CS60203: Design Optimization of Computing Systems *August 7, 2024*

Introduction to Systems Performance and Measurement

Instructor: Mainack Mondal *Scribe-By: Teerath Agarwal*

1 Summary

We need to build and maintain reliable systems with high performance when we talk about computing systems. One might think that it is not that necessary, but then we observe instances where such high performance systems become very critical for the survival of a big institution or a company. Or the counter case where absence of such a high performance system leads to miserable failures. Examples of both cases are widely available.

The metrics we need to improve to build a high performance system are reliability, throughput, scalability and latency. The first step to improve our system is to know where exactly it is facing a problem or reaching the bottleneck. For that we do instrumentation and measurement. Instrumentation means adding code or information in our system beforehand so that later we can take help of it to figure out the problem in our system. Measurement is reading and making some sense out of the data provided by instrumentation techniques.

2 Performance

When working with computing systems, one may prefer a different specification optimization over the other, and this may vary depending upon the use case. For example, a hospital chain would require a reliable, high security system to store patients' records, whereas an online gaming server would stress upon heavy graphical computation. The needs vary, but both of them require a high performance system. So, to generalize, we say, when a system behaves in a desirable manner, across all intended circumstances, and usage patterns, that system has a high performance. Improving a system's performance boils down to one or more of the following:

- Scaling systems to handle variable loads
- Reducing system latency
- Increasing system throughput
- Reliability, durability, and availability

3 Terminology and metrics

3.1 Basic Metrics

The basic metrics quantify the values we try to improve for high performance. These include:

- Throughput: Operations or data volume per unit time

- Scalability: A profile/spec of how the system behaves under varying load
- Latency: Operation time, as an average or percentile

3.2 Utilization

Utilization is the measure of how busy a resource is. Knowing the percentage utilization of different resources and IO devices can help us to know the bottleneck in our system. For example, a computing resource reaching 100% utilization is most likely the reason why the system is slow. However we must be careful while using it as a metric. It is easily available and easy to read, but it might be misleading at times. For example, in a scenario where CPU utilization is unusually high and GPU utilization is extremely low while executing a graphic intensive task, might lead us to increase the CPU's processing power, but in reality, it indicates that the GPU is not working properly. Utilization is broadly of two types:

3.2.1 Time Based

Utilization over a time period T is defined as B/T, where B is the time period for which the resource was busy. Time based utilization is extremely misleading. Some resources, for instance, disks, at any particular time may be in use but will have more capacity to read/write more data. In such a case, time based utilization will depict a 100% utilization, whereas it would not be the bottleneck. Another reason it is misleading is that a resource might be waiting on some other resources to get data for further usage. For example, a CPU waits on IO devices for the data. In this case as well, the utilization depicted will be unnaturally high.

3.2.2 Capacity Based

A system or component (such as a disk drive) is able to deliver a certain amount of throughput. At any level of performance, the system or component is working at some proportion of its capacity. That proportion is called the utilization. Capacity based utilization is a much more sensible form of utilization. Most of the software tools for this purpose show capacity based utilization.

3.3 Saturation

When a resource truly reaches its limits and needs to be upgraded to work more in the same timeframe, then we say that it has reached saturation. By definition, the degree to which more work is requested of a resource than it can process is saturation. From the understanding of saturation and utilization, it is easy to guess the relation, that saturation begins to occur at 100% capacity-based utilization, as extra work cannot be processed and begins to queue and cause undesired delay. Simply put, when a resource reaches saturation, it causes a bottleneck. It is valuable to understand the behavior of a system under saturation.

3.4 Queuing System

Most resources put the tasks they are assigned in queue. They are modeled as a queuing system. This makes it easier to predict their performance in various situations. For example, disks.

4 Instrumentation

Instrumentation refers to the tools, methods, and processes used to monitor and control a system. It involves the incorporation of various sensors, probes, and software tools designed to gather data about the system's behavior. To be specific, we prepare beforehand, put up checkpoints or use tools to navigate through and find the cause of the failure of an entity. The goal of instrumentation is to enable detailed monitoring and control of a system, for debugging, performance analysis, and optimization. When instrumentation is done during compile time, it is called static instrumentation, whereas when done during runtime, it's called dynamic instrumentation.

4.1 Static Instrumentation

Static instrumentation involves modifying the code of a program or system before it is executed. This modification is done at build time, and it typically involves inserting additional code or hooks that are used for monitoring purposes. Some examples of static instrumentation are logging and kernel tracepoints.

4.2 Dynamic Instrumentation

It is a technique to analyze and modify a running program's behavior at runtime. This is achieved by injecting additional code into a program without altering its original source code or requiring a recompilation. Injecting additional code during runtime is made possible by executing the code in a virtual environment. Once the machine code is compiled it cannot be changed, unless it is executed in a virtual environment. Using Dynamic instrumentation helps to identify bugs and failure much more efficiently. Some examples of tools that facilitate dynamic instrumentation are GDB, Valgrind, DynamoRIO, Dyninst, SystemTap.

5 Measurement

Measurement refers to the process of quantifying specific attributes or behaviors of a system. This involves collecting data that represents various performance metrics, resource usage, and operational parameters. It is the act of obtaining numerical values that describe how a system operates. We use this numerical data for our analysis to find the cause of failure.

5.1 Observability

Observability refers to understanding a system through observation, without changing the state of the system. This includes tools that use counters, profiling, and tracing. It does not include benchmark tools, which modify the state of the system by performing a workload experiment. Observability is on three levels, which each of them being deeper and more complex than the previous one:

- User Space: involves monitoring and analyzing applications and processes running in the user-mode, capturing logs, metrics, and traces for performance and debugging.
- Kernel Space: focuses on tracking and analyzing system-level operations and behaviors within the kernel, such as system calls, interrupts, and resource management.

- Hardware: involves monitoring and measuring hardware performance and health indicators, such as CPU temperature, power consumption, and hardware interrupts.

5.1.1 Counters

Counters are variables or registers that keep track of the number of occurrences of specific events or the count of particular items over time. For example, there are counters that keep a track of the number of page faults, the number of CPU migrations, etc. Similarly there are Performance Monitoring Counters (PMCs) that provide statistics about resource utilization.

5.1.2 Profiling

Profiling is the process of analyzing a program's runtime behavior to identify performance bottlenecks, resource usage, and execution patterns. It refers to the use of tools that perform sampling of the target which is a resource or a software. For example, if CPU is the target resource, the commonly used method to profile it involves taking timed-interval samples of the on-CPU code paths. Note that profilers often use 99 Hz sampling frequency instead of 100 Hz to avoid sampling in lockstep with target activity. Perf, for example, is a well known profiler.