Week 3:

Performance Evaluation

CSC 469 / CSC 2208

Fall 2018
Overview

• Common question: “How fast does program X run on machine Y?"
• Near-perfect timing measurements on a compute systems should be straightforward. *Or is it?*
• Many factors that can vary from one execution of a program to another
  • Switching between processes => scheduling processor resources depends on number of users sharing the system, network traffic, timing of disk operations
  • Access patterns to caches => not just current process, other concurrent processes too
  • Branch prediction logic => history can vary
• 2 basic mechanisms: interval counting, cycle counting
• Goal: methods to get reliable measurements of program performance
Performance Evaluation Topics

- Time scales
- Interval counting
- Cycle counting
- K-best measurement scheme
- Performance metrics
- Amdahl’s Law
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
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
    • Some 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 \((time \text{ executing instructions in the user process})\)

= system time \((time \text{ executing instructions in kernel on behalf of user process})\)

= some other user’s time \((time \text{ executing instructions in different user’s process})\)

= real (wall clock) time

We use the word “time” to refer to user time.
Activity Periods: Light Load

- Most of the time spent executing one process
- Periodic interrupts every 10ms
  - Interval timer
- Other interrupts
  - Due to I/O activity
  - Inactivity periods
    - System time spent processing interrupts
Activity Periods: Heavier 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
  - 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)
Interval Counting Example

(a) Interval Timings

(b) Actual Times

Imprecise: Timing at the granularity of the timer interval!
Accuracy of Interval Counting

• Worst case
  • Timer interval $\delta$
  • Single measurement can be off by +/- $\delta$
  • 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)

- Interval timer reports 70 ms
- Min Actual = 60 + e
- Max Actual = 80 - e
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 2 GHz machine
  - Low order 32-bits wrap around every \( \frac{2^{32}}{2 \times 10^9} = 2.1 \) seconds
  - High order 64-bits wrap around every \( \frac{2^{64}}{2 \times 10^9} = 9223372037 \) seconds
    - 293 years
- For 5 GHz machine
  - Low order 32-bits wrap every 0.86 seconds
  - High order 64-bits wrap every 116 years
- See tutorial notes for usage details
Measuring program execution

- Wrap rdtsc instruction within an `access_counter` function call (use inline assembly – see tutorial)

```c
static u_int64_t start = 0;
void access_counter(unsigned* hi, unsigned* low);

void start_counter() {
    unsigned hi, lo;
    access_counter(&hi, &lo);
    start = ((u_int64_t)hi << 32) | lo;
}

u_int64_t get_counter() {
    unsigned ncyc_hi, ncyc_lo;
    access_counter(&ncyc_hi, &ncyc_lo);
    return (((u_int64_t)ncyc_hi << 32) | ncyc_lo) - start;
}
```

Measuring Cycles (Basic Idea):
- Get current value of cycle counter
- Compute something
- Get new value of cycle counter
- Get elapsed time (in cycles) by subtraction
Measurement Pitfalls

- **Overhead**
  - Calling `get_counter()` incurs small amount of overhead
  - Want to measure long enough code sequence to compensate

- **Unexpected Cache Effects**
  - Warm vs cold caches, artificial hits or misses
  - e.g., these are actual measurements (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 (both data and instr)

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

- Do we expect code to access the same data repeatedly?
- Some functions don’t execute in the cache
- Depends on both the algorithm and the data set
Dealing with Overhead & Cache Effects

• What if it’s more likely that the code will access new data with each execution?
  • To force timing code to measure the performance of code under “cold caches”, we could **flush caches** before the actual experiment
  • Example: Write code that does some operations on a large dummy array to evict existing cached blocks
  • **Problem**: clearing caches may also **clear all instructions** of P() from L2
    • Overestimates time for P()
  • Side note: on Linux, can clear page cache using:
    • `sudo sh -c 'echo 1 >/proc/sys/vm/drop_caches'`
Dealing with Overhead & Cache Effects

- Execute P() once to warm up cache
- Keep doubling the number of times we execute P() until some threshold is reached
  - Used CMIN = 50000

```c
int count = 1;
double ctr_meas = 0;
double cycles;
do {
    int reps = count;
P(); /* Warm up cache */
    get_counter();
    while (reps-- > 0)
        P();
        ctr_meas = get_counter();
cycles = ctr_meas / count;
count += count;
} while (ctr_meas < CMIN); /* Make sure we 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 (in this example)
  - Some amount of overhead
  - Caches polluted by OS and other user’s code & data
    - Cold misses occur when process is restarted

- Measurement Strategy
  - Try to measure uninterrupted code execution
Detecting Context Switches

• Clock Interrupts
  • Processor clock causes interrupt every $\Delta t$ seconds
    • Typically $\Delta t = 1-10$ ms
    • Same as interval timer resolution
  
  Measurement takes place without interval timer advancing

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

```c
start = get_etime(); /* timer_gettime() wrapper, measures thread time */

/* 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 and generally take a long time to service

- Detecting
  - See if real time clock has advanced
    - Using coarse-grained timer
      - Similar to `get_etime()`, but uses `CLOCK_REALTIME`

```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
It’s worse than that…

• Modern OS may support multiple clock sources

  • E.g. Linux:

  wolf:~> cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm

  wolf:~> cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc

• But tsc may not be a stable source of timing information
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
Portability: Time of Day Clock

- Return elapsed time since some reference time (e.g., Jan 1, 1970)
- Example: Unix `gettimeofday()` command
- Coarse grained vs fine grained (e.g., ~3\(\mu\)sec resolution on older Linux, 10 msec resolution on Windows NT, same as cycle counter on new Linux)
- Possibly 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 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, otherwise accuracy low!
  • On what hardware / OS / OS version?
    • Accessing counters
      • How is gettimeofday() 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
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
  - Getting accurate timings on heavily loaded systems is especially difficult!
  - Frequency scaling may also be an issue
- It’s difficult to get repeatable times
  - Cache effects due to ordering and context switches
- Every system is different!
- Moral of the story:
  - Adopt a healthy skepticism about measurements!
  - Always subject measurements to sanity checks.
Modern CPUs contain counters for low-level architectural events e.g:

- instructions executed, branches taken, cache accesses, etc.

Example – using perf tools:

- perf stat --repeat $N -e $event1 -e $event2 -e $event3 -e $event4 -- someprogram
- perf stat --repeat 5 -e cycles -e instructions -e cache-references -e cache-misses -e syscalls:sys_enter -e syscalls:sys_exit -- sh -c "/usr/bin/postgres --single mydb -D $PGDATA < query.sql > /dev/null"

```
10074181817 cycles # 0.000 M/sec ( +/- 0.098% )
12648819036 instructions # 1.256 IPC ( +/- 0.003% )
73553771 cache-references # 0.000 M/sec ( +/- 0.152% )
476347 cache-misses # 0.000 M/sec ( +/- 0.915% )
55834 syscalls:sys_enter # 0.000 M/sec ( +/- 0.000% )
55834 syscalls:sys_exit # 0.000 M/sec ( +/- 0.000% )
3.806739355 seconds time elapsed ( +/- 0.160% )
```
Hardware Performance Monitoring

- Hard to use because
  - Limited number of counters ➔ can’t count all interesting events at the same time
  - Non-standard (libraries like PAPI help)
  - Poor documentation
  - Extracting performance insight from low-level microarchitectural events is tough
1. Metrics - a measurable quantity that is the basis for comparison
   - Choosing a good metric requires deciding what factors are most important
   - Latency, bandwidth, throughput are common in computer systems
   - Give me some other ones...
   - Capacity, utilization, overhead, useful work, etc..

2. A system to measure
   - Model
   - Simulation
   - “Live”

3. A set of tests to perform on the target system
   - Benchmarks

What else?
1. Choosing Metrics

• What performance metric should be used to compare the following?
  • Two disk drives
  • Two transaction processing systems
  • Two packet retransmission algorithms
  • Two clock scaling algorithms for reducing energy usage
Characteristics of a good performance metric

• Intuitive (For all stakeholders)
• Reliable
  • Trusted for useful comparison and prediction
  • Repeatable
• Easy to measure
  • No complicated metric that’s difficult to measure correctly
• Consistent
  • Definition is the same across different configurations and different systems
  • In many cases not necessarily true (ex. MIPS and MFLOPS)
• Independent of outside influences
  • No intervention from vendors to influence the composition of the metric to their benefit
2. Choosing a system to measure

- Models
  - rigorous mathematical model, insight into effects of different parameters
  - before a system is built

- Simulation
  - simulate the system operation (or small parts)

- Live System
  - implement the system in full and measure its performance directly
  - get the test infrastructure set up and test it “live”
Techniques – pros and cons

• Models
  + cheap, fast to develop
  - highly simplified, not always accurate, depends on accuracy of assumptions

• Simulation
  + flexibility: easy to vary parameters, test assumptions
  - cost/time depends on level of detail, less detailed simulations may leave out important factors

• Live System
  + real results, can’t overlook contribution of other components
  - can be hard to interpret, effects of specific parameters may be hard to isolate
  - expensive: buy test infrastructure, implement the system in full, etc.
General advice

- Simulation is generally the most widely-used
  - Not necessarily the best though
- Generally recommended to use combination of techniques, if possible
- Don’t trust the results produced by just one method
  - Validate one method with another
    - e.g., modeling + simulation, simulation + live system
3. Choosing experiments

• Suppose that the performance of a system depends on the following three factors:
  • Garbage collection technique used (concurrent, stop and copy, none)
  • Type of workload (office desktop computing, database server, scientific computing)
  • Type of CPU (Pentium, POWER PC)

How many experiments are needed?

How do you quantify the performance impact of each factor?
Designing Workloads

- Can’t always measure “real” workload
  - Want repeatability
  - Want to test new ideas, can’t deploy in real setting
- Macro benchmarks emulate typical workloads
- May be run by large community
  - SPEC
  - TPC
- Vigilance is still needed
  - E.g., scaling issues
  - E.g., file system benchmarking
And then there’s analysis

- Why performance analysis is an “art” not a “science”:
- Given the following measurements of throughput:

<table>
<thead>
<tr>
<th>System:</th>
<th>Workload 1</th>
<th>Workload 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

- What is a fair comparison?
Some possibilities…

- **Absolute:**

<table>
<thead>
<tr>
<th>System</th>
<th>Workload1</th>
<th>Workload2</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>20</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>20</td>
<td>15</td>
</tr>
</tbody>
</table>

- **Performance of A relative to B:**

<table>
<thead>
<tr>
<th>System</th>
<th>Workload1</th>
<th>Workload2</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2x</td>
<td>0.5x</td>
<td>1.25x</td>
</tr>
<tr>
<td>B</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
</tr>
</tbody>
</table>

- **Performance of B relative to A:**

<table>
<thead>
<tr>
<th>System</th>
<th>Workload1</th>
<th>Workload2</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>B</td>
<td>0.5x</td>
<td>2x</td>
<td>1.25x</td>
</tr>
</tbody>
</table>
General Advice

- Understand the goals
  - Solid understanding of the problem, Solid understanding of the system
  - Difficult => Goals may change once problem is better understood

- Be careful about bias
  - “system X is better than system Y”
  - Don’t select metric for highlighting a particular system, conduct a proper comparison
  - Findings may be skewed by bias => inaccurate/incomplete conclusions

- Use a systematic approach
  - Arbitrary selection of system parameters, metrics, workloads => inaccurate/incomplete conclusions
More advice

• Understand the phenomena being measured
  • Is variance caused by experimental noise or is there intrinsic variance?
• Decide if you want the minimum, mean or 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→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>
</tbody>
</table>

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

| T₁                           | T₂                           |

Old time: \( T = T₁ + T₂ \)

New program (enhanced)

| T₁' = T₁                     | T₂' ≤ T₂                    |

New time: \( T' = T₁' + T₂' \)

Speedup: \( S_{\text{overall}} = \frac{T}{T'} \)

When we speed up one part of a program, the overall system performance depends on both how significant the part is, and how much it was sped up.
Two key parameters:

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

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

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

- \( S_{overall} = \frac{1}{(1 - F_{enhanced}) + F_{enhanced} \frac{1}{S_{enhanced}}} \)

- Ex1: Calculate Min and Max speedup bounds:
  - \( Min \leq S_{overall} \leq Max \)

- Ex2: Calculate Max \( S_{overall} \) for the following values of \( F_{enhanced} \):
  - 0.9375
  - 0.96875
  - 0.984375
  - 0.9921875
  - 0
  - 0.5
  - 0.75
  - 0.875

- What do you notice?
Lessons from Amdahl’s Law

• Useful Corollary of Amdahl’s law:

\[ 1 \leq S_{overall} \leq \frac{1}{(1 - F_{enhanced})} \]

Remember: \[ S_{overall} = \frac{1}{(1 - F_{enhanced}) + \frac{F_{enhanced}}{S_{enhanced}}} \]

<table>
<thead>
<tr>
<th>(F_{enhanced})</th>
<th>Max (S_{overall})</th>
<th>(F_{enhanced})</th>
<th>Max (S_{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
  • Recall week1 (problems in complex systems)?