Beyond Performance Testing

by:

R. Scott Barber

Part 11: Collaborative Tuning

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. According to Roget's II: The New Thesaurus (Third Edition, 1995), to exploit is "to put into action or use: actuate, apply, employ, exercise, implement, practice, use, utilize." You exploit a bottleneck by building very specific tests that exercise the weakness in the system as an aid to the tuning effort.

This article concludes the four-article group on the theme "finding bottlenecks to tune." This is the last step down the tuning path where the performance test engineer serves as the lead. By the conclusion of this article, you should be confident in your ability to work with the development team to identify and exploit areas of concern in a way that adds significant value to the overall development process.

So far, this is what we've covered in this series:

Part 1: Introduction

Part 2: A Performance Engineering Strategy

Part 3: How Fast Is Fast Enough?

Part 4: Accounting for User Abandonment

Part 5: Determining the Root Cause of Script Failures

Part 6: Interpreting Scatter Charts

Part 7: Identifying the Critical Failure or Bottleneck

Part 8: Modifying Tests to Focus on Failure or Bottleneck Resolution

Part 9: Pinpointing the Architectural Tier of the Failure or Bottleneck

Part 10: Creating a Test to Exploit the Failure or Bottleneck

This article is intended for mid- to senior-level performance testers and members of the development team who work closely with performance test engineers. If you haven't read Parts 5, 6, 7, 8, and 9 of of this series, I suggest you do so before reading this article.

Why Exploit Identified Bottlenecks?

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 Part 7, in which we compared the flow of activity through a software system to the flow of water through a pipe system.

Figure 1 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 may or may not be open at a given point in time.

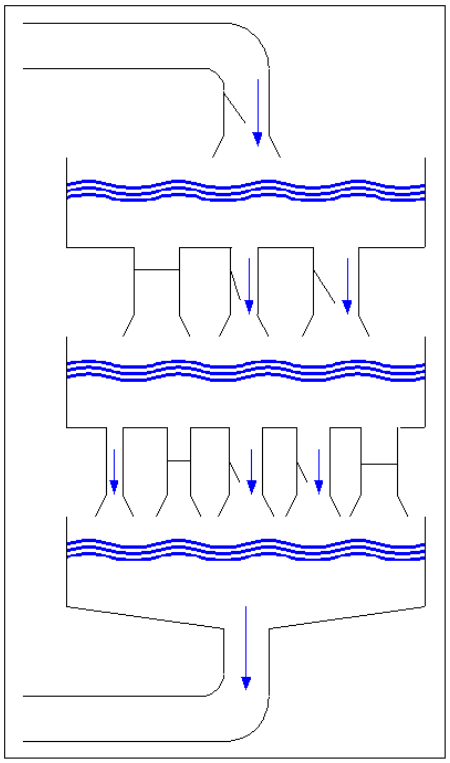
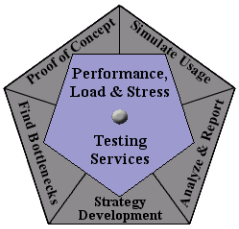


Figure 1: The tier as a hydraulics system

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 go 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

In Part 8, I discussed how to modify tests to focus on bottleneck resolution. Now you'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 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, like searching for all book titles that include the letter *t*, or executing an extremely high volume of searches. 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% utilized on the database server?
- How many rows of data must I be requesting before the system returns a time-out message?
- How many times can this activity be conducted in a 10-minute period before all of the available threads are consumed?

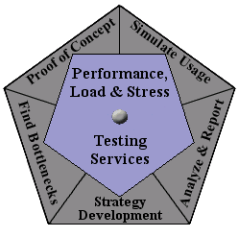
Each of these facts tells us something about bounds conditions. In both functional and performance testing, unexpected behavior tends to occur under these conditions. In performance testing, these unexpected behaviors often point us to the actual cause of the observed symptoms under expected usage conditions.

On a recent project, we found that after we applied SSL to our Web site all of the pages slowed down by about 30%. 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. 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% 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 not in TestManager but rather in the application server logs. Breakpoints are commonly identified by error messages being returned, system or browser time-outs 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 requested 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

We discussed how to monitor resource utilization in Part 9. During this monitoring, you and your development team should be looking for resource utilization that's above the expected volume and/or is 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 may still 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 example is memory utilization. Under large loads, one (or more) of your servers is likely to experience memory utilization consistently greater than 80%. Once this number grows to more than 80%, 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 and 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 simply by 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 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 test engineer leads and the development team assists to the phase where the development team leads and the performance test engineer 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. Be available to the development team and be open to building and executing tests on a moment's notice that you may not completely understand (though it's still a good idea to ask questions and gain understanding throughout the process). And don't be discouraged if developers start digging more independently at this point in the process.

Following the Development Team's Lead: Example 1

As an illustration of the crucial role the development team can play at this point, let's return to an example from Part 6. Figure 11 there, reproduced as Figure 2 below, is a scatter chart depicting a test where the response times experienced a significant slowdown about halfway through the test execution. As you may recall, the 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.

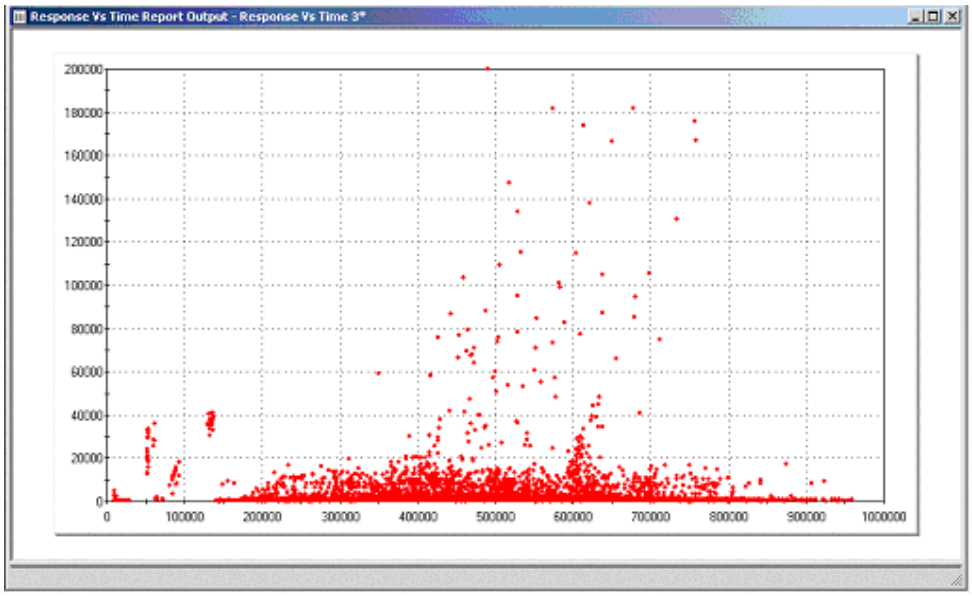


Figure 2: A scatter chart showing a slowdown midway through a test

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 (see Figure 3 below, a reproduction of Figure 20 in Part 6). In Part 6, I mentioned that *we* then 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 initially recommended.

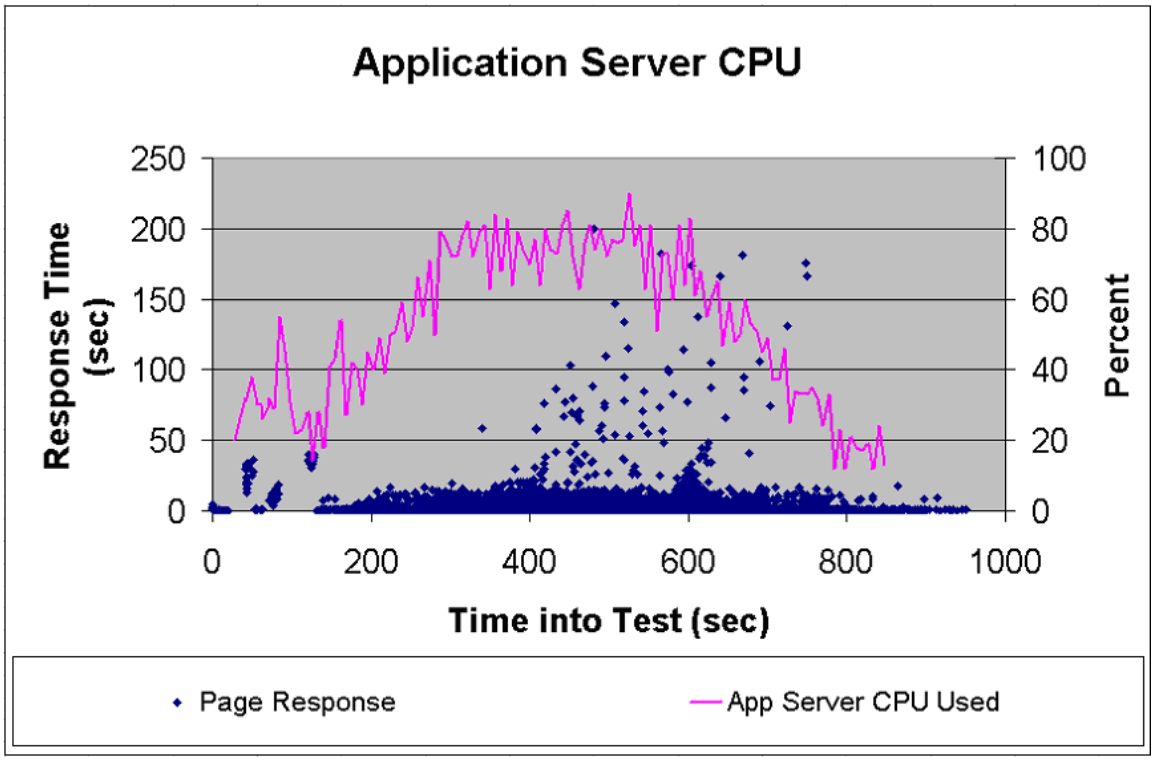


Figure 3: The scatter chart overlaid with application server CPU utilization data

Monitoring the CPU queue length resulted in the chart in Figure 4 below (a reproduction of Figure 21 in Part 6), which showed a direct correlation between the queue length and the 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 I saved at least one extra step and revealed the actual cause of the poor performance in this test.

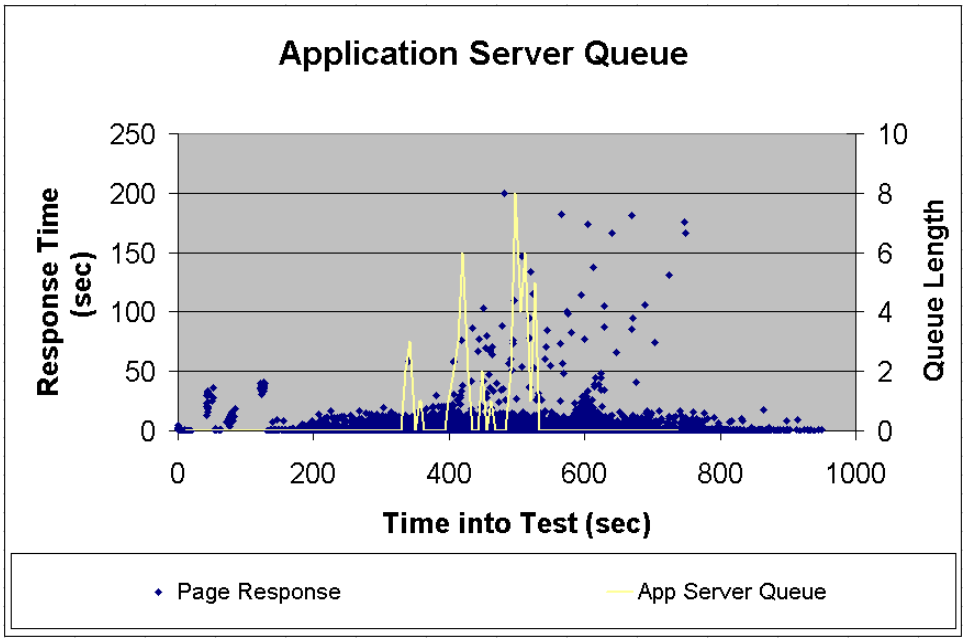


Figure 4: The scatter chart overlaid with application server queue length data

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 that activity and monitoring various resources allowed us to track the actual cause to code processing in the application server.

Following the Development Team's Lead: Example 2

Another example of following the development team's lead is the one illustrated in Figure 16 of Part 9, reproduced as Figure 5 below, where looking at response times by tier revealed that the Web server seemed to be "eating" 4 seconds every time a request went through it. When I reported that finding it seemed ridiculous to both the development team and I, so we developed some more tests.

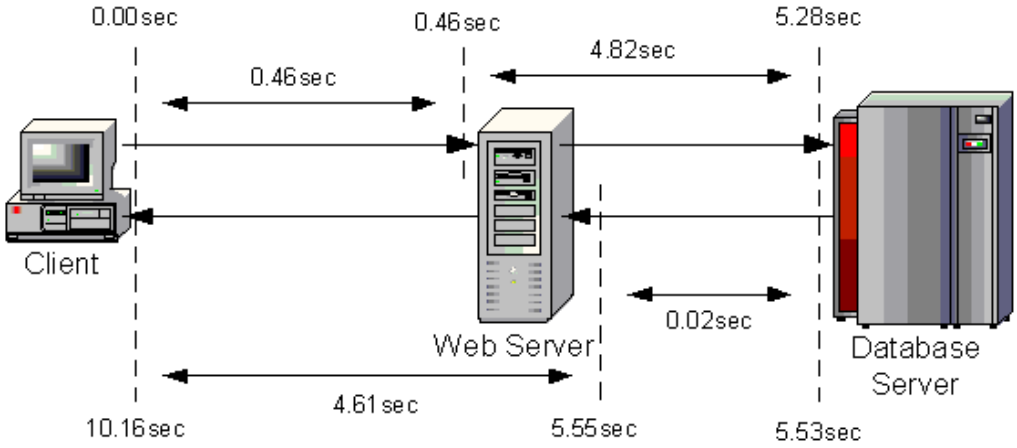
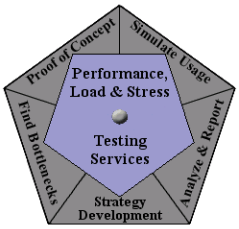


Figure 5: Response-time-by-tier graph



The first thing we did was put some graphics on the Web server of various sizes — 1 KB, 10 KB, 100 KB, and 1000 KB. I then manually wrote four test scripts in the IBM Rational® Robot software to retrieve each of those graphics and time the response. I executed these four scripts 100 times each. Looking at the results, I found something very interesting. The 1 KB graphic always returned in roughly 4.1, 8.1, or 12.1 seconds. The other graphics returned in roughly the same amount of time — for instance, the 100 KB graphic returned in roughly 4.3, 8.3, or 12.3 seconds. In all four cases, about 60% of the responses returned in a little more than 4 seconds, 30% returned in a little more than 8 seconds, and the remaining 10% returned in a little more than 12 seconds.

Having no idea what those measurements meant, we then added logging to the Web server, where we time-stamped the arrival of the request and the departure of the first byte of the response. In all cases, that measurement was well under .01 seconds, indicating that the problem wasn't actually in the Web server at all.

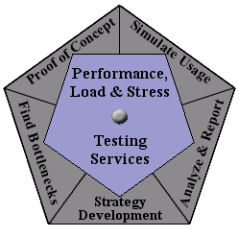
We then contacted our network administrators, since the only piece left was the network between the load-generation machine and the Web server. First they put a sniffer on the subnet of the load-generation machine and validated that our results matched what was appearing on the network. On a wild hunch, we then moved the network sniffer to the subnet containing the Web server. When we compared those numbers, we found that the round trip for the requests/responses on that subnet didn't have the "4-second steps," as we'd come to call them. Some analysis showed that the only things between those two subnets were some passive hubs and a router. After that, an expert on configuring that particular model of router was called and reconfigured the router so it wasn't imposing the artificial 4-second delay.

Moving into a Different Kind of Testing

All through the "User Experience, Not Metrics" series and up until Part 8 of this series, we 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 Cem Kaner, senior author of *Bad Software* and *Lessons Learned in Software Testing* and professor of computer sciences at the Florida Institute of Technology, "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 what might be called "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."

In Part 8, 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, 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 as performance testers conduct are gray box tests. While we begin designing our tests thinking about how users interact with the system, we then start thinking about how the system works and modify our initial design 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

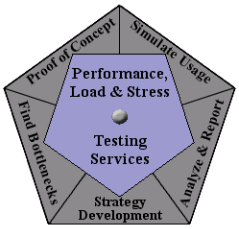
Load-generation tools can see only so far into the application. No matter how good your tests and analysis are, you'll sometimes have to dig into the application with your development team, often all the way to the code level. You could say that this is where you cross the line from gray box testing into white box testing. I'm not aware of a single load-generation tool on the market today that's designed for white box testing. 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 and custom (handwritten) script method that I mentioned in Part 9 to access the database directly. This method can be used to access virtually any component of the application, right down to an individual line of code. Most of the time the test harness is built by the developer to complement the performance test engineer'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.

Often, response time measurements from these tests are embedded in application logs that the developer reviews herself, and thus the performance test engineer rarely sees the results. This is completely natural. By this point, the developer is leading and you're assisting. In this case you're assisting by providing input into the system in a way that the developer either can't do or would find prohibitively difficult, and the developer is doing the analysis. This process is sometimes even thought of as collaborative unit testing.

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 IBM Rational TestStudio® software. As I mentioned in Part 9, there are many such tools on the market, and they're very specific to the application architecture. The tools I'm referring to are often classified as code analyzers, runtime analyzers, code profilers, or even performance profilers. As an example of the kinds of tools that are available, let's take a look at Rational's runtime analysis suite, IBM Rational® PurifyPlus. If you're not already familiar with this product, it's officially described this way:

"Rational PurifyPlus is a complete set of runtime analysis tools designed for improving application



reliability and performance. PurifyPlus combines memory error and leak detection, application performance profiling, and code coverage analysis into a single, complete package. Together, these functions help developers ensure the highest reliability and performance of their software from its very first release."

Many third-party tools, including PurifyPlus, are made to work independently of the load-generation tool but often provide even more value when used in conjunction with it. "Testing J2EE Applications with Rational PurifyPlus" by Goran Begic gives a pretty representative look at what tools of this type can do in terms of tracing code performance and memory usage down to the class or method level. You can add a lot of value by knowing how to use one or more of these third-party tools in conjunction with the load-generation and/or bottleneck-focused scripts you've already created. The bottom line is that familiarizing yourself with at least one of these tools will help you work more effectively with the development team.

Is It Time to Tune?

Now it's time to tune. If you've followed all the steps outlined in the last four articles, even to the point of obtaining and using a performance profiler, you've no doubt identified the cause of the performance symptoms and shared that information with your development team. As I've mentioned, tuning is an iterative process. Be prepared to re-execute all of the tests you've created to pinpoint and exploit the bottleneck, in reverse order, on request from the developer. This is done to ensure that the symptoms have been resolved all the way back to the expected user loads and that no new symptoms have arisen as a result of the tuning effort.

Summing It Up

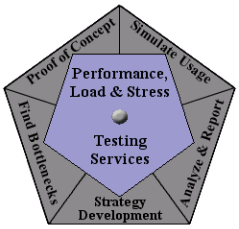
This concludes the four-article theme "finding bottlenecks to tune." In this group of articles I've discussed detecting performance suspects, distinguishing between failures, slow spots, and bottlenecks, and how to track down performance bottlenecks to a level of detail great enough for developers to tune them. I've stressed the increasing levels of interaction with the development team throughout this theme. Without a good relationship with the development team, it's unlikely that you'll ever be certain of anything more concrete than suspects, symptoms, and hunches. Working together, you and your development team should be able to detect and tune bottlenecks quickly and efficiently. In the next four articles I'll discuss more advanced areas where the performance test engineer can serve in a support role, to include tuning.

References

- "Testing J2EE Applications with Rational PurifyPlus" by Goran Begic

Acknowledgments

- Thanks go to Steve Tani for reviewing Parts 7 through 10 for content and consistency. His input greatly improved this four-article theme.
- The original version of this article was written on commission for IBM Rational and can be found



on the IBM DeveloperWorks web site

About the Author

Scott Barber is the CTO of PerfTestPlus (www.PerfTestPlus.com) and Co-Founder of the Workshop on Performance and Reliability (WOPR – www.performance-workshop.org). Scott's particular specialties are testing and analyzing performance for complex systems, developing customized testing methodologies, testing embedded systems, testing biometric identification and security systems, group facilitation and authoring instructional or educational materials. In recognition of his standing as a thought leading performance tester, Scott was invited to be a monthly columnist for Software Test and Performance Magazine in addition to his regular contributions to this and other top software testing print and on-line publications, is regularly invited to participate in industry advancing professional workshops and to present at a wide variety of software development and testing venues. His presentations are well received by industry and academic conferences, college classes, local user groups and individual corporations. Scott is active in his personal mission of improving the state of performance testing across the industry by collaborating with other industry authors, thought leaders and expert practitioners as well as volunteering his time to establish and grow industry organizations. His tireless dedication to the advancement of software testing in general and specifically performance testing is often referred to as a hobby in addition to a job due to the enjoyment he gains from his efforts.

About PerfTestPlus

PerfTestPlus was founded on the concept of making software testing industry expertise and thought-leadership available to organizations, large and small, who want to push their testing beyond "state-of-the-practice" to "state-of-the-art." Our founders are dedicated to delivering expert level software-testing-related services in a manner that is both ethical and cost-effective. PerfTestPlus enables individual experts to deliver expert-level services to clients who value true expertise. Rather than trying to find individuals to fit some pre-determined expertise or service offering, PerfTestPlus builds its services around the expertise of its employees. What this means to you is that when you hire an analyst, trainer, mentor or consultant through PerfTestPlus, what you get is someone who is passionate about what you have hired them to do, someone who considers that task to be their specialty, someone who is willing to stake their personal reputation on the quality of their work - not just the reputation of a distant and "faceless" company.