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
Two fundamental time scales:
- Processor: \( \sim 1 \) nanosecond \((10^{-9} \text{ secs})\)
- External events: \( \sim 10 \) milliseconds \((10^{-2} \text{ secs})\)
  - 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 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 $2^{32} / (550 \times 10^6) = 7.8$ seconds
  - High order 64-bits wrap around every $2^{64} / (550 \times 10^6) = 33539534679$ 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
  - Subtract to get elapsed cycles
- Needs inline assembly to get access to cycle counter register
  - Details in tutorial tomorrow
Time of Day Clock

- return elapsed time since some reference time (e.g., Jan 1, 1970)
- example: Unix gettimeofday() command
- coarse grained (e.g., ~3μsec 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

```c
#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 $C_{MIN} = 50000$

```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 += count;
} while (cmeas < CMIN); /* Make sure have enough */
return cycles / (1e6 * MHZ);
```
Context Switching

- Context switches can also affect cache performance
  - e.g., \((\text{foo1, foo2})\) 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 $\Delta t$ seconds
    - Typically $\Delta t = 10$ ms
    - Same as interval timer resolution

![Diagram showing time and $\Delta t$]

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

```c
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

```c
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

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.
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
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.

<table>
<thead>
<tr>
<th></th>
<th>Time MTL-&gt;TO-&gt;MTL</th>
<th>Total trip time</th>
<th>Speedup over VIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIA</td>
<td>9 hours</td>
<td>16 hours</td>
<td>1</td>
</tr>
<tr>
<td>WestJet</td>
<td>2.5 hours</td>
<td>9.5 hours</td>
<td>1.7</td>
</tr>
</tbody>
</table>

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

**Old program (unenhanced)**

$T_1$ $\quad$ $T_2$

Old time: $T = T_1 + T_2$

**New program (enhanced)**

$T_1' = T_1$ $\quad$ $T_2' \leq T_2$

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

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

$T_1$ = time that can NOT be enhanced.

$T_2$ = time that can be enhanced.

$T_2'$ = time after the enhancement.
Computing Speedup

Two key parameters:

\[ F_{\text{enhanced}} = \frac{T_2}{T} \quad \text{(fraction of original time that can be improved)} \]
\[ S_{\text{enhanced}} = \frac{T_2}{T'} \quad \text{(speedup of enhanced part)} \]

\[ T' = T_1' + T_2' = T_1 + T_2' = T(1 - F_{\text{enhanced}}) + T_2' \]
\[ = T(1 - F_{\text{enhanced}}) + (T_2/S_{\text{enhanced}}) \quad \text{[by def of } S_{\text{enhanced}}]\]
\[ = T(1 - F_{\text{enhanced}}) + T(F_{\text{enhanced}}/S_{\text{enhanced}}) \quad \text{[by def of } F_{\text{enhanced}}]\]
\[ = T((1 - F_{\text{enhanced}}) + F_{\text{enhanced}}/S_{\text{enhanced}}) \]

Amdahl’s Law:

\[ S_{\text{overall}} = \frac{T}{T'} = \frac{1}{((1 - F_{\text{enhanced}}) + F_{\text{enhanced}}/S_{\text{enhanced}})} \]

- **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 MTL to TO (15 minutes), or a wormhole opens between MTL and TO (0 minutes):

<table>
<thead>
<tr>
<th></th>
<th>Time MTL→TO→MTL</th>
<th>Total trip time</th>
<th>Speedup over VIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIA</td>
<td>9 hours</td>
<td>16 hours</td>
<td>1</td>
</tr>
<tr>
<td>WestJet</td>
<td>2.5 hours</td>
<td>9.5 hours</td>
<td>1.7</td>
</tr>
<tr>
<td>Rocket</td>
<td>0.25 hours</td>
<td>7.25 hours</td>
<td>2.2</td>
</tr>
<tr>
<td>Wormhole</td>
<td>0 hours</td>
<td>7 hours</td>
<td>2.3</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.
• What does this say about parallel systems?
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