Lab 1: Multi-Level Performance Analysis

> ***Goal:*** *Learn to trace a performance observation through multiple levels: algorithm → hardware → OS.*

Overview

This lab teaches you to analyze performance systematically. You'll start with simple measurements, then dig deeper to understand *why* things are slow.

Structure (Tiered Difficulty):

Part	Difficulty	Required?	Estimated Time
Part A	Basic	✅ Yes	30-40 min in class
Part B	Intermediate	✅ Yes	40-50 min (class + homework)
Part C	Advanced	⭐ Optional	1-2 hours (homework)

Everyone completes Parts A and B. Part C is required for Master/Ph.D students.

Prerequisites

Before starting, verify your environment:

```
# Check tools
gcc --version           # Need GCC
perf --version          # Need perf (may need sudo)
valgrind --version      # Need valgrind

> **VM note (VMware Fusion / VirtualBox):** many `perf stat -e ...` **hardware
counter** events (cycles, cache-misses, branch-misses, etc.) may show `<not
supported>` or 0. Keep the `perf` commands in your writeup, but when counters are
unavailable, use the **Valgrind fallbacks** given below (cachegrind/callgrind).
```

If any are missing:

```
sudo apt update
sudo apt install -y build-essential linux-tools-common linux-tools-$(uname -r)
valgrind
```

Part A: Quicksort Warmup (Basic — Required)

Goal: Get familiar with profiling tools and collect baseline data.

Step 1: Get the Code

The starter code is in `lab1_quicksort/` directory.

```
cd week2A/lab1_quicksort
ls
# Should see: Makefile, main.c, src/, datasets/, outputs/
```

Step 2: Build

```
make clean
make
```

Step 3: Generate Test Datasets

```
# Random integers
shuf -i 1-10000 -n 10000 > datasets/random_10000.txt
shuf -i 1-50000 -n 50000 > datasets/random_50000.txt

# Sorted
seq 1 10000 > datasets/sorted_10000.txt
seq 1 50000 > datasets/sorted_50000.txt

# Reverse sorted
seq 10000 -1 1 > datasets/reverse_10000.txt
seq 50000 -1 1 > datasets/reverse_50000.txt

# Nearly sorted (90% sorted, 10% random swaps)
seq 1 10000 | awk 'BEGIN{srand()} {if(rand()<0.1) hold=$0; else {print; if(hold)
{print hold; hold=""}}}' > datasets/nearly_10000.txt
```

Step 4: Run and Measure

For each dataset, collect:

Basic timing:

```
time ./qs datasets/random_10000.txt
time ./qs datasets/sorted_10000.txt
time ./qs datasets/reverse_10000.txt
```

Hardware counters:

```
sudo perf stat -e cycles,instructions,cache-references,cache-misses,branches,branch-
misses ./qs datasets/random_10000.txt
```

If hardware counters are unavailable in your VM (common):

- Cache + branch behavior (simulated):

  ```
  valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes ./qs
  datasets/random_10000.txt
  ```

```
# Inspect: D1 misses / LLd misses / Branches / Mispredicts
```

- Hot spots (instruction-level):

  ```
  valgrind --tool=callgrind ./qs datasets/random_10000.txt
  callgrind_annotate --auto=yes callgrind.out.* | head -40
  # Optional GUI: kcachegrind
  ```

Memory usage:

```
valgrind --tool=massif --massif-out-file=massif_random.out ./qs
datasets/random_10000.txt
ms_print massif_random.out | head -30
```

Step 5: Record Results

Fill in this table (run each **3 times**, record all values):

Dataset	Run 1 (s)	Run 2 (s)	Run 3 (s)	Mean	cache-misses	branch-misses
random_10000						
sorted_10000						
reverse_10000						
nearly_10000						

Step 6: Quick Explanation

Answer in 2-3 sentences:

1. Which dataset is slowest? Why?
2. Does branch-miss rate explain the slowdown?
3. What about cache-miss rate?

Expected insight: Sorted input causes O(n^2) behavior due to bad pivot choices. The algorithm itself is the problem, not hardware. But you should still see this reflected in cycle count and instruction count.

✅ Part A Checklist

- [ ] Built the quicksort program
- [ ] Generated all 4 dataset types
- [ ] Ran each dataset 3 times
- [ ] Collected `perf stat` data
- [ ] Wrote 2-3 sentence explanation

Part B: Deep Analysis (Intermediate — Required)

Goal: Find the point where quicksort transitions from CPU-bound to memory-bound.

The Question

At what point does quicksort transition from CPU-bound to memory-bound?

- For small N, everything fits in cache → CPU-bound
- For large N, working set exceeds cache → memory-bound
- Where is the transition?

Hypothesis Formation

Before experimenting, write down your hypothesis:

1. What is your L3 cache size?

   ```
   lscpu | grep "L3 cache"
   # or
   cat /sys/devices/system/cpu/cpu0/cache/index3/size
   # or
   You know that from your M4 specs
   ```

2. If each integer is 4 bytes, how many integers fit in L3 cache?

3. At what N do you expect to see cache misses increase dramatically?

Write this down before running experiments!

Experiment Design

Design an experiment to test your hypothesis:

Suggested approach:

```
# Generate datasets of increasing size
for N in 1000 5000 10000 20000 50000 100000 200000 500000; do
    shuf -i 1-$N -n $N > datasets/random_$N.txt
done

# Measure each
for N in 1000 5000 10000 20000 50000 100000 200000 500000; do
    echo "=== N = $N ==="
    sudo perf stat -e cycles,instructions,cache-misses,cache-references ./qs
datasets/random_$N.txt 2>&1 | grep -E "(cycles|cache|instructions)"
done
```

VM fallback (when perf hardware events are `<not supported>`):

Use cachegrind to track cache-miss trends across N (simulated counters):

```
for N in 1000 5000 10000 20000 50000 100000 200000 500000; do
    echo "=== N = $N ==="
    valgrind --tool=cachegrind --cache-sim=yes --branch-sim=no ./qs
datasets/random_$N.txt 2>&1 | \
```

```
      egrep "(D1  misses|LLd misses|D   refs|I1  misses)"
done
```

(IPC may not be available without `cycles` + `instructions`; in that case, reason from time scaling + cachegrind trends.)

Metrics to Collect

N	User time (s)	Cycles	Instructions	Cache refs	Cache misses	Miss rate
1,000						
5,000						
10,000						
20,000						
50,000						
100,000						
200,000						
500,000						

Analysis Questions

1. **Cache miss rate transition:** At what N does cache miss rate increase significantly?
2. **Scaling behavior:** Plot time vs N. Is it linear (O(n log n))? Where does it deviate?
3. **Instructions per cycle (IPC):** Calculate IPC = instructions / cycles.
    - High IPC (~2-4) = CPU-bound, good cache behavior
    - Low IPC (<1) = Memory-bound, waiting for data
    - At what N does IPC drop?
4. **Connecting to theory:** Your L3 cache is approximately X MB. An array of N 4-byte integers occupies N*4 bytes. At N = L3_size/4, you should see behavior change. Does it?

Deliverable for Part B

Write 1-2 pages explaining:

1. Your hypothesis (before running experiments)
2. Your experiment design
3. Your results (include the data table and/or a plot)
4. Your interpretation: Was your hypothesis correct? What mechanism explains the transition?
5. What surprised you?

✅ Part B Checklist

- ☐ Wrote hypothesis before experimenting
- ☐ Generated datasets of increasing size
- ☐ Collected metrics for all N values

- [ ] Calculated IPC for each N
- [ ] Identified the transition point
- [ ] Wrote 1-2 page explanation

Part C: Your Own Workload (Advanced — Optional)

Goal: Apply what you learned to a different program. This is for students who want a deeper challenge.

Choose ONE of the following options:

Option 1: Profile a Standard Tool

Pick one:

- `sort` (GNU coreutils)
- `grep` (GNU grep)
- `wc` (word count)
- `gzip` / `gunzip`

Profile it on a large file:

```
# Create a test file
yes "the quick brown fox jumps over the lazy dog" | head -1000000 > testfile.txt

# Profile
sudo perf stat sort testfile.txt > /dev/null
sudo perf stat grep "fox" testfile.txt > /dev/null
sudo perf stat wc testfile.txt
```

Answer:

1. Is this tool CPU-bound or memory/IO-bound?
2. What is the IPC?
3. What hardware counter best explains its performance characteristics?

Option 2: Profile Your Own Code

If you have a project, profile one function or operation:

1. Isolate the operation into a benchmark
2. Run `perf stat` and `perf record`
3. Identify the bottleneck

Option 3: Cache-Friendly vs Cache-Unfriendly

Write two programs that do the same computation but have different cache behavior:

Program A: Row-major traversal (cache-friendly)

```
int sum = 0;
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        sum += matrix[i][j];
```

Program B: Column-major traversal (cache-unfriendly)

```c
int sum = 0;
for (int j = 0; j < N; j++)
    for (int i = 0; i < N; i++)
        sum += matrix[i][j];
```

Measure cache miss rates for both. At what N does the difference become significant?

Deliverable for Part C

Write 1 page explaining:

1. What you profiled
2. What metrics you collected
3. What you found (bottleneck identification)
4. What mechanism explains it (connect to cache, branches, syscalls, etc.)

Submission

Submit a single PDF containing:

1. **Part A Results** (1 page max)
    - Data table
    - Brief explanation of worst-case behavior
2. **Part B Analysis** (1-2 pages)
    - Hypothesis
    - Experiment design
    - Results
    - Interpretation
3. **Part C Extension** (1 page, if completed)
    - What you profiled
    - Findings
    - Mechanism explanation

Use the template `lab1_report_template.md` as a starting point.

Due: Before Week 3 lecture

Grading Rubric

Criterion	Points
Part A (30 points)	
Data collected correctly (multiple runs)	15
Correct explanation of worst case	15

Part B (60 points)	
Clear hypothesis stated (before experiments)	10
Sound experiment design	15
Data collected and presented clearly	15
Interpretation connects to OS/hardware mechanism	20
Part C (10 bonus points)	
Meaningful analysis of new workload	10

Total: 90 points (+ 10 bonus)

Parts A and B are required (90 points possible). Part C adds up to 10 bonus points.

Common Pitfalls

1. **Not running multiple times** — Single runs have high variance. Always run 3+ times.
2. **Forgetting to disable frequency scaling** — CPU may throttle. For accurate measurements:

   ```
   sudo cpupower frequency-set --governor performance
   ```

   (If cpupower is not available, that's OK for this lab.)
3. **Measuring cold cache** — First run may be slower (code not in cache). Warm up with a dummy run first.
4. **Confusing cache-miss count with cache-miss rate** — A larger dataset naturally has more memory accesses. Look at the RATE (misses / references).
5. **Not explaining the mechanism** — "It's slow because cache misses" is not enough. WHY are there cache misses? (Working set > cache size? Bad access pattern?)

Troubleshooting

"perf not permitted"

```
sudo perf stat ...
# or set kernel parameter:
sudo sysctl kernel.perf_event_paranoid=-1
```

"No symbols found" Make sure you compile with `-g`:

```
make CFLAGS="-g -O2"
```

Results are inconsistent

- Close other applications

- Disable turbo boost if possible
- Run more iterations

Reference: Useful perf Commands

```
# Basic stats
sudo perf stat ./program

# Specific events
sudo perf stat -e cycles,instructions,cache-misses,branch-misses ./program

# Sample where time is spent
sudo perf record -g ./program
sudo perf report

# List available events
perf list

# Detailed cache events
sudo perf stat -e L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses
./program
```

Getting Help

- **During lab workshop:** Ask the instructor or TA
- **Outside of class:** Post on course forum, come to office hours
- **Debugging tips:** Start simple, verify each step works before moving on

Don't struggle alone! Performance analysis can be tricky. Ask for help early.