Diagnosing Symptoms And Solving Problems

How to Take a Doctor's Approach to Exploiting Performance Bottlenecks

Is your software performing poorly? It may be time to call in a specialist.

By Scott Barber

Suppose that one afternoon, you notice that you are feeling sluggish, your throat is a little sore, and you aren't doing your best work. You head home early and make an appointment with your family doctor. When you go to the doc, she pokes, prods, asks questions and performs a variety of other tests to determine what's wrong. Finally, she determines that it could be a sore throat, but based on your history she refers you to a specialist. The next day you find yourself in the office of the right specialist, who diagnoses the cause of your symptoms and gives you a combination of advice and prescriptions to help you return to peak performance.

Sound familiar? It's the same with performance testing. You, as a performance tester, observe that the application isn't performing properly, so you investigate a little further to make sure it's really the application and not just a fluke. After you convince yourself that the symptoms are real, you find a developer—the family doc, so to speak—and describe or demonstrate the symptoms. The developer has a pretty good idea about the type of thing that may be causing the trouble, but he isn't certain, so he sends you to the expert on that area of the application (the specialist), who finally diagnoses and resolves the problem, not just the symptoms.

That is what this article is about: the thought process behind determining the actual cause of observed performance issues. If you read my previous article "How to Identify the Usual Performance Suspects" (May 2005, pp. 16-22; www.stpmag.com/backissues.htm), you may have already compiled a fairly substantial catalog of information about your symptoms of poor performance, which we categorized as failures, slow spots or bottlenecks—yet it's likely that you still don't have enough information to effectively diagnose the problem.

Scott Barber is CTO of PerfTestPlus Inc. His specialty is context-driven performance testing and analysis for distributed multiuser systems. Contact him at sbarber@perftestplus.com.

A Refresher on *n*-Tier Architecture

"All parts should go together without forcing. You must remember that the parts you are reassembling were disassembled by you. Therefore, if you can't get them together again, there must be a reason. By all means, do not use a hammer."—IBM maintenance manual, 1925

Before we can really dig into chasing bottlenecks to and into a specific tier, we should review some *n*-tier architecture basics. One of the things that confused me early in my performance testing career was the difference between the logical and the physical architecture of a system. I remember one meeting where the developers were talking about the "authentication server." I walked over to the network diagram and asked, "Which of these machines is the authentication server?" In a dismissive tone I was told, "None of them." Not easily discouraged, I asked, "Then where is the authentication server?" To which a developer replied, "It's on Web1 and Web4." If that response confuses you as much as it confused me at the time, the rest of this section is for you.

Logical architecture. Architecture used to be easy. Either you had a client/server (two-tier) application or you had a Web-based application (normally three-tier). During the early days of three-tier architectures, the tiers often corresponded to physical machines (as shown in Figure 1), whose roles were defined as follows:

Client tier (the user's machine): Presents requested data.

Presentation tier (the Web server): Handles all business logic and serves data to the client.

Data storage tier (the database server): Maintains data used by the system, typically in a relational database.

The machine that made up the presentation tier came to be known as the Web server because it ran the software used to "serve Web pages."

At first, as architectures became more complex, individual machines were added whenever a new tier was needed. Later, tiers began to be made up of clusters of machines that served the same role (see Figure 2).

The truth of the matter is that no one actually uses the term "file storage tier." They refer to that functionality as "the file server," for the same reason that the presentation tier became synonymous with "Web server" for Web-based applications.

The key to understanding a logical architecture is simply this: In a logical architecture, each tier contains a unique set of functionality that's logically separated from the other tiers. But even if a tier is commonly referred to with the word "server," it's not safe to assume that every tier lives on its own dedicated machine.

Physical architecture. So, you may ask, what does the actual physical environment look like? That's an important question when it comes to performance testing—and one that most developers and stakeholders find hard to believe matters to the performance tester. The paradigm that most stakeholders and developers hold to is that "Testers don't need to know anything but how to access the system from the client machine." This is simply not true when it comes to performance testing. Be persistent and patient in your quest for information. Over time, they'll come to understand.

I've called this section "Physical Architecture," but that's actually one of the least-used terms for what we're talking about. Probably the most commonly used term is "environment" (that is, the test environment or hardware environment); it may also be called the "network architecture." Whichever name your organization uses, what we're referring to here is represented in diagrams where actual, physical, tangible computers are shown and labeled with the roles they play, along with the other actual, physical, tangible computers they talk to. Figure 3 shows the physical architecture of the system we looked at logically in Figure 2.

Figure 3, minus the color overlays and tier labels, is very similar to the diagram I was looking at when I asked the question "Where's the authentication server?" I'm sure you now understand my confusion a little better, since there's no machine in Figure 3 labeled "Authentication Server." Instead of trying to explain verbally how the authentication server relates to the physical architecture, I've included tier labels with color coding in Figure 3.

What we see here is that most logical tiers consist of more than one physical machine (often called clusters). We also see that the machines that make up the presentation tier (Web1 through Web4) are all serving double duty as either an authentication-tier server or a file-storage-tier server. As it turns out, it's just about as common for a logical tier to be spread over several machines as it is for a physical machine to host the functionality of more than one logical tier. It should be clear that if you can't identify which tier is holding up progress, tuning becomes an exercise in trial and error. So the next question is, how do you figure out which tier is causing the issue?

Capturing Resource Utilization And Response Time by Tier

Most operating systems come with resource-monitoring software, like Perfmon for Microsoft and PerfMeter for Solaris, to assist with this task. There are many other resource-monitoring tools on the market. It's usually just best to ask your developers and administrators which tools they're using, and use them. Note that the challenge when using a resource-monitoring tool in conjunction with your load-generation tool is to properly correlate your results, so work with your systems administrators on this.

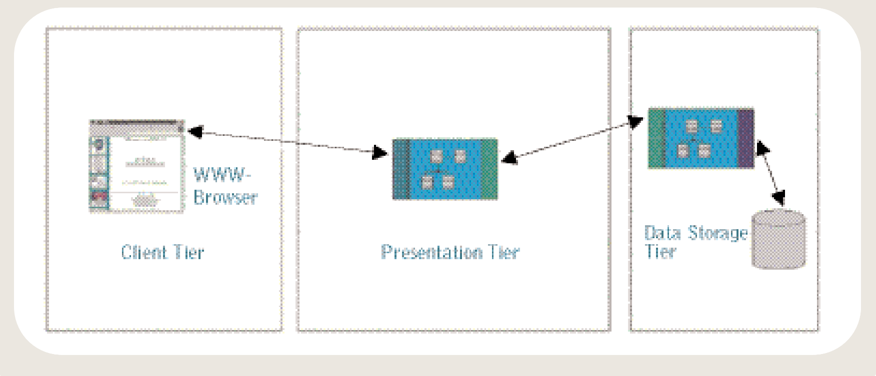
There are several ways to capture response time by tier involving either special tools or instrumentation. I should caution you that tools are generally very specific to your application architecture. These "activity timing" tools are commonly known as performance analysis tools and/or performance profilers. The methods involving instrumentation require close coordination with your developers and administrators and are also very specific to your application.

Because both methods are so specific to the environment you're testing and the tier you want to isolate, it's beyond the scope of this article to go into greater detail. Instead, allow me to describe a third method employing the load-generation tool you already have.

Let's assume that we have a simple three-tier system like the one shown in Figure 1. During our initial tests, our load-generation machine was located between the Client Tier and the Presentation Tier (the Web server).

Now, let's further assume that through testing we've determined that only transactions that interact with the database cause symptoms of poor performance. We've further established that these aren't failures; the symptoms span multiple activities, and the entire system is affected by the symptoms. By monitoring resources, we've found that the database often shows 100 percent CPU utilization and runs out of memory, and that there's often a queue of requests to enter the database under loads significantly below the target load.

Based on this, we decide with our team that we want to test just the database server under load and eliminate the Web server response times from the equation. We further decide to write some custom load-generation scripts by hand (that is, not using recording) to send SQL commands directly to the database. To use these custom scripts, we need to ensure that our load-generation machine can "see" the database server directly, which is conceptually equivalent to moving it from between the Client and Presentation tiers to between the Presentation and Data Storage (database server) tiers.



1. A THREE-TIER ARCHITECTURE

A second way is to use the tools we already have, placing a second load-generation machine between the Presentation and Data Storage tiers, to capture the traffic against the database generated by the load test as it is being executed by the first load-generation machine, located between the Client and Presentation tiers. This will give us a recorded script to edit that contains the entire load being placed on the database in a way that's easy to play back and evaluate.

In this case, our script represents the requests that are sent to the database by the Web server. Executing these scripts and reviewing the response times will show us conclusively how much of the end-to-end test time is being spent in the database. This information is almost always what the database administrator needs in order to find and/or tune the issue.

There are hundreds of third-party tools available to assist with the capture of resource-utilization statistics and response time by tier. There are fewer, but still many, third-party tools that provide a combination of these functions. These are commonly known as performance monitoring tools.

It's beyond the scope of this article to go into details about what a performance monitoring solution can add to your performance testing exercises, but I encourage you and your development team to jointly research a performance monitoring tool that fits your needs. If your organization conducts performance engineering testing exercises often, the time you'll save by obtaining and using one of these tools will far outweigh the cost of the tool in a short period of time.

Interpreting Tier-Specific Metrics

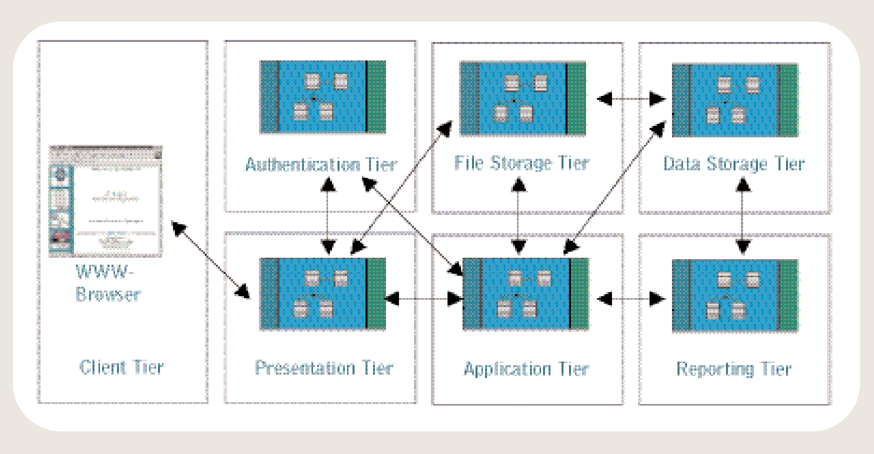
Often, tier-specific metrics leave little doubt as to their meaning, but even these detailed metrics may not hold all the answers. I've found a number of methods useful, individually and collectively, for interpreting tier-level metrics.

Look for the obvious. First and foremost, look for the obvious. In one situation, for example, the obvious was that the Web server (Presentation Tier, more precisely) was eating up four seconds every time data passed through it. Another example can be seen in the scatter chart shown in Figure 5. This chart represents a test where the response times experienced a significant slowdown about halfway through the test execution. These are the kinds of clues we're looking for. Unfortunately, in both cases, these were still both symptoms and not causes. In both cases, those symptoms gave the development team and myself ideas about where to look next.

Consult the development team. Once you find some obvious abnormalities, symptoms or clues—or you realize that you haven't found any obvious abnormalities, symptoms or clues—you should contact your development team and discuss what those findings mean. If you found no clues, maybe it means that the metrics you collected weren't the right ones, or that there wasn't enough load on the system, or that you eliminated a trigger event when you modified your tests. You probably won't know which (if any) of these is the case without help from your development team.

In the cases where you do find clues, the development team definitely wants to be involved. These clues are what point to either the next round of tests or to what they'll find themselves tuning in the next hours, days or weeks.

The point is, when you get this far into your performance testing, you and the development team really form a consolidated performance testing and tuning team. Most development teams aren't used to working like this, so it's up to you to be the team leader and ensure that there's constant two-way communication about tests, results, clues and suspicions. More than half the time, I find that I'm able to track down a bottleneck not by keen insight or superior testing knowledge, but rather by listening to developers when they say things like "I wonder if…," "Did you try…?" or "What if we…?" You'll also often find that after you show the results to your development team, one of them will come back to you later and say, "I found it," when you didn't even know he or she was looking for it.



2. A MULTI-TIER LOGICAL ARCHITECTURE

Change your test to prove your theory. Once you see your tier-specific results, it'll be almost impossible for you not to form theories about what caused those results. This would be like reading a mystery novel and not trying to guess "who dunnit" before the final chapter. You just can't do it.

Instead of trying to wait for the last chapter, I recommend embracing those theories and changing your test to prove or disprove them immediately. Once again, you'll likely need the assistance of the development team, but by this point you should have achieved a good working relationship with them. Besides, most developers I've worked with really seem to enjoy this part of the performance testing process. Honestly, I have to agree with them. To me, this is the fun part; it offers the same excitement as a treasure hunt did when I was a kid: "I wonder what we'll find if we follow all the clues correctly?"

Why Exploit Identified Bottlenecks?

Now that you know what the bottleneck is functionally and where it is architecturally, you're ready to track down the cause. If you've made it this far, none of your other tests have isolated the bottleneck sufficiently to resolve it. That's what exploiting bottlenecks is all about. Exploit means to apply, employ or exercise something. You exploit a bottleneck by building very specific tests that exercise the weaknesses in the system as an aid to the tuning effort.

Inevitably, whenever I get to this point in a training course, I'm asked, "If we know where the bottleneck is, why do we need to exploit it? Isn't that redundant?" The truth is that it's only redundant if the development team already knows what they need to tune and how to tune it. More often, just identifying the tier isn't enough. To explain why this is so, let's return to our hydrodynamics analogy from the May article, in which we compared the flow of activity through a software system to the flow of water through a pipe system.

Figure 4 is a simplistic representation of what the inside of a tier might look like if it were a hydraulics system. The pipe that represents our network comes into the tier from the top left. Once the water leaves that pipe, it enters a pool with various pipes exiting the bottom. This represents requests entering a processing queue where there are a limited number of processing units (likely threads) to handle those requests. Which "exit pipe" the request flows through is based on the type of request that's being made. Notice that the exit pipes are of various sizes and each one may or may not be open at a given point in time.

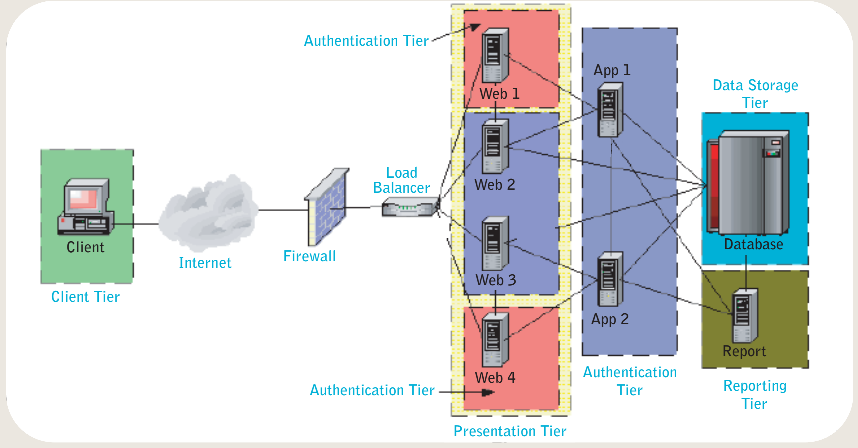
Without delving too deeply into the different possibilities for request processing, suffice it to say that any given tier can have more or fewer processes (pools) for a request to pass through, depending on the specific request and/or the design of your system. The number, size and availability of exit pipes from these processes can have a significant effect on the overall performance of the system. I'm sure you can see that just pointing to the tier and saying, "The bottleneck's in there" probably isn't good enough. To tune the system, we often need to help developers narrow the focus down to the specific process or even to the parameters (inputs and outputs) of that process (symbolized by an exit pipe). That's what we do when we exploit bottlenecks.

Ways to Exploit Identified Bottlenecks

So far we've been discussing how to modify tests to focus on bottleneck resolution. Now we're going to modify existing tests again and/or generate new ones to get more information about exactly what's causing the bottleneck in the tier you've identified. I'll explain how to exploit bottlenecks for tuning by finding bounds conditions, breakpoints and resource constraints.

Find bounds conditions. One of the ways to exploit bottlenecks is to execute tests that focus on identifying bounds

3. PHYSICAL ARCHITECTURE WITH LOGICAL OVERLAY



conditions rather than running under expected normal conditions. These bounds conditions are a little different from the bounds conditions we test during functional testing. We're not talking about testing to see if an input field accepts numbers with more than six digits correctly. We're talking about testing the bounds of performance—for example, seeing how the system performs when executing only searches that return excessively large amounts of data, such as searching for all book titles that include the letter "t" on Amazon.com or executing an extremely high volume of searches concurrently. These types of tests will often show results that allow us to say more than just "This search seems slow."

Under these extreme conditions, we look for information like the following:

- How many searches can I do before the memory starts to rise above 80 percent utilization on the database server?
- How many rows of data must I be requesting before the system returns a timeout message?
- How many times can this activity be conducted in a 10-minute period before all of the available threads are consumed?

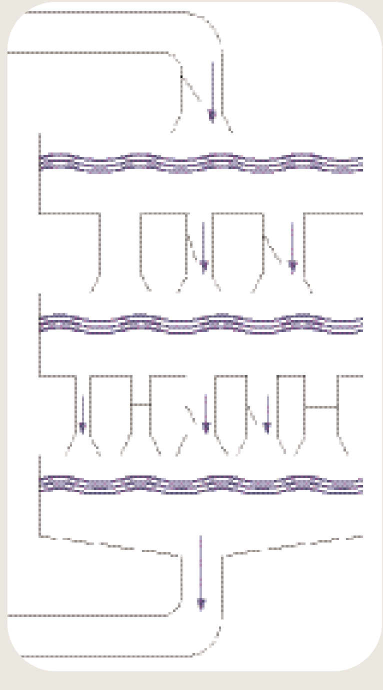
Each of these facts tells us something about bounds conditions. In both functional and performance testing, unexpected behavior tends to occur under these conditions. In performance testing, these unexpected behaviors often point us to the actual cause of the observed symptoms under expected usage conditions.

On a recent project, we found that after we applied SSL to our Web site, all of the pages slowed down by about 30 percent. While we expected the login activity to slow down, we didn't expect subsequent pages to be slower.

At first we thought that the login process itself was slowing down the entire Web server, so we created a "login only" test and monitored the resources on the Web server (where the logical Authentication Tier resided). This revealed that logging in under load was not the problem, so we decided to exploit the bottleneck instead by looking for the bound where performance degraded.

It turned out we didn't have to look far. We started by limiting our test to a single user logging in, navigating to the search screen, searching, and logging out, and we monitored the Authentication Tier through full logging. When we evaluated that log, we found that every page was checking with the Authentication Tier to see if the user had permission to access that page rather than simply getting the ACL (access control list) from a client-side cookie as intended.

4. TIERS AS A PIPE SYSTEM



Once the developers saw that the problem was with the "retrieve permissions" process in the Authentication Tier, they were able to resolve the problem in less than an hour. The performance of all pages improved to what it had been before SSL was applied, and login gained back 50 percent of the performance it had lost when SSL was applied.

Find breakpoints. Deliberately causing the application to fail by running it under conditions even more extreme than the ones under which it shows symptoms of poor performance is another method of exploiting a bottleneck. Such breakpoints, where the bottleneck becomes a failure, are often uncovered while searching for and testing at bounds conditions. Like finding bounds conditions, determining the point at which the system fails due to an extreme performance case will likely point to a cause.

Information about breakpoints will most often be found in the application server logs. Breakpoints are commonly identified by error messages being returned, system or browser timeouts occurring, and/or nothing returning at all (that is, the page just sitting there forever). Any of these conditions can yield valuable information to the developer who's trying to help track down and tune the bottleneck.

I also used this method on a recent project. In this case, we had determined that reports seemed to slow down dramatically under load. Through all of our monitoring, we were unable to track down the reason. Monitoring the report server seemed to point to the database returning data slowly, but monitoring the database showed the requests coming back quickly. We finally decided to just increase the number of reports until we received an error message.

After increasing the reporting load significantly, we did receive an error message—indicating an overflow error. While this error didn't make sense to most of the team, it did make sense to the administrator of the report server. From that error message she was able to determine that when the report server received a request for a report, it was sending a request to get the data from the database and putting that data into a single processing queue. This meant that the data for all of the reports was stacking up and only one report was being generated at a time, where the actual intent was for this server to have five parallel processes and not just one. After a few calls to support, the administrator was able to configure the report server to handle the five parallel processes, and our problem was resolved.

Find resource constraints. While monitoring resource utilization, you and your development team should be looking for resource utilization that's above the expected volume and/or above the recommended usage for that particular resource. If adding stress (such as adding additional high-volume searches) pushes resource utilization to a higher than expected rate, this may indicate that the activity being tested isn't managing that resource adequately during less stressful times, either. The inadequate resource utilization may not be obvious during low-stress situations, but it still may be the cause of the symptoms. Only by exploiting the bottleneck by intensifying it can we find out for sure if resource utilization is the cause.

The most common resource constraint is memory utilization. Under large loads, one or more of your servers is likely to experience memory utilization consistently greater than 80 percent. Once this number grows to more than 80 percent, performance almost always suffers. In these cases, it's up to the developers and architects to determine if the application is managing memory poorly, if configuration settings need to be adjusted or if more memory is required.

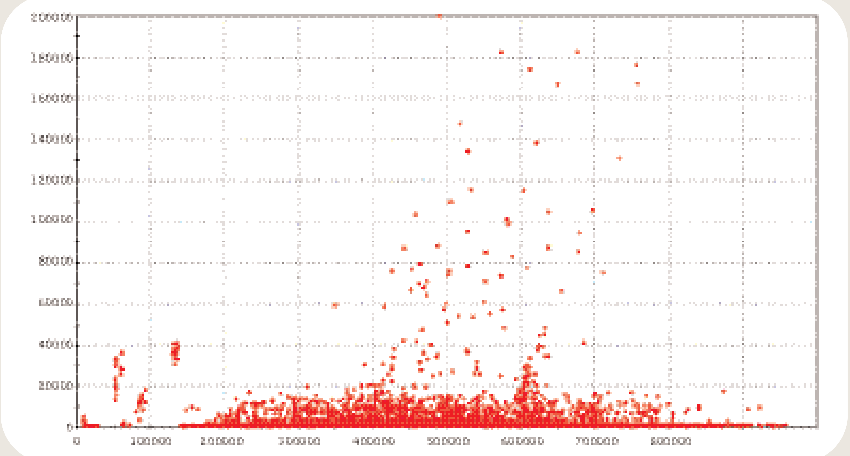
Handing Off Leadership To the Development Team

You may have noticed that the farther down the trail of chasing bottlenecks you go, the more closely you're working with the development team. Interacting with the development team is crucial to the process of building tests to exploit bottlenecks. You'll very rarely have a deep enough understanding of the system to build tests and collect data at this level on your own. In cases where you're able to exploit the bottleneck by simply modifying test data, inputs and load, the development team is still critical in the results interpretation stage.

As your tests get narrower and narrower, and closer and closer to the actual code, the development team becomes increasingly critical in the test development stage. The development team is also normally where the best guesses will come from as to what tests to develop to try to exploit a particular bottleneck, not just how to develop them, as the examples below will show.

This is the point of transition from the testing phase, where the performance tester leads and the development team assists, to the tuning phase, where the development team leads and the performance tester assists. It's important to explain to the development team that now you're helping them, not the other way around, and that you're going to exploit bottlenecks with the intent of helping them find the root cause of the symptom, not just to ferret out more symptoms.

5. SCATTER CHART DEPICTING A SLOWDOWN



Be available to the development team, and be open to building and executing on a moment's notice tests that you may not completely understand (though it's still a good idea to ask questions and gain understanding throughout the process). Don't be discouraged if developers start digging more independently at this point in the process.

As an illustration of the crucial role the development team can play at this point, let's review the scatter chart in Figure 5. This chart shows a test run with caching at the front, a good run for a period after that, then a mostly classic slowdown toward the midway point.

In an attempt to determine the cause of that slowdown, we looked at several common resource statistics associated with the servers involved. We found that the CPU utilization of the application server reached unacceptable levels shortly before the response times increased.

As you may already suspect, *we* decided to monitor the CPU queue length for that same test. The reason I stress "we" is that it was the developer's idea to look specifically at that metric, which wasn't among those that I had initially recommended.

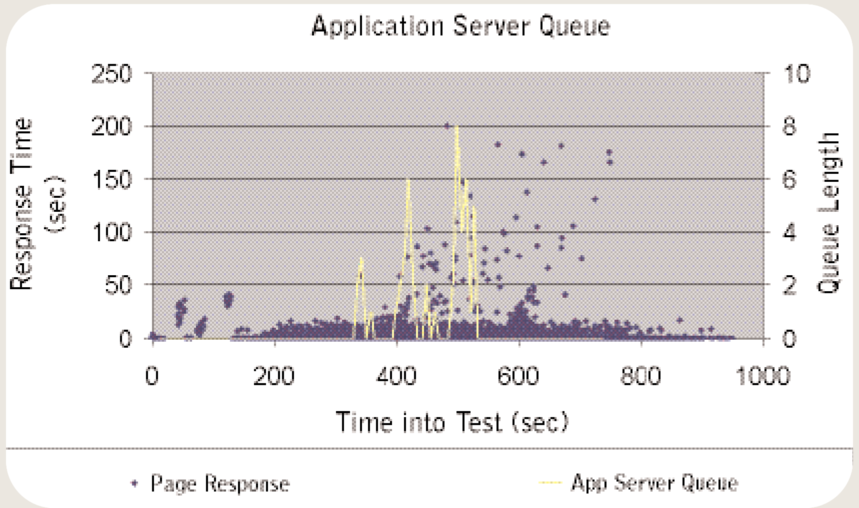
Monitoring the CPU queue length resulted in the chart in Figure 6, which shows a direct correlation between queue length and poor performance. I can't say whether I would have looked at that metric eventually, but for whatever reason, I wasn't planning to look at it initially. That open communication between the developer and myself saved at least one extra step and revealed the actual cause of the poor performance in this test.

Incidentally, the test that generated those results was a test that had been created to exploit what we thought was a database bottleneck. The initial symptoms had been that activities writing to the database were slow. Building tests that exploited those activities and monitoring various resources allowed us to track the actual cause to code processing in the application server.

Moving Into a Different Kind of Testing

The vast majority of performance testing classes and publications are focused on what could be categorized as black box performance tests—that is, tests created without reference to the source code or other information about the internals of the product. In the words of Florida institute of Technology professor Cem Kaner, author, along with James Bach and Bret Pettichord, of "Lessons Learned in Software Testing" (Wiley, 2001), "The black box tester consults external sources for information about how the product runs (or should run), what the product-related risks are, what kinds of errors are likely and how the program should handle them, and so on."

6. MONITORING THE APPLICATION SERVER'S CPU QUEUE



This is in contrast to white box testing, which Kaner defines as "testing with thorough knowledge of the code." In one discussion, Kaner goes on to say, "The programmer might be the person who does this. I've seen members of independent test groups do this type of testing. Some risks that are invisible to the black box tester aren't too hard to see in the source, such as weak error handling, a weak model of interrupt-triggering events, or excessive coupling of different parts of the program. The test groups that do this type of work usually specialize one or a few people who do nothing but read the source code looking for interesting/risky areas and then design thorough tests to exploit those risks."

When we began designing our new tests in interaction with the development team, we started getting into the area of tests that could be classified as gray box tests. According to Kaner, the design of gray box tests is educated by information about the code or the program operation of a kind that would normally be out of view of the tester. Kaner makes the point that the distinction between black box, white box and gray box testing is in the thinking of the tester. Thinking that's focused neither on the usage-related world external to the program nor on the source of the program, but is more focused on the technical relationship between the program and the system, is what he refers to as gray box testing.

Whether or not you like those particular terms, I'm certain you'll agree that at this level of bottleneck detection and tuning we've moved from user-experience (usage-related) tests to tests that are focused on the technical relationship between the program and the system. It's often the case that exploiting bottlenecks isn't going to be done simply by modifying user-centric load-generation scripts.

As it turns out, most of the tests we performance testers conduct are gray box tests. While we begin designing our tests thinking about how users will interact with the system, we then start thinking about how the system works and modify our initial designs accordingly. For example, we may add a script that runs a particular report simply because we know that it accesses data from a particularly large table in the database. The fact that we decide to create that script based on the design of the database makes it a gray box test.

Knowing When to Put The Load-Generation Tool Away

Load-generation tools can see only so far into the application. No matter how good your tests and analysis are, you'll sometimes have to dig into the application with your development team, often all the way to the code level.

Currently there are no load-generation tools on the market that are designed for this kind of grey/white box testing, though there are two scheduled for release this year. Because of this, one of the best ways you can assist the development team at this level is with tools that complement your load-generation tool.

An example of a tool at your disposal is the test harness. Most of the time the test harness is built by the developer to complement the performance tester's individual skills and scripting preferences to exploit a very specific area of the application that the developer wants to be able to test in a repeatable way.

If you can't exploit bottlenecks using test harnesses and hand-coded scripts, it's probably time for a third-party tool to complement the load-generation tool. The tools I'm referring to are often classified as "code analyzers," "runtime analyzers," "code profilers" or even "performance profilers." You can add a lot of value by knowing how to use one or more of these tools in conjunction with the load-generation and/or bottleneck-focused scripts you've already created.

Making A Successful Diagnosis

As you can see, exploiting performance bottlenecks really does follow the same thought process as a doctor employs in diagnosing the symptoms of an illness. I suspect that virtually all of us find the process of diagnosing an illness to be at least fairly natural. Applying the principles of the diagnostic process to observed symptoms of poor performance will likely have a similar result—grumbling about having to do it, but being very glad you did once you start feeling better. ☒