Measurement & Performance

Todd C. Mowry

CS 347

Jan 15, 1998

Topics:

- **Timers**
- **Performance measures**
- **Relating performance measures**
  - system perfomance measures
  - latency and throughput
  - Amdahl's law

The Nature of Time

real (i.e. wall clock) **time**

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

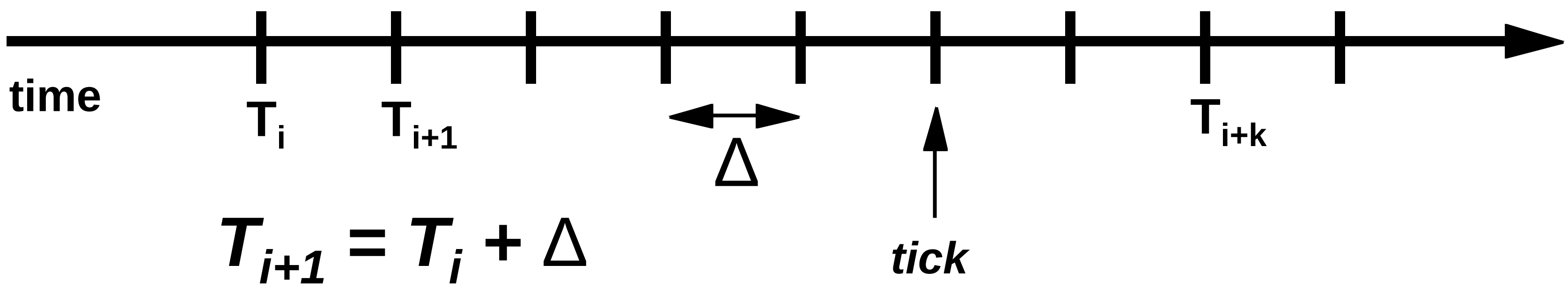
 = **System Time:** time spent executing instructions in the ***kernel*** on behalf of the user process

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

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

Unless otherwise specified, "time" often refers to "user time".

Anatomy of a Timer



$$T_{i+1} = T_i + \Delta$$

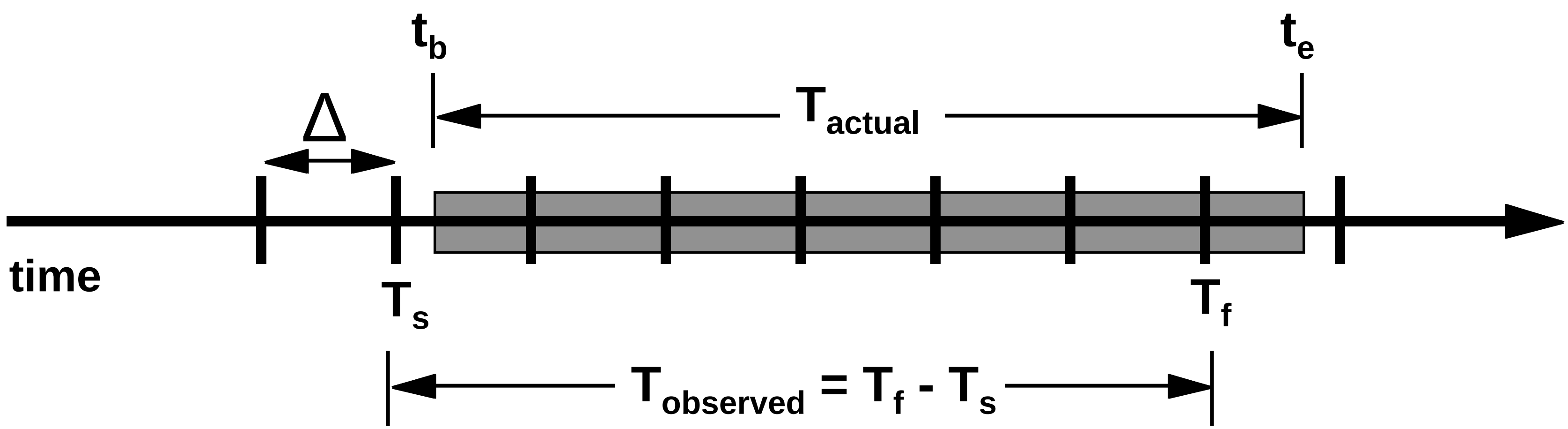
A counter value (T) is updated upon discrete *ticks*

- 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

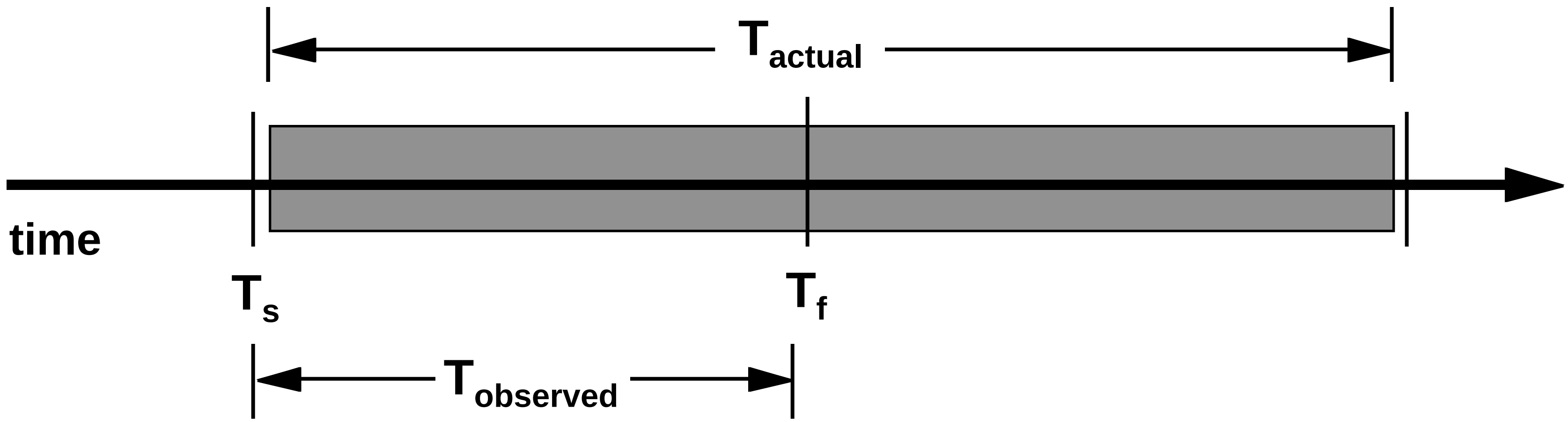


Estimating elapsed time:

- 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



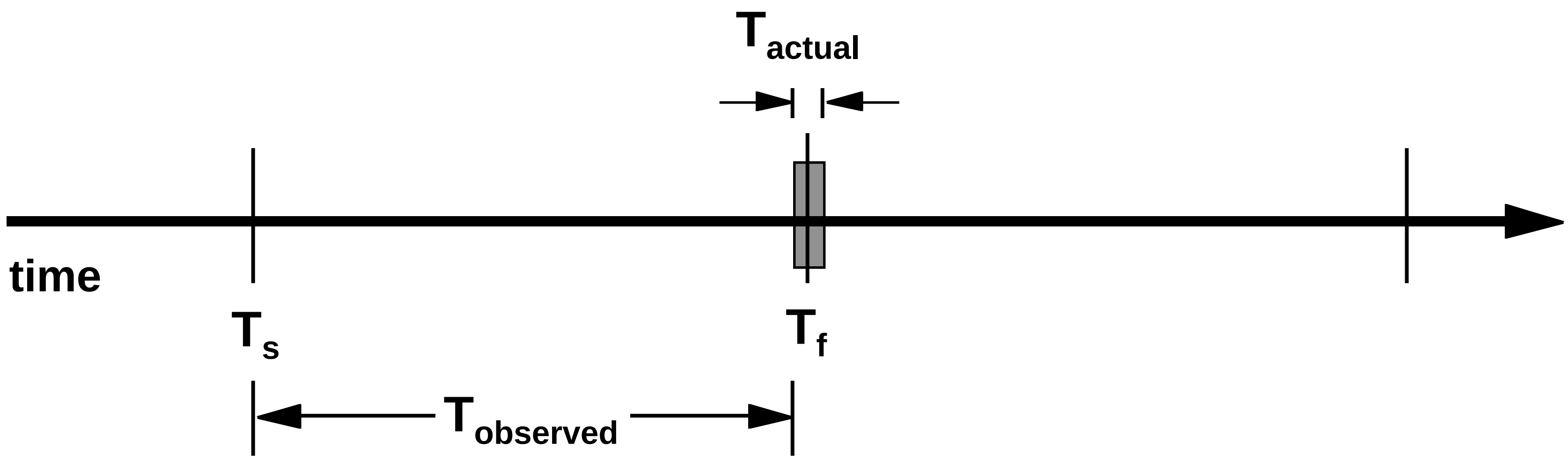
T_{actual}: ~ 2 Δ

$T_{observed}$: Δ

Absolute measurement error: ~ Δ

Relative measurement error: ~ Δ/ 2Δ = ~ 50%

Timer Error: Example #2



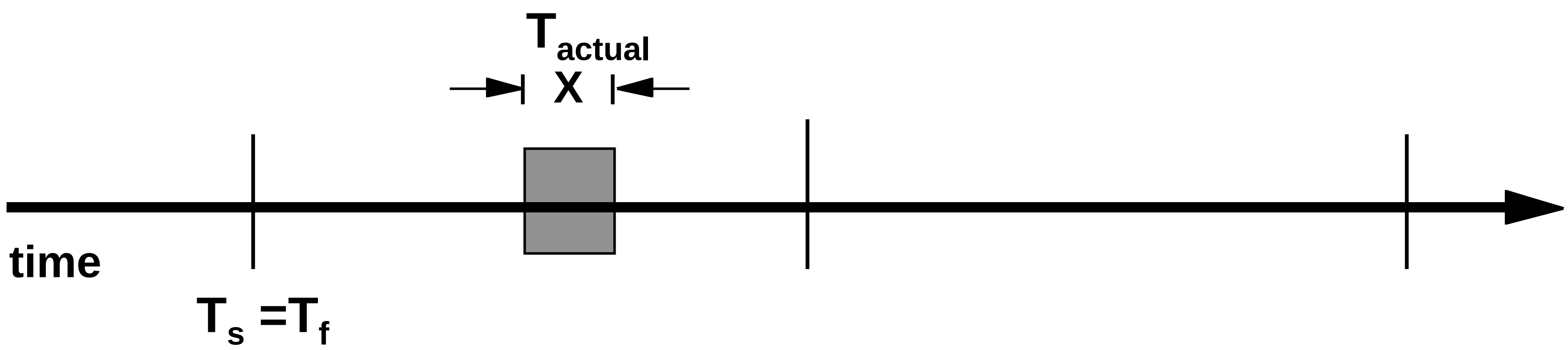
T_{actual}: ε (~ zero)

$T_{observed}$: Δ

Absolute measurement error: ~ Δ

Relative measurement error: ~ Δ / ε = ~ infinite

Timer Error: Example #3



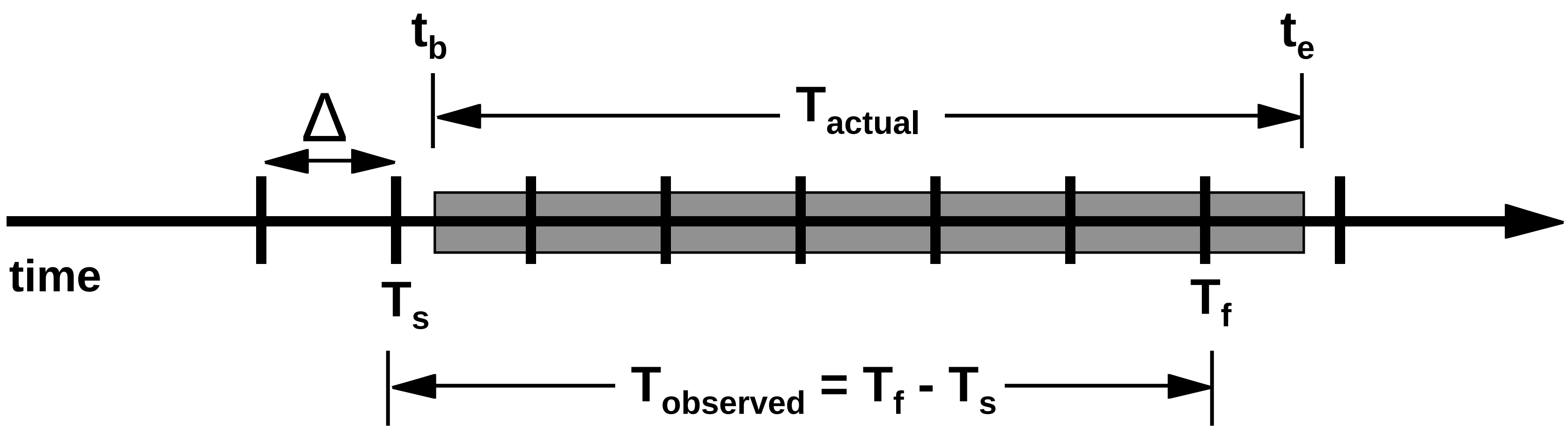
T_{actual}: X

$T_{observed}$: 0

Absolute measurement error: X

Relative measurement error: X / X = 100%

Timer Error: Summary



Absolute measurement error: +/- Δ

Key point: need a large number of ticks to hide error

- **can compute $T_{threshold}$ as a function of Δ and *E***
- **$T_{threshold}$ = minimum observed time to guarantee relative error bound**
- ***E* = maximum acceptable relative measurement error**

Homework 1 Timer Package

Unix interval countdown timer

- *decrements* timer value by Δ every Δ seconds
- setitimer(): initialize timer value
- getitimer(): sample timer value
- measures user time

“etime” package:

- based on Unix interval timers
- set_etime(): initializes timer
- get_etime(): returns elapsed time in seconds since last call to set_etime()

Performance expressed as a time

Absolute time measures

- difference between start and finish of an operation
- synonyms: running time, elapsed time, response time, latency, completion time, execution time
- most straightforward performance measure

Relative (normalized) time measures

- running time normalized to some reference time
- (e.g. time/reference time)

Guiding principle: Choose performance measures that track running time.

Performance expressed as a rate

Rates are performance measures expressed in units of work per unit time.

Examples:

- **millions of instructions / sec (MIPS)**
- **millions of floating point instructions / sec (MFLOPS)**
- **millions of bytes / sec (MBytes/sec)**
- **millions of bits / sec (Mbits/sec)**
- **images / sec**
- **samples / sec**
- **transactions / sec (TPS)**

Performance expressed as a rate(cont)

Key idea: Report rates that track execution time.

Example: Suppose we are measuring a program that convolves a stream of images from a video camera.

Bad performance measure: MFLOPS

- **number of floating point operations depends on the particular convolution algorithm: n^2 matix-vector product vs nlogn fast Fourier transform. An FFT with a bad MFLOPS rate may run faster than a matrix-vector product with a good MFLOPS rate.**

Good performance measure: images/sec

- **a program that runs faster will convolve more images per second.**

Performance expressed as a rate(cont)

Fallacy: Peak rates track running time.

Example: the i860 is advertised as having a peak rate of 80 MFLOPS (40 MHz with 2 flops per cycle).

However, the measured performance of some compiled linear algebra kernels (icc -O2) tells a different story:

Kernel	1d fft	sasum	saxpy	sdot	sgemm	sgemv	spvma
MFLOPS	8.5	3.2	6.1	10.3	6.2	15.0	8.1
%peak	11%	4%	7%	13%	8%	19%	10%

Relating time to system measures

Suppose that for some program we have:

- T seconds running time (the ultimate performance measure)
- C clock ticks, I instructions, P seconds/tick (performance measures of interest to the system designer)

$$T \text{ secs} = C \text{ ticks} \times P \text{ secs/tick}$$

$$= (I \text{ inst}/I \text{ inst}) \times C \text{ ticks} \times P \text{ secs/tick}$$

$$T \text{ secs} = I \text{ inst} \times (C \text{ ticks}/I \text{ inst}) \times P \text{ secs/tick}$$

running time

instruction count

avg clock ticks per instruction (CPI)

clock period

Pipeline latency and throughput

Latency (L): time to process an individual image.

Throughput (R): images processed per unit time

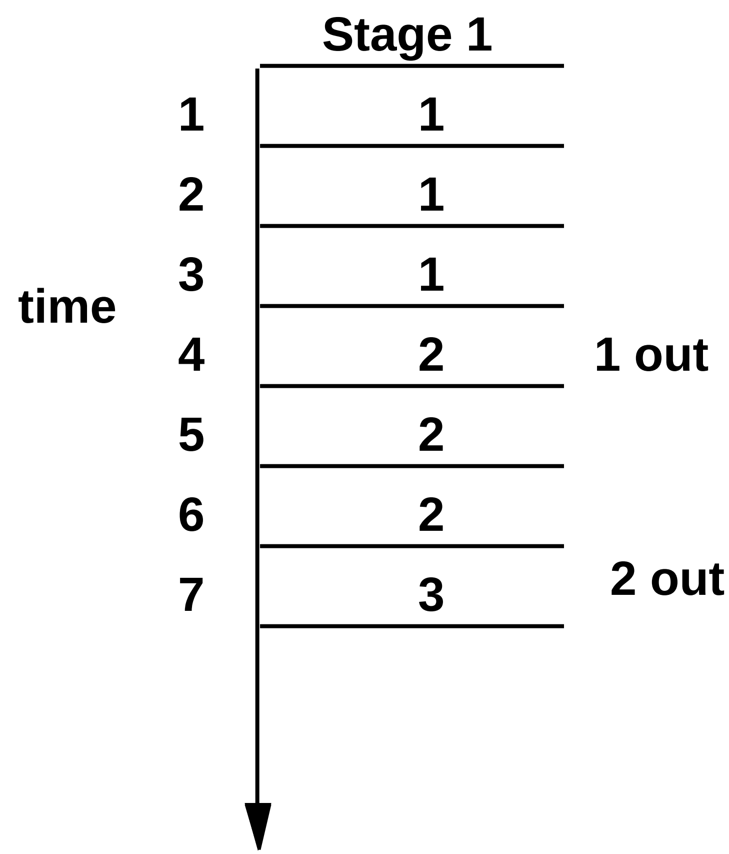
One image can be processed by the system at any point in time

Video system performance

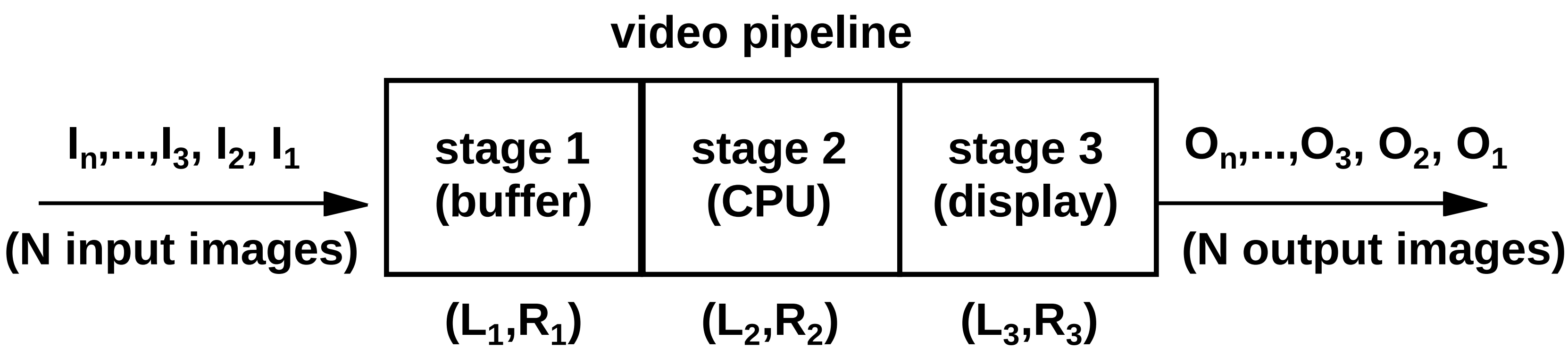
L = 3 secs/image.

R = 1/L = 1/3 images/sec.

T = L + (N-1)1/R
   = 3N



Pipelining the video system



One image can be in each stage at any point in time.

L_i = latency of stage i
R_i = throughput of stage i

$L = L_1 + L_2 + L_3$
$R = \min(R_1, R_2, R_3)$

Pipelined video system performance

Suppose:

$L_1 = L_2 = L_3 = 1$

Then:

L = 3 secs/image.

R = 1 image/sec.

$T = L + (N-1)1/R$
$= N + 2$

time

	Stage 1	Stage 2	Stage 3	
1	1			
2	2	1		
3	3	2	1	
4	4	3	2	1 out
5	5	4	3	2 out
6	6	5	4	3 out
7	7	6	5	4 out

Relating time to latency and thruput

In general:

- **T = L + (N-1)/R**

The impact of latency and throughput on running time depends on N:

- **(N = 1) => (T = L)**
- **(N >> 1) => (T = N-1/R)**

To maximize throughput, we should try to maximize the minimum throughput over all stages (i.e., we strive for all stages to have equal throughput).

Amdahl's law

You plan to visit a friend in Normandy France and must decide whether it is worth it to take the Concorde SST ($3,100) or a 747 ($1,021) from NY to Paris, assuming it will take 4 hours Pgh to NY and 4 hours Paris to Normandy.

	time NY->Paris	total trip time	speedup over 747
747	8.5 hours	16.5 hours	1
SST	3.75 hours	11.75 hours	1.4

Taking the SST (which is 2.2 times faster) speeds up the overall trip by only a factor of 1.4!

Amdahl's law (cont)

Old program (unenhanced)

T_1 | T_2

Old time: $T = T_1 + T_2$

New program (enhanced)

$T_1' = T_1$ | $T_2' <= T_2$

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: $S_{overall} = T / T'$

Amdahl's law (cont)

Two key parameters:

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

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

Amdahl's Law:

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

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

Amdahl's law (cont)

Trip example: Suppose that for the New York to Paris leg, we now consider the possibility of taking a rocket ship (15 minutes) or a handy rip in the fabric of space-time (0 minutes):

	time NY->Paris	total trip time	speedup over 747
747	8.5 hours	16.5 hours	1
SST	3.75 hours	11.75 hours	1.4
rocket	0.25 hours	8.25 hours	2.0
rip	0.0 hours	8 hours	2.1

Amdahl's law (cont)

Useful corollary to Amdahl's law:

- $1 \le S_{overall} \le 1 / (1 - F_{enhanced})$

$F_{enhanced}$	Max $S_{overall}$	$F_{enhanced}$	Max $S_{overall}$
0.0	1	0.9375	16
0.5	2	0.96875	32
0.75	4	0.984375	64
0.875	8	0.9921875	128

Moral: It is hard to speed up a program.

Moral++ : It is easy to make premature optimizations.