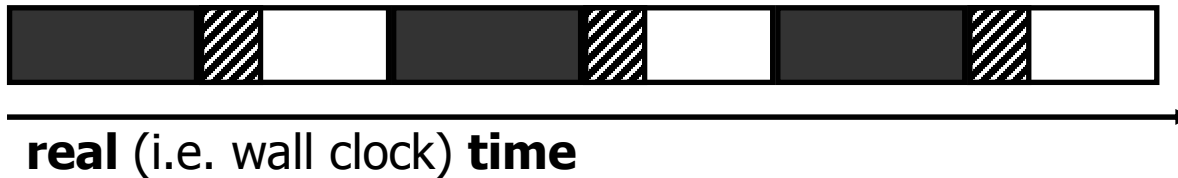


Measurement
&
Performance

Topics


- ◆ Timers
- ◆ Performance measures
- ◆ Time-based metrics
- ◆ Rate-based metrics
- ◆ Benchmarking
- ◆ Amdahl's law




The Nature of Time



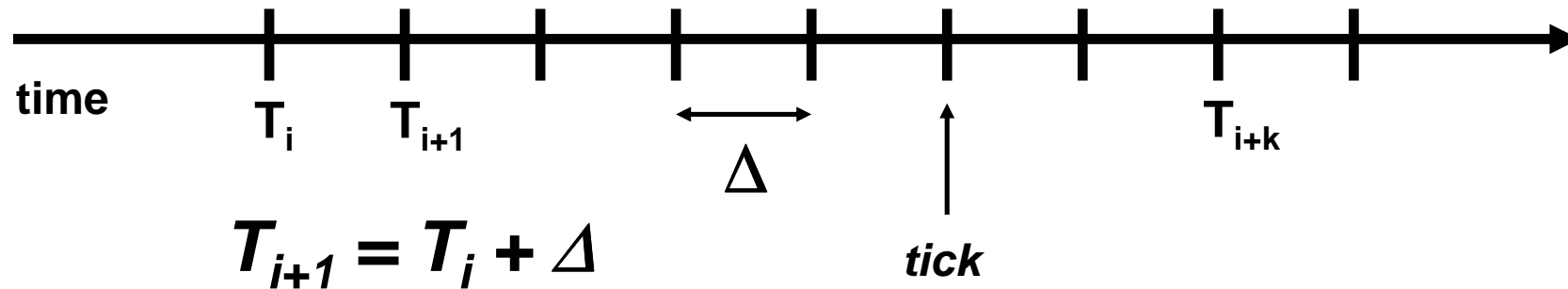
 = **User Time**: time spent executing instructions in the user process

 = **System Time**: time spent executing instructions in the *kernel* on behalf of the user process

 = **all other time** (either idle or else executing instructions unrelated to the user process)

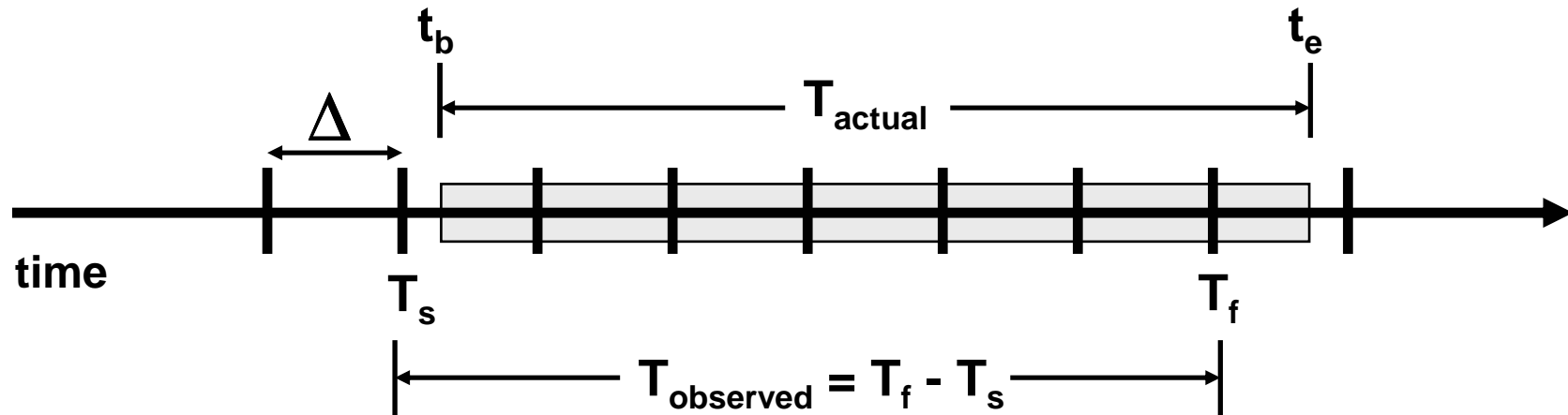
 +  +  = **real (wall clock) time**

Anatomy of a Timer



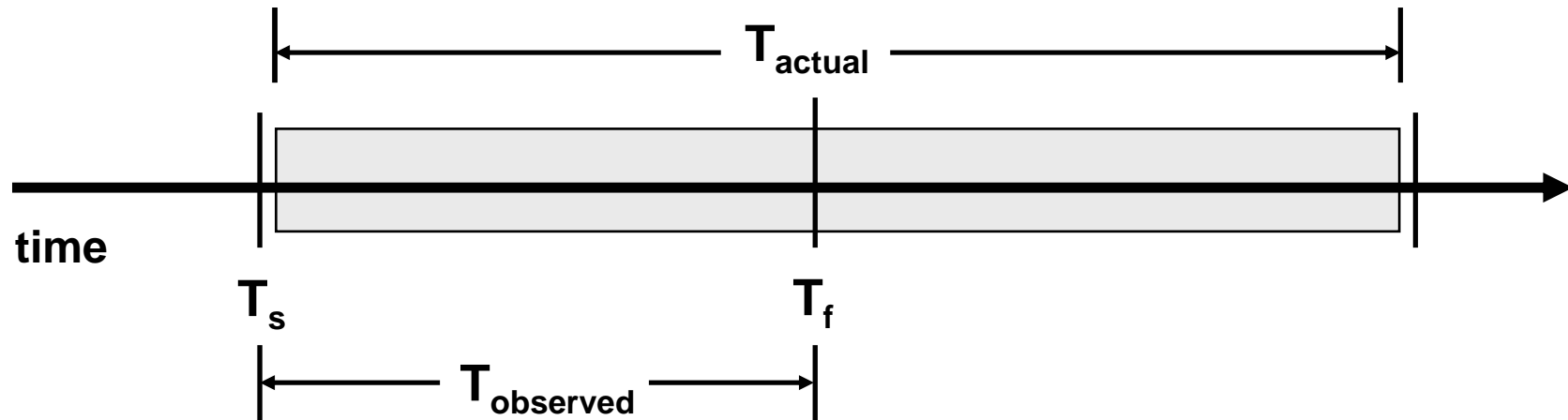
- ◆ A *counter value* (T) is updated upon discrete *ticks*
 - ◆ a tick occurs once every Δ time units
 - ◆ upon a tick, the counter value is incremented by Δ time units
- ◆ Some Terminology:
 - ◆ timer *period* = Δ seconds/tick
 - ◆ timer *resolution* = $1/\Delta$ ticks/second

Using Timers



- ◆ Estimating elapsed time:
 - ◆ based on discrete timer values before (T_s) and after (T_f) the event
- ◆ How close is T_{observed} to T_{actual} ?

Timer Error: Example #1



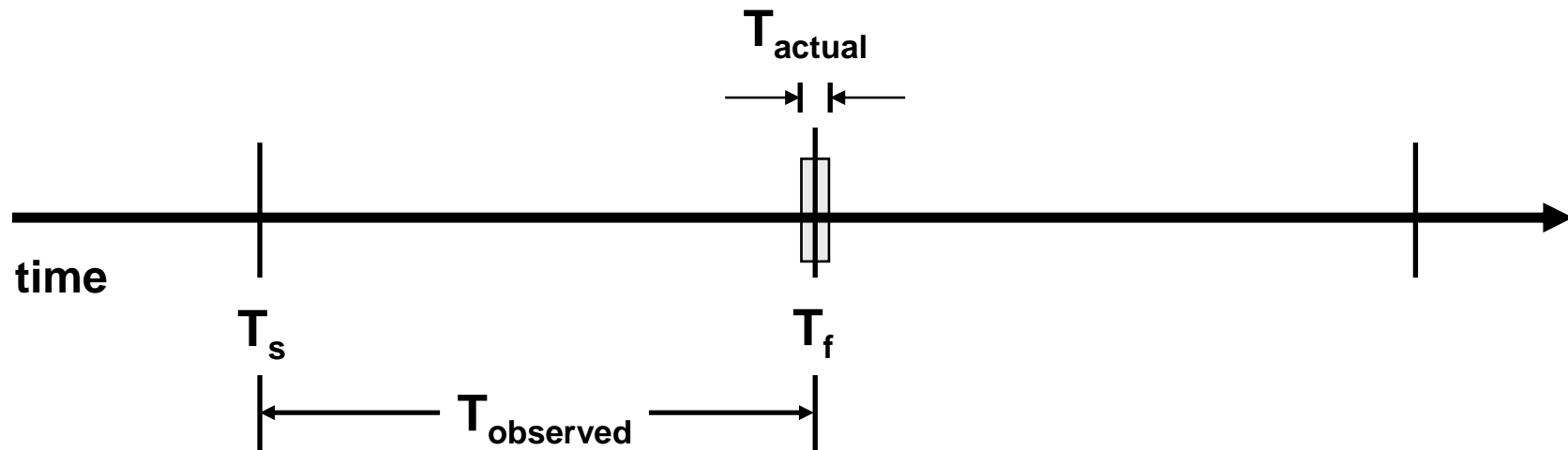
$$T_{\text{actual}}: \sim 2 \Delta$$

$$T_{\text{observed}}: \Delta$$

Absolute measurement error: $\sim \Delta$

Relative measurement error: $\sim \Delta / 2\Delta = \sim 50\%$

Timer Error: Example #2



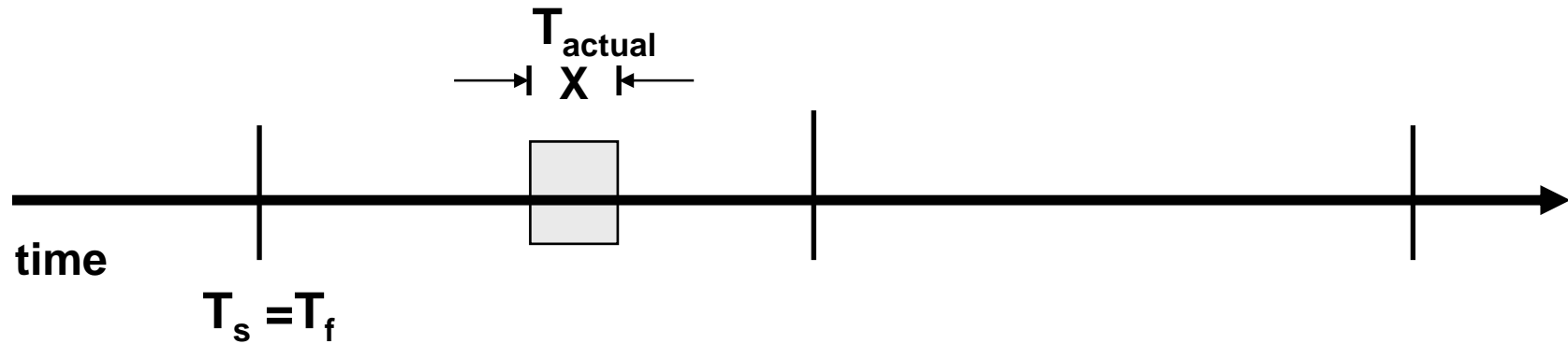
$T_{\text{actual}}: \epsilon (\sim \text{zero})$

$T_{\text{observed}}: \Delta$

Absolute measurement error: $\sim \Delta$

Relative measurement error: $\sim \Delta / \epsilon = \sim \text{infinite}$

Timer Error: Example #3

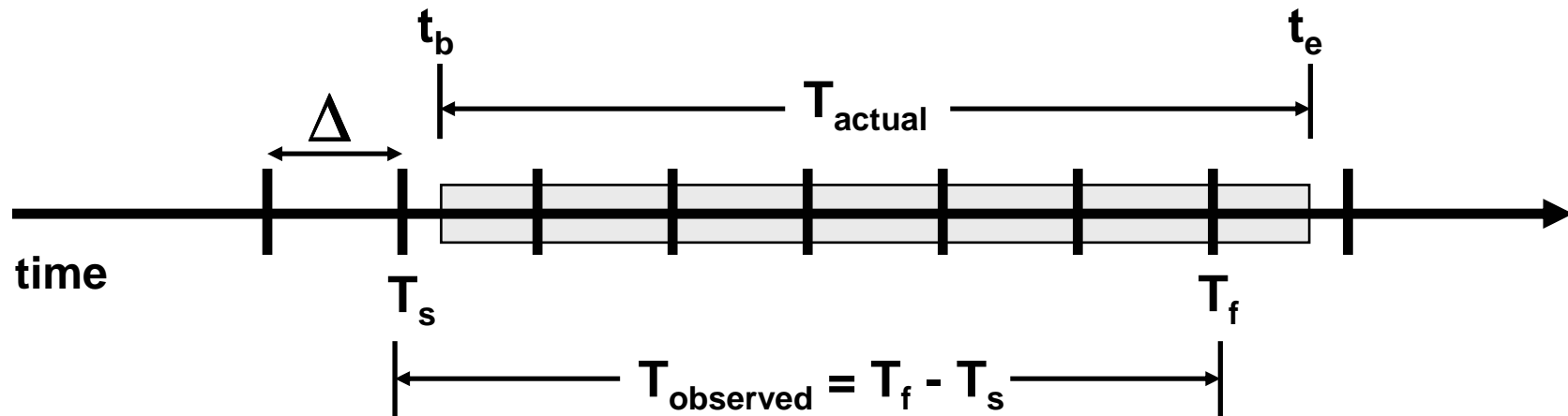


$T_{\text{actual}}: X$
 $T_{\text{observed}}: 0$

Absolute measurement error: X

Relative measurement error: $X / X = 100\%$

Timer Error: Summary



- ◆ Absolute measurement error: $\pm \Delta$
- ◆ Key point:
 - ◆ need a large number of ticks to reduce error
 - ◆ increase timer resolution ($= 1/\Delta$ ticks/second)

Performance

Performance expressed as a TIME

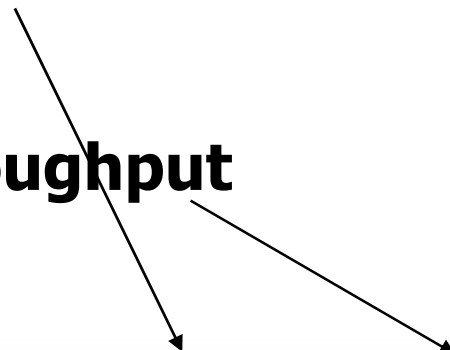
- ◆ **Absolute** time measures
 - ◆ Difference between start and finish of an operation
 - ◆ Examples:
 - *Running time (elapsed time, response time, latency, completion time, execution time)*
 - *Latency*
 - *CPU time*
 - ◆ Most straightforward performance measure
- ◆ **Relative** (normalized) time measures
 - ◆ Running time normalized to some reference time
 - (e.g. time/reference time)

Performance expressed as a RATE

- ◆ Rates are performance measures expressed in units of work per unit time.
 - ◆ Examples:
 - millions of instructions / s (MIPS)
 - millions of floating point instructions / s (MFLOPS)
 - MB/s = 2^{20} bytes / s
 - Mb/s = 10^6 bits / s
 - KB/s = 2^{10} bytes / s = 1024 bytes / s
 - Kb/s = 10^3 bits / s
 - frames / s (fps)
 - samples / s
 - transactions / s (TPS)

TIME vs. RATE

- ◆ Significance of the two classes of metrics depends on the context/application/system type
- ◆ Time-based metrics:
 - ◆ Closer to the concept of **performance**
- ◆ Rate-based metrics:
 - ◆ Closer to the concept of **throughput**
- ◆ Example:



<u>Airplane</u>	<u>Passengers</u>	<u>Range (mi)</u>	<u>Speed (mph)</u>	<u>pXmph</u>
Boeing 737-100	101	630	598	60,398
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79424

Time-based metrics

◆ Execution time:

- ◆ **Wall-clock time** elapsed from start to end of computation
- ◆ Includes:
 - CPU time
 - I/O time
- ◆ Ex: UNIX's `time` command:
 - `90.7u 12.9s 2:39 65%`
 - 90.7 user seconds
 - 12.9 system seconds
 - 2:39 wall clock time
 - 65% of the wall clock time was spent running on the CPU
- ◆ CPU time is closer to our notion of "performance"
 - ◆ Measures actual CPU performance

CPU performance

- ◆ Use clock cycles to compute CPU performance:

$$\boxed{\text{CPU}_{\text{Time}} = N_{\text{cycles}} * T_{\text{clock}}} \quad \text{or} \quad \boxed{\text{CPU}_{\text{Time}} = \frac{N_{\text{cycles}}}{f_{\text{Clock}}}}$$

- ◆ Introducing the number of executed instructions N_{inst}

$$\boxed{\text{CPU}_{\text{time}} = N_{\text{inst}} / N_{\text{inst}} * N_{\text{cycles}} * T_{\text{clock}} = N_{\text{inst}} * (N_{\text{cycles}} / N_{\text{inst}}) * T_{\text{clock}}}$$

- ◆ $(N_{\text{cycles}} / N_{\text{inst}}) =$ number of clock cycles per instruction = **CPI**

$$\boxed{\text{CPU}_{\text{time}} = N_{\text{inst}} * \text{CPI} * T_{\text{clock}}}$$

↓ ↓ ↓

Compiler+ISA ISA Technology(+ISA)

CPU performance (2)

- ◆ Previous formulation is too general!
 - ◆ CPI is not single
 - ◆ Different (class of) instructions will take different amounts of time
- ◆ Modified definition of CPI:

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}} = \frac{\sum_{i=1}^n (\text{CPI}_i * I_i)}{I_c} = \sum_{i=1}^n \left(\text{CPI}_i \frac{I_i}{I_c} \right)$$

- ◆ Where:
 - I_i = number of instructions of type i in the program
 - n = types or classes of instructions
 - I_c = total number of instructions in the program

CPU performance: example (1)

- ◆ A benchmark has 80 instructions:
 - ◆ 25 instructions are loads/stores (each takes 2 cycles)
 - ◆ 50 instructions are adds (each takes 1 cycle)
 - ◆ 5 instructions are square root (each takes 100 cycles)

$$\begin{aligned}\mathbf{CPI} &= ((25 * 2) + (50 * 1) + (5 * 100)) / 80 \\ &= (25/80 * 2) + (50/80 * 1) + (5/80 * 100) = \\ &= \mathbf{7.5}\end{aligned}$$

CPU performance: example

- ◆ Two machines:
 - ◆ Machine A: conditional branch is performed by a compare instruction followed by a branch instruction
 - ◆ Machine B performs conditional branch as one instruction.
 - ◆ On both machines, conditional branch takes **two clock cycles** and **the rest of the instructions take 1 clock cycle**.
 - A and B perform the same program
 - A: 20% of instructions are compares, 20% are jumps
 - B: 25% are conditional branches
 - ◆ Finally, clock cycle time of A is 25% faster than B's clock cycle time. Which machine is faster?

Solution

$$\text{CPU}_{\text{time}} = N_{\text{inst}} * \text{CPI} * T_{\text{clock}}$$

$$\text{CPI}_A = 1$$

$$\text{CPU}_A = N_A * \text{CPI} * t_A = N_A * 1 * t_A$$

$$t_B = 1.25 * t_A$$

$$\text{CPI}_B = 0.25 * 2 + 0.75 * 1 = 1.25$$

$$\begin{aligned} \text{CPU}_B &= N_B * \text{CPI} * t_B = N_B * 1.25 * t_B = 0.8 N_A * 1.25 * t_B \\ &= 0.8 N_A * 1.25 * 1.25 t_A = N_A * 1.25 * t_A \end{aligned}$$

A is faster!

CPU performance

- ◆ CPU time would be the “perfect metric”
 - ◆ Time is exactly what we need
- ◆ However, it can be difficult to compute
 - ◆ Lack of information (e.g., new ISA)
 - ◆ CPI_i cannot be a “static measure” (e.g., from a table)
 - *instruction cache*
 - *pipeline*
 - *I/O*
- ◆ *How about some “average” measure?*
 - ◆ For instance, a rate-base metric

Rate-based metrics

- ◆ Let's review some commonly used (in the past) rate-base metrics
 - ◆ MHz
 - ◆ MIPS
 - ◆ MFLOPS
- ◆ Always keep in mind:

$$\mathbf{CPU_{time} = N_{inst} * CPI * T_{clock}}$$

Using MHz

- ◆ MHz = millions of clock cycles/sec
- ◆ MHz does not predict running time:

- ◆ $\text{CPU}_{\text{time}} = \boxed{N_{\text{inst}} * \text{CPI}} * T_{\text{clock}}$ ←

- ◆ Example:

CPU	MHz	System	CPU time
Pentium Pro	180	Alder	6,440
POWER2	77	RS/6000 591	3,263

Using MIPS

- ◆ MIPS = millions of instructions / second
 - ◆ Very used in the late 80's / early 90's

$$\text{MIPS} = \frac{N_{\text{inst}}}{\text{CPU}_{\text{time}} * 10^6} = \frac{f_{\text{clock}}}{\text{CPI} * 10^6}$$

- ◆ Relation between CPU time and MIPS

$$\text{CPU}_{\text{time}} = \frac{N_{\text{inst}}}{\text{MIPS} * 10^6}$$

Using MIPS

- ◆ MIPS is not suitable to measure performance:
 - ◆ MIPS is dependent on the instruction set
 - Difficult to compare MIPS of computers with different instruction sets
 - ◆ MIPS is dependent on the test program
- ◆ MIPS is used to measure the complexity of an algorithm on a given platform
 - ◆ Not asymptotic but operative complexity
 - ◆ Multimedia compression algorithms can be classified according to MIPS

Using MIPS

- ◆ MIPS can vary inversely to performance

Using MIPS: Example

- ◆ Optimizing compiler can reduce 50% of ALU instructions only
- ◆ $f_{\text{clock}} = 50\text{MHz}$ ($T_{\text{clock}} = 20\text{ns}$)
- ◆ $\text{MIPS}_{\text{original}} = ?$
- ◆ $\text{MIPS}_{\text{optimized}} = ?$

Operation	Frequency	CPI
ALU ops	43%	1
Loads	21%	2
Stores	12%	2
Branches	24%	2

$$\text{CPI}_{\text{orig}} = 0.43*1+0.21*2+0.12*2+0.24*2 = 1.57$$

$$\text{MIPS}_{\text{orig}} = 50*10^6 / (1.57 * 10^6) = \mathbf{31.85}$$

$$\begin{aligned} \text{CPI}_{\text{opt}} &= (0.43/2*1+0.21*2+0.12*2+0.24*2) / (1-0.43/2) \\ &= 1.73 \end{aligned}$$

$$\text{MIPS}_{\text{opt}} = 50*10^6 / (1.73 * 10^6) = \mathbf{28.90}$$

Using MFLOPS

- ◆ MFLOPS = millions of floating operations /sec
 - ◆ Same as MIPS, but referred to a specific instruction type

$$\text{MFLOPS} = \frac{I_{c, \text{ floating point}}}{\text{CPU}_{\text{time}} * 10^6}$$

- ◆ Although focused on FP instructions only, same inconsistencies as MIPS

Benchmarking

Benchmarking

- ◆ **Goal:** Measure a set of programs (benchmarks) that represent the workload of real applications and that predict the running time of those applications
- ◆ **Steps in the benchmarking process:**
 - (1) Choose **representative** benchmark programs.
 - difficult to find realistic AND portable programs.
 - (2) Choose an **individual performance measure** (for each benchmark)
 - time, normalized time, rate?
 - (3) Choose an **aggregate performance measure** (for all benchmarks)
 - sum, normalized sum, mean, normalized mean?

Why Do Benchmarking?

- ◆ How we evaluate differences
 - ◆ Different systems and changes to single system
- ◆ Provide a target for system developers
 - ◆ Benchmarks should represent large class of important programs
 - ◆ Improving benchmark performance should help many programs
- ◆ Benchmarks shape a field:
 - ◆ Good ones accelerate progress
 - Good target for development
 - ◆ Bad benchmarks hurt progress
 - Inventions that help real programs don't help benchmark

Benchmark examples

- ◆ (Toy) Benchmarks
 - ◆ 10-100 line
 - ◆ e.g., :puzzle, quicksort, ...
- ◆ **Synthetic Benchmarks** [early 90's]
 - ◆ attempt to match average frequencies of real workloads
 - ◆ e.g., **Whetstone, Dhrystone**
- ◆ **Kernels**
 - ◆ Time critical excerpts of real programs
 - ◆ e.g., Livermore loops, fast Fourier transform
- ◆ **Real programs**
 - ◆ e.g., gcc, jpeg

Successful Benchmark Suite: SPEC

- ◆ 1987: processor industry mired in “bench marketing”:
 - ◆ “That is 8 MIPS machine, but they claim 10 MIPS!”
- ◆ 1988 : EE Times + 5 companies band together to perform *Systems Performance Evaluation Committee (SPEC)* in 1988
 - ◆ Sun, MIPS, HP, Apollo, DEC
- ◆ Create standard list of programs, inputs, reporting:
 - ◆ some real programs, includes OS calls, some I/O
- ◆ Currently SPEC is more than 40 computer companies:
 - ◆ Compaq, Cray, DEC, HP, Hitachi, IBM, Intel, Motorola, Netscape, SGI, Sun

www.specbench.org/osg/

SPEC Benchmarks

- ◆ New incarnations required every three years:
 - ◆ SPEC89, SPEC92, SPEC95, SPEC2000.
- ◆ Causes of benchmark obsolescence:
 - ◆ increasing processor speed
 - ◆ increasing cache sizes
 - ◆ increasing levels of caches
 - ◆ increasing application code size
 - ◆ library code dependences
 - ◆ aggressive benchmark engineering

SPEC2000 integer benchmarks

Benchmark	Ref Time (Sec)	Application Area	Specific Task
099.go	4600	Game playing; artificial intelligence	Plays the game Go against itself.
124.m88ksim	1900	Simulation	Simulates the Motorola 88100 processor running Dhrystone and a memory test program.
126.gcc	1700	Programming & compilation	Compiles pre-processed source into optimized SPARC assembly code.
129.compress	1800	Compression	Compresses large text files (about 16MB) using adaptive Lempel-Ziv coding.
130.li	1900	Language interpreter	Lisp interpreter.
132.jpeg	2400	Imaging	Performs jpeg image compression with various parameters.
134.perl	1900	Shell interpreter	Performs text and numeric manipulations (anagrams/prime number factoring).
147.vortex	2700	Database	Builds and manipulates three interrelated databases.

SPEC2000 floating point benchmarks

<u>Benchmark</u>	<u>Ref Time (Sec)</u>	<u>Application Area</u>	<u>Specific Task</u>
101.tomcatv	3700	Fluid Dynamics / Geometric Translation	Generation of a two-dimensional boundary-fitted coordinate system around general geometric domains.
102.swim	8600	Weather Prediction	Solves shallow water equations using finite difference approximations. (The only single precision benchmark in CFP95.)
103.su2cor	1400	Quantum Physics	Masses of elementary particles are computed in the Quark-Gluon theory.
104.hydro2d	2400	Astrophysics	Hydrodynamical Navier Stokes equations are used to compute galactic jets.
107.mgrid	2500	Electromagnetism	Calculation of a 3D potential field.
110.applu	2200	Fluid Dynamics/Math	Solves matrix system with pivoting.
125.turb3d	4100	Simulation	Simulates turbulence in a cubic area.
141.apsi	2100	Weather Predication	Calculates statistics on temperature and pollutants in a grid.
145.fpppp	9600	Chemistry	Performs multi-electron derivatives.
146.wave	3000	Electromagnetics	Solve's Maxwell's eqn on cartesian mesh.

Benchmark performance

Comparing performance

- ◆ Execution time of a benchmark set matches CPU time as close as possible
- ◆ But, how to measure it?
- ◆ Example:

	Machine A	Machine B
Program 1	2 sec	4 sec
Program 2	12 sec	8 sec

- ◆ How much faster is A than B?
- ◆ Attempt 1: **ratio of run times, normalized to A times**
 - program1:** 4/2 **program2 :** 8/12
 - A 2x faster on program 1, 2/3x faster on program 2
 - On **average**, A is **$(2 + 2/3) / 2 = 4/3$** times faster than B

Comparing performance (2)

◆ Example (cont.):

◆ Attempt 2: ratio of run times, normalized to B times

program 1: 2/4 **program 2 :** 12/8

- A 2x faster on program 1 and 2/3x faster on program 2
- On **average, $(1/2 + 3/2) / 2 = 1$**
- A is **1.0 times faster** than B

◆ Attempt 3: ratio of runtimes, total times, normalized to A

program 1: 2/4 **program2 :** 8/12

- Machine A took 14 seconds for both programs
- Machine B took 12 seconds for both programs
- A takes 14/12 of the time of B
- A is **6/7 faster** than B

Comparing performance (3)

- ◆ What is the right answer?
 - ◆ All calculations answer different questions...
- ◆ Not all “averages” correctly track execution time!
- ◆ Principle:
 - ◆ Conventional “average” is correct for absolute measures
 - **Arithmetic mean**
 - ◆ For rate-based measures:
 - **Harmonic mean**
 - ◆ For normalized measures:
 - **Geometric mean ?**

Means and Ratios

- ◆ Metrics that track CPU time
 - ◆ Total running time
 - ◆ Normalized total running time
 - ◆ Arithmetic average of running times

$$\frac{1}{n} \sum_{i=1}^n Time_i$$

- ◆ If not all benchmarks have equal importance:
weighted arithmetic mean

$$\sum_{i=1}^n Weight_i \times Time_i$$

Means and Ratios (2)

◆ Example:

	Machine A [s]	Machine B [s]	Machine C [s]
Prog 1	20	10	40
Prog 2	40	80	20
Total running time	60	90	60
Normalized total running time w.r.t. A	1	1.5	1
Normalized total running time w.r.t. B	0.66	1	0.66
Arithmetic Mean	30	45	30
Sum of normalized times w.r.t. A	2.0	2.5	2.5
Sum of normalized times w.r.t. B	2.5	2.0	4.25

$$A=C > B$$

Means and Ratios (Cont.)

- ◆ The harmonic mean (HM) is a measure for rates (and ratios in general) that predicts running time

- ◆ If *Rate* is the generic metric we want to average over n programs

$$\frac{n}{\sum_{i=1}^n \frac{1}{Rate_i}}$$

- ◆ If not all benchmarks have equal importance: weighted harmonic mean

$$\frac{n}{\sum_{i=1}^n \frac{Weight_i}{Rate_i}}$$

Means and Ratios (2)

◆ Example:

	Machine A [s]	Machine B [s]	Machine C [s]
Prog 1	20	10	40
Prog 2	40	80	20
Total running time	60	90	60
Normalized total running time w.r.t. A	1	1.5	1
Normalized total running time w.r.t. B	0.66	1	0.66
Arithmetic Mean	30	45	30
Harmonic mean*	1.33	0.88	1.33

* = Using MIPS (prog1= 40Minstr, prog2 = 40Minstr)

- $MIPS_{A,prog1} = 2$ $MIPS_{A,prog2} = 1$
- $MIPS_{B,prog1} = 4$ $MIPS_{B,prog2} = 0.5$
- $MIPS_{C,prog1} = 1$ $MIPS_{C,prog2} = 2$
- $HM_A = 2/(1/2+1/1) = 2/(3/2) = 1.33$
- $HM_B = 2/(1/4+1/0.5) = 2/(9/4) = 0.88$
- $HM_C = 2/(1/1+1/2) = 2(3/2) = 1.33$

Means and Ratios (3)

- ◆ How to average normalized values?

- ◆ **Use geometric mean**

$$GM = (\prod_{i=1,..n} NormTime_i)^{1/n}$$

- ◆ Property of geometric mean

$$GM(Xi)/GM(Yi) = GM (Xi/Yi)$$

- ◆ GM is consistent over different references

- ◆ **But is consistently wrong... !!**

- ◆ It does not track running time!

Means and Ratios (4)

◆ Example:

	Machine A (/A, /B)	Machine B (/A, /B)	Machine C (/A, /B)
Prog 1	(1,2)	(0.5,1)	(2,4)
Prog 2	(1,0.5)	(2,1)	(0.5, 0.25)
Total running time	60	90	60
Total normalized time w.r.t. A	2	2.5	2.5
Total normalized time w.r.t. B	2.5	2	4.25
Arithmetic Mean (of times norm. w.r.t. A)	1	1.25	1.25
Arithmetic Mean (of times norm. w.r.t. B)	1.25	1	2.125
Geometric Mean (of times norm. w.r.t. A)	1	1	1
Geometric Mean (of times norm. w.r.t. B)	1	1	1

Not dependent on the reference!

SPEC CPU performance measures

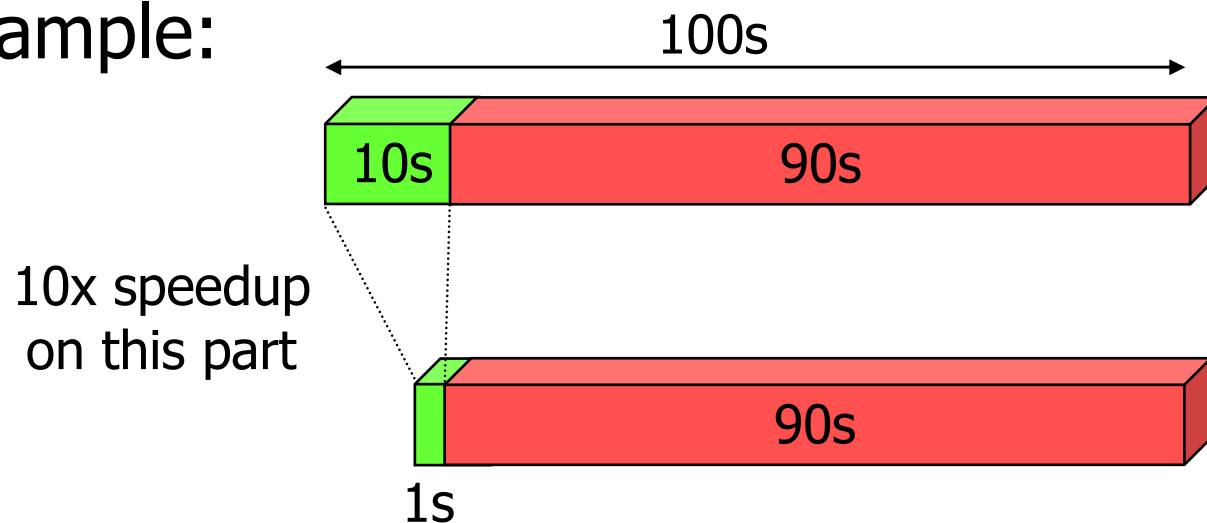
- ◆ SPECfp index = $(NT_1 \times NT_2 \times \dots \times NT_n)^{1/n}$
 - ◆ This is a **geometric mean**
 - ◆ Each NT_k is a normalized time:
 - = (ref. time for benchmark k)/(measured time for benchmark k)
 - reference times are measured on a SparcStation 10/40 (40 MHz Supersparc with no L2 cache)
- ◆ Problem: SPEC performance measures don't predict execution time!!!

system	total time	SPECfp95
166 MHz Pentium Pro	6470	5.47
180 MHz Pentium Pro	6440	5.40

Amdahl's law

- ◆ Speeding up a small fraction of the execution time of a program or a computation, the **WHOLE** computation will not be speed up by the same amount

- ◆ Example:



Total time = 100s (initial) 91s (after speedup)
Total speedup = $9/100 = 9\%$

Amdahl's law (cont.)

◆ Defining speedup:

Old program (not enhanced)



Old time: $T = T_1 + T_2$

New program (enhanced)



New time: $T' = T_1' + T_2'$

T_1 = time that can NOT be enhanced.

T_2 = time that can be enhanced.

T_2' = time after the enhancement.

Speedup: $S_{\text{overall}} = T / T'$

Amdahl's law (cont)

◆ Two key parameters:

- ◆ $F_{\text{enhanced}} = T_2 / T$ (fraction of original time that can be improved)
- ◆ $S_{\text{enhanced}} = T_2 / T_2'$ (speedup of enhanced part)

$$\begin{aligned} T' &= T_1' + T_2' = T_1 + T_2' = T(1 - F_{\text{enhanced}}) + T_2' \\ &= T(1 - F_{\text{enhanced}}) + (T_2 / S_{\text{enhanced}}) && \text{[by def of } S_{\text{enhanced}} \text{]} \\ &= T(1 - F_{\text{enhanced}}) + T(F_{\text{enhanced}} / S_{\text{enhanced}}) && \text{[by def. of } F_{\text{enhanced}} \text{]} \\ &= T((1 - F_{\text{enhanced}}) + F_{\text{enhanced}} / S_{\text{enhanced}}) \end{aligned}$$

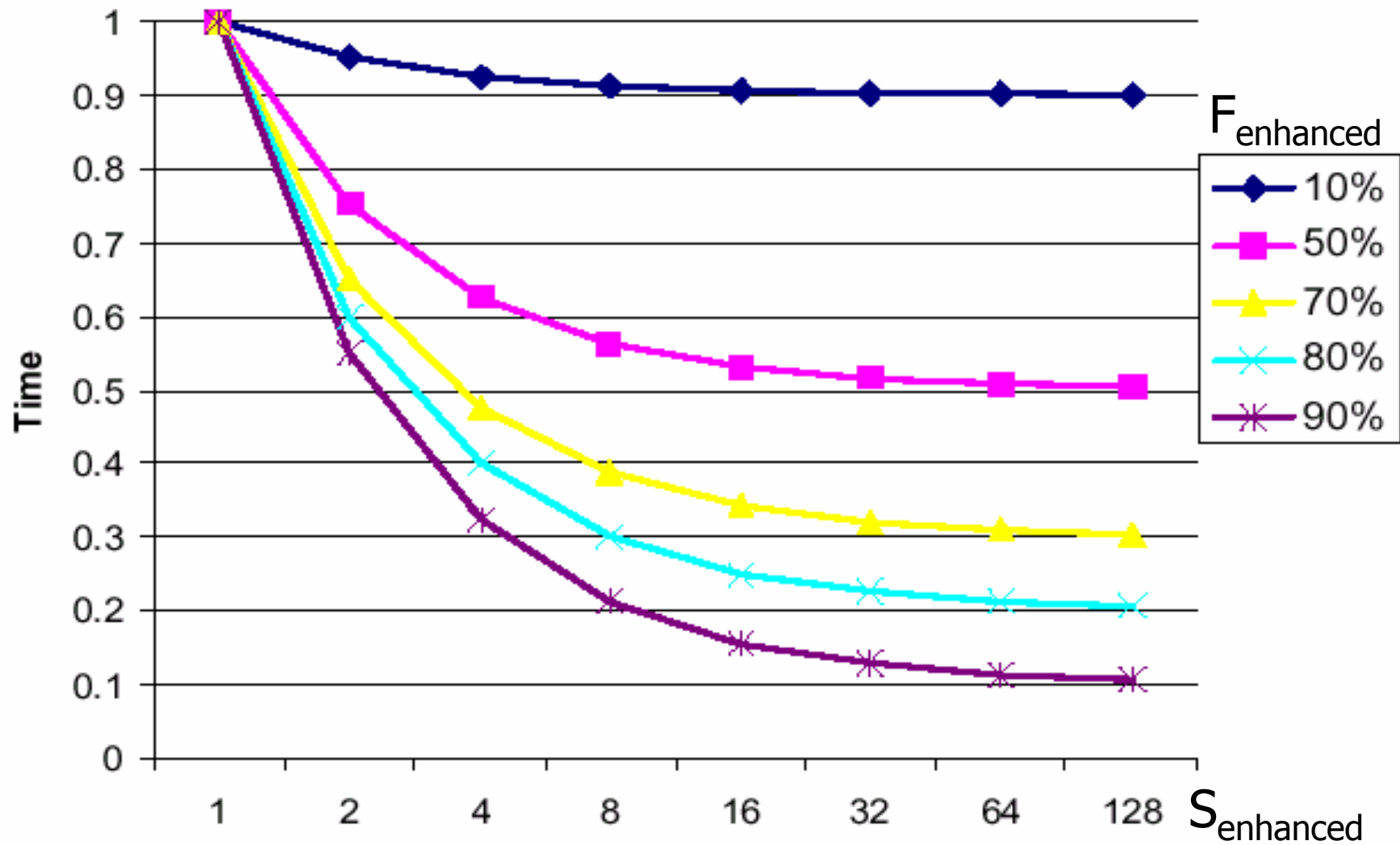
• Amdahl's Law:

$$S_{\text{overall}} = T / T' = 1 / ((1 - F_{\text{enhanced}}) + F_{\text{enhanced}} / S_{\text{enhanced}})$$

◆ Key idea:

- ◆ Amdahl's law quantifies the general notion of diminishing returns. It applies to any activity, not just computer programs.

Amdhal law (cont.)



Amdahl's law: example

- ◆ Program runs for 100 seconds on a uniprocessor
- ◆ 50% of the program can be parallelized on a multiprocessor
- ◆ Assume a multiprocessor with 5 processors (5x faster)

$$\text{Speedup} = \frac{1}{\frac{0.5}{5} + (1 - 0.5)} = 1/0.6 \approx \mathbf{1.7}$$

- Bottomline:
 - It is hard to speed up a program
 - It is easy to make premature optimizations.

Conclusions

- ◆ **Performance is important** to measure
 - ◆ For architects comparing different deep mechanisms
 - ◆ For developers of software trying to optimize code, applications
 - ◆ For users, trying to decide which machine to use, or to buy
- ◆ **Performance metrics are subtle**
 - ◆ Easy to mess up the "*machine A is XXX times faster than machine B*" numerical performance comparison
 - ◆ You need to know exactly what you are measuring: time, rate, throughput, CPI, cycles, ...
 - ◆ You need to know how combining these to give aggregate numbers
- ◆ **No metric is perfect**, so lots of emphasis on standard **benchmarks** today