Measuring Operating Systems Performance

- What do we mean by performance?
- Definitions of performance depend on what our own definition is
    - could be that we don't want to wait for commands to complete
    - could mean that we want to improve how fast we get results

Metrics

- If we care about system performance, we must quantify it
- Latency is likely to be expressed in units of time
- Throughput is likely to be expressed in some unit of work that is relevant to your system, divided by the unit of time. (Operations/sec, example)
- **Numbers used to characterize the performance of our system are called metrics**
- Metrics must be practically measurable by you
    - If you have a proprietary OS whose source code you cannot see, you will have a hard time measuring what's happening inside it
    - If you are measuring the performance of a web server that you do not run yourself, you probably can't run any code at all on that server
- We are often using the system whose performance we wish to measure to actually capture our results
    - If the process of performing the measurements itself has a large effect on performance, we may have obtained false readings for our metrics
    - ex: a measurement system that regularly writes records concerning file system behavior to the disk drive that stores that file system
    - experimental logging could be competing with the behavior you are actually trying to measure

Complexity and the Role of Statistics in Measurement

- Don't measure the event of interest on your system only once. Measure it many times and treat the set of measurements as a probability distribution
- Among the simplest tools are the simple mean, median, and mode
- Mean
    - getting a single number that somehow captures something important about the entire data set
- Median
    - Useful for getting a sense of where the measurements in a data set are "centered"
- Mode
    - most useful for seeing what value occurs the most
- How many experimental runs do you need to perform to fully capture the behavior of a system you are studying?
    - the greater the degree of variability in what you're measuring, the more independent measurements you'll need to perform to get a pretty confident picture of its full behavior

Comparing Alternatives

- Sometimes the purpose of your performance experiment is simply to characterize how a system performs according to one or more metrics
    - In other cases, the system can be built, configured, or used in several different ways, and you want to know how well it performs in those varying situations
- One issue is that possibly there are a large number of different options you could compare
    - ex: you might want to know what would happen if you increased or decreased the amount of RAM allocated to buffer spaces, or what would happen if you used several different scheduling disciplines
- Things you intentionally alter in performance experiments to determine which of several alternatives to use are called factors
    - Amount of buffer space allocated, the scheduling discipline, etc.
- One of the dangers of running performance experiments is getting too entranced with the many possibilities
    - You can ignore the things that you may consider not as important
- You can also control the number of runs of each alternative you make to capture statistical variation against the number of alternatives you look at
- Before you undertake a set of measurements, think first, measure second
- Treat each alternative you measure fairly -> don't create conditions that artificially favor one condition over another
    - ex: if you run process A, and then process B, B might be faster simply because of caching
- Make sure to use the same settings on all runs

Sources of Performance Problems

- ex: Overload resource (memory bandwidth or CPU cycles), solution built into software doesn't scale, etc
- If you run performance measurements to uncover a scaling issue, a test that only runs a small number of iterations or on a small version of the problem may not produce useful results
- You might know performance is bad, but to run an experiment to determine exactly why, you need to know why performance is bad
- You can often get clues without running any new experiments. The OS will tell you, on request, how much memory is being used, CPU utilization, and many other statistics
    - If there's plenty of free memory on the system but is still running poorly -> waste of time to run experiments

Workloads

- There are different aspects of workloads that you need to think about when designing performance experiments
- Traces

        - Take or otherwise obtain a detailed trace of the workload of the system in its ordinary activities
        - ex: for a web server, the trace is likely to be a set of web requests submitted to that server
        - Whatever, the trace may be of, you capture it from the running system and save it in a form that will allow you to recreate it in a faithful manner
        - Advantages
            - provides realism and reproducibility -> same trace be replayed over and over
        - Disadvantages
            - Not easily configurable
                - if your experiment needs to examine performance under controlled levels of workload, you might not be able to get a trace for each workload level you need
                - Merely running two copies of one trace in parallel might not realistically represent a true doubled load
            - Might be difficult to gather the information necessary to create the trace
- Live Workloads
    - You can perform measurements on a working system as it goes about its normal activities.
    - Data can be gathered as the system does work
    - The main **advantage** here is that **there is realism**
    - Disadvantages
        - Lack of control, which manifests itself both in not being able to reproduce the behavior seen in previous tests, and in not being able to scale loads up and down as desired
        - Your experimental framework needs to have minimal impact, both in performance and functionality
            - it is more important to complete the live work rather than the experimental tests
            - unless the impact is essentially nil, you are not likely to be able to run the performance measurements for very long on a working system
        - Consider privacy implications to you observation of the live workload
- Standard Benchmarks
    - Sets of programs or sets of data that are intended to drive performance experiments
    - May have been derived from real traces at some point
    - Advantages
        - Allow for easy comparison to other systems' performances, since the developers of those systems can also run the same benchmark
        - No privacy implications since they are artificial

        - Some benchmarks are built to be inherently scalable, allowing you to increase/decrease the workload
    - Disadvantages
        - Limited number of them available and there may not be one that is suited for a particular system/situation
        - Developing a good benchmark is quite a lot of work
- Simulated Workloads
    - Build models of the loads you are interested in, typically models instantiated in executable code
    - These models are parametrized, allowing for them to be scaled up or down, to alter the mix of different elements of the load
    - Advantage
        - easily customizable to different scenarios and possibilities -> flexibility
    - Disadvantages
        - Validity of the performance results you achieve is only as good as the quality of the models. -> parameters may be easily altered and scaled -> could lead to errors

Common Mistakes in Performance Measurements

- 1) Measuring latency without considering utilization. Everything runs fast on a lightly loaded system. Most often, one should examine the latency when the system is heavily loaded
- 2) Not reporting the variability of measurements: We must understand the distribution of the model rather than a single value such as a mean or median
- 3) Ignoring important special cases. You ignore the fact that a few special cases distort the measurement, given you a false sense of what happens in the general case
    - Common: Ignoring startup effects. Computers make use of caching -> programs loaded off disk may hang around memory for a while in case they will be run again
    - Translations of DNS names to IP addresses are stored to avoid having to make expensive network requests
    - TLB
    - Performance may degrade as space if filled up -> ex: a hash table that uses chaining to handle collisions
- 4) Ignoring the costs of your measurement program. You will most likely use your own system to measure your system. If you are running a separate process to perform your measurements, it is competing for CPU and memory with the processes you are trying to measure
    - Often impossible to completely ignore this, but minimizing impact is possible
        - Instead of writing each observed measurement to disk, save it in a RAM buffer -> write out the buffer at the end of experiment or infrequently write to disk if isn't too big

        - Make sure to keep your measurement code small and cheap -> if feasible, bundle it into the process you are measuring
- 5) Losing your data: Never throw away your experiment data, even if you think that you are finished with your experiment.
    - Remember to label your data
    - Important for looking back at historical purpose
- 6) Valuing numbers over wisdom: Point of performance is not to obtain numbers. It is to understand the performance characteristics of your system. Don't bother gathering numbers that are not going to lead to wisdom, and don't consider your task complete when you have numbers in hand

Load and Stress Testing

Introduction

- For most products, the vast majority of all tests exercise positive functional assertions (if situation x, program will y) -> positive (functionality) or negative (error handling) behavior
- A typical suite is a collection of tests cases, each of which have the general form:
    - establish conditions (c1, c2, …. cn)
    - perform operations (o1, o2, … on)
    - ascertain that assertions (a1, a2, … an) are satisfied
- Create test cases that
    - adequately exercises the program's capabilities
    - adequately captures and verifies the program's behavior
    - is small enough to be practically implementable
- Repeatability of tests, therefore become very important -> if all operations work and past the test, then it is okay!
- However, load and stress testing are quite different
    - the number of enumerated test cases is relatively small and the particulars of each test case may be particularly important
    - run test cases in pseudo-random order for unspecified periods of time
    - we have no expectation that results will be repeatable
    - No definitive pass indication. Rather, we can only say that the test has run for a period of time
    - We may not take the trouble to define a complete set of assertions to determine the correctness with which any particular operation has been performed. In some cases, we may even look at returned results

Load Testing

- The initial and primary purpose of load testing is to measure the ability of a system to provide service at a specified load level. Collected data includes things like
    - response time for each request
    - aggregate throughput
    - CPU time and utilization

    - disk I/O operations and utilization
    - network packets and utilization
- This information can be used to
    - measure the system's speed and capacity
    - analyze bottlenecks to enable improvements
- Key ingredient is a tool to generate the test traffic -> **load generators**

2.1: Load Generation

- Load generator is a system that can generate test traffic, corresponding to a specified profile, at a calibrated rate
- This traffic will be used to drive the system that is being measured. Such testing is normally performed on a "whole system"
    - if the system to be tested is a network server, the load generator will pretend to be multiple clients, sending requests
    - If the system to be tested is an application server, the load generator will create multiple test tasks to be run
    - if the system to be tested is an I/O device, the load generator will generate I/O requests
- Test load is broadly characterized in terms of
    - request rate
        - num of ops/sec
    - request mix
        - different types of clients use a system in different ways
        - If different types of requests exercise very different code paths, it is important that the load generator be able to accurately emulate all types of clients
    - sequence fidelity
        - in many situations, it is sufficient to merely generate the right mix of read and write operations. In other cases, it may be critical to simulate particular access patterns
- A good load generator will be tunable in terms of
    - both the overall request rate and the mix of operations that is generated
    - some load generators may merely generate random requests according to a distribution

2.2: Performance Measurement

- Few typical ways to use a load generator for performance assessment
    - 1) deliver requests at a specified rate and measure the response time
    - 2) deliver requests at increasing rates until a maximum throughput is reached
    - 3) deliver requests at a rate, and us this as a calibrated background for measuring the performance of other system services
    - 4) deliver requests at a rate, and use this as a test load for detailed studies performance bottlenecks

2.3: Accelerated Aging

- Load generators can be used to create realistic traffic to simulate normal traffic for long periods of time -> can also crank up to much higher rates in order to simulate aging

3: Stress Testing

- Load generators are fundamentally intended to generate request sequences that are representative of what the measured system will experience in the field
- Random stress testing is used to test residual errors that tend to involve combinations of unlikely circumstances
    - use randomly generated complex usage scenarios, to increase the likelihood of encountering unlikely combinations of operations
    - deliberately generate large numbers of conflicting requests -> e.g. multiple clients trying to update the same file
    - introduce a wide range of randomly generated errors and simulated resource exhaustions so that the system is continually experiencing it
    - Introduce wide swings in load, and regular overload situations
- Gives us serious confidence about the robustness and stability of our systems