DOI:10.1145/2408776.2408791

Article development led by acmqueue
queue.acm.org

The USE method addresses shortcomings in other commonly used methodologies.

BY BRENDAN GREGG

Thinking Methodically about Performance

PERFORMANCE ISSUES CAN be complex and mysterious, providing little or no clue to their origin. In the absence of a starting point—or a methodology to provide one—performance issues are often analyzed randomly: guessing where the problem may be and then changing things until it goes away. While this can deliver results—if you guess correctly—it can also be time consuming, disruptive, and may ultimately overlook certain issues. This article describes system-performance issues and the methodologies in use today for analyzing them, and it proposes a new methodology for approaching and solving a class of issues.

Systems-performance analysis is complex because of the number of components and their interactions in a typical system. An environment may be composed of databases, Web servers, load balancers, and custom applications, all running upon operating systems—either bare-metal or virtual. And that is just the software. Hardware and firmware, including external storage systems and network infrastructure, add many more components to the environment, any of which is a potential source of issues. Each of these components may require its own field of expertise, and a company may not have staff knowledgeable in all the components in its environment.

Performance issues may also arise from complex interactions between components that work well in isolation. Solving this type of problem may require multiple domains of expertise to work together.

As an example of such complexity within an environment, consider a mysterious performance issue we encountered at Joyent for a cloud-computing customer: the problem appeared to be a memory leak, but from an unknown location. This was not reproducible in a lab environment with the components in isolation. The production environment included the operating system and system libraries, the customer's own application code written in node.js, and a Riak database running on the Erlang VM (virtual machine). Finding the root cause required knowledge of the customer's code, node.js, Riak, Erlang, and the operating system, each of which was provided by one or more different engineers. The problem turned out to be in the system library, identified by engineers with operating-systems expertise.

Another complicating factor is that "good" or "bad" performance can be subjective: what may be unacceptable latency for one user may be acceptable for another. Without a means of clearly identifying issues, it can be difficult to know not only if an issue is present, but also when it is fixed. The ability to measure performance issues—for example, as an expression of response time—allows them to be quantified and different issues ranked in order of importance.

Performance-analysis methodology can provide an efficient means of analyzing a system or component and identifying the root cause(s) of problems, without requiring deep expertise. Methodology can also provide ways of identifying and quantifying issues, allowing them to be known and ranked.

Performance texts have provided methodologies for various activities, such as capacity planning,[1,16] benchmarking,[18] and modeling systems.[7,8,10] Methodologies for finding the root causes of performance issues, however, are uncommon. One example is the drill-down analysis method introduced in Solaris Performance and Tools,[13] which describes a three-stage procedure for moving from a high-level symptom down to analyzing the cause. These texts have typically covered analysis by use of ad hoc checklists of recent tips and tuning, and by teaching operating-systems internals and tools.[2,11,12,15] The latter allows performance analysts to develop their own methodologies, although this can take considerable time to accomplish.

Ad hoc performance checklists have been a popular resource. For example, *Sun Performance and Tuning*[2] includes "Quick Reference for Common Tuning Tips," which lists 11 tips, intended to find disk bottlenecks, network file system (NFS), memory, and CPU issues, and is both easy to follow and prescriptive. Support staff groups often use these lists because they provide a consistent check of all items, including the most egregious issues. This approach poses some problems, however. Observability is limited to the specific items in the list, and they are usually point-in-time recommendations that go out of date and require updates. These checklists also focus on issues for which there are known fixes that can be easily documented, such as the setting of tunable parameters, but not custom fixes to the source code or environment.

In this article I summarize several other methodologies for systems-performance analysis, including the USE method, which is explained in detail. I begin by describing two commonly used *anti*-methodologies—the blame-someone-else anti-method and the streetlight anti-method—that serve as comparisons with later methodologies.

The "Blame-Someone-Else" Anti-Method

The first anti-methodology follows these simple steps:

1. Find a system or environment component you are not responsible for.
2. Hypothesize that the issue is with that component.
3. Redirect the issue to the responsible team.
4. When proven wrong, go back to step 1.

For example, "Maybe it's the network. Can you check with the network team to see if they have had dropped packets or something?"

Instead of investigating performance issues, this methodology makes them someone else's problem, which can be wasteful of other teams' resources. A lack of data analysis—or even data to begin with—leads to the hypothesis. Ask for screen shots showing which tools were run and how their output was interpreted. These can be taken to someone else for a second opinion.

The Streetlight Anti-Method

While running tools and collecting data is better than wild hypotheses, it is not sufficient for effective performance analysis, as shown by the streetlight anti-method. This is the absence of any deliberate methodology. The user analyzes performance by selecting observability tools that are familiar, found on the Internet, or found at random and then seeing whether anything obvious shows up. This hit-or-miss approach can overlook many types of issues.

Finding the right tool can take a while. The most familiar tools are run first, even if they do not make the most sense. This is related to an observational bias called the *streetlight effect*,[17] named after a parable:

A policeman sees a drunk hunting for something under a streetlight and asks what he is looking for. The drunk says he has lost his keys. The policeman cannot find them either, and asks if he lost them under the streetlight. The drunk replies: "No, but this is where the light is best."

The performance equivalent would be looking at top(1), not because it makes sense but because the user does not know how to read other tools.

Learning more tools helps but is still a limited approach. Certain system components or resources may be overlooked because of a lack of observability tools or metrics. Furthermore, the user, unaware that the view is incomplete, has no way of identifying "unknown unknowns."

Better performance-analysis methodologies are available that may solve issues before you run any tools at all. These include the problem statement method, workload characterization, and drill-down analysis.

Problem Statement Method

The problem statement method, commonly used by support staff for collecting information about a problem, has been adapted for performance analysis.[9] It can be the first methodology attempted for performance issues.

The intent is to collect a detailed description of the issue—the problem statement—that directs deeper analysis. The description on its own may even solve the issue. This is typically entered into a ticketing system by asking the following questions:

- What makes you think there is a performance problem?
- Has this system ever performed well?

▸ What has changed recently? (Software? Hardware? Load?)

▸ Can the performance degradation be expressed in terms of latency or runtime?

▸ Does the problem affect other people or applications (or is it just you)?

▸ What is the environment? What software and hardware is used? Versions? Configuration?

These questions may be customized to the environment. While the questions may seem obvious, the answers often resolve a class of issues, requiring no deeper methodologies. When that is not the case, other methodologies can be called into service, including workload characterization and drill-down analysis.

The Workload Characterization Method

The workload can be characterized by answering questions such as:

▸ Who is causing the load? Process ID, user ID, remote IP address?

▸ Why is the load being called? Code path?

▸ What are other characteristics of the load? IOPS, throughput, type?

▸ How is the load changing over time?

This helps to separate *problems of load from problems of architecture*, by identifying the former.

The best performance wins often arise from eliminating unnecessary work. Sometimes these bottlenecks are caused by applications malfunctioning (for example, a thread stuck in a loop) or bad configurations (systemwide backups running during the day). With maintenance or reconfiguration, such unnecessary work can be eliminated. Characterizing the load can identify this class of issue.

The Drill-Down Analysis Method

Drill-down analysis involves peeling away layers of software and hardware to find the core of the issue—moving from a high-level view to deeper details. These deeper details may include examining kernel internals—for example, by using profiling to sample kernel stack traces, or dynamic tracing to examine the execution of kernel functions.

Solaris Performance and Tools[13] provides a drill-down analysis methodology for system performance. It follows three stages:

▸ *Monitoring.* This continually records high-level statistics over time across many systems, identifying or alerting if a problem is present.

▸ *Identification.* Given a system with a suspected problem, this narrows the investigation to particular resources or areas of interest using system tools and identifying possible bottlenecks.

▸ *Analysis.* This stage provides further examination of particular system areas, identifying the root cause(s) and quantifying the issue.

The analysis stage may follow its own drill-down approach, beginning with applications at the top of the software stack and drilling down into system libraries, system calls, kernel internals, device drivers, and hardware.

While drill-down analysis often pinpoints the root cause of issues, it can be time consuming, and when drilling in the wrong direction, it can waste a great deal of time.

"Good" or "bad" performance can be subjective: what may be unacceptable latency for one user may be acceptable for another. Without a means of clearly identifying issues, it can be difficult to know not only if an issue is present, but also when it is fixed.

The Need for a New Methodology

I recently analyzed a database performance issue on the Joyent public cloud, which began with a ticket containing a problem statement as described in the previous section. The statement indicated that there was a real issue that needed deeper analysis.

The issue had been intermittent, with some database queries taking seconds to complete. The customer blamed the network, hypothesizing that the query latency was caused by dropped network packets. This was not a wild hypothesis, as the ticket included output from `ping(1)` showing occasional high latency; `ping(1)` is a common and familiar tool, however, and with no other supporting evidence, this seemed to be an example of the streetlight anti-method.

The support team ran tools to investigate the network in much more detail, including examining TCP/IP stack network counters, without finding any problems. This analysis took time because there are dozens of such statistics, some of which are difficult to interpret and must be examined over time to look for correlations. While logged into the systems, the team also checked CPU usage vs. the cloud-imposed limits, following their own ad hoc checklist of common issues. Their conclusion was that there was no issue while they were watching: the network and CPUs were fine.

At this point, many system components and tens of thousands of system statistics had not yet been checked, as they were assumed to be unrelated to the issue. Without a direction to follow, checking everything across all systems in the customer's cloud environment could take days. The analysis to date had not found any evidence of a real issue, which was discouraging.

The next step was to try dynamic tracing of the originally reported problem (network packet drops), in the hope of finding something that the standard network counters had missed. I have used the DTrace tool many times to perform drill-down analysis of the TCP/IP stack. This can provide many details beyond the standard network observability toolset, including inspection of kernel-dropped packets and internal TCP state. It still can take hours to catch intermittent issues, however. I was tempted to begin drill-down analysis from the database query latency, in case the issue was not network related, or to begin characterizing the database workload over time, in case the problem was caused by a burst of load, but these approaches are also time consuming.

Before beginning deeper analysis, I wanted to perform a quick check of all system components, not just the network and CPUs, to look for bottlenecks or errors. For this to be quick, it would need to check only a limited number of statistics per system, not the tens of thousands available. And for this to be complete, it would need to check all components, including those that might be missed because they have no observability tools or statistics by default.

The utilization, saturation, and errors (USE) method provided one way of doing this. It quickly revealed that the database system was out of memory and was paging, and that the disks were occasionally running at saturation. Focusing troubleshooting efforts on networking early on had meant these areas were overlooked in the team's analysis. The real issues were in the system memory and disks, which were much quicker to read and interpret.

I developed the USE method while teaching classes in operating-systems performance. The goal was to help my students find common issues and to ensure that they were not overlooking important areas. I have used it successfully many times in enterprise and cloud-computing environments, but it does not solve all types of problems and should be treated as just one methodology in the toolbox.

The USE Method

The USE method is intended to be used early in a performance investigation, after the problem-statement method, to identify systemic bottlenecks quickly. It can be summarized as: *For every resource, check utilization, saturation, and errors.*

Resource in this case means all physical server functional components (CPUs, disks, buses, and so on) examined individually. Some software resources can be examined using the same methodology, provided the metrics make sense.

Utilization is the percentage of time that the resource is busy servicing work during a specific time interval. While busy, the resource may still be able to accept more work; the degree to which it cannot do so is identified by *saturation*. That extra work is often waiting in a queue.

For some resource types, including main memory, utilization is the *capacity* of the resource that is used. This is different from the time-based definition. Once a capacity resource reaches 100% utilization, no more work can be accepted, and it either queues the work (saturation) or returns errors, either of which is identified by the USE method.

Errors in terms of the USE method refer to the count of error events. Errors should be investigated because they can degrade performance, and they may not be immediately noticed when the failure mode is recoverable. This includes operations that fail and are retried, as well as devices that fail in a pool of redundant devices.

In contrast to the streetlight anti-method, the USE method iterates over system resources instead of starting with tools. This creates a complete list of questions to ask, and only then searches for the tools to answer them. Even when tools cannot be found to answer the questions, the knowledge that these questions are unanswered can be extremely useful for the performance analyst: they are now "known unknowns."

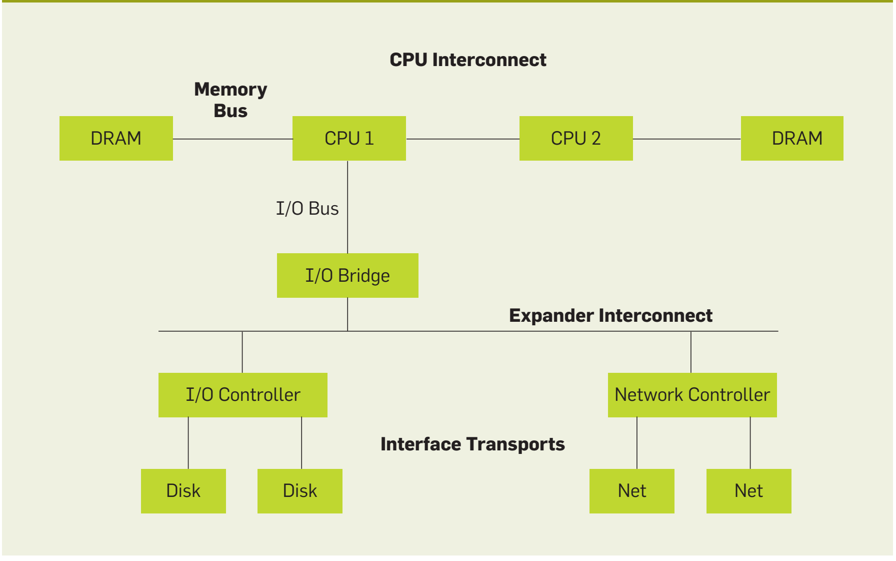
The USE method also directs analysis to a limited number of key metrics, so that all system resources are checked as quickly as possible. After this, if no issues have been found, you can turn to other methodologies.

The key metrics of the USE method are usually expressed as follows:

- Utilization as a percentage over a time interval (for example, one CPU is running at 90% utilization).
- Saturation as a wait queue length (for example, the CPUs have an average run queue length of four).
- Errors as the number of errors reported (for example, the network interface has had 50 late collisions).

It is also important to express the time interval for the measurement. Though it may seem counterintuitive, a

Figure 1. A two-socket system.



short burst of high utilization can cause saturation and performance issues, even though the overall utilization is low over a long interval. Some monitoring tools report utilization as five-minute averages. CPU utilization, for example, can vary dramatically from second to second, and a five-minute average can disguise short periods of 100% utilization and, therefore, saturation.

The first step in the USE method is to create a list of resources. Try to be as complete as possible. Here is a generic list of server hardware resources, with specific examples:

- **CPUs**—Sockets, cores, hardware threads (virtual CPUs).
- **Main memory**—DRAM.
- **Network interfaces**—Ethernet ports.
- **Storage devices**—Disks.
- **Controllers**—Storage, network.
- **Interconnects**—CPU, memory, I/O.

Each component typically acts as a single resource type. For example, main memory is a *capacity* resource, and a network interface is an I/O resource, which can mean either IOPS (I/O operations per second) or throughput. Some components can behave as multiple resource types—for example, a storage device is both an I/O resource and a capacity resource. Consider all types that can lead to performance bottlenecks. Also note that I/O resources can be further studied as *queueing systems*, which queue and then service these requests.

Some physical components can be left off your checklist, such as hardware caches (for example, MMU TLB/TSB, CPU Level-1/2/3). The USE method is most effective for resources that suffer performance degradation under high utilization or saturation, leading to bottlenecks; caches *improve* performance under high utilization.

Cache hit rates and other performance attributes can be checked after the USE method has been applied—that is, after systemic bottlenecks have been ruled out. If you are unsure whether to include a resource, go ahead and include it and then see how well the metrics work in practice.

Function Block Diagram

Another way to iterate over resources is to find or draw a function block diagram[3] for the system. This type of diagram also shows relationships, which can be very useful when looking for bottlenecks in the flow of data. Figure 1 is a generic diagram showing a two-socket system.

While determining utilization for the various buses, annotate each one on the functional diagram with its maximum bandwidth. The resulting diagram may pinpoint systemic bottlenecks before a single measurement has been taken. (This is also a useful exercise during hardware product design, while you still have time to change physical components.)

CPU, memory, and I/O interconnects are often overlooked. Fortunately, they are not commonly the cause of system bottlenecks. Unfortunately, when they are, the problem can be difficult to solve (maybe you can upgrade the main board or reduce load (for example, "zero copy" projects lighten memory bus load). At least the USE method takes interconnect performance into consideration. (See Analyzing the HyperTransport[4] for an example of an interconnect issue identified in this way.)

Once you have your list of resources, consider the types of metrics you need for each (utilization, saturation, and errors). Table 1 lists some example resources and metric types, along with possible metrics (from generic Unix/Linux). These metrics can be expressed either as averages per interval or as counts.

Repeat for all combinations and include instructions for fetching each metric. Take note of metrics that are not currently available: these are the "known unknowns." You will end up with a list of about 30 metrics, some of which are difficult to measure and some of which cannot be measured at all. Example checklists have been built for Linux- and Solaris-based systems.[5,6]

Fortunately, the most common issues are usually found with the easier metrics (for example, CPU saturation, memory capacity saturation, network interface utilization, disk utilization), so these can be checked first.

Table 2 lists some examples of harder combinations. Some of these metrics may not be available from

Table 1. Resources and metric types.

Resource	Type	Metric
CPU	utilization	CPU utilization (ideally per CPU)
CPU	saturation	dispatcher queue length (aka run-queue length)
Memory	utilization	available free memory (systemwide)
Memory	saturation	anonymous paging or thread swapping ("page scanning" is another indicator)
Network interface	utilization	RX/TX throughput / max bandwidth
Storage device I/O	utilization	device busy percent
Storage device I/O	saturation	wait queue length
Storage device I/O	errors	device errors ("soft," "hard")

Table 2. Harder combinations.

Resource	Type	Metric
CPU	errors	correctable CPU cache ECC events or faulted CPUs (if the OS+HW supports that)
Memory	errors	failed malloc()s (although this is usually because of virtual memory exhaustion, not physical)
Network	saturation	saturation-related NIC or operating-system errors (for example, Solaris "nocanputs")
Storage Controller	utilization	depends on the controller; it may have a max IOPS or throughput that can be checked vs. current activity
CPU interconnect	utilization	per port throughput/max bandwidth (CPU performance counters)
Memory interconnect	saturation	memory stall cycles, high CPI (CPU performance counters)
I/O interconnect	utilization	bus throughput/max bandwidth (performance counters may exist on your hardware; for example, Intel "uncore" events)

standard operating-system tools. I often have to write my own software for such metrics, using either static or dynamic tracing (DTrace) or the CPU performance counter facility.

Some software resources can be similarly examined. This usually applies to smaller components of software, not to entire applications. For example:

‣ **Mutex locks.** Utilization may be defined as the time the lock was held, saturation by those threads queued waiting on the lock.

‣ **Thread pools.** Utilization may be defined as the time threads were busy processing work, saturation by the number of requests waiting to be serviced by the thread pool.

‣ **Process/thread capacity.** The system may have a limited number of processes or threads whose current usage may be defined as utilization; waiting on allocation may indicate saturation; and errors occur when the allocation fails (for example, "cannot fork").

‣ **File descriptor capacity.** This is similar to the above, but for file descriptors.

If the metrics work well, then use them; otherwise, software troubleshooting can be left to other methodologies.

Suggested Interpretations

The USE method helps identify which metrics to use. After you learn how to read them from the operating system, your next task is to interpret their current values. For some metrics, interpretation may be obvious (and well documented). Others are not so obvious and may depend on workload requirements or expectations. Here are some general suggestions for interpreting metric types:

‣ **Utilization.** 100% utilization is usually a sign of a bottleneck (check saturation and its effect to confirm). High utilization (for example, beyond 60%) can begin to be a problem for a couple of reasons. First, when utilization is measured over a relatively long time period (multiple seconds or minutes), a total utilization of, say, 60% can hide short bursts of 100% utilization. Second, some system resources, such as hard disks, usually cannot be interrupted during an operation, even for higher-priority work. Compare this with CPUs, which can be interrupted ("preempted") at almost any moment. Once disk utilization is above 60%, queueing delays can become more frequent and noticeable, as the tail of the queue becomes longer. This can be quantified using queuing theory to model response time vs. utilization (for example, modeling a disk as M/M/1).

‣ **Saturation.** Any degree of saturation can be a problem (non-zero). This may be measured as the length of a wait queue or time spent waiting on the queue.

‣ **Errors.** Non-zero error counters are worth investigating, especially if they are still increasing while performance is poor.

It is easy to interpret the negative cases: low utilization, no saturation, no errors. This is more useful than it sounds. Narrowing the scope of an investigation can help you focus quickly on the problem area.

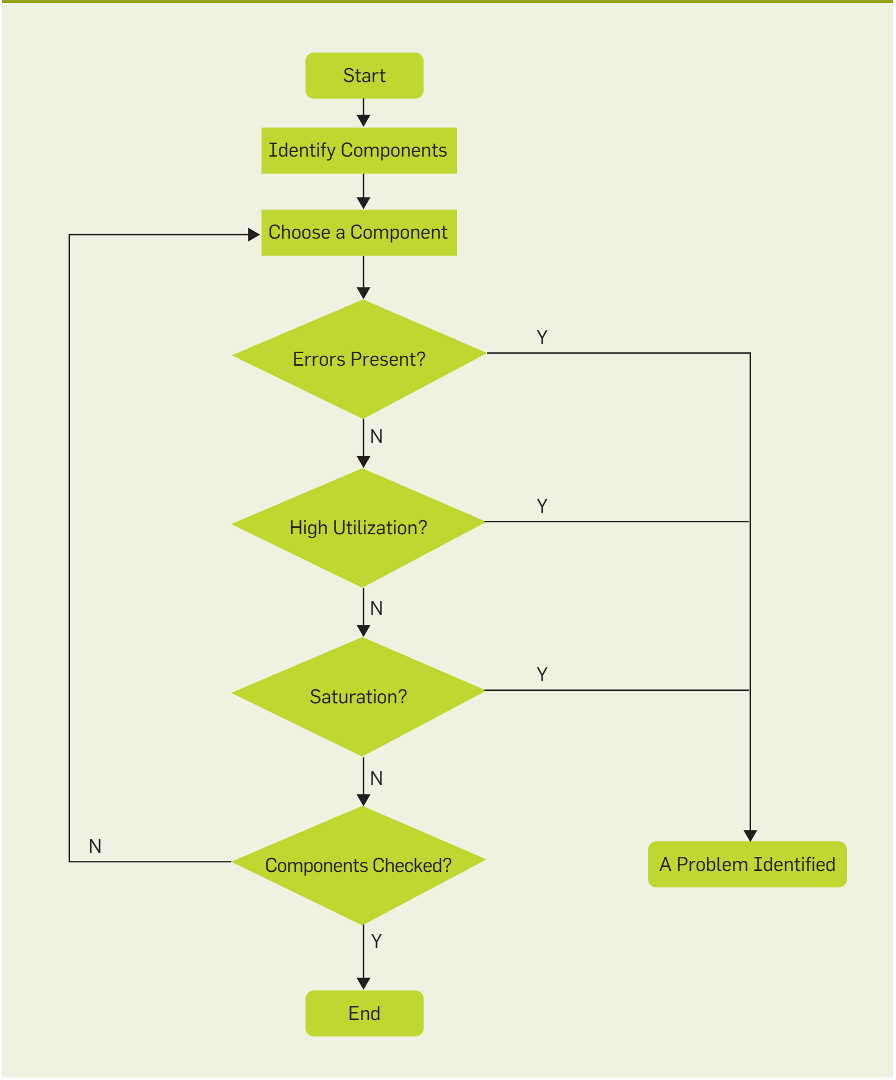
Cloud Computing

In a cloud-computing environment, software resource controls may be in place to limit or throttle tenants who are sharing one system. At Joyent, we primarily use operating-system virtualization (the SmartOS-based SmartMachine), which imposes memory and CPU limits, as well as storage I/O throttling. Each of these resource limits can be examined with the USE method, similar to examining the physical resources.

For example, in our environment *memory capacity utilization* can be the tenant's memory usage vs. its memory cap. *Memory capacity saturation* can be seen by anonymous paging activity, even though the traditional Unix page scanner may be idle.

‣ **Strategy.** The USE method is pictured as a flowchart in Figure 2. Errors come first, because they are usually easier and quicker to interpret than utilization and saturation.

Figure 2. The USE method.



The USE method identifies problems that are likely to be system bottlenecks. Unfortunately, a system may be suffering more than one performance problem, so the first issue you find may be a problem but not the problem. You can investigate each discovery using further methodologies before returning to the USE method as needed to iterate over more resources. Or you may find it more efficient to complete the USE method checklist first and list all problems found, then to investigate each based on its likely priority.

Methodologies for further analysis include the workload characterization method and drill-down analysis method, summarized earlier. After completing these (if needed), you should have evidence to determine whether the corrective action needed is to adjust the load applied or to tune the resource itself.

While the previous methodologies may solve most server issues, latency-based methodologies (for example, Method R[14]) can approach finding 100% of all issues. These methodologies, however, can take much more time if you are unfamiliar with software internals, and they may be more suited for database administrators or application developers who already have this familiarity.

Conclusion

Systems-performance analysis can be complex, and issues can arise from any component, including from interactions among them. Methodologies in common use today sometimes resemble guesswork: trying familiar tools or posing hypotheses without solid evidence.

The USE Method was developed to address shortcomings in other commonly used methodologies and is a simple strategy for performing a complete check of system health. It considers all resources so as to avoid overlooking issues, and it uses limited metrics so that it can be followed quickly. This is especially important for distributed environments including cloud computing, where many systems may need to be checked. This methodology will, however, find only certain types of issues—bottlenecks and errors—and should be considered as one tool in a larger methodology toolbox.

In a cloud-computing environment, software resource controls may be in place to limit or throttle tenants who are sharing one system.

Acknowledgments

Carry Millsap's methodology research including Method R has been inspirational and my impetus for documenting system methodologies. Thanks to Bryan Cantrill for helping with this article and for fathering DTrace—which has allowed system methodologies to be developed and used in practice far beyond what traditional observability allows. And thanks to Deirdré Straughan for editing and feedback.

Related articles on queue.acm.org

The Price of Performance
Luiz André Barroso
http://queue.acm.org/detail.cfm?id=1095420

Performance Anti-Patterns
Bart Smaalders
http://queue.acm.org/detail.cfm?id=1117403

Thinking Clearly about Performance
Cary Millsap
http://queue.acm.org/detail.cfm?id=1854041

References

1. Allspaw, J. *The Art of Capacity Planning.* O'Reilly, 2008.
2. Cockcroft, A. *Sun Performance and Tuning.* Prentice Hall, NJ, 1995.
3. Function block diagram; http://en.wikipedia.org/wiki/Function_block_diagram.
4. Gregg, B. 7410 hardware update, and analyzing the HyperTransport; http://dtrace.org/blogs/brendan/2009/09/22/7410-hardware-update-and-analyzing-thehypertransport/.
5. Gregg, B. The USE method: Linux performance checklist, 2012; http://dtrace.org/blogs/brendan/2012/03/07/the-use-method-linux-performance-checklist/.
6. Gregg, B. The USE method: Solaris performance checklist, 2012; http://dtrace.org/blogs/brendan/2012/03/01/the-use-method-solaris-performance-checklist/.
7. Gunther, N. *Guerrilla Capacity Planning.* Springer, 2007.
8. Gunther, N. *The Practical Performance Analyst.* McGraw Hill, 1997.
9. Hargreaves, A. I have a performance problem, 2011; http://alanhargreaves.wordpress.com/2011/06/27/i-have-a-performance-problem/.
10. Jain, R. *The Art of Computer Systems Performance Analysis.* Wiley, 1991.
11. Loukidas, M. *System Performance Tuning.* O'Reilly, 1990.
12. McDougall, R. and Mauro, J. Solaris Internals—*Solaris 10 and OpenSolaris Kernel Architecture.* Prentice Hall, 2006.
13. McDougall, R., Mauro, J. and Gregg, B. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris.* Prentice Hall, 2006.
14. Millsap, C. and Holt, J. *Optimizing Oracle Performance.* O'Reilly, 2003.
15. Musumeci, G.D. and Loukidas, M. *System Performance Tuning*, 2nd Edition. O'Reilly, 2002.
16. Schlossnagle, T. *Scalable Internet Architectures.* Sams Publishing, 2006.
17. Streetlight effect; http://en.wikipedia.org/wiki/Streetlight_effect.
18. Wong, B. *Configuration and Capacity Planning for Solaris Servers.* Prentice Hall, 1997.

Brendan Gregg is the lead performance engineer at Joyent, where he analyzes performance and scalability at any level of the software stack. He is the primary author of *DTrace* (Prentice Hall, 2011), and co-author of *Solaris Performance and Tools* (Prentice Hall, 2006), and numerous articles about systems performance. He was previously a performance lead and kernel engineer at Sun Microsystems, where he developed the ZFS L2ARC.