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 polks, 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 can categorize as failures, slow spots or bottlenecks—yet it’s likely that you still don’t have enough information to effectively diagnose the problem.

A Refresher on n-Tier Architecture

“All parts should go together without forcing. You must remember that the parts you are assembling 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 Web2.” 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. Let’s take as an example a three-tier architecture.

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 dual duty as either an authentication-server tier or a file-storage-tier server. As it turns out, it’s just

www.stpmag.com • 15 JULY 2005
monitoring resources, we’ve found that system is affected by the symptoms. By database cause symptoms of poor performance analysis tools and/or 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 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 of the response time by tier involving either special tools or instrumentation. I should caution 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, as well as some specific to your application architecture.

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 tests, our load-generation machine between the Presentation and Data Storage (database server) tiers.

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 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. Consolidated performance testing and tuning team. Most development teams aren’t used to working like this, so it’s up to the development leader and ensure that there’s constant two-way communication about tests, results, clues and suspicions. More than half the time I find that it’s important to look for the obvious. If you find some obvious abnormalities, consolidated performance testing and tuning team. Most development teams aren’t used to working like this, so it’s up to the development leader and ensure that there’s constant two-way communication about tests, results, clues and suspicions. More than half the time I find that it’s important to look for the obvious. If you find some obvious abnormalities, you realize that you haven’t found any obvious abnormalities, symptoms or clues—"you should consult 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 for a reason.

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 for a reason.

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?”

Capturing Resource Utilization And Response Time by Tier

Most operating systems come with resource-monitoring software, like PERF Mon 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 the response time by tier involving either special tools or instrumentation. I should caution 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 as well as specific to your application architecture.

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 tests, our load-generation machine between the Presentation and Data Storage (database server) tiers.

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 correct the issue.

There are hundreds of third-party tools available to assist with the capture of resource-utilization statistics and 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. Consolidated performance testing and tuning team. Most development teams aren’t used to working like this, so it’s up to the development leader and ensure that there’s constant two-way communication about tests, results, clues and suspicions. More than half the time I find that it’s important to look for the obvious. If you find some obvious abnormalities, symptoms or clues—"you should consult 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 for a reason.

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 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 not redundant if the development team already knows what they need to tune and how to tune it. More often, just identifying the bottleneck is enough. To exploit the bottleneck is to “pin down” the hydrodynamics analogy from the May article, in which we compared the flow of water through the pipes of a complex 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. The Web server is on top, the application tier is right below the Web server, and the database tier. The pipe that represents our network comes into the tier from the top left. Once the water leaves that pipe, it enters one of the various pipes connecting to 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.

Now that the exit pipes are of various sizes and each one may or may not be open at a given point in time. You need to study into the different possibilities for request processing, suffice to say that any given tier can have more or fewer processes (threads) depending on the specific request and/or the design of your system. The number, size and availability of exit pipes for these processes can have a significant effect on the overall performance of the system. I’m sure you can see that just pinning down the pipe by building very specific tests that exercise the weaknesses in the system as an aid to the tuning effort.

Find bottlenecks conditions. One of the ways to exploit bottlenecks is to execute tests that focus on identifying bottlenecks.
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 searchers) pushes resource utilization to a higher rate than the uncorrected 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 utiliza-
tion. 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 deter-
mine if the application is managing memory poorly, if configuration set-
tings need to be adjusted or if more memory is required.

Handing Off Leadership To the Development Team

You may want to take or delegate to the development team, and be open to building and execut-
ing on a moment’s notice tests that you may not completely understand. Your team’s still a good idea to ask ques-
tions and gain understanding through-
out the process. Don’t be discouraged if developers start digging more inde-
pendently 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. As you study the chart you’ll notice a steep drop in the performance measurements. You may have had a heart attack just seeing this. Don’t be discouraged. The important thing is to notice that, for the test run, you have a database bottleneck. The initial symp-
toms had been that activities writing to the database were slow. Building tests that exploit these activities and moni-
toring various resources allowed us to

5. SCATTER CHART DEPICTING A SLOWDOWN

16 • Software Test & Performance JULY 2005
17 • Software Test & Performance JULY 2005

conditions rather than running under expected normal conditions. These bottlenecks can be a little different from the bounds conditions we test during in-
fected as well. Taking about two to three digi-
tes correctly. We’re talking about test-
ing functional testing. We’re not talking about testing under conditions even more extreme than the ones under which it shows symptoms, and/or performance is another method of exploiting a bottleneck. Which breakpoints, where the bot-
tlenecks become a failure, are often the right place to start testing at bounds conditions. Like finding bottlenecks, determining the point at which the system fails due to an extremely heavy load case will likely point to a cause.

Information about breakpoints will come from the database 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 devel-
oper who’s trying to help track down the bottleneck. I also used this method on a recent project. In this case, we had determined that reports seemed to slow down dra-

natically under load. Through all of our monitoring, we were unable to track down the reason. At one point, the database returned data slowly, but monitoring the database showed the requests coming back quickly. We finally decided to just increase the number of reported requests 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 performance tester. From that error message she was able to determine that when the report server received 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 CPU of the database server received an unex-
tended 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 job 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 or would not at that metric eventually, but for what-
ever reason, I wasn’t planning to look at

THE TESTING PROCESS

16 • Software Test & Performance JULY 2005
17 • Software Test & Performance JULY 2005

4. TIERS AS A PIPE SYSTEM

THE TESTING PROCESS

16 • Software Test & Performance JULY 2005
17 • Software Test & Performance JULY 2005

THE TESTING PROCESS

16 • Software Test & Performance JULY 2005
17 • Software Test & Performance JULY 2005

THE TESTING PROCESS

16 • Software Test & Performance JULY 2005
17 • Software Test & Performance JULY 2005

THE TESTING PROCESS
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.”

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, 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-generating tools on the market that are designed for this kind of gray/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 profilers,” “runtime analyzers,” “code analyzers,” 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.