Lecture 10
Measurement and Timing

Charles E. Leiserson
```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

void my_sort(double *A, int n);
void fill(double *A, int n);

struct timespec start, end;

int main() {
    int max = 4*1000*1000;
    int min = 1;
    int step = 20 * 1000;
    double A[max];

    for (int n=min; n<max; n+=step){
        fill(A, n);
        clock_gettime(CLOCK_MONOTONIC, &start);
        my_sort(A, n);
        clock_gettime(CLOCK_MONOTONIC, &end);

        double tdiff = (end.tv_sec - start.tv_sec) + 1e-9*(end.tv_nsec - start.tv_nsec);
        printf("size %d, time %f\n", n, tdiff);
    }
    return 0;
}
```

Library for `clock_gettime()`

Sorting routine to be timed.

Auxiliary routine for filling array with random numbers.

Used by `clock_gettime()`:

```c
struct timespec {
    time_t tv_sec; /* seconds */
    long tv_nsec; /* nanoseconds */
};
```

Inspired by a study due to Sivan Toledo.
Timing a Code for Sorting

```c
#include <stdio.h>
#include <time.h>

void my_sort(double *A, int n);
void fill(double *A, int n);

struct timespec start, end;

int main() {
    int max = 4 * 1000 * 1000;
    int min = 500 * 1000;
    int step = 20 * 1000;
    double A[max];

    for (int n=min; n<max; n+=step) {
        fill(A, n);
        clock_gettime(CLOCK_MONOTONIC, &start);
        my_sort(A, n);
        clock_gettime(CLOCK_MONOTONIC, &end);
        double tdiff = (end.tv_sec - start.tv_sec) + 1e-9*(end.tv_nsec - start.tv_nsec);
        printf("size %d, time %f\n", n, tdiff);
    }
    return 0;
}
```

Loop over arrays of increasing length.

Measure time before sorting.

Sort.

Measure time after sorting.

Compute elapsed time.
Running Times for Sorting

- Measured running time
- Best fit to $c_1 \cdot n \lg n$
- Best fit to $c_2 \cdot n$

array size $n$

What is going on?
Dynamic Frequency and Voltage Scaling

**DVFS** is a technique to reduce power by adjusting the clock frequency and supply voltage to transistors.

- Reduce operating frequency if chip is too hot or otherwise to conserve (especially battery) power.
- Reduce voltage if frequency is reduced.

\[
\text{Power } \propto C \, V^2 \, f
\]

\(C = \text{dynamic capacitance}\)
\(\approx \text{roughly area } \times \text{activity (how many bits toggle)}\)
\(V = \text{supply voltage}\)
\(f = \text{clock frequency}\)

*Reducing frequency and voltage results in a cubic reduction in power (and heat).*

But it wreaks havoc on performance measurements!
How can one reliably measure the performance of software?
OUTLINE

• QUIESCING SYSTEMS
• TOOLS FOR MEASURING SOFTWARE PERFORMANCE
• PERFORMANCE MODELING
OUTLINE

• Quiescing Systems
• Tools for Measuring Software Performance
• Performance Modeling

© 2008–2018 by the MIT 6.172 Lecturers
Question: If you were an Olympic pistol coach, which shooter would you recruit for your team?

Answer: B, because you just need to teach B to shoot lower and to the left.

Performance-engineering lesson
If you can reduce variability, you can compensate for systematic and random measurement errors.
Sources of Variability

- Daemons and background jobs
- Interrupts
- Code and data alignment
- Thread placement
- Runtime scheduler

- Hyperthreading
- Multitenancy
- Dynamic voltage and frequency scaling (DVFS)
- Turbo Boost
- Network traffic
Unquiesced System

Experiment (joint work with Tim Kaler)
- Cilk program to count the primes in an interval
- AWS c4 instance (18 cores)
- 2-way hyperthreading on, Turbo Boost on
- 18 Cilk workers
- 100 runs, each about 1 second
Quiesced System

Experiment (joint work with Tim Kaler)
- Cilk program to count the primes in an interval
- AWS c4 instance (18 cores)
- 2–way hyperthreading off, Turbo Boost off
- 18 Cilk workers
- 100 runs, each about 1 second

© 2008–2018 by the MIT 6.172 Lecturers
Quiescing the System

- Make sure no other jobs are running.
- Shut down daemons and cron jobs.
- Disconnect the network.
- Don’t fiddle with the mouse!
- For serial jobs, don’t run on core 0, where interrupt handlers are usually run.
- Turn hyperthreading off.
- Turn off DVFS.
- Turn off Turbo Boost.
- Use `taskset` to pin Cilk workers to cores.
- Etc., etc. (Already done for you with `awsrun`.)
A small change to one place in the source code can cause much of the generated machine code to change locations. Performance can vary due to changes in cache alignment and page alignment.

**Similar:** Changing the order in which the *.o files appear on the linker command line can have a larger effect than going between `-O2` to `-O3`.

*cache and page alignment has changed*
LLVM Alignment Switches

LLVM tends to cache-align functions, but it also provides several compiler switches for controlling alignment:

- `-align-all-functions=<uint>`
  - Force the alignment of all functions.
- `-align-all-blocks=<uint>`
  - Force the alignment of all blocks in the function.
- `-align-all-nofallthru-blocks=<uint>`
  - Force the alignment of all blocks that have no fall-through predecessors (i.e. don't add nops that are executed).

Aligned code is more likely to avoid performance anomalies, but it can also sometimes be slower.
Data Alignment

A program’s name can affect its speed!

- [Mytkowicz, Diwan, Hauswirth, and Sweeney, “Producing wrong data without doing anything obviously wrong,” 2009.]

  • The executable’s name ends up in an environment variable.
  • Environment variables end up on the call stack.
  • The length of the name affects the stack alignment.
  • Data access slows when crossing page boundaries.
OUTLINE

• **Quiescing Systems**
• **Tools for Measuring Software Performance**
• **Performance Modeling**
Ways to Measure a Program

• Measure the program externally.
  ➢ /usr/bin/time

• Instrument the program.
  ➢ Include timing calls in the program.
  ➢ E.g., gettimeofday(), clock_gettime(), rdtsc().
  ➢ By hand, or with compiler support.

• Interrupt the program.
  ➢ Stop the program, and look at its internal state.
  ➢ E.g., gdb, Poor Man’s Profiler, gprof.

• Exploit hardware and operating systems support.
  ➢ Run the program with counters maintained by the hardware and operating system, e.g., perf.

• Simulate the program.
  ➢ E.g., cachegrind.
The \texttt{time} command can measure elapsed time, user time, and system time for an entire program. What does that mean?

- \texttt{real} is wall-clock time.
- \texttt{user} is the amount of processor time spent in user-mode code (outside the kernel) within the process.
- \texttt{sys} is the amount of processor time spent in the kernel within the process.

\begin{verbatim}
real 0m3.502s
user 0m0.023s
sys 0m0.005s
\end{verbatim}
clock_gettime(CLOCK_MONOTONIC, ...)

```c
#include <time.h>
struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);
function_to_measure();
clock_gettime(CLOCK_MONOTONIC, &end);

double tdiff = (end.tv_sec - start.tv_sec)
               + 1e-9*(end.tv_nsec - start.tv_nsec);
```

- On my laptop, `clock_gettime(CLOCK_MONOTONIC, ...)` takes about 83ns.
- That’s about two orders of magnitude faster than a system call.
- `clock_gettime(CLOCK_MONOTONIC, ...)` guarantees never to run backwards.
x86 processors provide a **time-stamp counter** (TSC) in hardware. You can read TSC as follows:

```c
static __inline__ unsigned long long rdtsc(void)
{
    unsigned hi, lo;
    __asm__ __volatile__ (
        "rdtsc : " =a"(lo), "=d"(hi));
    return ( ((unsigned long long)lo) |
             (((unsigned long long)hi)<<32));
}
```

- The time returned is “clock cycles since boot.”
- `rdtsc()` runs in about 32ns.
Don’t Use Lousy Timers!

• \texttt{rdtsc()} may give different answers on different cores on the same machine.
• TSC sometimes runs backwards.
• The counter may not progress at a constant speed.
• Converting clock cycles to seconds can be ... tricky.
• Don’t use \texttt{rdtsc()}!
• And don’t use \texttt{gettimeofday()}, either, because it has similar problems!
Interrupting

• **IDEA:** Run your program under `gdb`, and type `control-C` at random intervals.
• Look at the stack each time to determine which functions are usually being executed.
• Who needs a fancy profiler?
• Some people call this strategy the “Poor Man’s Profiler.”
• `pmprof` and `gprof` automate this strategy to provide profile information for all your functions.
• Neither is accurate if you don’t obtain enough samples. (`gprof` samples only **100** times per second.)
Hardware Counters

- **libpfm4** virtualizes all the hardware counters.
- Modern kernels make it possible for libraries such as **libpfm4** to measure all the provided hardware event counters on a per-process basis.
- **perf stat** employs **libpfm4**.
- There are many esoteric hardware counters. Good luck figuring out what they all measure.
- Watch out: You probably cannot measure more than 4 or 5 counters at a time without paying a penalty in performance or accuracy.
Simulation

• Simulators, such as cachegrind, usually run much slower than real time.
• But they can deliver accurate and repeatable performance numbers.
• If you want a particular statistic, you can go in and collect it without perturbing the simulation.
OUTLINE

• QUIESCING SYSTEMS

• TOOLS FOR MEASURING SOFTWARE PERFORMANCE

• PERFORMANCE MODELING
1. Measure the performance of Program A.
2. Make a change to Program A to produce a hopefully faster Program A’.
3. Measure the performance of Program A’.
4. If A’ beats A, set A = A’.
5. If A is still not fast enough, go to Step 2.

If you can’t measure performance reliably, it is hard to make many small changes that add up.
Problem

Suppose that you measure the performance of a deterministic program 100 times on a computer with some interfering background noise. What statistic best represents the raw performance of the software?

- arithmetic mean
- geometric mean
- median
- maximum
- minimum
Problem

Suppose that you measure the performance of a deterministic program 100 times on a computer with some interfering background noise. What statistic best represents the raw performance of the software?

- arithmetic mean
- geometric mean
- median
- maximum
- **minimum**

Minimum does the best at noise rejection, because we expect that any measurements higher than the minimum are due to noise.
Selecting among Summary Statistics

**Service as many requests as possible**
- Arithmetic mean
- CPU utilization

**All tasks are completed within 10 ms**
- Arithmetic mean
- Wall-clock time

**Meet a customer service-level agreement (SLA)**
- Some weighted combination
- multiple

**Fit into a machine with 100 MB of memory**
- Maximum
- Memory use

**Most service requests are satisfied within 100 ms**
- 90th percentile
- Wall clock time

**Least cost possible**
- Arithmetic mean
- Energy use or CPU utilization

**Fastest/biggest/best solution**
- Arithmetic mean
- Speedup of wall clock time
Summarizing Ratios

Program B is > 3 times better than A.

WRONG!
### Paradox
If we look at the ratio $B/A$, then $A$ is better by a factor of almost $3$.

### Observation
The arithmetic mean of $A/B$ is **NOT** the inverse of the arithmetic mean of $B/A$. 

[Turn the Comparison Upside-Down](#)
## Geometric Mean

<table>
<thead>
<tr>
<th>Trial</th>
<th>Program A</th>
<th>Program B</th>
<th>A/B</th>
<th>B/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>3</td>
<td>3.00</td>
<td>0.33</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
<td>4.00</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>20</td>
<td>0.10</td>
<td>10.00</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>2</td>
<td>5.00</td>
<td>0.20</td>
</tr>
<tr>
<td>Mean</td>
<td>(a) 7.25</td>
<td>(a) 6.75</td>
<td>(g) 1.57</td>
<td>(g) 0.64</td>
</tr>
</tbody>
</table>

### Formula

\[
\left( \prod_{i=1}^{n} a_i \right)^{1/n} = \sqrt[n]{a_1 a_2 \cdots a_n}
\]

### Observation

The geometric mean of A/B IS the inverse of the geometric mean of B/A.
Comparing Two Programs

Q. You want to know which of two programs, A and B, is faster, and you have a slightly noisy computer on which to measure their performance. What is your strategy?

A. Perform $n$ head-to-head comparisons between A and B, and suppose A wins more frequently. Consider the null hypothesis that B beats A, and calculate the *P-value*: “If B beats A, what is the probability that we’d observe that A beats B more often than we did?” If the P-value is low, we can accept that A beats B.

*(See Statistics 101.)*

**Note:** With a lot of noise, we need lots of trials.
Fitting to a Model

Suppose that I have gathered this data:

<table>
<thead>
<tr>
<th>Program</th>
<th>Time (s)</th>
<th>Instructions</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>python</td>
<td>34864</td>
<td>170889186565542</td>
<td>36615004052</td>
</tr>
<tr>
<td>java</td>
<td>2618</td>
<td>7509707536406</td>
<td>39322034007</td>
</tr>
<tr>
<td>C gcc -O0</td>
<td>1480</td>
<td>2274589361551</td>
<td>68047140354</td>
</tr>
<tr>
<td>C gcc -O3</td>
<td>430</td>
<td>278479001783</td>
<td>34049504541</td>
</tr>
</tbody>
</table>

I want to infer how long it takes to run an instruction and how long to take a cache miss.

I guess that I can model the runtime $T$ as

$$T = a \cdot I + b \cdot C,$$

where

- $I$ is the number of instructions, and
- $C$ is the number of cache misses.
A least-squares regression can fit the data to the model

\[ T = a \cdot I + b \cdot C, \]

yielding

- \( a = 0.2002 \text{ ns} \)
- \( b = 18.00 \text{ ns} \)

with \( R^2 = 0.9997 \), which means that 99.97% of the data is explained by the model.
Adding more basis functions to the model improves the fit, but how do I know whether I’m overfitting?

- Removing a basis function doesn’t affect the quality much.

Is the model predictive?

- Pick half the data at random.
- Use that data to find the coefficients.
- Using those coefficients, find out how well the model predicts the other half of the data.

How can I tell whether I’m fooling myself?

- Triangulate.
- Check that different ways of measuring tell a consistent story.
- Analogously to a spreadsheet, make sure the sum of the row sums adds up to the sum of the column sums.