

Lecture 5: Performance Evaluation

CSC 469H1F

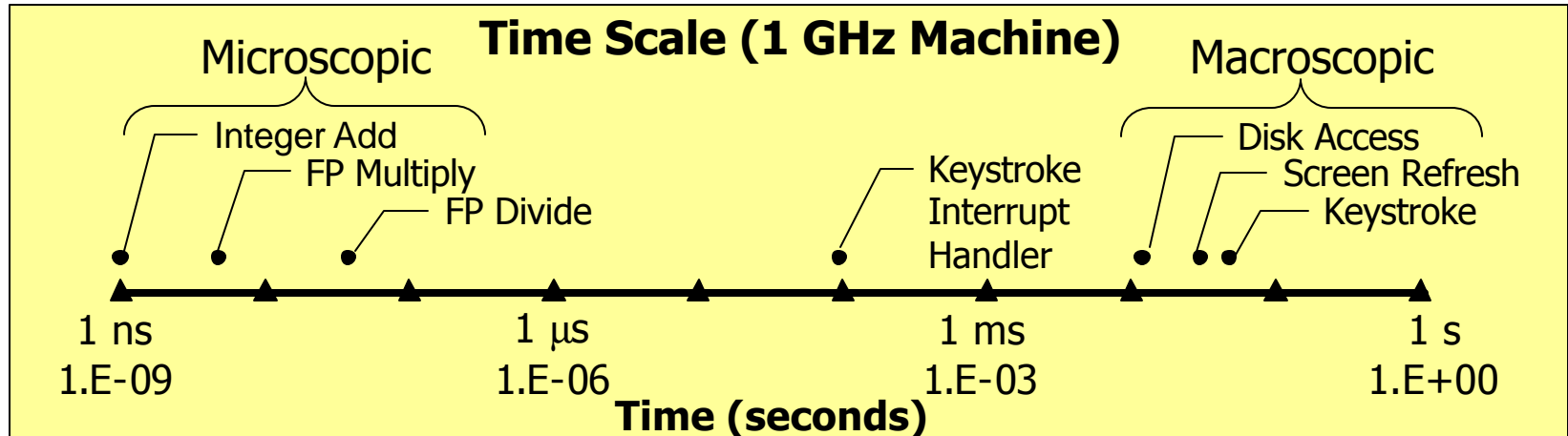
Fall 2006

Angela Demke Brown

Topics

- Time scales
- Interval counting
- Cycle counting
- K-best measurement scheme
- Amdahl's Law

Computer Time Scales



- Two fundamental time scales:
 - Processor: ~1 nanosecond (10^{-9} secs)
 - External events: ~10 milliseconds (10^{-2} secs)
 - Keyboard input, disk seek, screen refresh
- Implication
 - Can execute many instructions while waiting for external event
 - Basis for multiprogramming

Measurement

- What does it mean to ask “How much time does program X require?”
 - CPU time
 - How many total seconds are used when executing X ?
 - Measure used for most applications
 - Small dependence on other system activities
 - Actual (“Wall clock”) time
 - How many seconds elapsed between start and completion of X?
 - Depends on system load, I/O times, etc.
- How does time get measured?
- How does sharing impact measurement and performance?

"Time" on a Computer System



real (wall clock) time



= **user time** (time executing instructing instructions in the user process)



= **system time** (time executing instructing instructions in kernel on behalf of user process)



= **some other user's time** (time executing instructing instructions in different user's process)



+



+



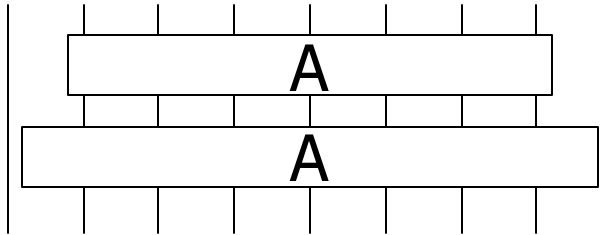
= **real (wall clock) time**

We will use the word "time" to refer to user time.

Interval Counting

- OS measures runtimes using interval timer
 - Maintain 2 counts per process
 - User time and system time
 - On each timer interrupt, increment counter for currently-executing process
 - User time if running in user mode
 - System time if running in kernel mode
 - Reported by unix "time" command (or getrusage in C program)

Accuracy of Interval Counting



- Interval timer reports 70 ms
- Min Actual = $60 + \epsilon$
- Max Actual = $80 - \epsilon$

- Worst case
 - Timer interval δ
 - Single measurement can be off by $\pm \delta$
 - No bound on error for multiple measurements
- Average case
 - Over/under estimates tend to balance out
 - Provided total run time is large enough (~ 100 timer intervals, or 1 second)

Cycle Counters

- Most modern systems have built in registers that are incremented every clock cycle
 - Very fine grained
 - Maintained as part of process state
 - Possible to save & restore with context switches
 - In Linux, counts elapsed global time
 - Special assembly code instruction to access
- On (recent model) Intel machines:
 - 64 bit counter.
 - RDTSC instruction sets `%edx` to high order 32-bits, `%eax` to low order 32-bits

Cycle Counter Period

- Wrap-around times for 550 MHz machine
 - Low order 32-bits wrap around every $2^{32} / (550 * 10^6) = 7.8$ seconds
 - High order 64-bits wrap around every $2^{64} / (550 * 10^6) = 33539534679$ seconds
 - 1065.3 years
- For 2 GHz machine
 - Low order 32-bits wrap every 2.1 seconds
 - High order 64-bits wrap every 293 years

Measuring with Cycle Counter

- Idea:
 - Get current value of cycle counter
 - Store as pair of unsigned's "cyc_hi" and "cyc_lo"
 - Compute something
 - Get new value of cycle counter
 - Perform double precision subtraction to get elapsed cycles

```
/* Keep track of most recent reading of counter */
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;
void start_counter()
{
    /* Get current value of cycle counter */
    access_counter(&cyc_hi, &cyc_lo);
}
```

Accessing the Cycle Counter

- *GCC* allows inline assembly code with mechanism for matching registers with program variables
- Code only works on x86 machine compiling with *GCC*

```
void access_counter(unsigned *hi, unsigned *lo)
{
    /* Get cycle counter */
    asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
        : "=r" (*hi), "=r" (*lo) /* output list */
        : /* No input */
        : "%edx", "%eax"); /* Clobbers list */
}
```

- Emit assembly with `rdtsc` and two `movl` instructions
- Code generates two outputs:
 - **Symbolic register** `%0` should be used for `*hi`
 - **Symbolic register** `%1` should be used for `*lo`
- Have to tell *GCC* about registers modified by assembly code
 - Old value in registers `%eax`, `%ebx` are "clobbered" by `rdtsc` instruction

Completing Measurement

- Get new value of cycle counter
- Perform double precision subtraction to get elapsed cycles
- Express as double to avoid overflow problems

```
/* global cyc_hi, cyc_lo store most recent reading */
double get_counter()
{
    unsigned ncyc_hi, ncyc_lo;
    unsigned hi, lo, borrow;
    /* Get current value of cycle counter */
    access_counter(&ncyc_hi, &ncyc_lo);
    /* do double precision subtraction */
    lo = ncyc_lo - cyc_lo;
    borrow = lo > ncyc_lo;
    hi = ncyc_hi - cyc_hi - borrow;
    return (double) hi * (1 << 30) * 4 + lo;
}
```

Timing with Cycle Counter

- Need to convert cycles into time
 - Determine clock rate of processor
 - Count number of cycles required for some fixed number of seconds
 - Simple version:

```
double MHZ;  
int sleep_time = 10;  
start_counter();  
sleep(sleep_time);  
MHZ = get_counter() / (sleep_time * 1e6);
```

- This is a bit too simple though
 - Assumes sleep() actually sleeps for 10 seconds
 - May be less (if interrupted) or more (especially if heavy load)
 - See "mhz: Anatomy of a micro-benchmark", Staelin & McVoy, Usenix Technical Conference 1998

Time of Day Clock

- return elapsed time since some reference time (e.g., Jan 1, 1970)
- example: Unix `gettimeofday()` command
- coarse grained (e.g., $\sim 3\mu\text{sec}$ resolution on Linux, 10 msec resolution on Windows NT)
 - Lots of overhead making call to OS
 - Different underlying implementations give different resolutions

```
#include <sys/time.h>
#include <unistd.h>

struct timeval tstart, tfinish;
double tsecs;
gettimeofday(&tstart, NULL);
P();
gettimeofday(&tfinish, NULL);
tsecs = (tfinish.tv_sec - tstart.tv_sec) +
        1e6 * (tfinish.tv_usec - tstart.tv_usec);
```

Measurement Pitfalls

- **Overhead**
 - Calling `get_counter()` incurs small amount of overhead
 - Want to measure long enough code sequence to compensate
- **Unexpected Cache Effects**
 - artificial hits or misses
 - e.g., these measurements were taken with the Alpha cycle counter:

```
foo1(array1, array2, array3);          /* 68,829 cycles */
```

```
foo2(array1, array2, array3);          /* 23,337 cycles */
```

vs.

```
foo2(array1, array2, array3);          /* 70,513 cycles */
```

```
foo1(array1, array2, array3);          /* 23,203 cycles */
```

Dealing with Overhead & Cache Effects

- Execute P() once to warm up cache
- Keep doubling number of times execute P() until reach some threshold
 - Used CMIN = 50000

```
int count = 1;
double cmeas = 0;
double cycles;
do {
    int c = count;
    P();                /* Warm up cache */
    get_counter();
    while (c-- > 0)
        P();
    cmeas = get_counter();
    cycles = cmeas / count;
    count += count;
} while (cmeas < CMIN); /* Make sure have enough */
return cycles / (1e6 * MHZ);
```

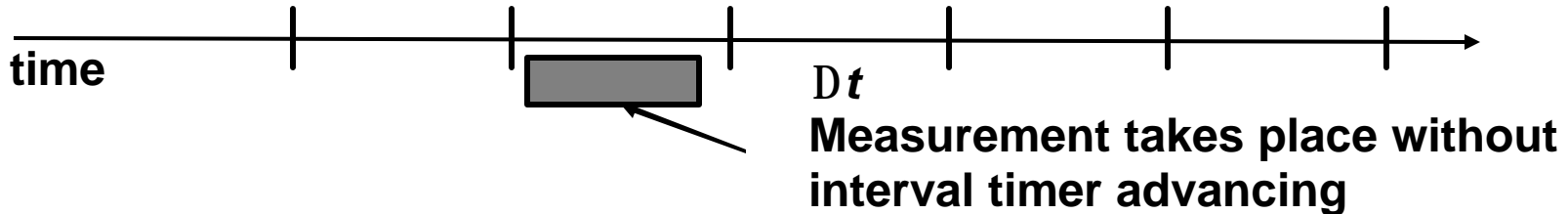
Context Switching

- Context switches can also affect cache performance
 - e.g., (foo1, foo2) cycles on an unloaded timing server:
 - 71,002, 23,617
 - 67,968, 23,384
 - 68,840, 23,365
 - 68,571, 23,492
 - 69,911, 23,692
- Why Do Context Switches Matter?
 - Cycle counter only accumulates when running user process
 - Some amount of overhead
 - Caches polluted by OS and other user's code & data
 - Cold misses as restart process
- Measurement Strategy
 - Try to measure uninterrupted code execution

Detecting Context Switches

- Clock Interrupts

- Processor clock causes interrupt every Dt seconds
 - Typically $Dt = 10$ ms
 - Same as interval timer resolution



- Can detect by seeing if interval timer has advanced during measurement

```
start = get_etime();

/* Perform Measurement */
. . .
if (get_etime() - start > 0)
    /* Discard measurement */
```

Detecting Context Switches (Cont.)

- External Interrupts
 - E.g., due to completion of disk operation
 - Occur at unpredictable times but generally take a long time to service
- Detecting
 - See if real time clock has advanced
 - Using coarse-grained interval timer

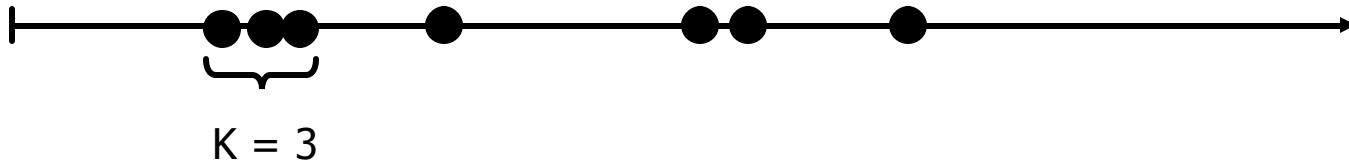
```
start = get_rtime();

/* Perform Measurement */
. . .
if (get_rtime() - start > 0)
    /* Discard measurement */
```

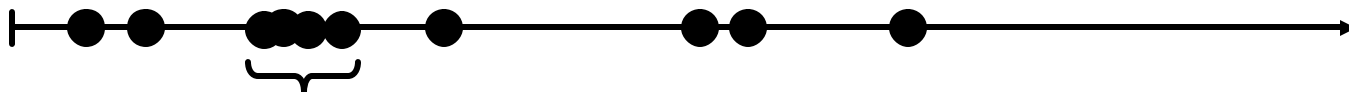
- Reliability
 - Good, but not 100%
 - Can't get clean measurements on heavily loaded system

Improving Accuracy

- K-Best Measurements
 - Assume that bad measurements always overestimate time
 - True if main problem is due to context switches or interference effects
 - Take multiple samples (e.g., $N = 20$) until lowest K are within some small tolerance of each other
 - Choose fastest measurement from the K-Best



- In some cases, errors can both under and overestimate time (e.g., when using interval timers)
 - Look for cluster of samples within some tolerance of each other



Measurement Summary

- It's difficult to get accurate times
 - compensating for overhead
 - but can't always measure short procedures in loops
 - global state
 - mallocs
 - changes cache behavior
- It's difficult to get repeatable times
 - cache effects due to ordering and context switches
- Moral of the story:
 - Adopt a healthy skepticism about measurements!
 - Always subject measurements to sanity checks.

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 Pittsburgh to NY (and 4 hours NY back to Pgh)

	Time NY→Paris→NY	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!

Speedup

Old program (unenhanced)



Old time: $T = T_1 + T_2$

T_1 = time that can NOT be enhanced.

T_2 = time that can be enhanced.

New program (enhanced)



New time: $T^c = T_1^c + T_2^c$

T_2^c = time after the enhancement.

Speedup: $S_{\text{overall}} = T / T^c$

Computing Speedup

Two key parameters:

$$F_{\text{enhanced}} = T_2 / T \quad (\text{fraction of original time that can be improved})$$

$$S_{\text{enhanced}} = T_2 / T_2^c \quad (\text{speedup of enhanced part})$$

$$\begin{aligned} T^c &= T_1^c + T_2^c = T_1 + T_2^c = T(1 - F_{\text{enhanced}}) + T_2^c \\ &= T(1 - F_{\text{enhanced}}) + (T_2 / S_{\text{enhanced}}) \\ &= T(1 - F_{\text{enhanced}}) + T(F_{\text{enhanced}} / S_{\text{enhanced}}) \\ &= T((1 - F_{\text{enhanced}}) + F_{\text{enhanced}} / S_{\text{enhanced}}) \end{aligned}$$

[by def of S_{enhanced}]

[by def of F_{enhanced}]

Amdahl's Law:

$$S_{\text{overall}} = T / T^c = 1 / ((1 - F_{\text{enhanced}}) + F_{\text{enhanced}} / S_{\text{enhanced}})$$

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

Trip example revisited

- Suppose you have the option of taking a rocket from NY to Paris (15 minutes), or a wormhole opens between NY and Paris (0 minutes):

	Time NY→Paris→NY	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
Wormhole	0 hours	8 hours	2.1

Lessons from Amdahl's Law

- Useful Corollary of Amdahl's law:
 - $1 \leq S_{\text{overall}} \leq 1 / (1 - F_{\text{enhanced}})$

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.

Other Maxims

- Second Corollary of Amdahl's law:
 - When you identify and eliminate one bottleneck in a system, something else will become the bottleneck
- Beware of Optimizing on Small Benchmarks
 - Easy to cut corners that lead to asymptotic inefficiencies