informIT

Analyzing Performance-Testing Results to Correlate Performance Plateaus and Stress Areas

Date: Jul 8, 2005 By Michael Kelly.

Mike Kelly builds on Scott Barber's work to show how you can combine performance-degradation curves and complex performance scenarios to help determine "good enough" quality for an application in terms of performance.

A year or so ago I had the pleasure of attending a conference at which Scott Barber gave two presentations on performance testing. The first presentation was on the effective presentation of performance test data; the second was on the modeling of application user communities. After watching both presentations and talking with Scott, I was able to draw a couple of insights. First, many times we focus too much on the problems that are easy to identify, rather than taking the time to determine where the real problems may be hiding. Second, we tend to performance test for the sake of performance testing, rather than taking the time to understand the usage of the application and the business drivers within which it operates. This behavior results in not knowing what to test and in not understanding how much performance testing is enough for the application.

In this article, I'll build on some of Scott's work to show how we can combine performance-degradation curves and complex performance scenarios to help determine "good enough" quality for an application in terms of performance. Throughout the article, I'll refer to Scott's work by providing a quick summary and stealing an example for illustration, and then move on to the next topic. I leave it to you to do the research necessary to fully understand the summarized content. This article is intended for the experienced performance tester or test manager.

Performance-Degradation Curves

Before we jump into the guts of this article, it might be good to establish some working definitions and concepts. Let's start with performance-degradation curves. In his article on creating a performance degradation curve, Scott Barber outlines a basic response-time degradation curve. If you're not familiar with this work, take a minute to read that article first; it sets the stage for what we're about to cover.

Figure 1 is an example of a response-time degradation curve. Degradation curves are common among performance testers; they go by various names, so forgive me if you know this curve by another name. A response-time degradation curve plots the response time experienced by the user against the user load. It's worth pointing out that the various user loads represented on one of these plots all use the same user-community model (explained in more detail later in this article). Later on, I'll discuss how to compare loads based on different models. This example shows the response times for two web pages (the home page and page 1) under differing loads (from 1 to 200 users). Curves like the one in Figure 1 are good for comparing specific page-response times across multiple tests using the same model, graphically displaying where performance starts to decline and where performance becomes unacceptable.

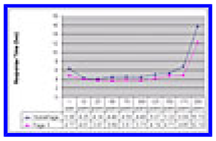


Figure 1 A basic response-time degradation curve.

The shape of a typical response-time degradation curve can be broken down into four regions (see Figure 2):

- The *single-user region* is just that—the response time for a single user on the system. This is useful for establishing a point of reference.

- The *performance plateau* shows the best performance you can expect under the specific conditions of that particular test without further performance tuning. This area represents good candidates for baselines and/or benchmarks.
- The *stress region* is where the application "degrades gracefully." Typically, the max recommended user load is the beginning of the stress region.
- The *knee in performance* is the point where performance "degrades ungracefully."

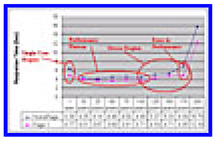


Figure 2 Four regions of the response-time degradation curve.

These regions are typically used by testers to help them determine where performance starts to degrade for any given portion of the application. It has been my experience that these charts are used primarily for two purposes:

- **The effective display of performance information, in an effort to show "good enough" performance or poor performance in relation to some stated requirement.** For example, if I had a requirement stating that the home page must load in under six seconds with 100 concurrent users, I could confirm that requirement using the chart in Figure 2. If the requirement was for 200 users, I could use the same graph to show that more work needs to be done to meet the requirement.
- **As a tool used to determine the knee in performance while performance tuning.** Where the knee occurs is the absolute maximum load you ever want your application/system to encounter. Data collected after the knee is the load data that exploits your critical bottleneck; this data is then used to research and correct performance bottlenecks. Many times this is an iterative process in an effort to push the knee in performance further to the right (or to a higher load).

While many testers are interested in the stress region and the knee in performance, in this article we'll take a slightly different view on this data.

Complex Performance-Testing Scenarios

Now that you know what a performance-degradation curve is, the next thing you need to understand is the modeling of performance scenarios. We'll build on the work of others again and review a modeling tool and practice that's already available. In his article on the user community modeling language (UCML™), Scott Barber shows a method to visually depict complex workloads and performance scenarios. When applied to performance testing, UCML can "serve to represent the workload distributions, operational profiles, pivot tables, matrixes, and Markov chains that performance testers often employ to determine what activities are to be included in a test and with what frequency they'll occur." I've used UCML diagrams to help me plan my performance testing, to document the tests I executed, and to help elicit performance requirements.

Figure 3 shows a sample UCML diagram for an online bookstore. This example has four types of users: new and existing users of the site, web site administrators, and vendors (people who sell their books on the web site). Each user starts at the home page; from there, their paths vary depending on what functions they need to perform. Some functions are unique to certain roles, and others are shared between roles. The model shows us all the potential paths through the application, as well as expected percentages of users for any given role or for any given path through the model. The legend in Figure 4 may be helpful as you review the diagram.

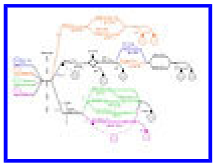
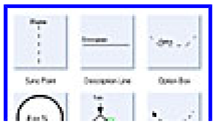


Figure 3 Sample UCML diagram for an online bookstore.



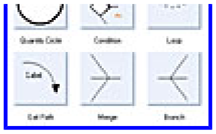


Figure 4 UCML 1.1 symbols.

Models like this allow us to create realistic (or reasonably realistic) performance test scenarios. Using a model like this, we can create a series of performance tests to measure application performance in terms of a specific user type or an aggregate of all user types, overall response given a specific load, effects of load on any given type of user, etc. The power behind a modeling approach like this is that it's intuitive to developers, users, managers, and testers alike. That means faster communication, clearer requirements, and better tests.

The Problem

We've looked at two commonly used tools for performance testers: performance-degradation curves and performance-usage models. Like any popular tool, these tools are often misused. I've seen performance-degradation curves (and/or very similar tools) used to justify more performance testing than is probably necessary. Many times, a performance tester will receive a requirement stating that a web page should load within X time under Y load. Using a performance degradation curve, the performance tester can find the knee in performance; if it's below the specified threshold, the cycle of application fix/modification and repeat testing can go on indefinitely—often with the only goal being rightward movement of the knee in performance along the performance-degradation curve.

While this is certainly a part of performance testing, when taken out of the context of the expected usage of the application, these requirements can be misleading. For which users should the page load in X seconds? Which users are logged into the system to create a load of Y? Is this requirement for the month of December, when the bookstore's web site is under the highest load; or is it during July, when there is very low traffic; or is it based on average values? Simply moving the knee in performance is not enough. We need to better understand the context in which that knee exists. Perhaps that knee is acceptable for certain circumstances and unacceptable in others.

Instead of focusing on what's wrong with the application performance, if we focus on what's right we may be able to better determine where problems exist—if we even *have* problems. By focusing on performance plateaus and stress areas (where the application is performing well) instead of the knee in performance, we're able to better understand the larger picture of how the application operates. Let's look at an example.

Developing Scenarios and Degradation Curves

Let's look at how we might determine whether our web site fulfills a performance requirement. Suppose we have a requirement that the home page should load within six seconds under a 150-user load. Using our user community model above, we can create complex workloads to execute against the site. We come up with the performance-degradation curve shown in Figure 5.

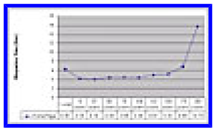


Figure 5 Home page basic response-time degradation curve.

We can see in Figure 5 that we've fulfilled our basic requirement. The results for the test represented in our usage model show that we have a response time around 5.33 seconds for the web site's home page. This is good news. Unfortunately, this is where most performance testers stop. While it's meaningful information, it's not enough information. We still know very little about how the home page loads with a load of 150 users. We only know how it loads with *those* 150 users ("those users" referring to the user percentages and user types defined in the model).

If we've taken the time to develop a model like the one in Figure 5, and then developed our performance scripts to represent that model, most likely it will be very easy for us to modify those scripts to reflect workload changes in the model. For example, in many tools, if I swap my New User and Member percentages, it's a simple variable change to reflect that in my scenario. I can then run the test again and create the response-time degradation curve for that new scenario. It might look something like Figure 6.

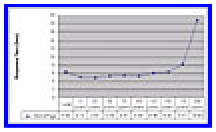


Figure 6 Home page response-time degradation curve with a greater percentage of new users.

What we find with this scenario is that by increasing the number of new users, we're increasing the number of users who need to create an account before they can make a purchase. This has an overall effect of raising the home page response time by about 20% because of the increased load on the server, due to users remaining in session longer and the server having to process more transactions for each user.

This variance is a good example of the ways in which you may want to vary your workload models while testing. This variant works on the assumption that the model initially developed was for the application while it was in production, after it had been in production for some time. That model neglected the fact that when the application is initially released, there will be no existing members. *All* users will be new users. If we had gone to production based on the first test alone, we would have lost the business of the unfortunate new users who couldn't access the home page because our server was too busy creating accounts for other users.

This is an example of including business scenarios in your testing. Following are some other examples:

- **Seasonal sales variances.** Fewer users search during the winter because they know what they're going to purchase. More users search in the summer because they're looking for a couple of new books to read over a break, but have no real preferences in mind when they go to the site.
- **New product releases.** We decide to add CD sales to the web site. Not only will it shift how many administrators are on the site adding new product; it may also increase searching on the web site for the first few months in which the new products are available, and then over time the usage pattern simply matches that of books.
- **Advertising campaigns or a partnership with another vendor.** Suppose we partner with Discover Card to offer a 10% discount to all Discover Card users on the site. While this may or may not affect load, it will most likely have a short-term effect on the usage pattern of existing members as they update their account information to switch their primary method of payment to Discover.

Many more examples are possible, and this is just for an eCommerce web site. None of the examples above *necessarily* affect load. That is, regardless of the scenario, we're still attempting to prove our 150-user requirement. Notice that I didn't include purchasing a Super Bowl commercial. The resulting 500,000-user load generated from that scenario is outside the requirement's scope.

The intent is to learn new information about the application so we can determine whether a) we have the correct requirement; b) a requirement has been fulfilled; and c) we need to perform other tests based on the new information uncovered with this testing. Through this type of testing, we can both confirm that the application really does what we think it does and learn information that may help us identify possible requirements omissions.

Correlating Performance Plateaus and Stress Areas for Each Scenario

So what do you do with all of these degradation curves once you've generated them? Let's look at correlating that data into some more useful format. Conceptually, we want to overlay all of this information so we can look at performance plateaus and stress areas side by side (see Figure 7).

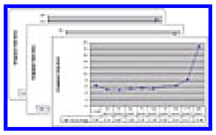


Figure 7 Conceptual overlay of response-time degradation curves.

Remember that we're not interested in what's broken—well, we *are*, but that's not what we're focusing on right now. We're focusing on what works within the system. We want to know what functionality appears to be stable, regardless of how we (realistically) vary the user scenarios for any given user load. To do that, we need to look at performance plateaus (the best performance you can expect without further performance tuning) and stress areas (the max recommended user load at which the application "degrades gracefully").

By looking at these areas instead of the knee in performance, we shift our focus from "performance tuning for the sake of performance tuning" to one of "good enough" performance, so we can concentrate on improving quality in those areas that really need improvement. The basic idea works like this: Because we can't possibly test everything, we need to continuously compare the present quality of the product against the cost and value of further improvement. "Good enough" testing is the process of developing a sufficient assessment of the current quality of the application, and then making both wise and timely decisions concerning the cost and value of further improvement. (James Bach has given a much better and more detailed definition than my feeble attempt to paraphrase his work; see the References section at the end of this article.)

In terms of performance testing, this means assessing what works, not only what doesn't. By knowing our plateaus and stress areas (those areas where our application works), we gain an accurate assessment of the quality of the product. If we never do this and we focus only on the areas that are broken, it's much more difficult to know the value of more testing, because we don't have a full picture of where the software is right now. How can we predict how much better the application can be with 40 more hours of testing, if we don't know how good it is now?

Back to our example. We've correlated the data and come up with a chart similar to the one in Figure 8. (Note that there are several acceptable ways to correlate this data; this is just one that works well for this specific example.)

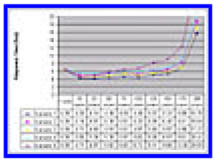


Figure 8 Correlated data for each scenario.

By analyzing this data, we can see that for three out of our five scenarios we didn't meet the requirement. That may be okay. It's not up to us to decide, we just report that information so project stakeholders can make informed decisions about the application's quality. What's more interesting is that by lining up all the scenario data we can see that four out of the five scenarios have the same basic performance plateaus and stress areas. The only exception is scenario 5. In this scenario, we enter our stress area right away at 25 users. A little more research may show that in this scenario our "vendor" users (6% of the total load based on the performance model above) run a specific type of report. This may be a problem that someone thinks we should fix, or it may be something we decide to go to production with because we think few vendors will run that specific report.

Let's look at another example. Figure 9 shows response-time degradation curves for the time needed to execute a search for our predicted usage patterns in each yearly quarter.

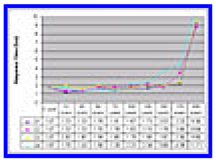


Figure 9 Search response-time degradation curves for each yearly quarter.

Q1, Q2, and Q3 are all fairly similar, as usage patterns remain about the same. That is, any given 100 users will do about the same thing regardless of the quarter, as long as it's not Q4. According to our usage models, users are doing much more searching in Q4. That is, for any given 100 users, the probability rises of those users executing several more searches than they did in the previous quarters. By correlating our data, we find that this has an effect on search response time for that quarter.

If a requirement says we need a year-round search response time of less than three seconds at a 150-user load, by using this information we can actually figure out what those users are doing differently. We can go back to the model, make changes, rerun the tests, re-correlate the data, and by using both tools attempt to determine where the problem is. Or we can tune our servers and make application changes and then rerun and re-correlate the tests. If we don't correlate the plateaus and stress areas, all we know is that if we make changes the response time may decrease. If we don't correlate this data, we don't know if it went down for all quarters, down for Q4 and up for the others, or if we're sliding the entire curve(s) left or right. Correlating the data gives us the bigger picture of the effect(s) of our changes on the entire system.

Going Forward

We covered a lot of ground in this article. If you're relatively new to performance testing, I advise reading the resources listed in the References section, working through any examples they may have, and then attempting to generate curves and models like these for your application. Once you build some familiarity with the tools, start combining them to better assess the current quality of your application. You can then use this information to improve your level of performance testing by better identifying the necessary exploratory performance tests and better informing project stakeholders about application performance.

References

More on performance testing:

- "Creating a Degradation Curve," part 10 of Scott Barber's "User Experience, Not Metrics" series
- "User Community Modeling Language (UCML™) v1.1 for Performance Test Workloads," by Scott Barber
- *The Web Testing Handbook*, by Steven Splaine and Stefan P. Jaskiel (STQE Publishing, 2001)

More on "good enough" quality, all by James Bach:

- "The Challenge of 'Good Enough' Software"
- "Good Enough Quality: Beyond the Buzzword"
- "A Framework for Good Enough Testing"

NOTE

Special thanks to Scott Barber for permission to reuse his materials and for his review, comments, and additions to this article. If you have specific questions related to his work, contact him at sbarber@perftestplus.com.

---

800 East 96th Street Indianapolis, Indiana 46240