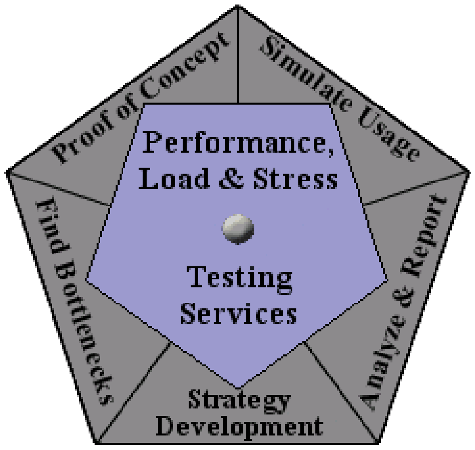
PerfTestPlus
Better Testing... Better Results

Part 7:Identify the Critical Failure or Bottleneck

Beyond Performance Testing

by:

R. Scott Barber



So you found an odd pattern in your scatter chart that appears to be a bottleneck. What do you do now? How do you gather enough information to refute the inevitable response, “The application is fine, your tool/test is wrong”? And how do you present that information conclusively up front so you can get right down to working collaboratively with the development team to solve the problem? Those are the questions we’ll be addressing in this article. In addition, I’ll be giving you eight rules about bottlenecks that I’ve found to be both significant and useful during my tenure as a performance test engineer.

This kicks off a four-article theme I call “finding bottlenecks to tune.” We’ve already explored the entry-level analysis that points us in the direction of a bottleneck or failure, and I’ve given some hints on how to ferret them out. Now it’s time to get under the hood. By the conclusion of Part 10, you should be confident in your ability to work with the development team to identify and exploit these 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

This article is intended for mid- to senior-level performance testers and members of the development team working closely with performance testers. If you haven’t read Parts 5 and 6 of this series, I suggest you do so before reading this article.

What Exactly *Is* a Bottleneck?

“If there were no mystery left to explore, life would get rather dull, wouldn’t it?”

— Sydney Buchman (www.quotablequotes.net)

Through years of experience in detecting bottlenecks, explaining bottlenecks, and teaching others how to do the same, I’ve learned something very important. It’s not safe to assume that most people know what a bottleneck is. Even if you've been doing performance testing for a long time and use the term bottleneck regularly in everyday speech, I recommend that you don't skip this section, particularly since the distinction between a system failure, a slow spot, and a bottleneck will be central to our discussions throughout the rest of this series.

The Dictionary Version

Here's how the *Webster's Millennium™ Dictionary of English* (Lexico Publishing Group, 2003) defines *bottleneck*:

n: a narrowing that reduces the flow through a channel [syn: constriction] v 1: slow down or impede by creating an obstruction; "His laziness has bottlenecked our efforts to reform the system" 2: become narrow, like a bottleneck; "Right by the bridge, the road bottlenecks"

This definition is understandable and hints at the origin of the term (referring to the narrow part of a jar or bottle) but is most useful to us if we note what it doesn't say as well as what it does say:

"reduces the flow," *not* "ceases the flow"

"slow down or impede by creating an obstruction," *not* "stop by creating an obstruction"

"become narrow," *not* "become impassable"

The reason this is important is that it's the basis for the distinction between a performance bottleneck and a performance-related failure that we'll explore below. I summarize that point as . . .

Scott's first rule of bottlenecks: A bottleneck is a slowdown, not a stoppage. A stoppage is a failure.

Something else the dictionary definition doesn't mention that will become relevant to us is load or volume. The fact that the definition above says nothing about the cause of a bottleneck suggests that it shouldn't be assumed that a bottleneck exists only under load or volume. In my experience, most folks assume that bottlenecks don't exist unless a certain "trigger load" is applied to a system. This is both contrary to the definition of *bottleneck* and often untrue when applied to software systems, thus bringing us to . . .

Scott's second rule of bottlenecks: Bottlenecks don't *only* exist under load.

The Hydrodynamics Version

Next let's take a look at bottlenecks from a hydrodynamics perspective. I have a BS degree in civil engineering, which means that I took roughly 16 credits of hydraulics and hydrodynamics in college long before I became a performance test engineer. What I've realized since then is that a useful comparison can be made, at least conceptually, between the flow of water through a pipe system and the flow of activity through a software system.

Figure 1 shows the simplest possible version of a bottleneck in a pipe (you may notice that at first glance, the drawing resembles a bottle). Without getting into complex formulas, you can see that more water can flow through the section of pipe on the left than on the right, given a constant pressure, over time. The arrows in this diagram depict velocity, or the speed that the water is actually moving through the pipe; the shorter the arrow, the slower the flow of water.

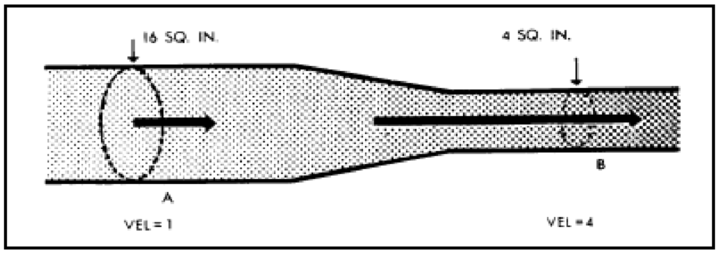


Figure 1: A simple pipe bottleneck

What you see here is that the water moves faster through the narrower section of pipe. This concept seems counterintuitive to most people at first, but the explanation is really rather simple. For the same total volume of water to move through a narrower section of pipe, that water has to move faster to make room for the water in the wider section of the pipe. This is a classic example of a queue.

To illustrate the concept of a queue, think now of that pipe holding sand instead of water. Each grain of sand in the wider part of the pipe must stop and wait for an opening in the narrower section of the pipe, and thus moves very slowly until it reaches the "release point," roughly where the narrower section of the pipe begins. Once it reaches the release point, that grain of sand starts moving much faster. That "stop and wait" period is a queue. The bottleneck is the cause of the queue; it's not the queue itself. What's important to note here is that the place where the pipe narrows is the bottleneck, but the sand (or water) actually moves most slowly right before the pipe begins to narrow. This brings us to . . .

Scott's third rule of bottlenecks: The symptoms of the bottleneck are (virtually) never observed at the actual location of the bottleneck.

In hydraulics, there's another useful concept: the "critical" bottleneck, defined as the one bottleneck that unless resolved will dictate the flow characteristics of a system. In Figure 2, you can see three sets of obstacles restricting the flow of water through the pipe. It's easy to see that obstacle 2 is restricting the flow the most. In this case obstacle 2 is the critical bottleneck, meaning that removing obstacles 1 and 3 won't actually improve the flow of water through the pipe.

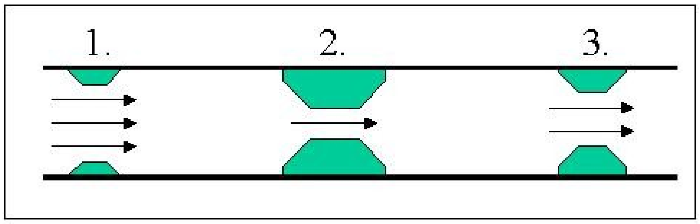


Figure 2: Multiple bottlenecks

More simply put . . .

Scott's fourth rule of bottlenecks: The critical bottleneck is the one bottleneck along a particular user path the removal of which will improve both performance and the ability to find other bottlenecks.

Exploring critical bottlenecks introduces us to the concept of multiple paths through a system. When you extend a system beyond a single pipe into a closed system, you often add alternate paths through that system. Figure 3 is an example of a closed hydraulics system.

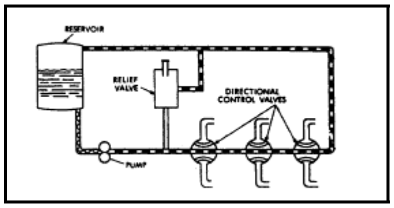


Figure 3: A closed hydraulics system

The difficulty of detecting bottlenecks in a system increases nearly exponentially with the number of possible paths through that system. Glancing at Figure 3, you can see that if there were a bottleneck in the pipe on the right side, the water could flow through the pipe in the center instead. This could lead to the appearance of a bottleneck in the center pipe, even though the bottleneck isn't there (see the third rule). Thus, it's important to remember . . .

Scott's fifth rule of bottlenecks: If you have multiple paths through a system and think there's a bottleneck, you should isolate each path and evaluate it separately.

In the system depicted in Figure 3, you can see some items other than pipes — pumps, valves, and a reservoir. If you think of the pipes as your network and the other items as your hardware (Web server, routers, and so forth), you quickly come up with . . .

Scott's sixth rule of bottlenecks: The bottleneck is more likely to be found in the hardware than in the network, but the network is easier to check.

The analogy between a closed hydraulics system and a Web-based application can actually go a lot further, but I think that's enough for now.

The Software Version

When people started using the term *bottleneck*, the concept of software hadn't even been invented. That fact alone should make us realize that the term probably has a special meaning when applied to software. Often the term *bottleneck* is used to refer to anything perceived to be slow in a software system, but this use of the term is imprecise and should be avoided. For instance, suppose one page on a Web site has several large graphic images on it. When a user loads this page, it may take a long time. But unless downloading the graphics causes some other activity in the system to slow down, it's not a bottleneck; it's just a slow page, or what I call a "slow spot."

The following definition was taken from *Load Testing for eConfidence*:

A bottleneck *is a point in a Web application where congestion and delay occur, slowing down the processing of requests and causing users to experience unacceptable service delays.*

The key to this definition is the word *congestion*. The next rule summarizes this example and definition.

Scott's seventh rule of bottlenecks: Unless other activities and/or users are affected by the observed slowness or its cause, it's not a bottleneck but a slow spot.

We'll discuss why this is significant in the next section.

Failure vs. Bottleneck vs. Slow Spot

In the course of defining *bottleneck*, we've made some distinctions that I'd like to spend a little more time on. The first is the distinction between a bottleneck and a failure. I think I've made it clear that a bottleneck is a slowdown, not a stoppage — meaning that the expected outcome is eventually achieved, regardless of how long it takes. For example, if you wait a long time but the requested Web page does eventually display properly, you've encountered a slow spot or a bottleneck. If, however, you wait and eventually are presented with an error page instead of the requested Web page, this is a failure.

The interesting twist to this distinction is that sometimes a very minor change can transform the failure back into a bottleneck. Consider the example above. It's entirely possible that in the second situation, a time-out (failure) occurred due to a Web server setting. Changing that setting and reexecuting your test may result in all activities being completed successfully but taking an unacceptable amount of time and slowing down all users (bottleneck).

For our purposes, anytime an error occurs, whether caused by a bottleneck or not, that error is a failure (you may prefer to call it a bug, defect, issue, or area of interest) and should be reported as such. When that failure causes other users to be unable to complete their tasks in the expected manner, that's a critical failure.

The main difference between a bottleneck and a slow spot is that a bottleneck has widely felt performance effects. A single large graphic can cause an annoying slow spot that may need to be resolved, but unless there are just a ton of people downloading that graphic (bottleneck caused by a popular activity) or your Web server is underpowered (bottleneck caused by insufficient infrastructure), it's just a slow spot with no real effect on the rest of the system.

I'm making a big deal of these distinctions because as we go through this group of articles about bottlenecks, we'll continually find failures and slow spots while we're chasing bottlenecks, and we'll need to distinguish among them to be able to take appropriate action.

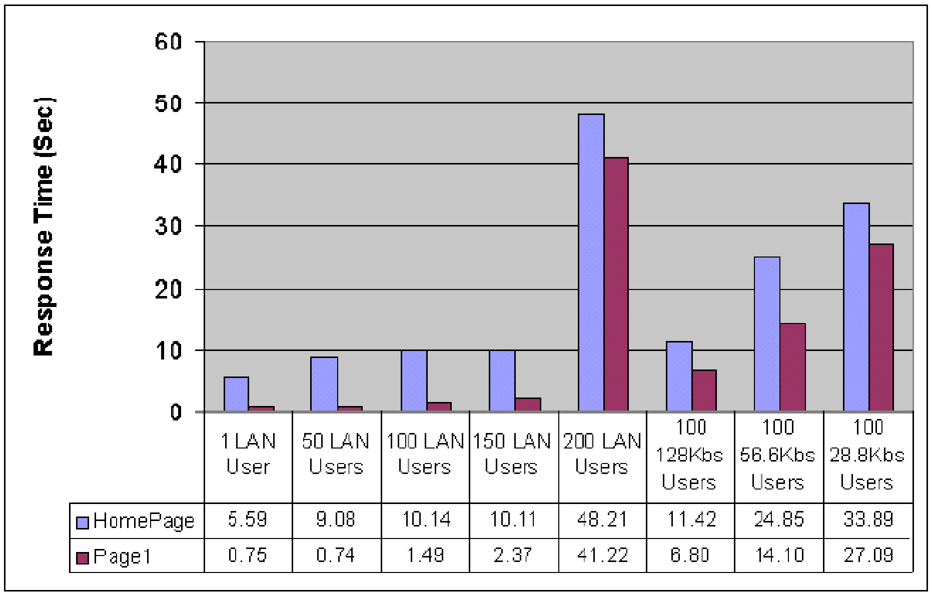
Identifying Bottleneck Suspects

There are at least as many ways of identifying bottleneck suspects as there are people who've ever observed something being slow when working on a computer. It's our job as performance test engineers to identify as many of those suspects as possible and then sort, categorize, verify, test, exploit, and potentially help resolve them. Let's discuss some common ways to identify bottleneck suspects. For now, let's not worry about whether these suspects might turn out to be failures or slow spots instead of bottlenecks.

Examine Response vs. Time Charts/Tables

If you're already executing performance tests, the most obvious place to look for bottleneck suspects is the response vs. time charts and tables. I'm assuming that by now you're familiar with these charts and tables, but if you'd like a refresher, Parts 6, 7, 8, and 9 of the "User Experience, Not Metrics" series discuss them in detail. By looking at the default charts that are displayed at the end of a test execution in TestManager, you'll immediately be able to see which timers or command IDs are noticeably slower than the others. Every one of these is a bottleneck suspect. Additionally, every timer or command ID that has a very large standard deviation (for example, a mean time that's *much* smaller than the 90th percentile time) is a suspect. While it's more likely that each of these is a symptom, a slow spot, or a failure, they're all worth noting and evaluating further.

If you've executed several tests under different loads, you could create the response time by test execution chart (described in "User Experience, Not Metrics," Part 9) to see if you have any load-related suspects. In Figure 4, for example, we see that performance seems to degrade significantly when there are more than 150 users and when slower connection speeds are emulated. These are examples of strong bottleneck suspects.



	1 LAN User	50 LAN Users	100 LAN Users	150 LAN Users	200 LAN Users	100 128Kbs Users	100 56.6Kbs Users	100 28.8Kbs Users
▨ HomePage	5.59	9.08	10.14	10.11	48.21	11.42	24.85	33.89
▨ Page1	0.75	0.74	1.49	2.37	41.22	6.80	14.10	27.09

Figure 4: Response time by test execution chart

Study Scatter Charts

Since the previous article in this series (Part 6) is devoted entirely to scatter charts, we won't spend much time on them here. In case you hadn't guessed it from reading Part 6, scatter charts are my favorite analysis tool, and the ease of identifying bottleneck suspects using them is one of the reasons. Simply put, any pattern that shows more than one dot (outlier) outside of your predefined acceptable performance levels is a potential bottleneck. The most likely suspects are patterns like the classic slowdown and banding patterns with bands above your acceptable performance level. Caching patterns are also good places to look, but as before, stacking patterns are more likely to result from bad test models or system failures.

Rely on Personal Observation

Personal observation is one of your best tools for identifying suspected bottlenecks. As you're creating scripts, you're using the application. You get to "feel" what performance is like, and you get a good idea of what types of activities cause the application to perform differently (better or worse) before you ever execute your first load test. These observations are extremely valuable, not only as a method of validating your scripts but also as a way to identify bottleneck suspects. Don't assume that your tool is better at detecting bottlenecks than you are. The ultimate users of the system are people, not load-generation tools; that in itself makes your opinion (based on observation) more valuable than the numbers the tool reports.

Listen to Third-Party Comments

Ultimately, other people will start using the system — generally while testing it. These folks will find all kinds of failures, slow spots, and bottlenecks, whether they realize it or not. It's important to talk to them and even observe them periodically to see and hear what they think of the system from a performance perspective. A simple comment like "I don't remember that search taking that long in the last version" is a big red flag that there may be a bottleneck hiding somewhere in the search activity. The best part about that flag is that the search may have seemed pretty fast to you, since you may never have used the last version. Don't assume that your personal observation will result in the same suspects as the observations of a casual user.

Confirming Suspects

After identifying a list of bottleneck suspects, the next step is to confirm them. Confirming a suspect won't necessarily confirm that the suspect is a bottleneck but rather will only verify that you've encountered some kind of performance issue (that is, either a bottleneck, a slow spot, or a failure) that warrants further research. In some cases you'll know at the time of confirmation which it is, and in other cases you won't know until much later in the process.

The key to confirming suspects is the ability to reproduce the results both exactly and manually. Until you can do at least one of those two things, the suspect is unconfirmed; and although unconfirmed suspects aren't necessarily invalid, they're generally given lower priority than confirmed suspects. Reproducing results with similar tests, with minimalist tests, and with not-so-similar tests can also offer clues to help you distinguish bottlenecks from other types of performance problems.

Reproduce Results Exactly

The single most important requirement for confirming a bottleneck suspect is the ability to reproduce the results that you or others have identified as indicating the suspect — that is, the symptoms. If the symptoms can't be reproduced, it's often the case that the observed condition that led to identifying the suspect was caused by something unrelated to the test.

For instance, while I'm building my scripts, I often end up with a whole list of bottleneck suspects from observation that I dismiss a week later when I can't reproduce them. The reason I dismiss them so easily is that I'm often developing my scripts against a development environment that's in a state of flux. If I can't reproduce my observation in the test environment, I do make a note to myself, but I've found that it's generally safe to assume that the development environment isn't stable enough to put much faith in my findings there.

This is just one example. I'm not recommending blindly dismissing everything you observe in a development environment. What I'm saying is use common sense. If you know that the developers are refreshing the database, promoting code, and rebooting servers multiple times a day, you can feel pretty confident that your suspect bottlenecks are suspect.

Reproduce Results Manually

If the suspected bottlenecks were observed while you were using a tool, you need to do whatever you can to reproduce the symptoms of the potential bottleneck manually. It's always possible that your test is causing a symptom that real users wouldn't encounter. Validate the accuracy of your scripts before considering a suspect bottleneck that was detected by a script to be confirmed. Even then, you'll want to try to reproduce that suspected bottleneck manually both while no one else is on the system and while the test is executing with the load at which the suspect bottleneck was first detected. The ability to observe the symptoms under one or both of those scenarios confirms a bottleneck suspect.

If the suspected bottleneck was identified through a third-party comment, try to reproduce the symptoms yourself. If you can't reproduce the symptoms, try to get the person who made the comment to reproduce the symptoms for you. If you and the person who made the comment have trouble reproducing the symptoms, take the time to try to determine what other factors could have contributed to the observation — for instance, the application being run on a different environment or a patch being applied that day.

Reproduce Results with Similar Tests

Without beating a dead horse, if you observe symptoms of a bottleneck using tools, be sure you can reproduce those symptoms with the same or similar tests — preferably with some variances, such as time of day, load, varying data, or additional activities that seem to be unrelated to the symptoms. The ability to reproduce the symptoms in similar situations is a strong indicator that the issue deserves further research and is therefore a confirmed bottleneck suspect.

Reproduce Results with Minimalist Tests

While you're confirming your suspects, you should try to reproduce the symptoms with the simplest test (manual or automated) possible. For instance, try to reproduce the symptoms without load, or without performing any other activities while logged in as that user. It's not absolutely necessary to be able to recreate the symptoms of the suspected bottleneck with a minimalist test for that suspect to be confirmed, but it will answer one of the first questions that the stakeholders are bound to ask and will aid in your ability to demonstrate the suspect.

Reproduce Results with Not-So-Similar Tests

Just like reproducing results with minimalist tests, reproducing results or symptoms with not-so-similar tests will help you demonstrate the existence of the suspect. It's also a big step toward identifying the differences among a slow spot, a failure, and a bottleneck. For example, a not-so-similar test may show that searching for a book as well as searching for a store near you on a retail site are both slow. If only searching for a store near you were slow, you might be tempted to think that something specific to that search was slow (slow spot), but knowing that both types of searches are slow may lead you to think that the database is poorly tuned (bottleneck).

Reporting Confirmed Suspects

I hinted at the importance of effectively reporting confirmed suspects earlier in this article. My experience shows that reporting suspected performance issues (failures, slow spots, and bottlenecks) is tricky business. You're often met with skepticism, disbelief, defensiveness, or dismissiveness . . . sometimes from different stakeholders in the same meeting! The first thing to remember is that you're not alone. Every performance test engineer who's ever reported a suspected issue has faced this. The second thing to remember is that if you've followed the approach outlined above, you have a confirmed, reproducible suspect to report. If you report it well, no one will be able to refute that it's a valid suspect.

On the other hand, if you present your suspects poorly, overstate or understate them, or don't report them at all, they may never get addressed. It's our job as performance test engineers to ensure that these suspects get taken seriously and addressed appropriately. Over the next few paragraphs I'll share with you some hints that I've found useful when reporting suspected bottlenecks.

Report Verbally

"Once long, long ago, in a galaxy far, far away," I was on my first performance-testing project as the performance test lead. I had developed and executed some tests that I was really proud of. I started analyzing the results of one test and found something. I was smart enough to execute that test again to verify that I could repeat it. As soon as I saw that I could repeat the pattern I'd found, I picked up the phone and called the lead architect.

Me: "I just ran some tests — you have a memory leak."

Architect: "Your tests are wrong — there's no memory leak."

Me: "I can reproduce it, you monitor the box, I'll rerun the test."

Architect: "OK, but there's no memory leak."

Fifteen minutes later I called back.

Me: "See, I told you there's a memory leak."

Architect: "Huh? Memory usage hasn't changed. I was about to call you to ask if you'd started your test yet."

Me: "What?!? The site is down — I just checked it manually. Are we looking at the same instance?!?"

Architect: "There's only one instance and the site is fine. Try it again."

Me: "OK . . . What did you do? How did you get it back up?"

Architect: "Nothing. Double-check your tests — there's no memory leak."

As you might imagine, that story could go on for a long, long time. In the end, after I'd lost virtually all of my credibility, we found that the pattern I was seeing was caused by a temporary license (a limited number of concurrent connections) being installed on one of our servers. As it turned out, I had a completely valid confirmed bottleneck suspect that I reported poorly. My mistake was deciding that I *knew* what the problem was instead of calling the lead architect and saying, "Hey, I'm getting some odd responses when I run tests with ten or more virtual users. It seems to have some symptoms like a memory leak, but I can't tell for sure what it is. When you have a chance, can you come down and take a look?" This was one of the biggest lessons of my career as a performance test engineer, now summarized as . . .

Scott's eighth rule of bottlenecks: When reporting bottleneck suspects, don't assume you know the cause, just report the symptoms.

More specifically, I've found the following advice to be useful when reporting suspected bottlenecks:

- Don't report suspected bottlenecks in a way that implies fault.
- Do describe all of the symptoms you've identified, not just the one you think is most relevant.
- Don't speculate on the cause of the bottleneck, even if you think you know what it is.
- Do describe all of the ways you've found to cause the symptoms.
- Don't get defensive when challenged — it really might be the fault of your test.
- Do be prepared to support your claims.

Report Visually

Most of the time, stakeholders will want to see charts and graphs demonstrating the symptoms of the suspected bottleneck. Since I devoted Parts 6 through 10 of the "User Experience, Not Metrics" series and Part 6 of this series to discussing how to display and interpret data visually, I won't discuss here which types of charts and tables are best to use. I will say that you should spend some time finding the best way to visually depict those symptoms and have those charts and/or tables ready to show when you report the suspected bottleneck.

Having the data available visually will almost always shift the focus of conversation from "Your test is wrong" to "Hmm . . . I wonder what would cause this odd pattern," and that's exactly what you're hoping for. At this stage your goal is to show the developers that you want to work with them, and the more information you can give them to help them draw their own conclusions the more they'll want to work with you to get even more information later.

Report Via Demonstration

Finally, no matter how smooth your words are or how convincing your charts and tables are, some folks will only believe you if you demonstrate the symptoms to them — or if you allow them to experience the symptoms themselves. Much like writing a good defect report for a functional test case, you should always have a step-by-step process prepared that others can follow to reproduce the symptoms on demand. It's even better if the process doesn't involve the test tool. Having this step-by-step process will save your credibility every time — especially if you remember my eighth rule and report only symptoms.

Is It Time to Tune?

Each of the four "finding bottlenecks to tune" articles will end with the question "Is it time to tune?" In this section we'll discuss when to jump out of the bottleneck detection and exploitation cycle and jump into the bottleneck resolution cycle.

While it may seem that all I've shown you how to do in this article is identify symptoms of potential issues (failures, slow spots, or bottlenecks) and report them, this is actually when most tuning begins. More times than I would have thought, when I report bottleneck suspects someone in the room responds, "Oops. I know what that is. Scott, I'll call you in a few hours and ask you to rerun your tests. I'm pretty sure this will be fixed."

As far as I'm concerned as a performance test engineer, this is the ideal situation. Nine times out of ten, I go back to my desk and work on something else for an hour or two, the developer calls, I rerun the test, and that suspected performance issue is gone. Some companies or project managers will ask you to document that. I actually recommend against documenting any performance issue that takes less than a day to resolve, but you'll obviously have to follow the guidance of your organization.

The bottom line is that while you're reporting the symptoms of your suspected performance issues, you'll generally find yourself engaged in conversations about the cause of the symptoms. If there's consensus as to both the cause of the symptoms and how it should be resolved, the attempt should be made to resolve that issue (that is, to tune) immediately. If either the cause or the resolution is unclear, you'll want to modify your tests to focus on resolving the issue. This is the topic of Part 8, "Modifying Tests to Focus on Failure or Bottleneck Resolution." There I'll show you how to further categorize performance issues into failures, slow spots, and bottlenecks, and how to isolate symptoms with the intent of gaining more information about the cause.

Summing It Up

Identifying symptoms of performance issues is the first step toward actually improving the performance of a system. In fact, simply identifying and reporting the symptoms is often enough to lead to performance improvement. Remember, however, that at this stage you shouldn't speculate as to the cause of the identified issue, and you should be on the lookout for the characteristic differences between failures, slow spots, and bottlenecks. In many cases, you won't know which it is until you've taken another step or two in the process, but you should always be on the lookout for telltale signs.

References

- *Load Testing for eConfidence* by Stefan Asböck (available from the Segue Web site)

Acknowledgments

- 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.