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

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?

- 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
“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)

- 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 + ε
  - Max Actual = 80 - ε

- **Worst case**
  - Timer interval δ
  - Single measurement can be off by +/- δ
  - 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

```c
/* 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
- Emit assembly with `rdtsc` and two `movl` instructions
- Code generates two outputs:
  - Symbolic register %0 should be used for `cyc_hi`
  - Symbolic register %1 should be used for `cyc_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

```c
/* 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:

```c
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)

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:

```c
foo1(array1, array2, array3); /* 68,829 cycles */
foo2(array1, array2, array3); /* 23,337 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

```c
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 *= 2;
} 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, 23,365
    - 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 $\Delta t$ seconds
  - Typically $\Delta t = 10$ ms
  - Same as interval timer resolution

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

- Measurement takes place without interval timer advancing

```
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

- $K = 3$

- 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)

<table>
<thead>
<tr>
<th></th>
<th>Time NY→Paris→NY</th>
<th>Total trip time</th>
<th>Speedup over 747</th>
</tr>
</thead>
<tbody>
<tr>
<td>747</td>
<td>8.5 hours</td>
<td>16.5 hours</td>
<td>1</td>
</tr>
<tr>
<td>SST</td>
<td>3.75 hours</td>
<td>11.75 hours</td>
<td>1.4</td>
</tr>
</tbody>
</table>

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

Speedup

<table>
<thead>
<tr>
<th>Old program (unenhanced)</th>
<th>New program (enhanced)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>T₂</td>
</tr>
</tbody>
</table>

Old time: T = T₁ + T₂

New time: T = T₁' + T₂'

Speedup: S_{overall} = T / T'
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):

<table>
<thead>
<tr>
<th></th>
<th>Time NY→Paris→NY</th>
<th>Total trip time</th>
<th>Speedup over 747</th>
</tr>
</thead>
<tbody>
<tr>
<td>SST</td>
<td>3.75 hours</td>
<td>11.75 hours</td>
<td>1.4</td>
</tr>
<tr>
<td>Rocket</td>
<td>0.25 hours</td>
<td>8.25 hours</td>
<td>2.0</td>
</tr>
<tr>
<td>Wormhole</td>
<td>0 hours</td>
<td>8 hours</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Lessons from Amdahl’s Law

- Useful Corollary of Amdahl’s law:
  \[ 1 \leq S_{\text{overall}} \leq \frac{1}{1 - F_{\text{enhanced}}} \]

<table>
<thead>
<tr>
<th>( F_{\text{enhanced}} )</th>
<th>( \text{Max } S_{\text{overall}} )</th>
<th>( F_{\text{enhanced}} )</th>
<th>( \text{Max } S_{\text{overall}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1</td>
<td>0.9375</td>
<td>16</td>
</tr>
<tr>
<td>0.5</td>
<td>2</td>
<td>0.96875</td>
<td>32</td>
</tr>
<tr>
<td>0.75</td>
<td>4</td>
<td>0.984375</td>
<td>64</td>
</tr>
<tr>
<td>0.875</td>
<td>8</td>
<td>0.9921875</td>
<td>128</td>
</tr>
</tbody>
</table>

- 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