Check for updates

DOI:10.1145/3213770

Performance measurements often go wrong, reporting surface-level results that are more marketing than science.

BY JOHN OUSTERHOUT

Always Measure One Level Deeper



PERFORMANCE MEASUREMENT IS one of the most important parts of software development. In academic research a thorough performance evaluation is considered essential for many publications to prove the value of a new idea. In industry, performance evaluation is necessary to maintain a high level of performance across the lifetime of a product. For example, cloud services promise to maintain particular performance levels; service providers must thus be able to detect when performance drops below acceptable levels and quickly identify and fix the problem.

A good performance evaluation provides a deep understanding of a system's behavior, quantifying not only the overall behavior but also its internal mechanisms and policies. It explains why a system behaves the way it does, what limits that behavior, and what problems must be addressed in order to

» key insights

- Performance measurement is less straightforward than it might seem; it is easy to believe results that are incorrect or misleading and overlook important system behaviors.
- The key to good performance measurement is to make many more measurements besides the ones you think will be important; it is crucial to understand not just the system's performance but also why it performs that way.
- Performance measurement done well results in new discoveries about the system being measured and new intuition about system behavior for the person doing the measuring.

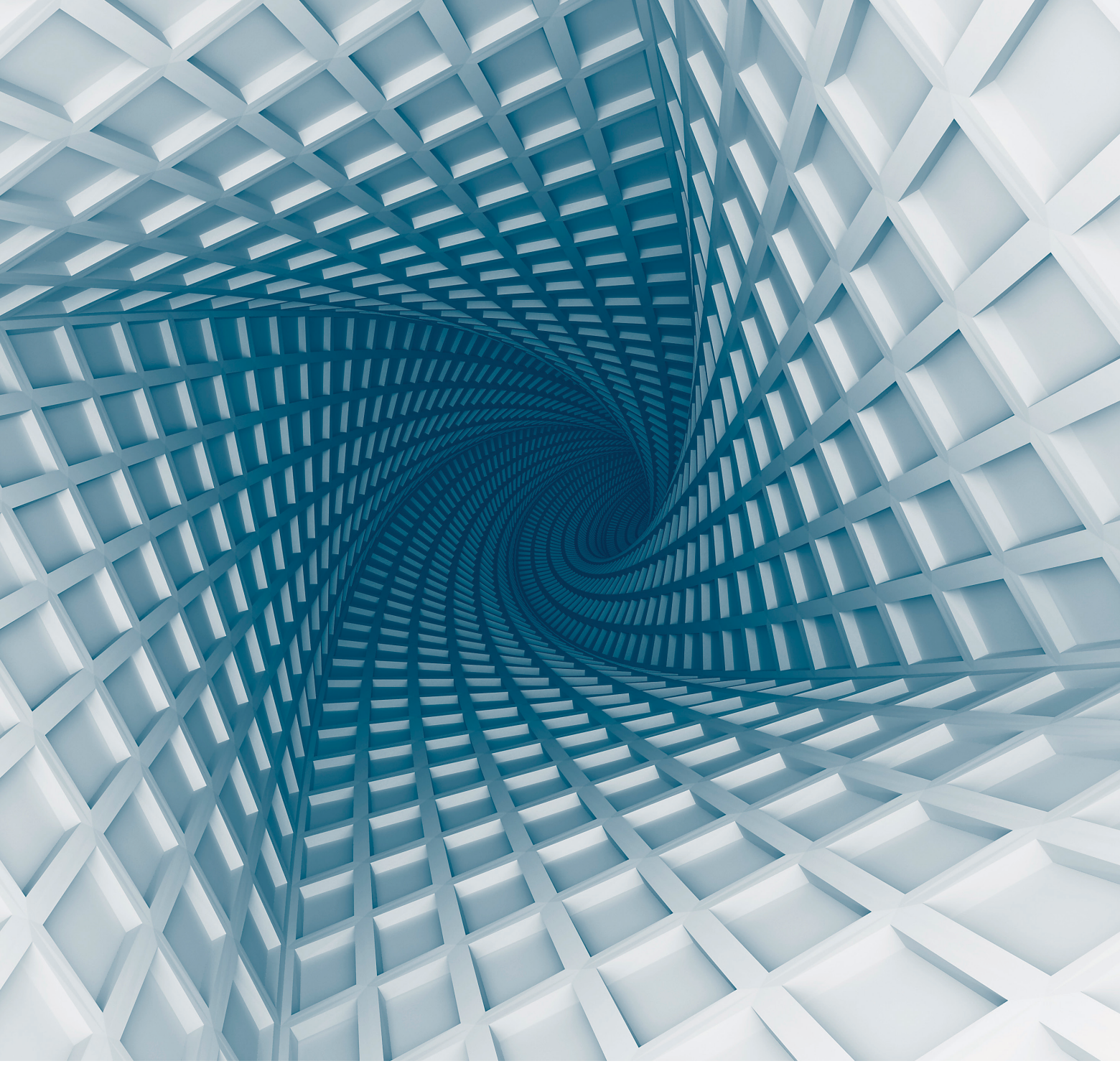


IMAGE BY VERSUS STUDIO

improve the system. Done well, performance evaluation exposes interesting system properties that were not obvious previously. It not only improves the quality of the system being measured but the developer's intuition, resulting in better systems in the future.

Unfortunately, there is no widespread understanding or agreement as to how to measure performance. Performance evaluation is rarely taught in computer science classes. And new faculty lack well-developed performance-measurement skills, making it difficult for them to train their students. The only way to become expert is through trial and error.

As a result, performance measurement is often done poorly, even by experienced developers. For example, if you have written a conference paper on a software system, it probably unfolded like this: The system implementation took longer than expected, so performance evaluation could not begin until a week or two before the paper submission deadline. The first attempts to run benchmarks resulted in system crashes, so you spent the next week fixing bugs. At this point the benchmarks ran, but the system's performance was not much better than the comparison systems. You tried different experiments, hoping to find one where the system looked good; this exposed yet more bugs that had to be fixed. Time was running out, so you stopped measuring as soon as you found an experiment that produced positive results. The paper focused on this experiment, omitting the results that were less favorable. There were a few things about these results that did not make complete sense, but you did your best to come up with plausible explanations

for them. There was not enough time to validate or double-check the numbers, and you could only hope there were not too many errors.

Measurements gathered this way are likely incomplete, misleading, or even erroneous. This article describes how to conduct performance measurement well. I first discuss five mistakes that account for most of the problems with performance measurements, all of which occurred in the scenario I just outlined. I then spell out four rules to follow when evaluating performance. These rules will help you avoid the mistakes and produce high-quality performance evaluations. Finally, I offer four suggestions about infrastructure to assist in performance evaluation.

The most important idea overall, as reflected in this article's headline, is to dig beneath the surface, measuring the system in depth and detail from multiple angles to create a complete and accurate understanding of performance.

Performance measurements should be considered guilty until proven innocent.

Most Common Mistakes

When performance measurements go wrong, it is usually due to five common mistakes:

Mistake 1: Trusting the numbers. Engineers are easily fooled during performance measurements because measurement bugs are not obvious. Engineers are used to dealing with functional bugs, which tend to be noticeable because they cause the system to crash or misbehave. If the system produces the desired behavior, it is probably working. Engineers tend to apply the same philosophy to performance measurements; if performance numbers are being generated and the system is not crashing, they assume the numbers are correct.

Performance-measurement code is just as likely to have bugs as any other code, but the bugs are less obvious. Most bugs in performance-measurement code do not cause crashes or prevent numbers from appearing; they simply produce incorrect numbers. There is no easy way to tell from a number whether it is right or wrong, so engineers tend to assume the numbers are indeed correct. This is a mistake. There are many ways for errors to creep into performance measurements. There may be bugs in the benchmarks or test applications, so the measured behavior is not the desired behavior. There may be bugs in the code that gathers metrics and processes them, as when, say, a clock is read at the wrong time or the 99th percentile is miscomputed. The system being measured may have functional bugs. And, finally, the system may have performance bugs, so the measurements do not reflect the system's true potential.

I have been involved in dozens of performance-measurement projects and cannot recall a single one in which the first results were correct. In each case there were multiple problems from the list just outlined. Only after working through them all did my colleagues and I obtain measurements that were meaningful.

Mistake 2: Guessing instead of measuring. The second common mistake is to draw conclusions about a system's performance based on educated guesses, without measurements to back them up. For example, I found the following explanation in a paper I reviewed recently: " ... throughput does not increase with the number of threads ... This is because the time taken to traverse the relatively long linked list bounds server performance." There was no indication that the authors had measured the actual length of the list or the time taken to traverse it, yet they stated their conclusion as fact. I frequently encounter unsubstantiated conclusions in papers; there were at least five other occurrences in the paper with the quote.

Educated guesses are often correct and play an important role in guiding performance measurement; see Rule 3 (Use your intuition to ask questions, not answer them). However, engineers' intuition about performance is not reliable. When my students and I designed our first log-structured file system,[4] we were fairly certain that reference patterns exhibiting locality would result in better performance than those without locality. Fortunately, we decided to measure, to be sure. To our surprise, the workloads with locality behaved worse than those without. It took considerable analysis to understand this behavior. The reasons were subtle, but they exposed important properties of the system and led us to a new policy for garbage collection that improved the system's performance significantly. If we

had trusted our initial guess, we would have missed an important opportunity for performance improvement.

It is unsafe to base conclusions on intuition alone, yet engineers do it all the time. A common mistake is for an engineer to hypothesize that a particular data structure is too slow and then replace it with a new data structure the engineer believes will be faster. If the problem is not verified by measuring performance, there is a good chance the optimization will not improve performance. The code change will simply waste a lot of time and probably introduce unnecessary complexity.

When I find a guess presented as fact and ask for justification, I sometimes get this response: "What else could it possibly be?" But this is a cop-out, suggesting it is up to others to prove the theory wrong and OK to make unsubstantiated claims until someone else proves them false. In some cases the person making the comment feels a process of elimination had been used, ruling out all possible alternatives. Unfortunately, a process of elimination is not reliable in performance evaluation, because it is not possible to know with certainty that every possible cause has been considered. Many factors can influence performance, and the ultimate cause of behavior is often something non-obvious, meaning a process of elimination will not consider it. It is unsafe to present an explanation as fact unless measurements confirm the specific behavior(s).

Mistake 3: Superficial measurements. Most performance measurements I see are superficial, measuring only the outermost visible behavior of a system (such as the overall running time of an application or the average latency of requests made to a server). These measurements are essential, as they represent the bottom line by which a system is likely to be judged, but they are not sufficient. They leave many questions unanswered (such as "What are the limits that keep the system from performing better?" and "Which of the improvements had the greatest impact on performance?"). In order to get a deep understanding of system performance, the internal behavior of a system must be measured, in addition to its top-level performance.

Superficial measurements are often combined with Mistake 1 (Trusting the numbers) and Mistake 2 (Guessing instead of measuring); the engineers measure only top-level performance, assume the numbers are correct, and then invent underlying behaviors to explain the numbers. For example, I found the following claim in a paper I reviewed recently (system names obscured to preserve author anonymity): "Unlike YYY, XXX observes close-to-linear-throughput scaling with more publishers due to its lock-free resolution of write-write contentions." The paper measured scaling, but there were no measurements of write-write contention, and systems XXX and YYY differed in many ways, so other explanations were possible for the performance difference.

Mistake 4: Confirmation bias. Performance measurement is rarely indifferent; when you measure performance, you are probably hoping for a particular outcome. If you have just built a new system or improved an existing one, you probably hope the performance measurements will show your ideas were good ones. If the measurements turn out well, it increases the likelihood your paper will be accepted or your boss will be impressed.

Unfortunately, such hope results in a phenomenon called "confirmation bias."[1] Confirmation bias causes people to select and interpret data in a way that supports their hypotheses. For example, confirmation bias affects your level of trust. When you see a result that supports your hypothesis, you are more likely to accept the result without question. In contrast, if a measurement suggests your new approach is not performing well, you are more likely to dig deeper to understand exactly what is happening and perhaps find a way to fix the problem. This means that an error in a positive result is less likely to be detected than is an error in a negative result.

When choosing benchmarks, you are more likely to choose ones that produce the desired results and less likely to choose ones that show the weaknesses of your approach. For example, a recent paper described a new network protocol and compared it to previous proposals. The previous proposals had all measured latency using the 99th-percentile worst case, but this particular paper measured at the median. The results appeared favorable for the new proposal. My students reran the measurements for the new protocol and discovered its 99th-percentile latency was significantly worse than the comparison protocols. We wondered if the paper's authors had intentionally switched metrics to exaggerate the performance of their protocol.

Confirmation bias also affects how you present information. You are more likely to include results that support your hypothesis and downplay or omit results that are negative. For example, I frequently see claims in papers of the form: "XXX is up to 3.5x faster than YYY." Such claims cherry-pick the best result to report and are misleading because they do not indicate what performance can be expected in the common case. Statements like this belong in late-night TV commercials, not scientific papers.

If applied consciously, bias is intellectually dishonest. Even if not applied consciously, it can cause results to be reported in a way that is more marketing than science; it sounds like you are trying to sell a product rather than uncover the truth about a system's behavior. Confirmation bias makes it more likely that results will be inaccurate (because you did not find bugs) or misleading (because you did not present all relevant data).

Mistake 5: Haste. The last mistake in performance evaluation is not allowing enough time. Engineers usually underestimate how long it takes to measure performance accurately, so they often carry out evaluations in a rush. When this happens, they will make mistakes and take shortcuts, leading to all the other mistakes.

The time issue is particularly problematic when working under a deadline (such as for a conference publication). There is almost always a rush to meet the submission deadline. The system implementation always takes longer than expected, delaying the start of performance measurement; there is often only a week or two for evaluation before the submission deadline, resulting in a sloppy evaluation. In principle, authors could keep working on the measurements while waiting for the paper to be reviewed, but in practice this rarely happens. In-

stead, they tell themselves: "Let's not spend more time on the paper until we see whether it is accepted." Once the paper is accepted, there are only a few weeks before the deadline for final papers, so there is yet another rush.

Keys to High-Quality Performance Analysis

Consider four rules that are likely to prevent the mistakes from the preceding section and lead to trustworthy and informative evaluations:

Rule 1: Allow lots of time. The first step toward high-quality performance measurements is to allow enough time. If you are measuring a non-trivial system, you should plan on at least two to three months. I tell my graduate students to aim for a complete set of preliminary measurements at least one month before the submission deadline; even this is barely enough time to find and fix problems with both the measurements and the system.

Performance analysis is not an instantaneous process like taking a picture of a finished artwork. It is a long and drawn-out process of confusion, discovery, and improvement. Performance analysis goes through several phases, each of which can take anywhere from a few days to a few weeks. First, you must add instrumentation code to the system to record the desired metrics. You must then get benchmark applications running, either by writing them or by downloading and installing existing programs. Running benchmarks will probably stress the system enough to expose bugs, and you will need to then track down and fix them. Eventually, the system will run well enough to start producing performance numbers. However, these numbers will almost certainly be wrong. The next step is to find and fix bugs in the measurements. Once you have verified the accuracy of the measurements, you will start to uncover problems with the system itself. As you look over the performance measurements, you will probably uncover additional functional bugs. Once they have been fixed, you can start analyzing the performance in depth. You will almost certainly discover opportunities to improve performance, and it is important to have enough time to make these improvements. You will encounter many things that do not make sense; in order to resolve them, you will need to add new metrics and validate them. To get the best results, you must iterate several times improving the metrics, measuring performance, and improving the system.

Rule 2: Never trust a number generated by a computer. Under Mistake 2 (Guessing instead of measuring), I discussed how it is tempting to believe performance numbers, even though they are often wrong. The only way to eliminate this mistake is to distrust every measurement until it has been carefully validated. Performance measurements should be considered guilty until proven innocent. When students come to me with measurements, I often challenge them by asking: "Suppose I said I don't believe these measurements. What can you say to convince me that they are correct?"

The way to validate a measurement is to find different ways to measure the same thing:

Take different measurements at the same level. For example, if you are measuring file-system throughput, do not measure just the throughput seen by a user application; also measure the throughput observed inside the operating system (such as at the file block cache). These measurements should match;

Measure the system's behavior at a lower level to break down the factors that determine performance, as I discuss later under Rule 4 (Always measure one level deeper);

Make back-of-the-envelope calculations to see if the measurements are in the ballpark expected; and

Run simulations and compare their results to measurements of the real implementation.

As you begin collecting measurements, compare them and be alert for inconsistencies. There will almost always be things that do not make sense. When something does not make complete sense, stop and gather more data. For example, in a recent measurement of a new network transport protocol, a benchmark indicated that a server could handle no more than 600,000 packets per second. However, my colleagues and I had seen servers process more than 900,000 packets per second with other protocols and believed the new protocol was at least as efficient as the old ones. We decided to gather additional data. As a result, we discovered a bug in the flow-control mechanism on the client side: clients were not transmitting data fast enough to keep the server fully loaded. Fixing the bug improved performance to the level we expected.

Further analysis will sometimes show the unexpected behavior is correct, as in the log-structured file system example discussed under Mistake 2 (Guessing instead of measuring); such situations are usually interesting, and you will learn something important as you resolve the contradiction. In my experience, initial performance measurements are always riddled with contradictions and things we don't understand, and resolving them is always useful; either we fix a problem or we deepen our understanding of the system.

Above all, do not tolerate anything you do not understand. Assume there are bugs and problems with every measurement, and your job is to find and fix them. If you do not find problems, you should feel uneasy, because there are probably bugs you missed. Curmudgeons make good performance evaluators because they trust nothing and enjoy finding problems.

Rule 3: Use your intuition to ask questions, not to answer them. Intuition is a wonderful thing. As you accumulate knowledge and experience in an area, you will start having gut-level feelings about a system's behavior and how to handle certain problems. If used properly, such intuition can save significant time and effort. However, it is easy to become overconfident and assume your intuition is infallible. This leads to Mistake 2 (Guessing instead of measuring).

The best way to use intuition is to identify promising areas for further exploration. For example, when looking over performance measurements, ask yourself if they make sense. How does the performance compare to what you expected? Does it seem too good to be true? Does the system scale more poorly than you had hoped? Does a curve jump unexpectedly when you expected it to be smooth? Do some benchmarks exhibit behavior that is dramatically

different from others? Consider anything that does not match your intuition a red flag and investigate it, as described in Rule 2 (Never trust a number generated by a computer). Intuition can be very helpful in identifying problems.

Intuition is great for directing attention but not reliable enough to make decisions on it alone. Intuition should always be validated with data before making decisions or claims. If your intuition suggests why a particular result is occurring, follow it up with measurements that prove or disprove the intuition. Draw conclusions based on the measurements, not the intuition, and include some of the measured data in the conclusion, so others know it is not just a guess.

If you continually form intuitions and then test them you will gain knowledge that helps you form better intuition in the future. Every false intuition means there was something you did not fully understand; in the process of testing it and discovering why it is false, you will learn something useful.

Rule 4: Always measure one level deeper. If you want to understand the performance of a system at a particular level, you must measure not just that level but also the next level deeper. That is, measure the underlying factors that contribute to the performance at the higher level. If you are measuring overall latency for remote procedure calls, you could measure deeper by breaking down that latency, determining how much time is spent in the client machine, how much time is spent in the network, and how much time is spent on the server. You could also measure where time is spent on the client and server. If you are measuring the overall throughput of a system, the system probably consists of a pipeline containing several components. Measure the utilization of each component (the fraction of time that component is busy). At least one component should be 100% utilized; if not, it should be possible to achieve a higher throughput.

Measuring deeper is the best way to validate top-level measurements and uncover bugs. Once you have collected some deeper measurements, ask yourself whether they seem consistent with the top-level measurements and with each other. You will almost certainly discover things that do not make sense; make additional measurements to resolve the contradictions. For example, in a recent analysis of a distributed transaction processing system, deeper measurements by my students included network throughput and disk throughput. We were surprised to see that the network throughput was greater than the disk throughput; this did not make sense, since every byte had to pass through both the network and the disk. It turned out that the disk subsystem had been configured with no limit on queue length; the disk was not keeping up, and its output queue was growing without bound. Once the students set a limit on queue length, the rest of the system throttled itself to match the disk throughput. Unfortunately, this meant our initial measurements of overall throughput were too optimistic.

Curmudgeons make good performance evaluators because they trust nothing and enjoy finding problems.

Measuring deeper will also indicate whether you are getting the best possible performance and, if not, how to improve it. Use deeper measurements to find out what is limiting performance. Try to identify the smallest elements that have the greatest impact on overall performance. For example, if the overall metric is latency, measure the individual latencies of components along the critical path; typically, there will be a few components that account for most of the overall latency. You can then focus on optimizing those components.

In recent measurements of a new network transport, one of my students found that round-trip tail latency was higher than our simulations had predicted. The student measured software latency in detail on both the sending and the receiving machines but found nothing that could account for the high tail latency. At this point we were about to conclude that the delays must be caused by the network switch. What else could it be? This would have been Mistake 2 (Guessing instead of measuring). Before giving up, we decided to dig deeper and measure precise timings for each individual packet. The measurements surprised us, showing that outlier delays were not isolated events. Delay tended to build up over a series of packets, affecting all of the packets from a single sender over a relatively long time interval, including packets for different destinations. This was a crucial clue. After several additional measure-

ments, the student discovered that long queues were building up in the sender's network interface due to a software bug. The transport included code to estimate the queue length and prevent queue buildup, but there was a bug in the estimator caused by underflow of an unsigned integer. The underflow was easy to fix, at which point tail latency dropped dramatically. Not only did this process improve the system's performance, it taught us an important lesson about the risks of unsigned integers.

Another way to measure deeper is to consider more detail. Instead of just looking at average values, graph the entire distribution and noodle over the shape to see if it provides useful information. Then look at some of the raw data samples to see if there are patterns. In one measurement of RPC latency, a student found that the average latency was higher than we expected. The latency was not intolerably high, and it would have been easy to simply accept this level of performance. Fortunately, the student decided to graph the times for individual RPCs. It turned out the data was bimodal, whereby every other RPC completed quickly, but the intervening ones were all significantly slower. With this information, the student tracked down and fixed a configuration error that eliminated all of the slow times. In this case, the average value was not a good indicator of system behavior.

The examples in this article may seem so esoteric that they must be outliers, but they are not. Every performance measurement project I have seen has had multiple such examples, which are extremely subtle, difficult to track down, and defy all intuition, until suddenly a simple explanation appears (such as unsigned integer underflow). Deeper measurements almost always produce substantial performance improvement, important discoveries about system behavior, or both.

Do not spend a lot of time agonizing over which deeper measurements to make. If the top-level measurements contain contradictions or things that are surprising, start with measurements that could help resolve them. Or pick measurements that will identify performance bottlenecks. If nothing else, choose a few metrics that are most obvious and easiest to collect, even if you are not sure they will be particularly illuminating. Once you look at the results, you will almost certainly find things that do not make sense; from this point on, track down and resolve everything that does not make perfect sense. Along the way you will discover other surprises; track them down as well. Over time, you will develop intuition about what kinds of deeper measurements are most likely to be fruitful.

It can be as fancy as an interactive webpage or as simple as a text file, but a dashboard is essential for any nontrivial measurement effort.

Measuring deeper is the single most important ingredient for high-quality performance measurement. Focusing on this one rule will prevent most of the mistakes anyone could potentially make. For example, in order to make deeper measurements you will have to allocate extra time. Measuring deeper will expose bugs and inconsistencies, so you will not accidentally trust bogus data. Most of the suggestions under Rule 2 (Never trust a number generated by a computer) are actually examples of measuring deeper. You will never need to guess the reasons for performance, since you will have actual data. Your measurements will not be superficial. Finally, you are less likely to be derailed by subconscious bias, since the deeper measurements will expose weaknesses, as well as strengths.

Measurement Infrastructure

Making good performance measurements takes time, so it is worth creating infrastructure to help you work more efficiently. The infrastructure will easily pay for itself by the time the measurement project is finished. Furthermore, performance measurements tend to be run repeatedly, making infrastructure even more valuable. In a cloud service provider, for example, measurements must be made continuously in order to maintain contractual service levels. In a research project, the full suite of performance measurements will be run several times (such as before submission, after the paper is accepted, and again during the writing of a Ph.D. dissertation). It is important to have infrastructure that makes it easy to rerun tests.

Automate measurements. It should be possible to type a single command line that invokes the full suite of measurements, including not just top-level measurements but also the deeper measurements. Each run should produce a large amount of performance data in an easy-to-read

form. It should also be easy to invoke a single benchmark by itself or vary the parameters for a benchmark. Also useful is a tool that can compare two sets of output to identify nontrivial changes in performance.

Create a dashboard. A dashboard is a display that shows all performance measurements from a particular run of a particular benchmark or from a deployed system. If you have been measuring deeply, the dashboard can easily show hundreds of measurements. A good dashboard brings together a lot of data in one place and makes it easy to examine performance from many angles. It can be as fancy as an interactive webpage or as simple as a text file, but a dashboard is essential for any nontrivial measurement effort.

Figure 1 shows approximately one-third of a dashboard my students created to analyze the performance of crash recovery in a distributed storage system.[3] In this benchmark, one of the system's storage servers has crashed, and several other servers ("recovery masters") reconstruct the lost data by reading copies stored on a collection of backup servers. This sample illustrates several important features of dashboards. Any dashboard should start with a summary section, giving the most important metric(s)—total recovery time in this case—and the parameters that controlled the benchmark. Each of the remaining sections digs deeper into one specific aspect of the performance. For example, "Recovery Master Time" analyzes where the recovery masters spent their time during recovery, showing CPU time for each component as both an absolute time and a percentage of total recovery time; the percentages help identify bottlenecks. It was important for the storage system being analyzed to make efficient use of the network during recovery, so we added a separate section to analyze network throughput for each of the servers, as well as for the cluster as a whole. Most measurements in the dashboard show averages across a group of servers, but in several cases the worst-case server is also shown. The dashboard also has a special section showing the worst-case performance in several categories, making it possible to see whether outliers are affecting overall performance.

You should create a simple dashboard as soon as you start making measurements; initially, it will include just the benchmark parameters and a few overall metrics. Every time you think of a new question to answer or a deeper measurement to take, add more data to the dashboard. Never remove metrics from the dashboard, even if you think you will never need them again. You probably will.

If you make a change and performance suddenly degrades, you can scan the dashboard for metrics that have changed significantly. The dashboard might indicate that, for example, the network is now overloaded or the fraction of time waiting for incoming segments suddenly increased. You can maintain a "good" dashboard for comparing with later dashboards and record dashboards at regular time intervals to track performance over long periods of time. A dashboard serves a purpose for performance measurement similar to that of unit tests for functional testing—providing a detailed analysis and making it easy to detect regressions.

Do not remove the instrumentation. Leave as much instrumentation as possible in the system at all times, so performance information is constantly available. For online services that run continuously, the dashboard should take the form of a webpage that can be displayed at any time. For applications that are run manually, it is convenient to have a command-line switch that will cause performance metrics to be recorded during execution and dumped when the application finishes.

One simple-yet-effective technique is to define a collection of counters that accumulate statistics (such as number of invocations of each externally visible request type and number of network bytes transmitted and received). Incrementing a counter is computationally inexpensive enough that a system can include a large number of them without hurting its performance. Make it easy to define new counters and read out all existing counters. For long-running services, it should be possible to sample the counters at regular intervals, and the dashboard should display historical trends for the counters.

Presentation matters. If you want to analyze performance in depth, measurements must be displayed in a way that exposes a lot of detail and allows it to be understood easily. In addition, the presentation must clarify the things

Figure 1. Excerpt from the dashboard used to evaluate crash recovery in a large-scale main memory storage system.[3]

```
=== Summary ===
Recovery time:                              2.58 s
Failure detection time:                     0.32 s
Recovery + detection time:                  2.90 s
Masters:                                      73
Backups:                                     146
Total nodes:                                  73
Replicas:                                      3
Objects per master:                       592950
Object size:                            1055.81 bytes
Total recovery segment entries:         43685317
Total live object space:                   43584 MB
Total recovery segment space w/ overhead:  43713 MB

=== Recovery Master Time ===
Total (90.5% of recovery time):       2333.64 ms avg / 2533.71 ms max /  100.00% avg
Waiting for incoming segments:         766.78 ms avg /  924.04 ms max /   32.86% avg
Inside recoverSegment:                1283.48 ms avg / 1657.36 ms max /   55.00% avg
Final log sync time:                    21.20 ms avg /   50.85 ms max /    0.91% avg
Removing tombstones:                     0.00 ms avg /    0.00 ms max /    0.00% avg
Other:                                 262.18 ms avg /  673.43 ms max /   10.17% avg

=== Network Utilization ===
Aggregate:                             994.43 Gb/s avg /    54.49%
Master in:                               4.57 Gb/s avg /    333.82 Gb/s total
Master out:                              6.35 Gb/s avg /    463.59 Gb/s total
Master out during replication:           7.61 Gb/s avg /    555.87 Gb/s total
Master out during log sync:             12.06 Gb/s avg /    880.42 Gb/s total
Backup in:                               3.63 Gb/s avg /    530.01 Gb/s total
Backup out:                              3.63 Gb/s avg /    529.98 Gb/s total

=== Slowest Servers ===
Backup opens, writes:                 rc26 / 729.2 ms
Stalled reading segs from backups:    rc21 / 924.0 ms
Reading from disk:                    rc29 / 170.3 MB/s
Writing to disk:                      rc23 /  71.3 MB/s
```

that are most important. Think of this as feeding your intuition. The way to discover interesting things is to absorb a lot of information and let your intuition go to work, identifying patterns, contradictions, and things that seem like they might be significant. You can then explore them in more detail.

When students bring their first measurements to me, the measurements are often in a barely comprehensible form (such as unaligned comma-separated values), telling me they did not want to spend time on a nice graph until they knew what data is important. However, the early phase of analysis, where you are trying to figure out what is happening and why, is when it is most important for information to be presented clearly. It is worth getting in the habit of always presenting data clearly from the start (such as with graphs rather than tables). Do not waste time with displays that are difficult to understand. Making graphs takes little time once you have learned how to use the tools, and you can reuse old scripts for new graphs. Consider clarity even when printing raw data, because you will occasionally want to look at it. Arrange the data in neat columns with labels, and use appropriate units (such as microseconds), rather than, say, "1.04e-07."

Figure 2 and Figure 3 show how the organization of a graph can have a big effect on how easy (or difficult) it is to visualize performance data. In Figure 2 my students and I aimed to understand tail latency (99.9th or 99.99th percentile worst-case performance) for write requests in the RAMCloud storage system. A traditional cumulative distribution function (CDF) like the one in Figure 2a emphasizes the mean value but makes it difficult to see tail latency. When we switched to a reverse cumulative distribution function with log-scale axes (see Figure 2b) the complete tail became visible, all the way out to the slowest of 100 million total samples. Figure 2b made it easy to see features worthy of additional study (such as the "shoulders" at approximately 70 µs and 1 ms); additional measurements showed the shoulder at 1 ms was caused by interference from concurrent garbage collection. If we had only considered a few discreet measurements of tail latency we might not have noticed these features.

Figure 2. Two different ways to display tail latency: (a) a traditional CDF with linear axes; and (b) a complementary CDF (each y-value is the fraction of samples greater than the corresponding x-value) with log-scale axes.

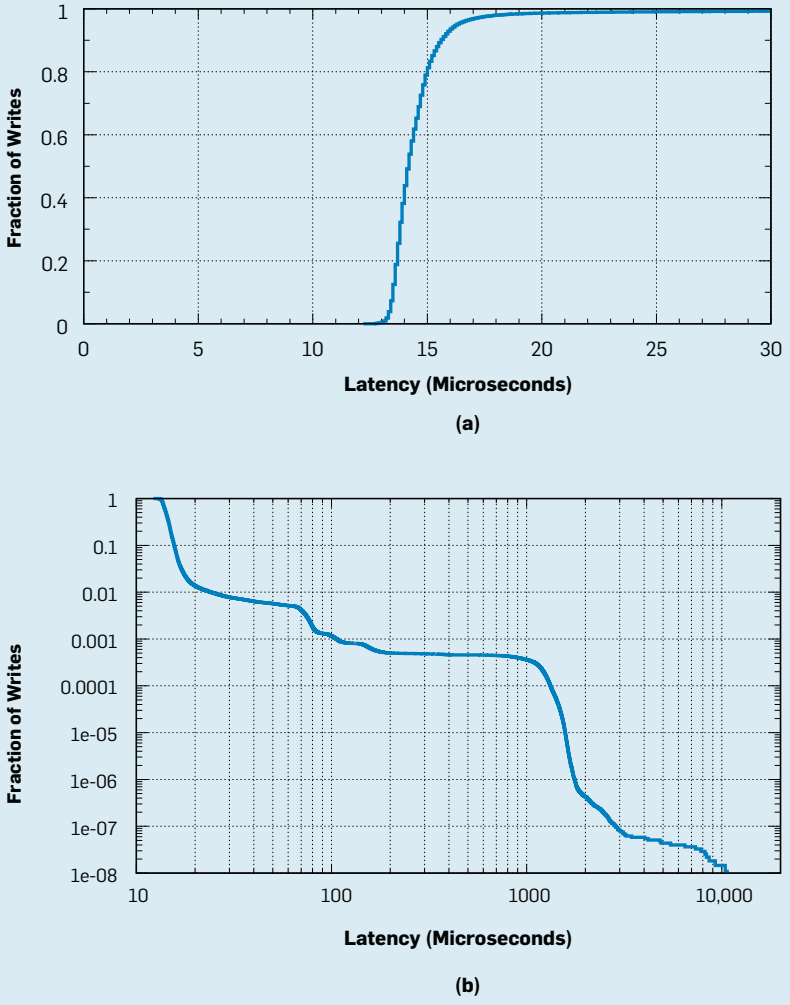
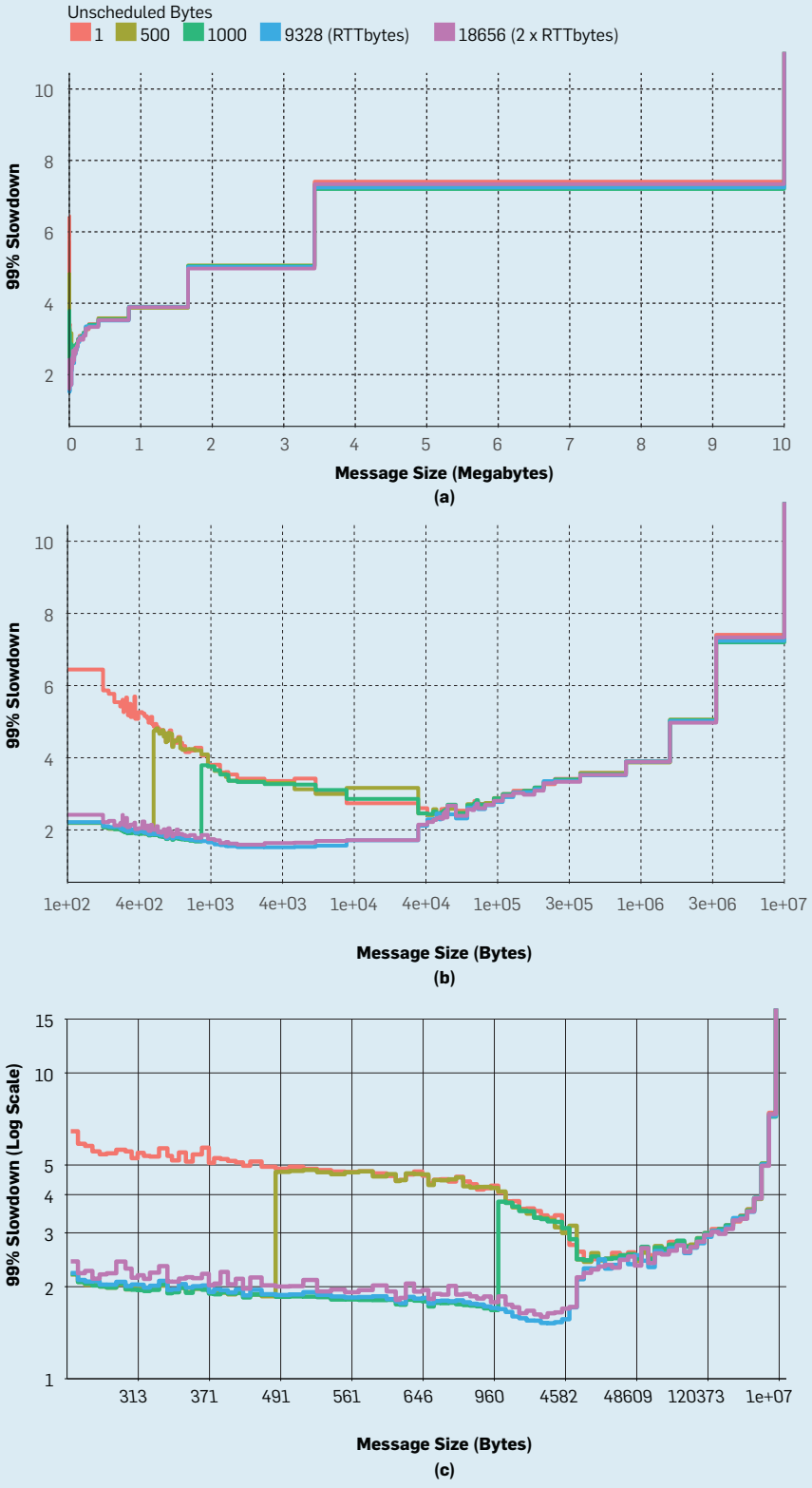


Figure 3 arose during development of a new network transport protocol. My students and I wanted to understand the effect of a particular parameter setting on the delivery time for messages of different size in a given workload. The first question we had to address in graphing the data was what metric to display. Displaying the absolute delivery times for messages would not be very useful, since it would not be obvious whether a particular time is good. Furthermore, comparisons between messages of different lengths would not be meaningful, as longer messages inherently take more time to deliver. Instead, we displayed slowdown, the actual delivery time for a message divided by the best possible time for messages of that size. This choice made it easy to see whether a particular time is indeed good; 1.0 is perfect, 2.0 means the message took twice as long as necessary, and so on. Slowdown also made it possible to compare measurements for messages of different length, since slowdown takes into account the inherent cost for each length.

The second question was the choice of the *x*-axis. An obvious choice would have been a linear *x*-axis, as in Figure 3a. However, the vast majority of messages is very small, so almost all the messages are bunched together at the left edge of that graph. A log-scale *x*-axis (see Figure 3b) makes it easier to see the small messages but still does not indicate how many messages were affected by each value of the parameter. To address this problem, we rescaled the *x*-axis to match the distribution of

Figure 3. Each figure displays 99th-percentile slowdown (delivery time for messages of a given size, divided by the best possible time for messages of that size) as a function of message size in a given workload: (a) *x*-axis is linear; (b) *x*-axis is logarithmic; and (c) *x*-axis is scaled to match the CDF of message lengths. Different curves correspond to different settings of the "unscheduled bytes" parameter.



message lengths (see Figure 3c); the *x*-axis is labeled with message size but is linear in number of messages, with each of the 10 tickmarks corresponding to 10% of all messages. With this view of the data it became easy to see that the parameter matters, as it affected approximately 70% of all messages in the experiment (those smaller than approximately 5 Kbytes).

Figure 3c includes more information than the other graphs; in addition to displaying slowdown, it also displays the CDF of message sizes via the *x*-axis labels. As a result it is easy to see that messages in this workload are mostly short; 60% of all messages require no more than 960 bytes. Figure 3c makes it clear that Figure 3a and Figure 3b are misleading.

Conclusion

The keys to good performance evaluation are a keen eye for things that do not make sense and a willingness to measure from many different angles. This takes more time than the quick and shallow measurements that are common today but provides a deeper and more accurate understanding of the system being measured. In addition, if you apply the scientific method, making and testing hypotheses, you will improve your intuition about systems. This will result in both better designs and better performance measurements in the future.

Acknowledgments

This article benefited from comments and suggestions from Jonathan Ellithorpe, Collin Lee, Yilong Li, Behnam Montazeri, Seojin Park, Henry Qin, Stephen Yang, and the anonymous *Communications* reviewers.

References

1. Nickerson, R.S. Confirmation bias: A ubiquitous phenomenon in many guises. *Review of General Psychology 2*, 2 (June 1998), 175–220.
2. Ousterhout, J. *A Philosophy of Software Design*. Yaknyam Press, Palo Alto, CA, 2018.
3. Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S.J., Qin, H., Rosenblum, M. et al. The RAMCloud storage system. *ACM Transactions on Computer Systems 33*, 3 (Aug. 2015), 7.
4. Rosenblum, M. and Ousterhout, J.K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 26–52.

John Ousterhout (ouster@cs.stanford.edu) is the Bosack Lerner Professor of Computer Science at Stanford University, Stanford, CA, USA.

© 2018 ACM 0001-0782/18/7 $15.00