CME 212: Timing and measuring performance

Lecture 10

February 10, 2012

What is performance?

- computers are systems of components that interact with each other
- complex system, hard to model analytically
- performance analysis is an experimental discipline of computer science
  - measurement
  - interpretation
  - communication
- very important for many vendors, easy to trick customers

Goals of performance analysis

- compare alternatives when buying computers
- determining the impact of a feature when designing a system or upgrading
- system tuning
- identify relative performance - that is, relative to previous systems
- performance debugging
- set expectations

Basic methodologies

Measurements

- not very general
- hard to change parameters
- time consuming
- intrusion of probes

Simulation (measure something like the real thing)

- easy to change parameters
- hard to model every detail
- needs validation

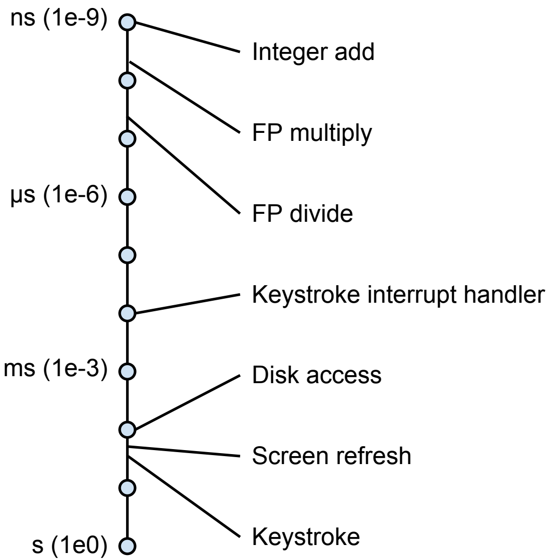
Analytical modelling

- hard

Measuring execution time

- why not use a stopwatch?
- in a modern time sharing operative system your code may not execute the entire time.
    - processes are interrupted by i/o, kernel activity and other processes
- the **wall clock time**, is the classic stopwatch time (time until prompt returns)
- the cpu time is the accumulated time the process actually ran on a cpu.
    - this can further be divided into **system** and **user** cpu time

Time scales (≈ 1GHz machine)



Measurement challenge

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") time
  - how many seconds elapse between the start and the completion of x?
  - depends on system load, i/o times, etc.

Confounding factors

- they way time is measured
- many processes share computing resources
- there are transient effects when switching from one process to another

"time" on a computer system

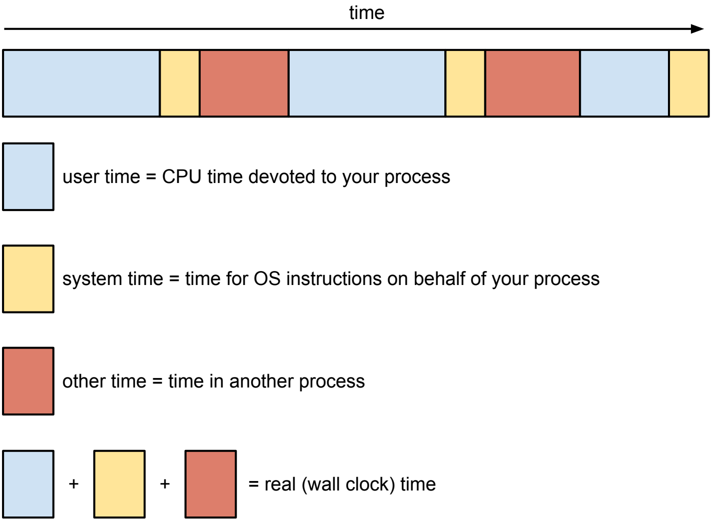
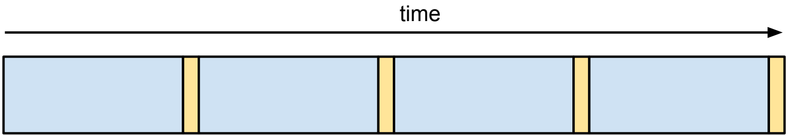


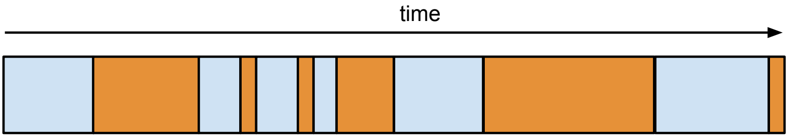
Figure: diagram of cpu time breakdown

Light load



- most of the time spent executing one process
- periodic interrupts every 10ms
- keep system from executing one process to exclusion of others

Heavy load



- sharing processor with one other active process
- from perspective of this process, system appears to be “inactive” for about 50% of the time

bash time

```
$ cd lapack
$ time make
...
gfortran  -O2 -c dlatms.f -o dlatms.o
gfortran  -O2 -c dlatme.f -o dlatme.o
...
real  2m25.957s
user  1m53.530s
sys 0m9.460s
```

This command is built into the `bash` shell. It does not provide very detailed information.

/usr/bin/time -v make -j8

See `man time`...

```
Command exited with non-zero status 2
  Command being timed: "make -j8"
  User time (seconds): 14.36
  System time (seconds): 0.69
  Percent of CPU this job got: 639%
  Elapsed (wall clock) time (h:mm:ss or m:ss): 0:02.35
  Average shared text size (kbytes): 0
  Average unshared data size (kbytes): 0
  Average stack size (kbytes): 0
  Average total size (kbytes): 0
  Maximum resident set size (kbytes): 121440
  Average resident set size (kbytes): 0
  Major (requiring I/O) page faults: 0
  Minor (reclaiming a frame) page faults: 532188
  Voluntary context switches: 964
  Involuntary context switches: 3179
  Swaps: 0
  File system inputs: 0
  File system outputs: 8552
  Socket messages sent: 0
  Socket messages received: 0
  Signals delivered: 0
  Page size (bytes): 4096
  Exit status: 2
```

Accessing timers in code

There are several methods to do this. We will discuss three

- `clock()` in `time.h`
- `gettimeofday()` in `sys/time.h`
- `x86/x86-64` cycle counter, accessible through an assembly instruction

It's good to know where your timers come from

- `time.h` is an interval counter in standard C
- `sys/time.h` comes from POSIX standards, these have changed over time
- Cycle counters are highly system dependent

Interval counting

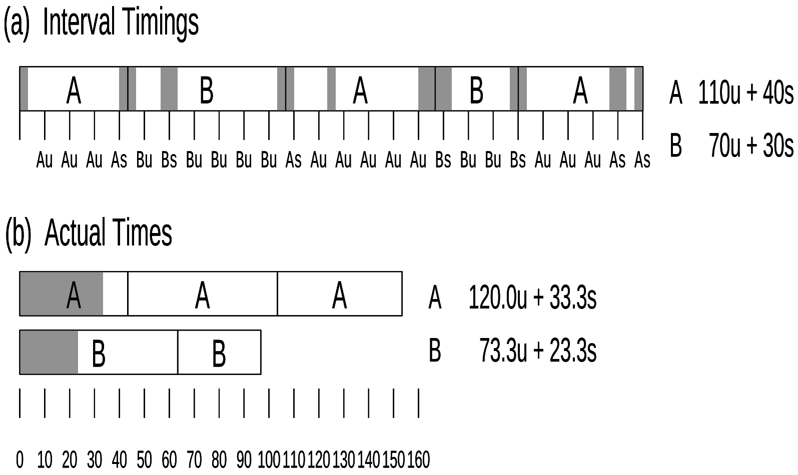
OS measures runtimes using interval timer

- maintains 2 counts per process
    1. user time
    2. system time

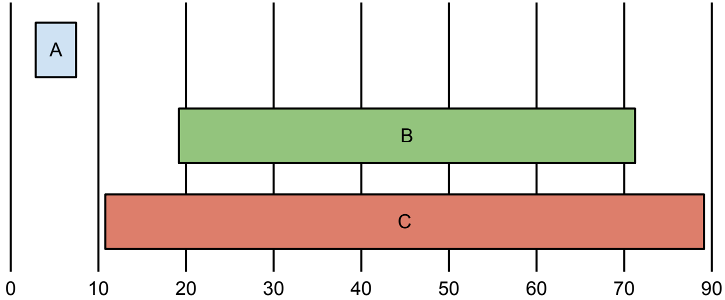
Each time you get a timer interrupt, increment counter for executing process

- this is called a clock tick
- increment user time if running in user mode
- increment system time if running in kernel mode

Interval counting example



Interval counting errors



- A is not counted because it feel between the intervals
- B is over timed
- C is under timed
- No bound for this, counters can consistently over or under measure
- Things tend to average out in the long run

Basic use of `time.h`

```c
#include <time.h>
...
clock_t start;
clock_t finish;
double run_time;

start = clock();
// do something
finish = clock();

run_time = finish-start;
run_time /= CLOCKS_PER_SEC;
```

- see `$ man 3 clock`
- On my computer: `CLOCKS_PER_SEC = 1000000`
- Has a resolution of 0.01 s
- There is overhead in call to `clock()`
- `run_time` is now in seconds
- Only user process time is counted

An experiment with `clock`

Let's measure the time it takes to call `y = exp(x)`

- allocate two `double` arrays of length N, say `x` and `y`
- loop over the arrays with the assignment `y[i] = exp(x[i])`
- time the loop using `clock()`
- repeat this process M times to get a good estimate

To get estimate of final time

- take average over M samples and divide by N

Any guesses?

The code

Inner loop:

```
void vec_exp(vec_t *v1, vec_t *v2) {
  for (size_t i = 0; i != v1->n; ++i)
    v2->a[i] = exp(v1->a[i]);
}
```

Sample loop:

```
clock_t start, finish;
for (size_t i=0; i != num_run; ++i) {
  start = clock();
  vec_exp(v1,v2);
  finish = clock();
  time_vec->a[i] = finish-start;
  time_vec->a[i] /= CLOCKS_PER_SEC;
}
// compute summary
r.mean = vec_mean(time_vec);
r.min = vec_min(time_vec);
r.max = vec_max(time_vec);
```

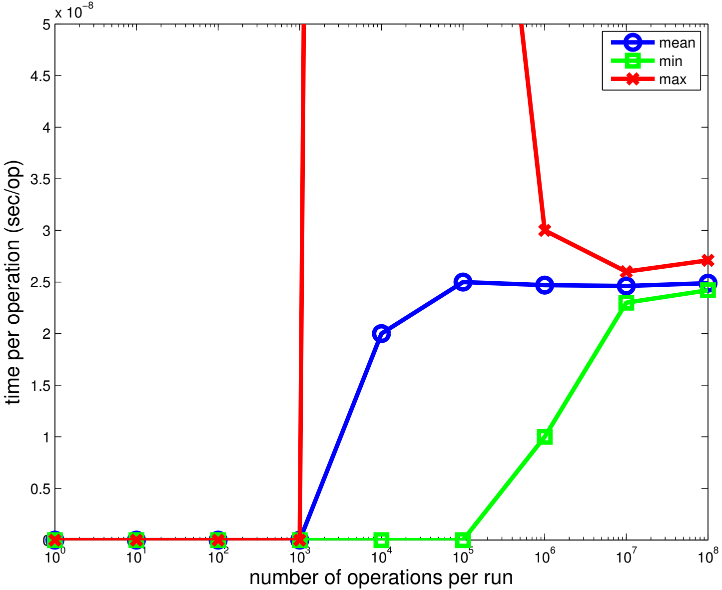


Figure: Timing results from `clock()` for different values of N

`clock()` table 1

Table: time per operation vs sample size

size	expected	mean	min	max
1e+01	2.5e-08	0.0e+00	0.0e+00	0.0e+00
1e+02	2.5e-08	0.0e+00	0.0e+00	0.0e+00
1e+03	2.5e-08	0.0e+00	0.0e+00	0.0e+00
1e+04	2.5e-08	2.0e-08	0.0e+00	1.0e-06
1e+05	2.5e-08	2.5e-08	0.0e+00	1.0e-07
1e+06	2.5e-08	2.5e-08	1.0e-08	3.0e-08
1e+07	2.5e-08	2.5e-08	2.3e-08	2.6e-08
1e+08	2.5e-08	2.5e-08	2.4e-08	2.7e-08

`clock()` table 2

Table: sample time vs sample size

size	expected	mean	min	max
1e+00	2.5e-08	0.0e+00	0.0e+00	0.0e+00
1e+01	2.5e-07	0.0e+00	0.0e+00	0.0e+00
1e+02	2.5e-06	0.0e+00	0.0e+00	0.0e+00
1e+03	2.5e-05	0.0e+00	0.0e+00	0.0e+00
1e+04	2.5e-04	2.0e-04	0.0e+00	1.0e-02
1e+05	2.5e-03	2.5e-03	0.0e+00	1.0e-02
1e+06	2.5e-02	2.5e-02	1.0e-02	3.0e-02
1e+07	2.5e-01	2.5e-01	2.3e-01	2.6e-01
1e+08	2.5e+00	2.5e+00	2.4e+00	2.7e+00

Repeat the experiment with `gettimeofday()`

- See `$ man gettimeofday`
- Interface:

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

struct timeval {
  time_t      tv_sec;     /* seconds */
  suseconds_t tv_usec;    /* microseconds */
};

int gettimeofday(struct timeval *tv, \
                 struct timezone *tz);
```

- In linux, the second argument should always be `NULL`
- Note the microsecond resolution
- Returns the wall clock time

Some of the code for gettimeofday()

```
typedef struct timeval timeval_t;
double elapsed_time(timeval_t start, timeval_t finish) {
  double start_s = (double)start.tv_sec +
    1.0e-6*(double)start.tv_usec;
  double finish_s = (double)finish.tv_sec +
    1.0e-6*(double)finish.tv_usec;
  return finish_s-start_s;
}
...
  // run all tests
  timeval_t start;
  timeval_t finish;
  for (size_t i=0; i != num_run; ++i) {
    gettimeofday(&start,NULL);
    vec_exp(v1,v2);
    gettimeofday(&finish,NULL);
    time_vec->a[i] = elapsed_time(start,finish);
  }
```

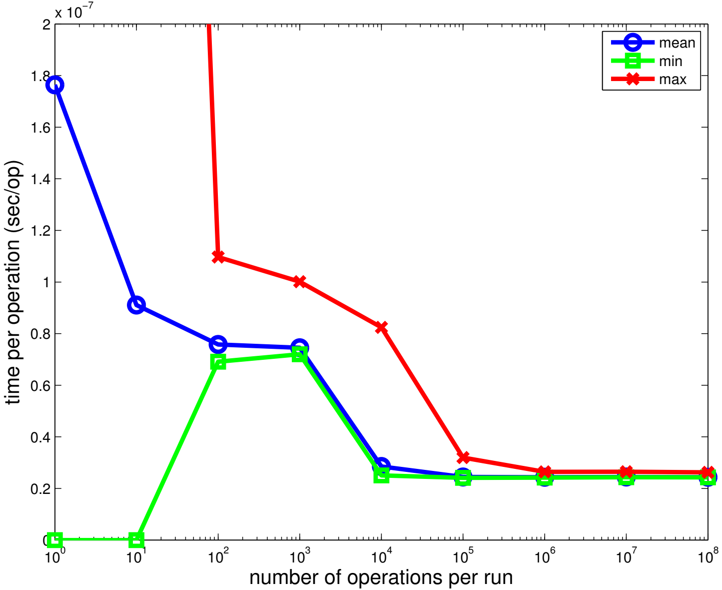


Figure: Timing results from `gettimeofday()` for different values of N

`gettimeofday()` table 1

Table: time per operation vs sample size

size	expected	mean	min	max
1e+00	2.5e-08	1.8e-07	0.0e+00	6.9e-06
1e+01	2.5e-08	9.1e-08	0.0e+00	9.1e-07
1e+02	2.5e-08	7.6e-08	6.9e-08	1.1e-07
1e+03	2.5e-08	7.5e-08	7.2e-08	1.0e-07
1e+04	2.5e-08	2.8e-08	2.5e-08	8.2e-08
1e+05	2.5e-08	2.4e-08	2.4e-08	3.2e-08
1e+06	2.5e-08	2.4e-08	2.4e-08	2.6e-08
1e+07	2.5e-08	2.4e-08	2.4e-08	2.6e-08
1e+08	2.5e-08	2.4e-08	2.4e-08	2.6e-08

`gettimeofday()` table 2

Table: sample time vs sample size

size	expected	mean	min	max
1e+00	2.5e-08	1.8e-07	0.0e+00	6.9e-06
1e+01	2.5e-07	9.1e-07	0.0e+00	9.1e-06
1e+02	2.5e-06	7.6e-06	6.9e-06	1.1e-05
1e+03	2.5e-05	7.5e-05	7.2e-05	1.0e-04
1e+04	2.5e-04	2.8e-04	2.5e-04	8.2e-04
1e+05	2.5e-03	2.4e-03	2.4e-03	3.2e-03
1e+06	2.5e-02	2.4e-02	2.4e-02	2.6e-02
1e+07	2.5e-01	2.4e-01	2.4e-01	2.6e-01
1e+08	2.5e+00	2.4e+00	2.4e+00	2.6e+00

Cycle counters

- Most modern systems have built-in registers that are incremented every clock cycle
- Very fine-grained measurement
- Need special assembly instructions to access
- On recent Intel machines:
  - 64-bit counter
  - Use RDTSC instructions to access
  - Need to worry about out of order execution
  - On Core i7 can use RDTSCP, otherwise call LFENCE before
- Let's see it in action...

RDTSC example: estimate clock speed

```
typedef unsigned long long ticks;
static __inline__ ticks getticks(void)
{
  unsigned a, d;
  asm("lfence");
  asm volatile("rdtsc" : "=a" (a), "=d" (d));
  //core i7 and beyond can use following w/o lfence
  //asm volatile("rdtscp" : "=a" (a), "=d" (d));
  return ((ticks)a) | (((ticks)d) << 32);
}
...
ticks a = getticks();
sleep(1);
ticks b = getticks();
printf("b-a␣=␣%llu\n",b-a);
```

On my computer:

```
b-a = 2926902987 // thats 2.93 GHz!
```

Some samples:

Just the timing calls:

```
ticks a = getticks();
ticks b = getticks();
```

b-a = 106 (~36 ns)

Second call to `exp()`

```
y = exp(z);
ticks a = getticks();
y = exp(x);
ticks b = getticks();
```

b-a = 704 (~240 ns)

Call to `exp()`

```
ticks a = getticks();
y = exp(x);
ticks b = getticks();
```

b-a = 37445 (~13000 ns)

Still not quite 25 ns!

Some notes on timers and counters

- Process counters will overflow, more of an issue on 32-bit machines
- `gettimeofday()` will give wall time, not process time
- The hardware count includes cycles for all other processes

Summarizing rates

- Let's say you are setting up an experiment to time 100,000 floating point operations.
- You do 2 experiments, the first finishes in 1 second and the next finishes in 2 seconds. Thus,
  - experiment 1 rate: 100 Mflop / sec
  - experiment 2 rate: 50 Mflop / sec
- If we use the arithmetic mean

$$\frac{100+50}{2} = 75 \text{ Mflop / sec}$$

- Is this an appropriate measure?

Harmonic mean

The **harmonic mean** of a set of positive real numbers $x_1, x_2, \ldots x_n > 0$ is defined

$$H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_1} + \cdots + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}.$$

From the example:

$$\frac{2}{\frac{1}{100} + \frac{1}{50}} = 66.6 \text{ Mflop / sec}$$

This is equivalent to:

$$\frac{100 \text{ Mflop} + 100 \text{ Mflop}}{3 \text{ sec}} = 66.6 \text{ Mflop / sec}$$

Key: use harmonic mean for **rates**!

Measurement summary

Timing is highly case and system dependent

- What is overall duration being measured?
  - > 1 second: interval counting is OK
  - $<< 1$ second: must use cycle counters

On what hardware / OS / OS version?

- Accessing counters
  - How `gettimeofday()` is implemented
- Timer interrupt overhead
- Scheduling policy

Devising a Measurement Method

- Long durations: use Unix timing functions (interval)
- Short durations
  - If possible, use `gettimeofday()`
  - Otherwise must work with cycle counters