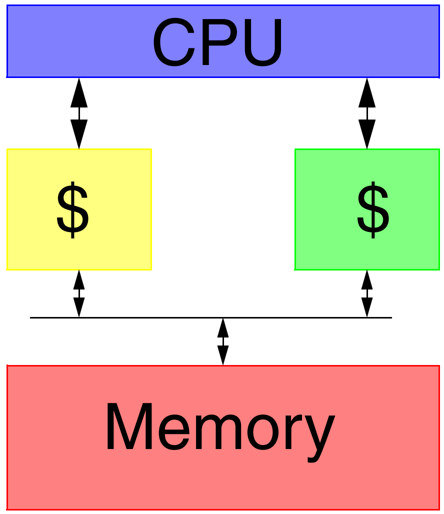
IPC = 1.5 (1/CPI)

30% loads and stores

90% data cache hit rate

95% icache hit rate

All cache misses require 32 bytes



So, processor memory demand is:

1.5 * 1.0 * 0.05 * 32 + 1.5 * 0.3 * 0.10 * 32 = 3.8 bytes/clock

To keep the processor busy memory needs to supply this bandwidth

Balance

Given a resource: If **demand bandwidth = supply bandwidth**

then the computation is that **resource-bound**

e.g., if memory bandwidth = processor demand for program P

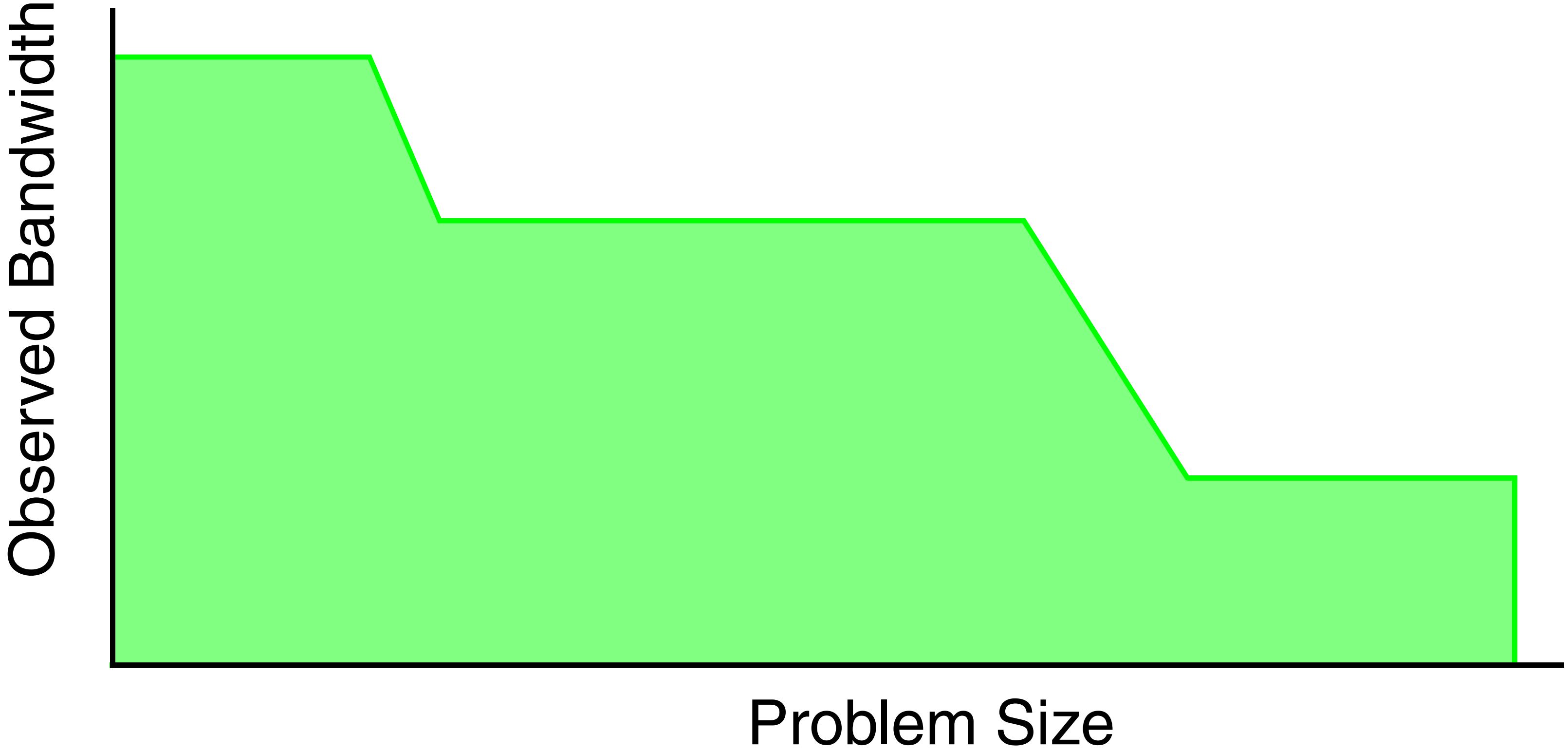
then P is said to be memory-bound

same for CPU-bound, disk-bound or I/O bound

GOAL: to be bound everywhere.

Memory Bandwidth

- copy: a[i] = b[i]   scale: a[i] = q*b[i]
- sum: a[i] = b[i] + c[i]   triad: a[i] = b[i] + q*c[i] (saxyp)



Memory Bandwidth (uniprocessor)

Memory bandwidth of real systems (MB/s)

System	copy	scale	sum	triad
Alpha ES45/1000	1946	1940	1978	1978
Cray T932	11341	10221	13014	13682
SUN UE 10k/400	364	215	287	296
Athlon 1333	941	592	727	685
PwrMac G4/867	629	615	609	680
PentiumIII/800	424	424	569	554
SparcClassic	57	48	48	43
AMD 386	7.4	5.7	7.7	6.4
Pentium4	1437	1431	1587	1575

(www.streambench.org)

Balance (again)

Storage capacity and bandwidth requirements

- e.g., large cache => higher hit rate => lower demand
- Or large memory => less paging => lower I/O demand

Amdahl's rule:

- 1 MIPS <=> 1 MB memory <=> 1 Mbits/s I/O
- if corrected to 1 Mbytes/s of I/O, the rule is still good!

Bursty Behavior

To get 2 IPC how many instructions should you -

- fetch per cycle?
- issue per cycle?
- complete per cycle?
- Is the answer 2?

instructions are not like sand where peaks and valleys are leveled

An Example

- A = B + C
- D = E + F

2-way issue:

0	load B	load C
1	load E	load F
2	add B, C	
3	store A	add E, F
4		store D

4-way issue:

0	load B	load C	load E	load F
1				
2	add B, C	add E, F		
3	store A	store B		

It takes a 4-way processor to get 2 IPC!

Design for higher PEAK rate to achieve a desired AVERAGE level of performance

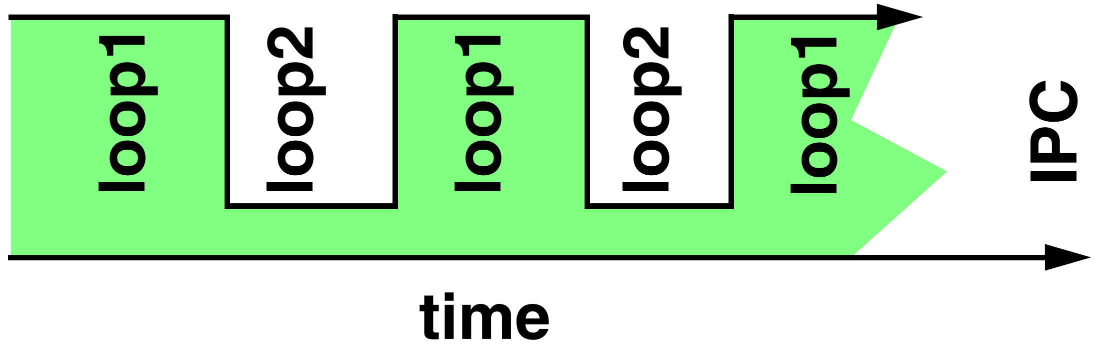
Bursty Behavior

loop1:

a[i] = a[i-1] + doo

loop 2:

a[i] = b[i] + c[i]



Dependences will cause pipeline stalls (or bubbles or wait times)

So sometimes pipeline will be full and at other only partially full

a higher PEAK level is need for a desired AVERAGE level performance