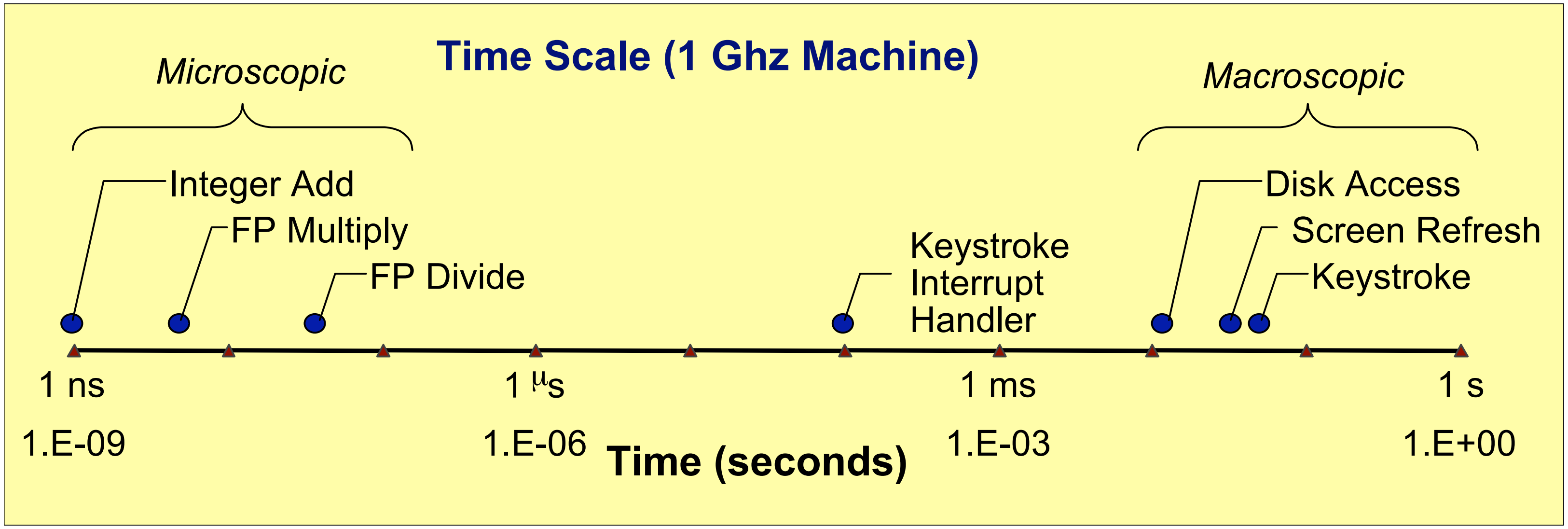
Time Measurement

CS 201
Gerson Robboy
Portland State University

Topics

- Time scales
- Interval counting
- Cycle counters
- K-best measurement scheme

Computer Time Scales



Two Fundamental Time Scales

- Processor: ~10^{-9} sec.
- External events: ~10^{-2} sec.
  - Keyboard input
  - Disk seek
  - Screen refresh

Implication

- Can execute many instructions while waiting for external event to occur
- Can alternate among processes without anyone noticing

Measurement Challenge

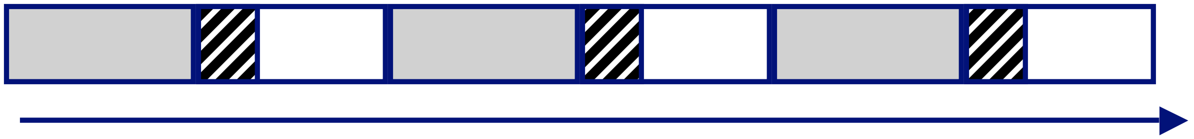
How Much Time Does Program X Require?

- CPU time
  - How many total seconds are used when executing X?
  - 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

- How does time get measured?
- Many processes share computing resources

“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)*

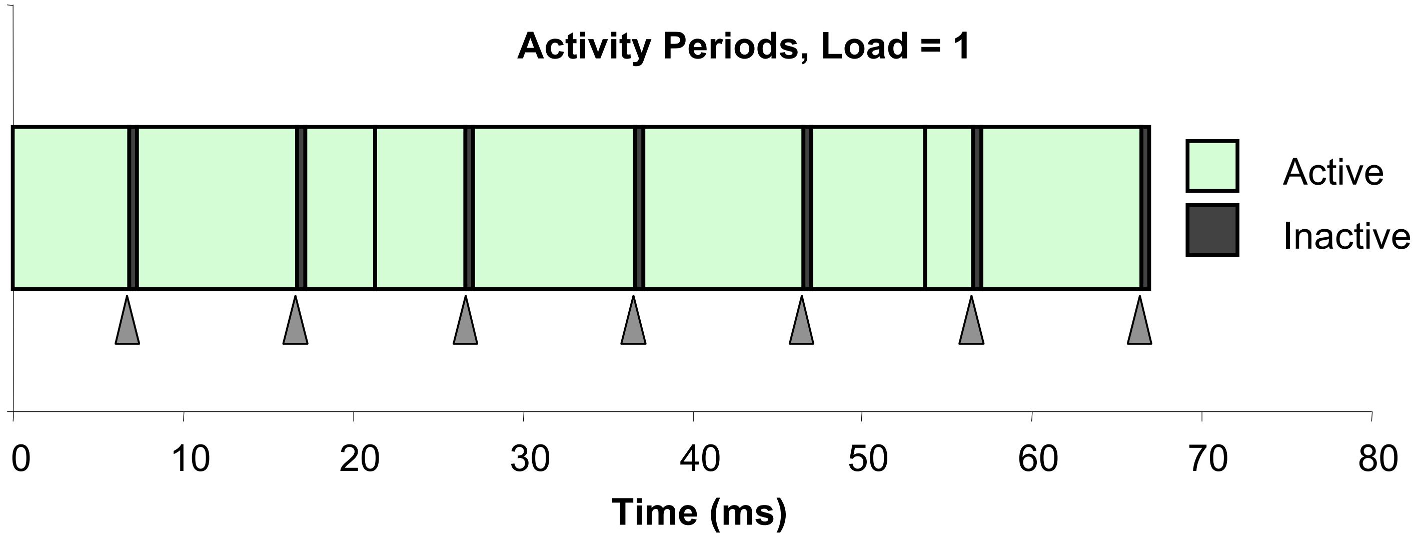
□ **= some other user’s time** *(time executing instructions in different user’s process)*

▒ + ▨ + □ **= real (wall clock) time**

We will use the word “time” to refer to user time.

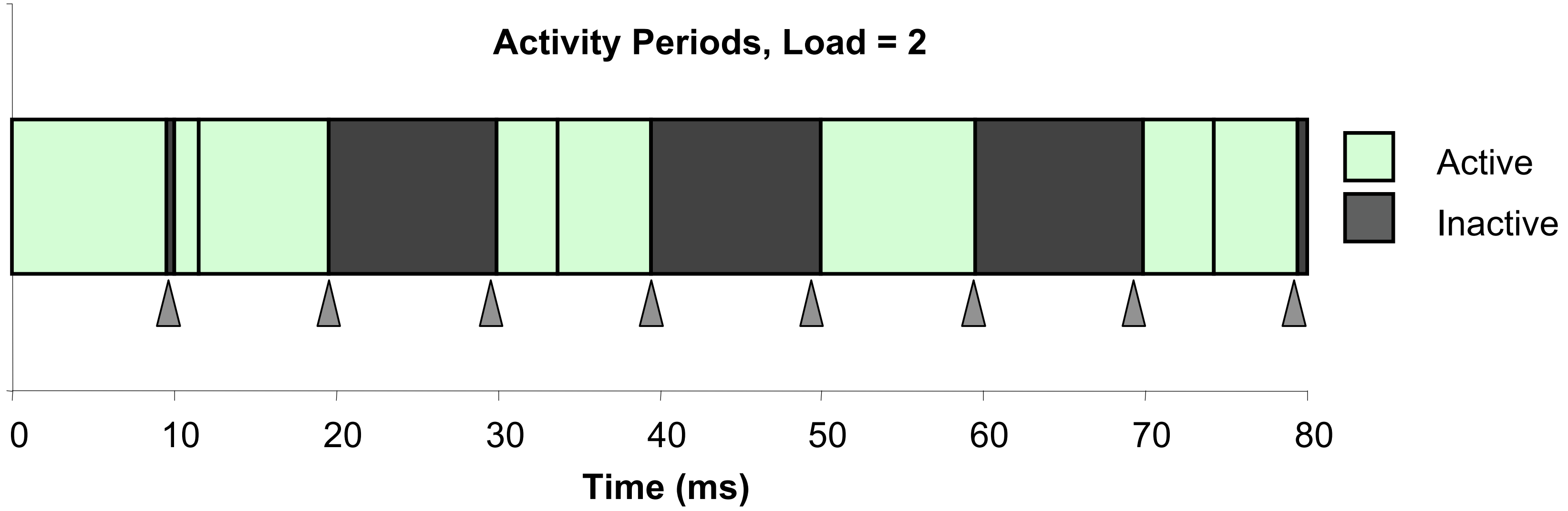
▒▒▒ **cumulative user time**

Activity Periods: Light Load



- Most of the time spent executing one process
- Periodic interrupts every 10ms
  - Interval timer
  - Schedules processes to run
- Other interrupts
  - Due to I/O activity
- Inactivity periods
  - System time spent processing interrupts
  - ~250,000 clock cycles

Activity Periods: Heavy Load



- Sharing processor with one other active process
- From perspective of this process, system appears to be “inactive” for ~50% of the time
  - Other process is executing

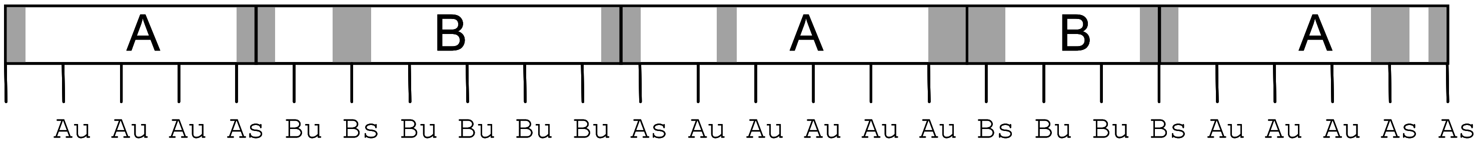
Interval Counting

OS Measures Runtimes Using Interval Timer

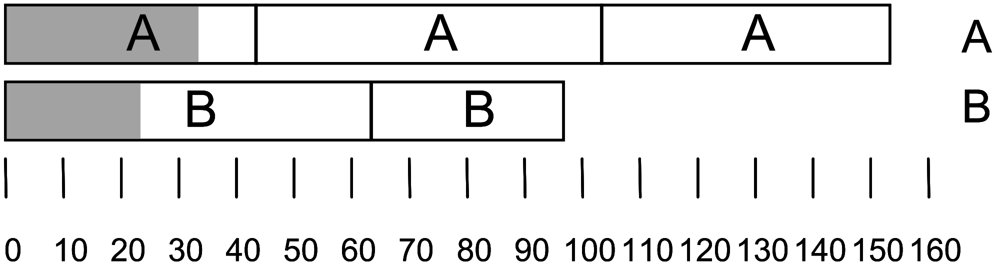
- In other words, statistical sampling.
  - Similar to profiling.
- Maintain 2 counts per process
  - User time
  - System time
- On each timer interrupt, increment counter for executing process
  - User time if running in user mode
  - System time if running in kernel mode

Interval Counting Example

(a) Interval Timings


A
B
A
B
A
Au Au Au As Bu Bs Bu Bu Bu Bu As Au Au Au Au Au Bs Bu Bu Bs Au Au Au As As

(b) Actual Times



A	120.0u + 33.3s
B	73.3u + 23.3s

Exercise: What timings does the interval timer give us for Au, As, Bu, and Bs? How far off are they from “actual?”

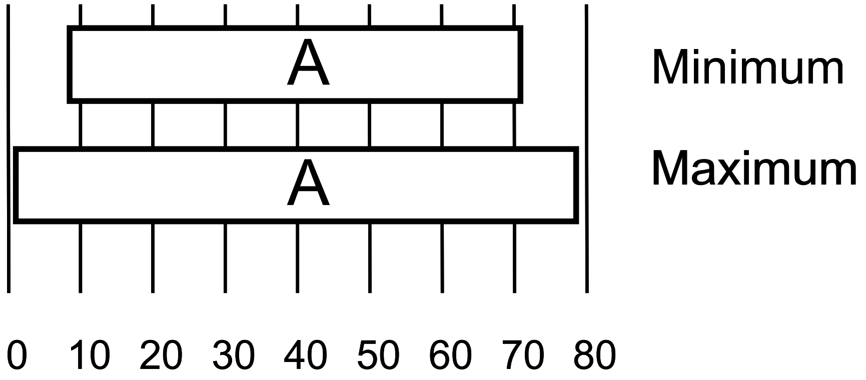
Unix `time` Command

```
time make osevent
gcc -O2 -Wall -g  -march=i486 -c clock.c
gcc -O2 -Wall -g  -march=i486 -c options.c
gcc -O2 -Wall -g  -march=i486 -c load.c
gcc -O2 -Wall -g  -march=i486 -o osevent osevent.c . . .
0.820u 0.300s 0:01.32 84.8%     0+0k 0+0io 4049pf+0w
```

- **0.82 seconds user time**
  - **82 timer intervals**
- **0.30 seconds system time**
  - **30 timer intervals**
- **1.32 seconds wall time**
- **84.8% of total was used running these processes**
  - **(.82+0.3)/1.32 = .848**

Exactly what process or processes are using that .82 of user time?

Accuracy of Interval Counting

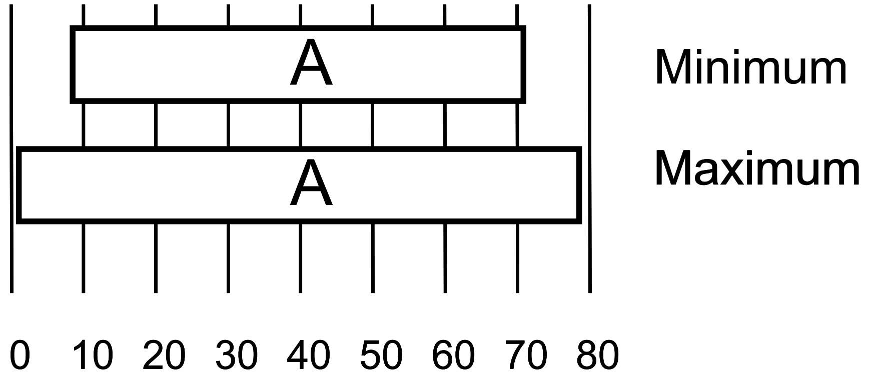


- **Computed time = 70ms**
- **Min Actual = 60 + ε**
- **Max Actual = 80 – ε**

Worst Case Analysis

- **Single process segment measurement can be off by how much?**
- **No bound on error for multiple segments**
  - **Could consistently underestimate, or consistently overestimate**
  - **Pretty unlikely**

Accuracy of Int. Cntg. (cont.)



- Computed time = 70ms
- Min Actual = 60 + ε
- Max Actual = 80 – ε

Average Case Analysis

- Over/underestimates tend to balance out
- As long as total run time is sufficiently large
  - Min run time ~1 second
  - 100 timer intervals
- Consistently miss 4% overhead due to timer interrupts

Cycle Counters

- Most modern systems have built in registers that are incremented every clock cycle
  - Very fine grained
  - Elapsed global time (“wall clock” time)
- Accessed with a special instruction
- On (recent model) Intel machines:
  - It’s a 64 bit counter called the time stamp counter.
  - Accessed with RDTSC instruction
  - RDTSC 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 years

For 2 GHz machine

- Low order 32-bits every 2.1 seconds
- High order 64 bits every 293 years

Measuring with Cycle Counter

Main 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 cycle 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);
}
```

So how do you implement *access_counter()* in C?

Need to do an RDTSC instruction.

One way: Write a function in assembly language, in a separate source file, conforming to the C language call/return conventions.

An easier way with gcc: Use inline assembly code

- When you see the syntax, you may not believe it’s easier

Accessing Linux Cycle Counter

GCC allows inline assembly code with mechanism for matching registers with program variables

This 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)
      : /* No input */
      : "%edx", "%eax");
}
```

Emit assembly with rdtsc and two movl instructions

Extended ASM: A Closer Look

```
asm(“Instruction String"
      : Output List
      : Input List
      : Clobbers List);
}
```

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

Instruction String

- Series of assembly commands
  - Separated by “;” or “\n”
  - Use “%%” where normally would use “%”

Extended ASM, Cont.

```
asm(“Instruction String"
      : Output List
      : Input List
      : Clobbers List);
}
```

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

Output List

- Expressions indicating destinations for values %0, %1, …, %*j*
  - Enclosed in parentheses
  - Must be *lvalue*
    - » Value that can appear on LHS of assignment
- Tag "=r" indicates that symbolic value (%0, etc.), should be replaced by register

Extended ASM, Cont.

```
asm(“Instruction
String"
       : Output List
       : Input List
       : Clobbers List);
}
```

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

Input List

- Series of expressions indicating sources for values %*j*+1, %*j*+2, …
  - Enclosed in parentheses
  - Any expression returning value
- Tag "r" indicates that symbolic value (%0, etc.) will come from register

Extended ASM, Cont.

```
asm(“Instruction
String"
       : Output List
       : Input List
       : Clobbers List);
}
```

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

Clobbers List

- List of register names that get altered by assembly instruction
- Compiler will make sure doesn’t store something in one of these registers that must be preserved across asm
  - Value set before & used after

Accessing Cycle Counter, Cont.

Emitted Assembly Code

```
movl 8(%ebp),%esi      # hi
movl 12(%ebp),%edi     # lo

rdtsc; movl %edx,%ecx; movl %eax,%ebx

movl %ecx,(%esi)       # Store high bits at *hi
movl %ebx,(%edi)       # Store low bits at *lo
```

- Used %ecx for *hi (replacing %0)
- Used %ebx for *lo (replacing %1)
- Does not use %eax or %edx for value that must be carried across inserted assembly code

Accessing the Cycle Cntr. (cont.)

Now are you convinced you’ll never write one of those?

Relax, no one ever does, from scratch. Start with an existing snippet of code that looks pretty much like what you want.

Inline assembly really comes in handy sometimes, once you get used to it.

Completing the Measurement

- Perform double precision subtraction to get elapsed cycles
- Express as `double` to avoid overflow problems

```
double get_counter()
{
  unsigned ncyc_hi, ncyc_lo
  unsigned hi, lo, borrow;
  /* Get 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;
}
```

- *Why do they do this?*

```
borrow = lo > ncyc_lo;
```

Timing With Cycle Counter

Determine Clock Rate of Processor

- Count number of cycles required for some fixed number of seconds

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

Time Function P

- First attempt: Simply count cycles for one execution of P

```
double tsecs;
start_counter();
P();
tsecs = get_counter() / (MHZ * 1e6);
```

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 Cache Effects

- Execute function once to “warm up” the cache
  - Both data and instructions

```
P();                  /* Warm up cache */
start_counter();
P();
cmeas = get_counter();
```

- Issue: Some functions don’t execute in cache
- Depends on both the algorithm and the data set
  - Need to do measurements with appropriate data

Multitasking Effects

Cycle Counter Measures Elapsed Time

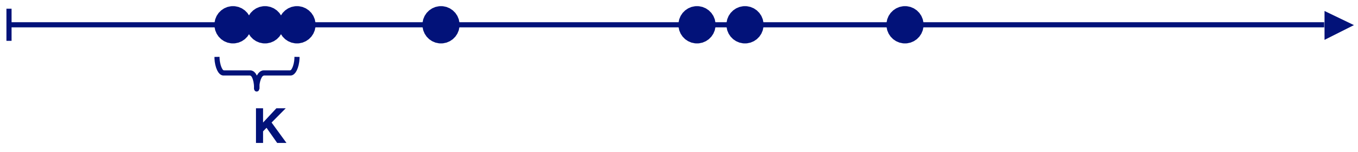
- Keeps accumulating during periods of inactivity
  - System activity
  - Running other processes

Key Observation

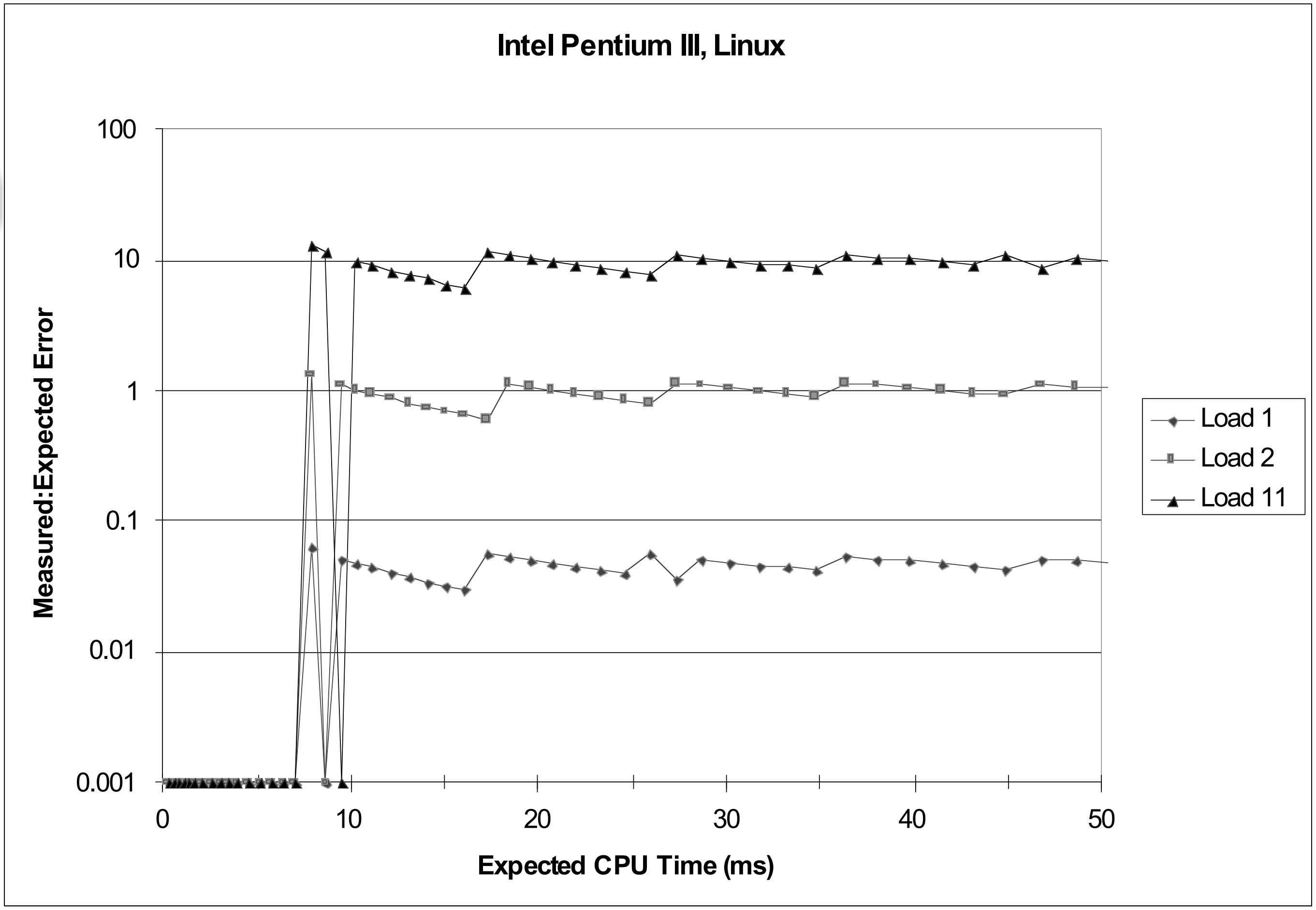
- Cycle counter never underestimates program run time
- Possibly overestimates by large amount

K-Best Measurement Scheme

- Perform up to N (e.g., 20) measurements of function
- See if fastest K (e.g., 3) within some relative factor ε (e.g., 0.001)



K-Best Validation



K = 3, ε = 0.001

Very good accuracy for < 8ms

- Within one timer interval
- Even when heavily loaded

Less accurate of > 10ms

- Light load: ~4% error
  - Interval clock interrupt handling
- Heavy load: Very high error

Time of Day Clock

- Unix `gettimeofday()` function
- Return elapsed time since reference time (Jan 1, 1970)
- Implementation
  - Uses interval counting on some machines
    - » Coarse grained
  - Uses cycle counter on others
    - » Fine grained, but significant overhead and only 1 microsecond resolution

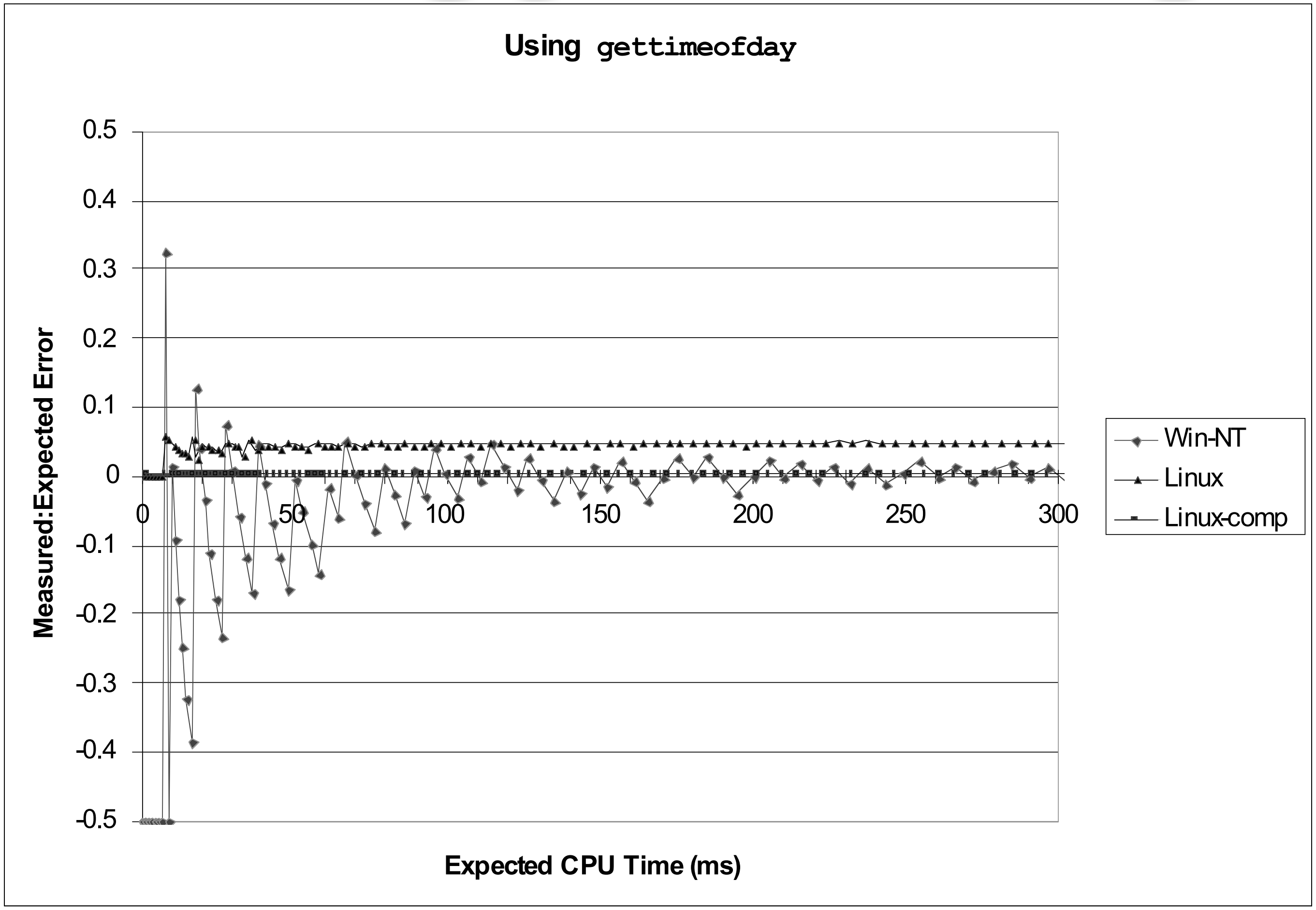
```
#include <sys/time.h>
#include <unistd.h>

  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);
```

Nice things about *gettimeofday()*

- Portable!
- No need to do double-precision integer arithmetic
- 1 Microsecond resolution is pretty good for most purposes

K-Best Using `gettimeofday`



Linux

- As good as using cycle counter
- For times > 10 microseconds

Windows

- Implemented by interval counting
- Too coarse-grained

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
- Short durations
  - If possible, use `gettimeofday`
  - Otherwise must work with cycle counters
  - K-best scheme most successful