

Lecture 4: Performance Evaluation

CSC 469H1F
Fall 2007
Angela Demke Brown

Topics

Today:

- Time scales
- Interval counting
- Cycle counting

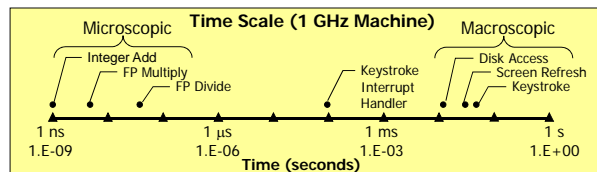
Monday:

- K-best measurement scheme
- Amdahl's Law

CSC469



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

CSC469



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?


CSC469





"Time" on a Computer System






real (wall clock) time

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

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

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

 +  +  = real (wall clock) time

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

CSC469



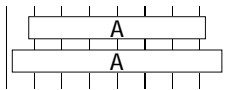
Interval Counting

- 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)

CSC469



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)

CSC469



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

CSC469



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

CSC469



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
 - subtract to get elapsed cycles
- Needs inline assembly to get access to cycle counter register
 - Details in tutorial tomorrow

CSC469



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 older Linux, 10 msec resolution on Windows NT, same as cycle counter on new Linux)
 - 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);
```

CSC469



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 */
```

CSC469



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);
```

CSC469



Context Switching

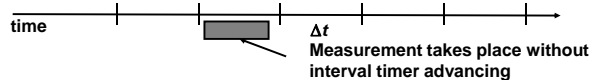
- Context switches can also affect cache performance
 - e.g., (E001, E002) cycles on an unloaded timing server:
 - 71,002, 23,617
 - 67,968, 23,384
 - 68,840, 23,365
 - 68,571, 23,492
 - 69,911, 23,692
- Why Do Context Switches Matter?
 - Cycle counter only accumulates when running user process
 - Some 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

CSC469



Detecting Context Switches

- Clock Interrupts
 - Processor clock causes interrupt every Δt seconds
 - Typically $\Delta t = 10$ 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 */
```

CSC469



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 */
```

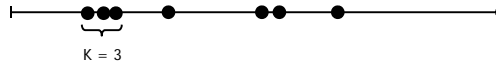
- Reliability
 - Good, but not 100%
 - Can't get clean measurements on heavily loaded system

CSC469

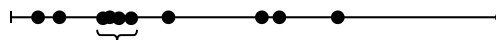


Improving Accuracy

- 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



CSC469



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.

CSC469



Advice

- Understand the phenomena being measured
 - Is variance caused by experimental noise or is there intrinsic variance?
- Decide if you want the minimum or the median
- Avoid common pitfalls
 - Measure the whole operation (e.g. file read vs. mmap)
 - Measure the operation you intend to measure
- Combine micro and macro benchmarks

CSC469



Amdahl's Law

- A friend is planning to visit you from Montreal, and you are driving to Algonquin Park for a week of camping. Your friend must choose between Via Rail (\$114, 9 hours, return) and WestJet (\$267, 2.5 hours, return). The drive to Algonquin park will take 3.5 hours each way.

	Time MTL→TO→MTL	Total trip time	Speedup over VIA
VIA	9 hours	16 hours	1
WestJet	2.5 hours	9.5 hours	1.7

- Taking the plane (which is 3.6 times faster) speeds up the overall trip by only a factor of 1.7!

CSC469



Speedup

Old program (unenhanced)

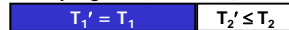


Old time: $T = T_1 + T_2$

T_1 = time that can NOT be enhanced.

T_2 = time that can be enhanced.

New program (enhanced)



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

T_2' = time after the enhancement.

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

CSC469



Computing Speedup

Two key parameters:

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

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

$$T' = T_1' + T_2' = T_1 + T_2' = T(1 - F_{\text{enhanced}}) + T_2'$$

$$= T(1 - F_{\text{enhanced}}) + (T_2 / S_{\text{enhanced}}) \quad [\text{by def of } S_{\text{enhanced}}]$$

$$= T(1 - F_{\text{enhanced}}) + T(F_{\text{enhanced}} / S_{\text{enhanced}}) \quad [\text{by def of } F_{\text{enhanced}}]$$

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

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.

CSC469



Trip example revisited

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

	Time MTL→TO→MTL	Total trip time	Speedup over VIA
VIA	9 hours	16 hours	1
WestJet	2.5 hours	9.5 hours	1.7
Rocket	0.25 hours	7.25 hours	2.2
Wormhole	0 hours	7 hours	2.3

CSC469



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.
- What does this say about parallel systems?

CSC469



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

CSC469

