Software Performance Engineering

Recitation 1.6

Sophia Sun
Tuesday, October 7, 2025

Some due dates

- Project 1 Final is due this Thursday!
- You can use late days for your writeup but not your code submission.
- Make sure to run the correctness check first!
- Reach tier 35 to get at least a B grade for final.

- Homework 4 is due next Monday (October 13).

The Final Writeup

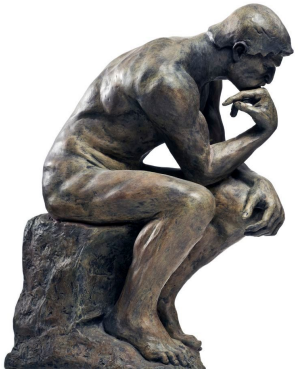
- Your final write-up can be an update of your beta write-up. In addition to the beta, it should include the following:

- An overview of changes you made to your beta release and what motivated you to do them
  - surprised by performance ranking
  - ideas conceived before the beta deadline but ran out of time implementing them
- Any updates to the acknowledgment, including specifically which of classmates' beta codes may have inspired you.

Performance Engineering

Performance Engineering

think,



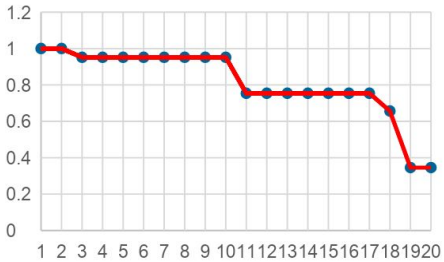
code,

run,

measure,

Basic Performance-Engineering Workflow

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

Tools to Measure Software Performance

How Much to Measure

- Measure the **whole** program
  - e.g., `/usr/bin/time`
  - e.g., `perf stat`, `cachegrind`, `strace`
- Measure just the **part** of the program we care about
  - Insert timing calls in the program
  - e.g., `clock_gettime()`, `gettimeofday()`, `rdtsc()`
- Create a **profile** of the program
  - e.g., `gdb`, `gprof`
  - use `perf record/report`, based on HW counters that OS and HW support
  - Using sampling or instrumentation

/usr/bin/time

The `time` command can measure elapsed time, user time, and system time for an entire program.

```
$ /usr/bin/time my-program arg1 arg2
real  0m3.502s
user  0m0.023s
sys       0m0.005s
```

What does that mean?

- `real` is wall-clock time (elapsed time).
- `user` is the amount of processor time spent in user-mode code (outside the kernel) within the process.
- `sys` is the amount of processor time spent in the kernel within the process

clock_gettime(CLOCK_MONOTONIC, …)

```
#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);
```

- Typically, `clock_gettime(CLOCK_MONOTONIC, …)` is fast
  - roughly 80ns — about 10^2 faster than an ordinary system call
- `clock_gettime(CLOCK_MONOTONIC, …)` has nice guarantees
  - it never runs backwards
- But `clock_gettime(CLOCK_MONOTONIC, …)` isn't always fast

rdtsc()

x86 processors provide a time-stamp counter (TSC) in hardware.

You can read TSC as follows:

For older compilers

```
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));
}
```

For newer compilers

```
__builtin_readcyclecounter();
```

- The time returned is “clock cycles since boot.”
- `rdtsc()` runs in about 32ns.

Issues with TSC

Use `rdtsc()` with caution!

- May give different answers on different cores on the same machine
- TSC can appear to run backwards, due to process migration
- Not all cycles take the same amount of time!
  - Modern processors change clock frequencies dynamically, e.g. DVFS, Turbo Boost
  - Processors reduce their clock frequency for AVX, AVX2, and AVX512 instructions
- Converting clock cycles to seconds can be tricky

Program Instrumentation

We can make a profiler by instrumenting the program.

```
static vec_t vec_add(vec_t a, vec_t b) {
  clock_gettime(CLOCK_MONOTONIC, &start);
  vec_t sum = { a.x + b.x, a.y + b.y };
  clock_gettime(CLOCK_MONOTONIC, &end);
  report_time("vec_add", &start, &end);
  return sum;
}

static vec_t vec_scale(vec_t v, double a) {
  clock_gettime(CLOCK_MONOTONIC, &start);
  vec_t scaled = { v.x * a, v.y * a };
  clock_gettime(CLOCK_MONOTONIC, &end);
  report_time("vec_scale", &start, &end);
  return scaled;
}
```

Instrumentation

Instrumentation

Instrumentation

Instrumentation

Caution: Watch out for probe effect, where program instrumentation alters the program's behavior in unintended ways.

Profiling by Sampling

IDEA: Interrupt the running program at regular intervals, and look at the stack each time to determine which functions are usually being executed.

- `pmprof`, `gprof`, `perf record`, and `gperftools` automate this strategy to provide profile information for all functions

- This approach is not accurate if it doesn't obtain enough samples. (`gprof` samples only `100` times per second.)

- You can do this yourself!
  - "Poor Man's Profiler": Run your program under `gdb`, and type control-C, `but` at random intervals

Simulation

Simulators can deliver accurate and repeatable perf numbers

- e.g. `cachegrind`, `gem5`

- For deterministic programs, you need only run the simulator once
- Simulators are robust to probe effect
  - If want a particular metric, you can go in and collect it without perturbing the simulation
- But they often run much slower than real time
- Simulators don't capture all relevant performance phenomena
  - e.g. `cachegrind` does not model all memory-system features, e.g., prefetching

Performance Surrogates

What could we measure as a surrogate for time?

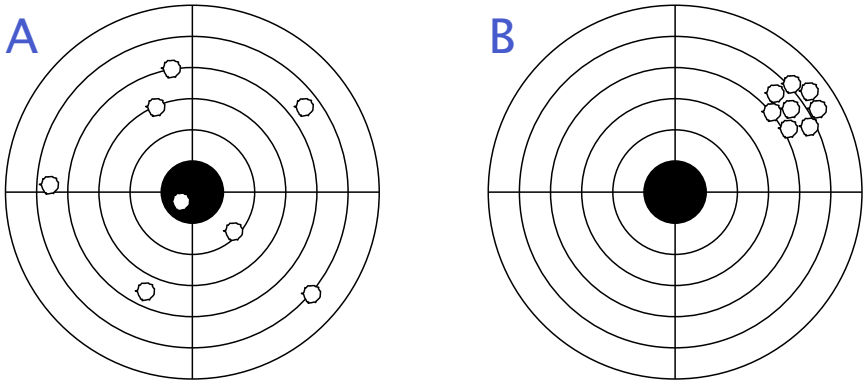
- Work: Number of executed instructions
  - Using hardware counters (on systems that support them) or program instrumentation
- Processor cycles
  - Using `rdtsc()`
- Memory accesses, or cache hits and misses
  - Using hardware counters
  - `cachegrind` simulation (gives repeatable numbers but slow)
- Span
  - Using Cilkscale

QUIESCING SYSTEMS

Genichi Taguchi and Quality

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.

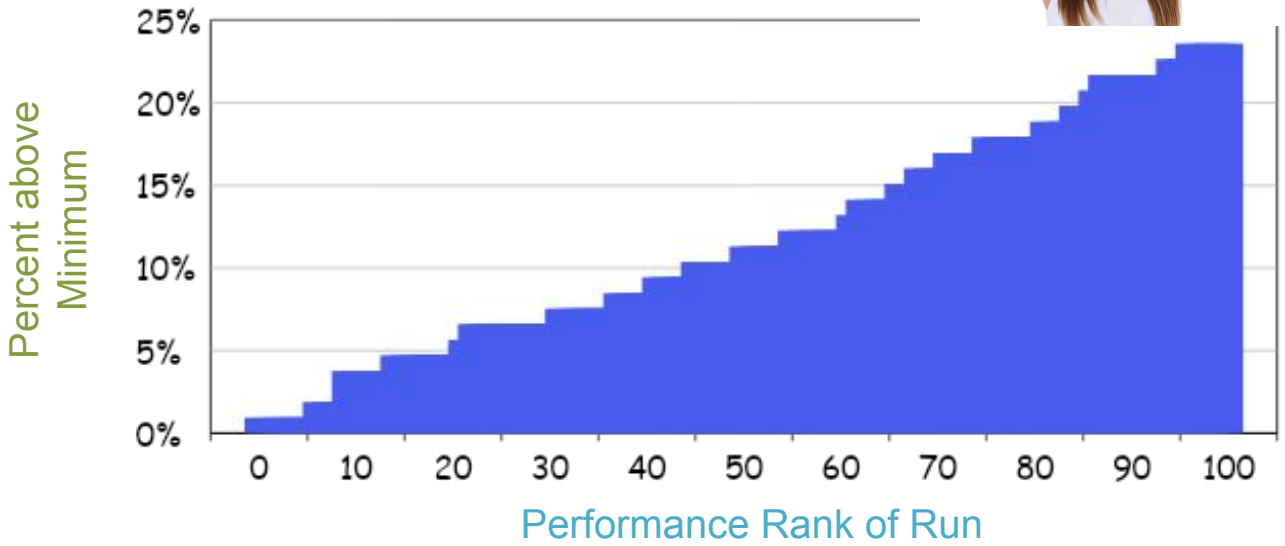
Sources of Variability

- Daemons and background jobs
- Interrupts
- Code and data alignment
- System calls
- Operating-system process scheduling
- Thread placement
- Runtime scheduler
- DVFS and Turbo Boost
- Network traffic
- Multitenancy
- Virtualization
- Hyperthreading

Unquiesced System

Experiment

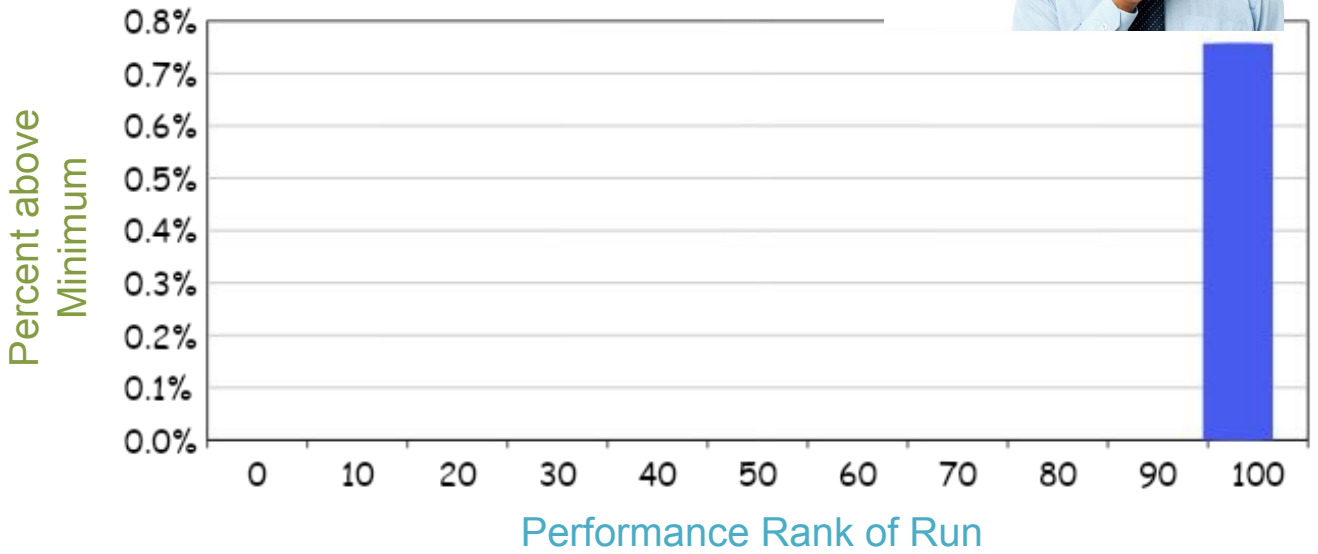
- Cilk program to count the primes in an interval
- AWS c4 instance (18 cores)
- 2-way hyperthreading on, Turbo Boost on
- 18 Cilk workers
- 100 runs, each about 1 second



Quiesced System

Experiment

- 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



Quiescing the System

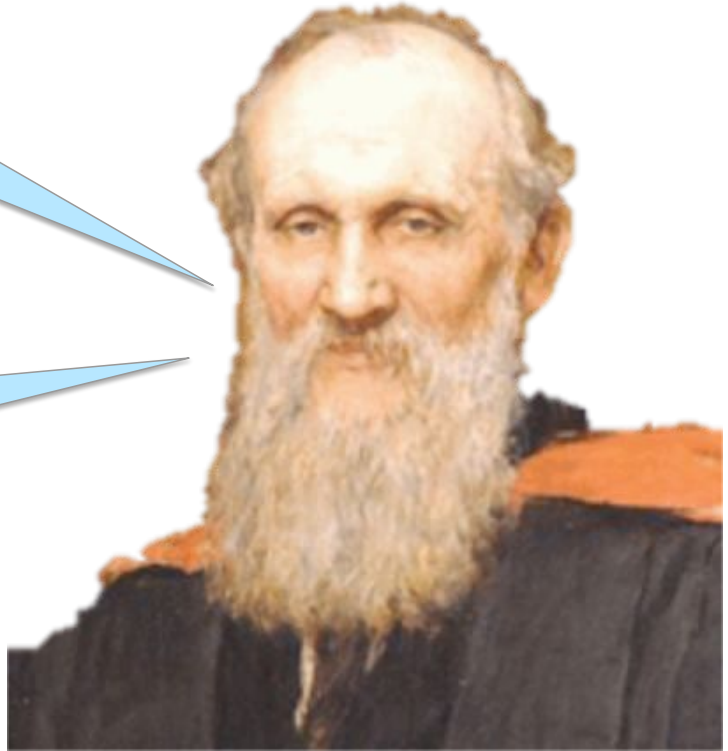
- Minimize the number of other jobs running
  - Shut down daemons and cron jobs
  - Disconnect the network
  - Don’t fiddle with the mouse!
  - For serial jobs, don’t run on core 0, where many interrupt handlers are usually run, see `/proc/interrupts`
- Use Linux CPU frequency governor to control DVFS and Turbo Boost
- Use `taskset` to pin Cilk workers to cores or hardware threads and avoid hyperthreading
- Many of these mitigations have been done for you in `telerun`

Wise Words from a Giant of Science

paraphrased

To measure is to know.

If you cannot measure it, you cannot improve it.



William Thomson, a.k.a., Lord Kelvin

Coding time before final due!

If you still haven't, think about these…

- How did you set up your temp buffer for loading and storing your blocks? How well does it utilize your cache?
- Any possibility that you can try out some different buffer size? How do you allocate your buffer if you want to load a different block size?
- How can you overlap some direct memory access time with the actual rotation time? (Hint: think about prefetching)
- In what order are you accessing the matrix right now when reading and writing blocks? How can you access your matrix in a cache-friendly way?