Cloud Systems (H/M) – Lab 1 – Benchmarking Practice

Last Changed: 11/01/2025

The primary goal of this lab is to gain experience in **system performance benchmarking**. With this, the lab builds towards the first assessed exercise.

We recommend using your own machine for the labs and exercises of CS(H/M), as you need privileged access for much of the practice.

We also recommend working in small groups (of 2-4 students) to pool compute resources (e.g. access to different machines and free cloud credits) and cover more ground (e.g. in this task, split up looking at different benchmarking tools). The two assessed exercises are also designed for teamwork (though you can work on your own if you prefer). Ideally, you form teams early on and work with the same colleagues throughout the course.

In the next labs and the first assessed exercise, we will measure the performance of virtual resources (virtual machines and containers). To prepare you for this, **this lab highlights important considerations around (micro)-benchmarking** and specific Linux tools for you to try yourself.

0. Prerequisites

You should use a Unix-like platform. **Linux is recommended**. If you use Windows, we suggest you use Linux in a virtual machine (e.g. Windows Subsystem for Linux or using Virtual Box).

1. Benchmarking Basics

Measuring the ***performance*** of a computer system is often not trivial.

Performance is usually given as a *load* that a computer system can handle (using a certain set of computing resources). So, what is load then? A load equals performing certain work, described by the work's properties (e.g. doing a particular calculation, serving a certain number of requests per second, or carrying out a specific mix of disk reads and writes).

Such **a load can be captured as a microbenchmark**, which is a program with the sole purpose of providing load for a measurement, specifically to understand the performance of a computer system. Typically, microbenchmarks measure the performance of individual operations or resources, in contrast to (end-to-end) application benchmarks, which are used to measure the performance of whole systems. Widely used microbenchmarks target, for instance, how fast arithmetic calculations are done by a CPU, how fast memory can be accessed, how fast disk writes and reads can be performed, or how much network bandwidth is available.

Performance measurements of real computer systems are often subject to some variance. This is the case, for example, because of **caching**. Caches need some time to warm up and performance might be drastically different once caches have warmed up. Similarly, there might be **interference** with other workloads, such as other user applications running at the same time or periodic operating system processes or language runtime functions. Consider, for example, measuring the performance of a Web application running on a language runtime (e.g. the JVM) that supports garbage collection, so programmers do not have to manually allocate and free memory. Many garbage collectors must “Stop the World” to perform their function, which might happen at different times, potentially introducing variance to application performance.

Therefore, any **load measurements are usually repeated** to report a mean, median, or x-percentile. Using a median or percentile is useful to exclude outliers. It is, for example, common practice to repeat computer systems experiments 3, 5, or 7 times and then report metrics for a median run (e.g. the run with a median runtime).
Similarly, it makes sense to **consider longer measurements** (e.g. increasing the load, so a benchmark runs minutes or hours) and to focus on a specific part of an execution (e.g. deliberately excluding a warm-up time from measurements).

At the same time, the prevalence of variance might be important, so you could **give a measure of variance**. This could simply be the standard deviation, next to the mean and median. It could also involve percentile. A widely used way to show data distributions around a median are box plots, which present the median, minimum, maximum, and the first and third quartile: https://en.wikipedia.org/wiki/Box_plot.

It is not always wise to just discard outliers, as they might be highly relevant for your system’s performance. Consider a system that is used so seldomly that caches are always cold or perhaps that the system must typically be (re-)started first. In that case, if you repeat a measurement a few times without time for the system to cool down in between, measurements might not be representative of actual performance. That is, it might sometimes be sensible to specifically **investigate outliers**.

Performance variance is especially relevant for cloud systems, as these are deployed onto shared infrastructure, co-located to applications of other users, as well as onto complex technology stacks with many layers (such as physical hardware, hypervisors, the operating system of a guest VM, a container runtime, and a cloud application platform). Hence, there are many potential sources of interference and warm-up effects. Furthermore, real cloud computing loads are commonly dynamic, with load stemming from user interactions.

An important aspect of good benchmarking practice is ***reproducibility***. Reproducibility refers to the ability to reproduce the results (within a certain error margin) by repeating an experiment, using the same experimental setup, test conditions, and measurement methodology. This often involves running the same benchmark code and configuration, using the same test inputs, on resources with the same specifications, and using the same monitoring tools used originally. This is typically supported by taking note of all relevant conditions and configurations and, as much as possible, preserving experiments in the form of executable benchmark scripts.

2. Benchmarking Tools

This section outlines three widely used open-source (micro-)benchmarking tools: **Sysbench** for general system performance, **fio** for storage, and **iPerf** for networking, as well as a basic C program **forksum.c** (to show that you can write your own microbenchmarks). You can find links to these tools below that provide guidance on installing them – except for forksum.c, which you can find on Moodle.

For each tool, there is a small practice task to complete (a specific command) after installation. You should run the commands but also investigate what the tool does.

It might be sensible to aim to examine only one or two of these tools as part of this lab or to form a team of up to four students to try all of them.

2.1 Sysbench

Sysbench, https://github.com/akopytov/sysbench, is a (micro-)benchmarking suite for measuring the performance of CPUs, memory, and I/O on Linux systems. In that, it is a versatile and general tool to benchmark the performance of a computer system. Sysbench is especially popular for assessing CPU performance.

Practice task: Run `sysbench cpu --cpu-max-prime=500 --threads=1 run` and observe the CPU speed (given as events per second). The benchmark identifies prime numbers (testing numbers up until a specified number) as much as possible within 10 seconds (i.e. starting again to look at the number 3 when the maximal number has been tested). What happens if you vary the number of threads?

2.2 fio

fio (flexible I/O tester), https://github.com/axboe/fio, is a tool for testing and benchmarking disk I/O performance. It supports sequential and random reads/writes, different test files and file sizes, and other possibilities for configuration (e.g. thread count). It can measure, for instance, read/write speeds, IOPS (Input/Output Operations Per Second), and latency. fio is popular for evaluating storage devices and filesystem performance.

Practice task: Run `fio --name=fiotest --size 1Gb --rw=read --runtime=15 --startdelay=15 --direct=1 --bs=1M --iodepth=16`. Try to understand what each parameter does. What does `--direct=1` do? How does removing it impact performance?

2.3 iPerf/iPerf3

iPerf/iPerf3, https://iperf.fr/, is a tool for measuring network performance, primarily focusing on bandwidth. It can be used to test TCP and UDP throughput between two endpoints – typically a server and a client. The tool is widely used for network testing and troubleshooting.

Practice task: Run `iperf3 -s` in one terminal window and `iperf3 -c localhost -u` in another one on the same host. This sends data over UDP, but the datagrams are sent and received by the same computer. Can you run the benchmark over two different computers?

2.4 forksum.c

The **`forksum.c`** program, as available on Moodle, performs a parallel computation, summing up an integer range. The C program receives a start and end value of an integer range and computes the sum of all numbers in that range. For this, the range is recursively split into two halves, and two child processes are forked to compute the respective subproblem, until the recursion finally ends when the start and end of the range are the same. The program outputs the computed sum and the number of forks. This tests not only the performance of arithmetic operations but also process spawning and function calls.

Practice task: Compile forksum.c with a C compiler (such as Clang) and then run, for example, `./forksum 1 500`. You can use the Linux tool **time** to measure the runtime (`time ./forksum 1 500`). This will also show you a time that was spent in system mode ("sys"), as opposed to time spent in user mode ("user"), with the operating system being called upon to fork processes. You might notice that system time significantly outweighs user time and even goes beyond the real (wall clock) time due to parallelism. What happens if you change the range? Note, though, that there are limits for forking processes, so forking as a service of the operating system might become unavailable.

3. Open-Ended Task Towards AE1: Performance Benchmarking

Benchmark the following properties of one or more computer systems, taking into account the considerations for performance benchmarking discussed in Section 1 and using the benchmarking tools presented in Section 2:

- CPU speed,
- memory access,
- random-access and sequential disk read speed,
- network upload speed,
- and how fast new processes can be spawned

Write a **script** to make your benchmarking easily **configurable** and repeatable/**reproducible**.

We recommend using Python for all scripting in this course, but you are free to use any other programming language, including Bash.
Using Python is supported by a set of answers to Frequently Asked Questions (FAQs) on installing, running and debugging, and programming (object-orientated) Python. These FAQs stem from NOSE2 and were prepared by a former NOSE2 tutor as part of a University-funded summer project.

Repeat your measurements, take note of the variance, and vary experimental conditions. Can you observe any impact of keeping background load to a minimum (or deliberately introducing interference through simultaneous loads)?