Measuring Operating Systems Performance

No matter how fast our hardware gets, the performance of our system always matters. Programmers tend to add complexity and sophistication to their systems to match any hardware performance improvements, so there is always a need to design software that achieves good performance on whatever hardware is available to us.

This instantly raises a question: what do we mean by performance? Most of us have an informal sense of what we mean. Perhaps we mean we don't want to wait for our commands to complete. Perhaps we mean we want to run complex calculations on very large data sets fast enough for the results to be useful. Perhaps we mean we want to get the most possible work through a given piece of hardware that we possibly can. Perhaps we mean we want to use as little space as possible on a storage device, or send as few bits across a network as possible. Maybe we care about how long a user has to wait before he starts to see some response from the system, but maybe we care about how long it takes for the entire job to complete.

As these potential answers to the performance question suggest, there's a lot more to understanding performance in a computer system than one might initially think. And there are yet more complexities in actually providing a valid answer to a particular performance question, such as "how many web requests per second can my server handle," or "will my VoIP call provide comprehensible speech to the listener if run it over a particular network," or "how many buffers should I allocate in my operating system to make sure that I/O is not delayed too much?"

Metrics

Perhaps the first and most important step in determining the performance of your system is achieving clarity about what you really care about. If you don't know what you want to measure, you can be quite sure you won't successfully measure it. (If only mere clarity were enough to ensure that; but it is a vital first step.) What makes a difference in your system? What must go fast, and what is less important to speed up? In a word, what are your goals?

As a rule, if we care about system performance, we must quantify it. Saying your system is "fast" doesn't mean much. Saying it can perform a complex operation in 10 nanoseconds makes it a lot clearer what's going on. So any good performance investigation is targeted towards reducing the observed system behavior to some set of characteristic numbers. These numbers must have useful meanings relative to your goals. So latency is likely to be expressed in some unit of time, and throughput is likely to be expressed in some unit of work that is relevant to your system divided by some unit of time. The numbers we choose to characterize the performance of our systems are called *metrics*, and obviously their proper choice is of critical importance. Just because some quantity is measurable, however, does not make it a good metric. For instance, you can put your smart phone on a scale and measure its weight. That's a metric, but it won't tell you much about whether you can render video at a high enough frame rate to be

tolerable. On the other hand, it might be relevant in a usability study. The metrics you choose need to be relevant to your goals.

One more important point about metrics is that they must be practically measurable by you. That's limited by your ability to probe hardware and alter software. If you have a proprietary operating system whose source code you cannot see, you will have a hard time measuring what's happening inside it. You will probably need to be satisfied with observing what happens as you enter and leave the operating system, perhaps augmented by whatever information the system itself will divulge to you on request. Similarly, 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, and you must choose metrics observable from places that you can reach. On the other hand, if you are measuring software running on top of an operating system, you might have more liberty to alter it for measurement purpose. Or you might not.

Another important point is that 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 the performance, we may have obtained false readings for our metrics. Consider, for example, a measurement system that regularly writes records concerning file system behavior to the disk drive that stores that file system. Chances are that this experimental logging is competing, in a performance sense, with the actual behavior you are trying to measure. Instead of simply observing how long it would take to perform a set of reads and writes from a group of files in the file system, we are also moving the head to another place on the disk where we are storing our log. Those head movements would make the file system appear to be slower than it actually would be if your experimental framework were not logging data. If your experimental framework interferes with the processes you are trying to measure, you end up with a false reading for your metric that will not accurately describe the system's behavior when you aren't running your experiment.

Complexity and the Role of Statistics in Measurement

Given that you've determined a good set of metrics, what next? By this point in your studies, you should be painfully aware of the complexities of large systems, particularly of operating systems. That complexity is going to have a big impact on your performance measurement studies. Consider what you already have learned about operating systems. They make heavy use of caching, with an expectation that some operations will be cache hits (probably fast) and some will be cache misses (probably slow). Interrupts will occur at unpredictable times, possibly resulting in altered performance of the code they break up. Different scheduling disciplines will insert varying delays into the performance of individual processes. Even if you have access to the operating system code, down at the hardware level there are caches and pipelines and other optimizations that you can't see at all, except that they produce varying performance results.

All of these characteristics mean that if you measure some metric on your system once, and then repeat that measurement again, you might see very different values. So what's

the right value? The one that makes your system look best? The worst one you've ever seen? Something else?

If you have taken a course in statistics, you may have a pretty good idea already about how to handle this issue. 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. You can then use the rich set of tools that this field of mathematics offers us to analyze your performance. Among the simplest of these tools are *mean*, *median*, and *mode*. The mean of a set of measurements is its average, the median is its middle point, and the mode is the most common value in the set. Means are useful for getting a single number that somehow captures something important about the entire data set. Medians are useful for getting a sense of where the measurements in a data set are "centered," in some sense. Mode is typically most useful when one value occurs far more often than any other, since then it can give you an idea of what result is most probable from a given data set. Mode tends to be less useful when the measurements are pretty evenly spread out.

It's often helpful to take a step further and work with quantities called indices of dispersion, which essentially describe how spread out a set of measurements are. *Range* is one such index of dispersion, describing the highest and lowest value observed. *Standard deviation* is another, describing the most commonly occurring range of values around the mean within the set. *Confidence intervals* describe the probability that a particular measurement is within a certain range.

In figure 1, we show a small set of sample data of latencies (in milliseconds) to access a disk block. Figure 2 shows various statistical properties of that data set. We calculated the mean by adding the 11 latencies and dividing by 11. We calculated the median by ordering the measurements and choosing the one in the middle. We calculated the mode by counting the number of times particular latencies occurred and choosing the one that occurred most often. To determine the range, we simply found the lowest measurement and the highest measurement. Standard deviation is calculated, typically, with a somewhat more complicated formula, but it is described in any detailed treatment of statistical properties.

Trial	Latency
1	27
2	31
3	28
4	26
5	30
6	35
7	31
8	29
9	32
10	25
11	33

Figure 1. Sample disk drive latencies

Mean	29.7
Median	30
Mode	31
Range	25-35
Standard deviation	3.07

Figure 2. Some statistical properties of the data in figure 1

These statistical properties are actually a bit more complex than they appear at first glance, and full understanding of their proper use is beyond the scope of either this chapter or an introductory course on operating systems. For example, there are actually several different ways to calculate means, and calculation of confidence intervals is typically based on assumptions about the probability distribution of the measurements. We will not further describe the many useful tools that the field of statistics offers, beyond recommending that those interested in understanding the performance of their systems really need good mastery of some of these tools.

You might wonder how many experimental runs you need to perform to fully capture the behavior of a system you're studying, given that there is possible statistical variation in the performance of each run. The field of statistics has useful tools for this purpose, as well, but they are beyond the scope of this chapter. In brief, 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 important 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. When this kind of comparative form of performance measurement is what you need, you have to take some care in how you go about doing it.

One issue is that possibly there are a large number of different options you could compare. There may be multiple dimensions in which you could examine the system. For example, 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, or what would happen if you replaced the Ext3 file system with Btrfs. You might want to know what would happen if you made several of these changes in different ways.

Things you intentionally alter in performance experiments to determine which of several alternatives to use are called *factors*. In the paragraph above, the amount of buffer space allocated might be one factor, the scheduling disciplined used might be a second factor, and the file system you chose might be a third factor. Factors can be set independently. For example, I might want to look at allocating 1Gbyte of buffer space, using Linux's Completely Fair Scheduling, and Btrfs, vs. allocating .5 Gbytes of buffer space, simple round robin, and Ext3. For each choice of settings of factors you want to investigate, you'll need to run some experiments. As mentioned earlier, you'll probably need to perform multiple runs to capture statistical variations.

Think about this a moment. If there are four different buffer sizes I want to investigate, three different scheduling disciplines, and two file systems, and I care about all possible combinations, how many sets of experiments am I going to have to run? In this relatively limited case, we're up to 24 sets of experiments, each of which will need to be run multiple times. One of the dangers of running performance experiments is getting too entranced with the many possibilities. If you're not careful, you might spend the rest of your life investigating some of these issues. Or, more likely, you'll get exhausted and give up before you get around to looking at the situations that are really important. A key element in successfully obtaining good performance results is striking a balance between investigating everything that's very important versus avoiding a combinatorial explosion of experimental settings. Maybe, on careful consideration, looking at two buffer sizes and ignoring one of the scheduling disciplines will still tell you what you need to know.

Another element controlling how much work you need to do is how many runs of each alternative you make to capture statistical variation against the number of alternatives you look at. Generally, the more you make, the higher your confidence in the statistical representativeness of your results (though there is a point of diminishing returns). Is it more important to consider more options or to have greater confidence in the performance of the options you do consider? That's for you to determine.

One can further generalize this issue to obtain perhaps the most important piece of advice we can give you before you undertake a set of performance measurements: **think first, measure second**. Determine exactly what you want to know and what you need to do to learn it, then design and perform experiments targeted at that required knowledge. Otherwise, you can spend arbitrarily large amounts of times hacking your way through the performance measurement jungle, possibly emerging at the end without having learned anything of importance.

There are other important issues in comparing the performance of various alternatives. One is to treat each alternative you measure fairly. Don't create experimental conditions that artificially favor one alternative over another. For example, in some cases things go slower the first time you do them than the second or later times. Caching often causes this effect. So if you ran alternative A first, then ran alternative B, B might appear to be faster not because it's better, but because it benefited from caching. Merely switching the order isn't enough, because now you've favored A. Running each alternative multiple times will help wash out these effects, particularly if you randomly intersperse the alternatives in different runs. (That's not always easy, if you need to perform heavyweight reconfiguration of the system to enable different alternatives.)

That is merely one example of being fair. Using the same settings for all runs (other than those that define the alternatives themselves) is another example. Resetting the system to the same state before starting each run is a third example. Even if caching is not involved, there may be some initial work that each system might need to do. If the system is not reset, the later runs can unfairly benefit from the configuration work done by earlier runs. Isolating the system from unrelated effects (such as updates to key pieces of software, external work loads not intended to be captured by your experiment, filling up the disk drive with your own experimental results, and so on) is also important.

Sources of Performance Problems

We can experience poor performance in our systems for many reasons. Sometimes there is an overloaded resource, such as memory or network bandwidth or CPU cycles. Sometimes a solution built into the software doesn't scale, so performance seems fine until the load on the system gets high. Then suddenly we fall off a performance cliff. Sometimes we have built an inefficient implementation that puts in unnecessary overheads, such as copying a piece of data many times or making lots of calls to recursive functions to perform a tiny amount of real work.

The problem you have will affect how you go about looking for it. 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. If your problem is a bottleneck in your network, running tests that don't send messages across the network will never find it. If your problem is contention in scheduling, a performance experiment embedded in a single process won't provide much insight.

There's an obvious chicken-and-egg problem here. You might know performance is bad, but to run an experiment to determine exactly why, you need to know why performance is bad. Otherwise, you might waste your time running an irrelevant experiment that tells you nothing. So how do you get started?

In actuality, you often can get some clues without running any new experiments. The operating system will tell you, on request, how much memory is being used, CPU utilization, and many other statistics concerning the behavior of your processes. If there's plenty of free memory and the system is still running poorly, you'd probably waste your time building a performance experiment based on investigating the effects of varying memory usage. If there are rarely any ready processes waiting to run, scheduling is very likely not the source of your problem. Knowledge of the general architecture and expected behavior of the poorly performing system component can help, as well. If you know that the software that is running slow always works on pretty much the same quantity of data, it's probably not a scaling problem.

But usually these kinds of hints will only get you so far, and often they will provide indications, not actually identify the source of the problem. What then? Take the best knowledge you can easily obtain about your code and the observed performance problem and generate a hypothesis about why it's happening. Design an experiment that will test that hypothesis, proving or disproving it. Run the experiment and determine if your hypothesis is born out. If not, generate a new hypothesis (with, one would hope, a deeper knowledge base to work from than before) and try again. Obviously, there are elements of art, experience, and even luck in this process. But you've seen this kind of process before. It's much like finding a bug in a program, where you observe the erroneous behavior, make a hypothesis for its cause, add fixes or extract new information relevant to the hypothesis, and test it, until the bug is found and repaired. Generally, finding performance problems is harder than finding bugs, since it's harder to narrow the field in which you're searching, but the basic approach is similar.

Like finding functionality bugs, finding performance problems is a skill you are likely to develop with practice. You will come to learn the signals that point towards particular

classes of performance problems and develop instincts that lead you in the right direction more often than the wrong one. However, never mistake your experience or a good hunch for the results of an actual measurement program. Ultimately, the point of performance measurement is to reveal the actual truth of what's happening, and nothing short of measuring it is a substitute for that evidence.

Workloads

One important aspect of running a performance experiment is the workload you use. In some cases, you are examining the performance of a particular program or operating system element, in which case you will tailor the workload to exercise that software. In many cases, you are looking for general performance in the face of typical overall system loads. In that situation, you need to generate a realistic workload for your system. Either way, somehow you must provide data sets, background activities, network traffic, and various other types of workload-related effects to test the performance.

There are different aspects of workloads that you need to think about when designing performance experiments. Your system is designed to do certain things: schedule processes, lay out a file system on a flash drive, respond to web requests, and so on. Obviously, one important aspect of the workload is the tasks you provide to your system directly related to its purpose: the set of processes to be scheduled, the files and file accesses to be handled, the web requests that clients generate. An equally important aspect of the workload, however, is based on the fact that operating systems are complex and involve simultaneous interactions of many different components that might affect each other in unpredictable ways. How your file system would perform if the only activity on the operating system was reads and writes to it is not the question you need to answer, as a rule. The important question is how it would perform in the face of all the other ordinary activities that the operating system would be doing in a real world setting. So your workload must also capture those background activities.

There are several different types of workloads typically used for performance measurement.

1. Traces – Take or otherwise obtain a detailed trace of the workload of the system in its ordinary activities. What such a trace consists of depends on the nature of what you are testing. For a web server, the trace is likely to be a set of web requests submitted to the server. For a mail server, it is likely to be a set of messages delivered to that server. For a file system, it might be a set of opens, reads, writes, and other file system operations. For an operating system component, it might be a set of applications that are run in a particular order with specified inputs. Whatever the trace might consist of, you capture it from the running system, saving it in a form that will allow you to recreate it in a faithful manner. Then, for each experimental run, you start from the beginning and run it to the end.

   Traces have good and bad properties for performance experiments. A good property is realism, since they represent realistic activities that you would actually want your system to handle well. Another good property is reproducibility. The same trace can be replayed over and over, identically for each run. There is an

issue here if the performance of the system has an impact on what would have happened in the real system. For example, a trace of a network protocol that sends a message and receives an acknowledgement before sending the next message would have run differently if the acknowledgement had been produced in half the time, double the time, or at some other delay than it had been when the trace was gathered. If the system being tested is the one generating the acknowledgements, that can result in the replayed trace producing unrealistic results.

A disadvantage of a trace is that it is not easily reconfigurable. 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. Cutting out portions of a trace might not realistically represent a smaller workload, either. Scaling a trace up or down is usually hard. Another frequent disadvantage is availability. Good traces are not easy to come by, and if your system is not yet in production, you might be unable to gather your own. Except for freshly gathered traces of your own, most traces you can find will be somewhat (to very) old. Another disadvantage in some cases is that it might be difficult to gather the information needed to create the trace from the tools available to you. You might not be able to capture all the system calls applications perform, for instance. Also, any particular single trace might or might not represent the typical activity of the system. The moment at which it was gathered might have been unusual, compared to the ordinary activities of your system. Depending on exactly what you are tracing, there may be privacy implications to saving it in a trace. For certain kinds of system, such as those dealing with medical records, you may have legal obligations to handle some of the data in particular ways. Be aware of any such privacy problems before you store data for a trace.

2. Live workloads – Sometimes you can perform measurements on a working system as it goes about its normal activities. A production system can also instrumented and data gathered as it does its work. Realism is a clear advantage here. Also, provided you can continue to do tests on the system indefinitely, with enough time you can capture a very wide range of real system behavior. You are likely to need to take little or no effort to establish realistic background loads, since they establish themselves, in essence.

   This approach has its own disadvantages. One is 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. Another is that your experimental framework usually needs to have minimal impact, both in performance and functionality, on the running system, since it is presumably more important to complete its live work than to gather your measurements. Unless this impact is essentially nil, you are not likely to be able to run the performance measurements for very long on a working system, since those tasked with getting it to do its job will not appreciate your experiments getting in the way. As with

traces, consider whether there are privacy implications to your observation of the live workload.

3. Standard benchmarks – These are either sets of programs or sets of data that are intended to drive performance experiments, typically on some particular thing, such as a file system, a database, a web server, or an intrusion detection system. They may have been derived from real traces at some point or they may be built from models of system behavior. They are typically designed to be usable by many developers, so it is often fairly easy to integrate them into your experiments, provided you are working in the same general framework they were designed for. (For example, a file system benchmark might generate Posix-compliant file operations, so any file system that is compatible with Posix can use it for testing.) They allow for easy comparison to other systems' performance, since the developers of those system can also run the same benchmark, or, indeed, you can yourself, if those other systems are also available for testing. A well-designed benchmark is likely to exercise a wide range of system behaviors, so the results you get from it may give you a fairly complete picture of your system's performance under different realistic conditions. Widely used benchmarks have been heavily studied themselves, and are unlikely to have many bugs, and likely to be relatively good representations of the kind of workload they are intended to mimic. Some benchmarks (though not all) are built to be inherently scalable, allowing you to adjust the workload up or down with little more than changing a line or two in a configuration file. Since benchmarks are artificial, there are usually no privacy implications to using them.

   As you no doubt expect, though, standard benchmarks have their own set of disadvantages. First, there are a limited number of them available, and there might not be one suited for the system or situation you want to test. One aspect of this characteristic is that standard benchmarks might not include portions of the workload space that are unusual in general, but important for your case. Another aspect of this characteristic is that it's tempting to use a standard benchmark that isn't quite right for your situation just because it's easy to do so. Resist such temptations. Second, since developing a good benchmark is quite a lot of work, they tend to be used for a very long time, running the risk of representing archaic workloads that no longer match what would happen on a current system.

4. Simulated workloads – In this approach, you build models of the loads you are interested in, typically models instantiated in executable code. These models are usually parameterized, allowing them to be scaled up or down, to alter the mix of different elements of the load, and otherwise to create variations on the load. When testing a system's performance, one decides which parameter settings are most relevant and uses the simulated workload models with suitable settings. This approach has the advantage of being easily customized to many different scenarios and possibilities, since you need merely alter the model parameters accordingly. One important aspect of this flexibility is good handling of scaling, either up or down. Assuming that there is no true randomization in the models, they are infinitely repeatable, allowing you to perform directly comparable tests

of different system alternatives. As with standard benchmarks, the artificiality of simulated workloads has the benefit of avoiding privacy considerations.

However, the validity of the performance results you achieve is only as good as the quality of the models. It is not easy to produce good models of complex systems and phenomena, and one can easily overlook important features of real loads in building one's models. While parameters can be easily altered and scaled, even if the model was faithful to real load for some settings, it may prove unrealistic at others. It may also be unclear how to set the various parameters to produce simulated load that matches a particular real load. If the parameters are set incorrectly, one may get a very false picture of how a real system would behave in those situations.

Common Mistakes in Performance Measurements

At the highest level, most mistakes in performance measurements can be attributed to insufficient thought by the experimenter. If you leap into a measurement program without giving some careful thought to what you are going to do, it may be hard to predict what problem you're going to encounter, but you've very likely to encounter some problem. Be clear on what issues you are investigating and what methods you are using to investigate them. Most frequently, you will benefit from writing down what you propose to do before you do it. That allows you to go back and check that you are continuing down the path you'd planned and hitting all the points you thought were important. It's OK to alter your plans as new information arises, but do so knowingly, not because you're blindly flailing around in a huge space of possible performance experiments.

At the next level of detail, problems tend to arise in areas like not measuring the right thing, not measuring accurately, not measuring in situations matching real world behavior, and not understanding what your measurements are telling you. These issues are so broad and have some many variants of the mistakes you can make, which are often quite specific to the system that you are measuring, that it is not necessarily helpful to pin down too many particular mistakes.

But there are certain more specific mistakes in measuring system performance that are sufficiently common that they are worth calling out in detail. We'll go through a few of these.

1. Measuring latency without considering utilization. Everything runs fast (or at least faster) on a lightly loaded system. Measuring the latency of an operation when absolutely nothing else is going on in the system is only worthwhile if the question to be answered is what is the fastest possible time in which it will complete. For anything else, one should measure the latency when the system has a characteristic background load. Most often, one should also examine the latency when the system is heavily loaded, as well, since that condition is likely to arise sooner or later in most systems.
2. Not reporting the variability of measurements. Sometimes this mistake is even more egregious, when a quantity is measured only once and that value reported as entire truth of the performance. Even if multiple measurements are taken,

however, merely reporting the average of the values will often give a false impression of the performance observed. For most phenomena, one needs to understand the distribution of those values. Is it basically bi-modal? Is there one very common value and some outliers? Are the values uniformly spread across some range? What behavior you will observe in the real world and whether you will be happy with your system or miserable may depend on the answers to those questions, so a good performance experiment should offer you some insight into them.

3. Ignoring important special cases. This mistake comes in two varieties. In one, you ignore the fact that a few special cases distort the measurement, given you a false sense of what happens in the more general case. In the other, while you carefully measure the ordinary case, you fail to consider that there will be some special circumstances that are very important and that are likely to display different performance.

   Perhaps the most common version of the first variety is ignoring startup effects. Computers make effective use of caching in many different ways. Programs loaded off disk may hang around in 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 multiple times. Hardware caches recently run instructions to avoid the cost of fetching them out of RAM when executing a loop. Caching is so ubiquitous and built into so many levels of a system that you are unlikely to predict all of its uses. That means you should regard the first few runs of a performance experiment as being potentially biased. They may have paid higher penalties than subsequent runs in order to warm up some caches. That does not necessarily mean you should discard them or disregard them, since, after all, every cache in a real system pays a performance penalty the first time the data is used, and that is a real element of system performance. But it does mean you should not compare different alternatives when one alternative has had the benefit of a warm cache while the other has not.

   A similar problem can work in the opposite way. Sometimes we have a data structure of a limited size, and as long as we are working within that size, things go quickly. When we have more elements than the data structure can hold, performance degrades. For example, consider a hash table that uses chaining to handle collisions. If the table is relatively empty, every read will hit the element it was looking for immediately, and performance will be fast. When the table starts to fill up, some probes will need to follow a chain of entries to find the one they are looking for in that cache bucket; performance will slow down in some cases, while remaining fast for others. If the table is very full, performance will slow down for almost everything, since most probes will require searching a chain. File systems that use various kinds of indirect blocks are another example. Accesses to the first few blocks will avoid the indirect block and will be fast. Accesses further into a file will require indirect, doubly indirect, or triply indirect access, and will be slower, depending on the access pattern. Again, these are genuine performance effects, but only if you are trying to measure performance for conditions where they might occur.

The other variant is also important, because sometimes these genuine effects are critical to what you need to measure. If you only measure a file system's performance on short files, you may never learn that it is very slow once files exceed a certain size until, in production use, your system suddenly slows to a crawl. Special cases can be very important. Sometimes what you really need to know, for example, is how long servicing a web request will take under the worst circumstances likely to arise, or how your system will behave under extremely high load, or what will happen if a piece of hardware experiences partial failure. This issue returns to the point of understanding what you are measuring and why you are measuring it.

4. Ignoring the costs of your measurement program. In a few cases, you may be measuring a system using tools that are entirely external to the system and impose little or no load on that system. Sniffing traffic on a network is one example. More commonly, especially for operating system measurement, you are using your system to measure your system. You're not only sharing the processor, memory, network, and secondary storage devices with the system under study, but you're sharing some of the abstractions the operating system offers. For example, if you are logging information from your measurement code for later examination, you are probably exercising the file system. Does that matter? If what you're measuring is file system performance, it almost certainly does, and it might even if you are measuring something that does not have any obvious relationship to the file system, such as the scheduler or the memory manager. The file system is obviously not the only example. If you are running a separate process to perform your measurements, for example, it is competing for CPU and memory with the processes you are trying to measure.

   Ideally, you want to avoid having your performance measurement program affect the behavior of the observed system at all. Often, this is impossible, in which case minimizing the impact and understanding it are the next best alternatives. There are useful techniques to minimize measurement costs. For example, instead of writing each observed measurement to disk (an expensive operation that interferes with other disk behavior), save it in a RAM buffer. Either write out the buffer to disk at the end of the experiment, if you can buffer that much data without affecting the experiment in other ways, or infrequently write it to disk if you can't keep a buffer that big. Keep your measurement code small and cheap. If feasible, bundle it into the process you are measuring, rather than running it as a separate process, unless you have reason to believe that it will cause less interference as a separate process. Avoid doing data analysis while gathering the data, since such analysis may be computationally expensive. Do it when you have finished the experiment and its calculations will not interfere with the experiment's timings. Start up anything computationally expensive related to your experiment before you start measuring the system you are investigating.

5. Losing your data. Never throw away experimental data, even if you think that you are finished with your experiment, nor even if you think the data in question was gathered in an erroneous way. Data has a way of proving useful for many

purposes, but discarded data is never useful. Of course, you should particularly avoid carelessly losing data. One common beginner mistake is to inadvertently overwrite the data from a previous experiment with data being gathered for the next experiment. Also remember to label your data. Even if you have kept every byte of data you've gathered, if you can't tell which bytes are related to which parts of your experiment, the data is as good as lost. This advice is for the long term. Ideally, you should be able to go back and look at data you gathered twenty years ago. Maybe you will never look at a lot of the data you gathered in the past again, but probably you will eventually want to look at some of it, and it's hard to predict what's going to prove useful in the future. So save it all, if possible. It's also important to keep the metadata around, which in this case means information about how you set up and ran your experiments. Which version of the operating system was it that you used on that experiment you ran five years ago? Chances are you won't remember, so make sure it's written down somewhere you can find.

6. Valuing numbers over wisdom. Remember, the point of your performance experiment is not to obtain a set of numbers. It's to understand important performance characteristics of your system. The numbers are the means to an end, not themselves the end. Don't bother gathering numbers that are not going to lead to wisdom, and don't consider your task complete when you have the numbers in hand. You actually have the most important step still to go: using the numbers to understand your system performance and, if necessary, using them to guide redesign or reconfiguration of your system in ways that are likely to lead to better performance.

   Unfortunately, it's hard to offer general advice on how to extract wisdom from sets of numbers. That's a task you will need to perform on a case-by-case basis. But do remember that performing that task is the goal, the entire point of running an elaborate performance experiment. Without the resulting wisdom, the work you did to get the numbers will be wasted.

Load and Stress Testing

Mark Kampe
$Id: loadstress.html 7 2007-08-26 19:52:08Z Mark $

1. Introduction

Load and Stress Testing is very different from most other testing activities.

For most products, the vast majority of all tests exercise positive functional assertions (if situation x, the program will y). Such assertions may describe either positive (functionality) or negative (error handling) behavior. A typical suite is collection of test cases, each of which has the general form:

- establish conditions (c_1, c_2, ... c_n)
- perform operations (o_1, o_2, ... o_n)
- ascertain that assertions (a_1, a_2, ... a_n) are satisfied.

The art of creating such a test suite is being able to define a set of test cases that simultaneously:

a. adequately exercises the program's capabilities
b. adequately captures and verifies the program's behavior
c. is small enough to be practically implementable

If all operations are performed, and all of the required assertions were satisfied, the program has passed the test. Such "functional validation" suites form the foundation for automated software testing, and are the primary basis for determining whether or not a product "works". As such, repeatability of results is considered to be very important.

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.
- we will run these test cases in pseudo-random orders for unspecified periods of time.
- we have no expectation that results will be repeatable (we are, in fact, depending on this).
- in many cases, there is 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 not even look at returned results.

This note is a brief introduction to the goals and methods of load and stress testing.

2. Load Testing

The initial (and perhaps still primary) purpose of load testing is to measure the ability of a system to provide service at a specified load level. A test harness issues requests to the component under test at a specified rate (the offered load), and collects performance data. The collected performance data depends on what is to be measured, but typically includes things like:

- response time for each request
- aggregate throughput
- CPU time and utilization

- disk I/O operations and utilization
- network packets and utilization

The resulting information can be used to:

- measure the system's speed and capacity
- analyze bottlenecks to enable improvements

The key ingredient in this process is a tool to generate the test traffic. Such tools are called **load generators**.

2.1 Load Generation

A 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", and is typically performed through the system's primary service interfaces:

- If the system to be tested is a network server, the load generator will pretend to be numerous clients, sending requests over a network.
- 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.

In all cases, the test load is broadly characterized in terms of:

- request rate
  the number of operations per second
- request mix
  Different types of clients use a system in different ways. A database might do a large number of small (and relatively random) disk reads and writes, while an streaming video server would do a much smaller number of huge contiguous transfers, all reads. If different types of requests exercise very different code paths, it is important that the load generator be able to accurately emulate all of the various 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 (timed sequences operations on related objects). In some situations it may be necessary to simulate realistic scenarios against pre-determined or random objects.

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. Others may have rule grammars that describe typical scenarios, and use these to generate complex realistic usage scenarios.

Most such load generators are proprietary tools, developed and maintained by the organizations that build the products they measure. Some have been turned into products (or open source tools) and are widely used. Some have become so widely used that they have been adopted as "standard performance benchmarks".

2.2 Performance Measurement

There are a 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 use this as a calibrated back-ground for measuring the performance other system services.

4. Deliver requests at a rate, and use this as a test load for detailed studies performance bottlenecks.

In the first two usages, the load generator is a measurement tool. In the other usages, it provides a calibrated activity mix, to exercise the system.

2.3 Accelerated Aging

Many errors (e.g. memory leaks) can have trivial consequences, and only accumulate to measurable effects after long periods of time. Load generators can be used to create realistic traffic to simulate normal traffic for long periods of time. Alternatively, they can be cranked up to much higher rates in order to simulate accelerated aging. It is common for test plans to require products to undergo (at least) months of continuous load testing to look for the accumulation of such problems.

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. Accelerated Aging uses cranked up load generators to simulate longer periods of use. If we go farther (into simulating scenarios far worse than anything that is ever expected to really happen) we enter the realm of stress testing.

Any error that results from a simple situation is likely to be easily recognized, tested, and (therefore) properly handled. In mature and well tested products, the residual errors tend to involve combinations of unlikely circumstances. These problems are much easier to find and fix in design review than they are to debug, but we still need to test for them. But how can we test for combinations of circumstances that we can't even enumerate?

The answer is random stress testing.

- use randomly generated complex usage scenarios, to increase the likelihood of encountering unlikely combinations of operations.
- deliberately generate large numbers conflicting requests (e.g. multiple clients trying to update the same file).
- introduce a wide range randomly generated errors, and simulated resource exhaustions, so that the system is continuously experiencing and recovering from errors.
- introduce wide swings in load, and regular overload situations.

Such testing takes situations that might normally occur only once or twice per year, and makes them occur (in combinations) hundreds of times per minute. Take a large number of such systems, and run them in this mode for several months. **This** will give us some serious confidence about the robustness and stability of our systems. Such testing is extremely demanding, and (in fact) very few software products receive (or survive) such testing. This is, however, typical methodology for mission critical and highly available products.

4. Conclusions

In the early stages of the product, most of the bugs are found by reviews and functional validation tests. These remain valuable (as regression tools) throughout the life of the product, but they are not actually expected to find many more bugs once they have initially succeeded.

Once a product has passed this "adolescent" stage, most of the bug reports result from new usage scenarios associated with adoption by real customers. There is a wider diversity of use here, and it may take a while so shake out all of these problems. But again, once the product has been brought up to the demands of every-day customer use, the number of bug reports resulting from this falls off sharply.

If we ignore new features (which are, in some sense, new software), the improvements in mature software tend to be in performance and robustness. The tools and techniques for finding functionality problems are neither designed nor adequate for driving improvements in these areas. Load and stress testing are very different from functional testing. The goals are different, the tools are different, the techniques are different, and the testing programs are different.

Functional quality starts with good design, and is then complemented by good review, and secured by a complete and well considered set of functional test suites. Performance and robustness also start with good design, are complemented by review, and secured by testing. Functionality tools and testing are, for the most part, done when the product ships. Load and stress testing will continue to be a critical element of product quality throughout its lifetime, and (unlike functional test cases) the load and stress testing tools may receive almost as much design and development attention as the tested product does.

In mature products, the difference between a good product and a great one is often found it the seriousness of the ongoing performance and robustness programs. **Performance** and **availability** don't just happen. They must be **earned**.