WHITE PAPER

now from symantec. VERITAS™

Reducing the Risks of Performance Problems in J2EE Projects

BY JACK SHIRAZI

Author of Java Performance Tuning (O'Reilly®)

TABLE OF CONTENTS

Why you should read this 3
J2SE Performance monitoring and tuning 3
    J2SE profilers 3
    J2SE performance tuning 3
J2EE applications need different tuning tools 4
    J2SE profiler disadvantages in J2EE systems 5
J2EE performance monitoring requirements 5
Performance monitoring with VERITAS Indepth for J2EE and SmarTune™ Technology 7
Optimizing your project's chances of discovering performance inefficiencies 7
    Make a Performance Plan 7
    Include a Performance Testing Environment 7
    Model Scaled Behaviour as Early as Possible 7
    Performance Test, Performance Test, Performance Test 7
Performance testing never stops for J2EE: Plan for the production environment 8
Performance trends in production are "early warning" signs 8
In Summary, J2EE needs 9
About the author 9

WHY YOU SHOULD READ THIS

If you are delving into or entrenched in J2EE applications you will need to read this white paper because it will save you hundreds of hours spent in support of J2EE application performance management. The white paper highlights some of the key J2EE performance management considerations as you develop and deploy applications. Specific tips and recommendations are provided to help reduce risks and ensure overall project success.

J2SE PERFORMANCE MONITORING AND TUNING

Most J2EE projects will have one or more developers with some J2SE (Java 2 standard edition) performance tuning experience, and it is expected that they will be able to carry this experience over to J2EE (Java 2 enterprise edition) projects. Unfortunately this is not necessarily the case since J2EE performance management is significantly different from J2SE performance management. Being aware of the differences between J2SE and J2EE performance management will enable your J2EE performance management to be more productive and run more smoothly.

J2SE profilers

Let's take a look at what J2SE performance tuning involves. The typical J2SE application runs in one JVM (Java virtual machine) and can be tuned by running the application in the JVM with a profiler attached to find the performance problems (also known as bottlenecks or hot spots). Profilers are applications that monitor the JVM while the application is running. Profilers typically record:

- Which methods are running
- How long each method took run
- How many times particular methods have been invoked
- Which objects are being created
- Which objects are being garbage collected
- How many objects have been created and garbage collected

In most profilers, this information is recorded with stack traces, so that the stack can be analyzed to identify the location in the application where the method was invoked from or the object created. Profilers also offer further features such as snapshots, which allows you to profile and analyze a small portion of application activity independently of previous and subsequent activity; and threaded profiling so that you can focus on the threads that are causing the bottlenecks rather than having to untangle method invocations summarized over multiple threads.

Many IDEs (integrated development environments) come with their own profiler, and there are also many dedicated profilers which are mature independent products. A list of profilers can be found at http://www.JavaPerformanceTuning.com/resources.shtml. The Sun SDK is also delivered with several different profilers, though they are not as fully featured as most other profilers.

J2SE performance tuning

The design of the application is actually the most important thing affecting the performance, but that is what good designers and design patterns are for. If you get the design wrong, performance tuning the implementation is of limited help in reaching your performance targets. Assuming that the design does produce a potentially efficient application, then the application can still be slow because of the implementation. At this point the application needs to be performance tuned.

J2SE applications can usually be performance tuned by focusing on three main aspects of the runtime execution:

- Object creation
- Method execution time
- Deadlock avoidance

The primary function of a profiler is to identify those methods that are causing the bottlenecks in the application, and those objects which are being created the most or causing the most amount of heap usage. The two most useful reports that a profiler provides you with is the method profile and the object creation or heap profile. The method profile provides a list of the methods that the JVM has spent the most time executing in, thus identifying the hot spot methods in the application, i.e. those methods which, if they could be speeded up, will most effectively make the application run faster. The object creation or heap profile identifies which objects are being created most frequently, or are using the most amount of heap space. Combined with garbage collection statistics, the object creation profile helps to identify which objects should be targeted for reduced creation using some of the many performance tuning techniques that address excessive object creation (see for example Chapter 4 of Java Performance Tuning, O'Reilly, available online at http://www.oreilly.com/catalog/javapt/chapter/ch04.html).

Deadlock avoidance is a different problem. Some tools provide thread interaction analysis which help to detect potential deadlocks. But actual deadlock detection is fairly easy in the single JVM environment typical of J2SE applications. You wait for the application to freeze, and when it does you force a stack dump from the JVM and that stack dump will tell you exactly where the deadlock occurred. The stack dump can be generated by sending a "kill -QUIT" signal (kill -3) to the JVM process (on Unix), or alternatively entering the key sequence "<ctrl>\" in the window where the Java program was started (on Unix), or entering the key sequence "<ctrl><break>" in the window where the Java program is running (on Windows), or clicking the Close button on the command window (on Windows). The stack dump lists the Java stack of every thread currently running, and also gives the state of the thread. In the case of a deadlock, two or more threads will be in the "W" or wait state, indicating that they are waiting for locks to be released. The method listed at the top of the stack listing is the "current" method, i.e. the method that requested a lock and has caused the thread to move into a "wait" state as it waits for the lock to be granted. So it is quite easy to identify which methods are causing the deadlock.

J2EE APPLICATIONS NEED DIFFERENT TUNING TOOLS

J2SE applications can be effectively tuned by repeatedly running the application with a profiler, analyzing the profile reports to identify the current bottleneck and successively eliminating each identified bottleneck. The basic profiling skills are not too difficult to acquire, though working out how to speed up particular bottlenecks can be challenging.

J2EE applications include all the types of bottlenecks that you can find in J2SE applications, but also include more serious bottlenecks that involve resource allocation.

The distributed nature of J2EE applications expands the possible locations and types of bottlenecks. No two application environments are exactly alike; however J2EE applications have some common characteristics. There are often many components running in parallel across multiple JVMs. Data is passed to and received from multiple different databases and legacy systems. Interactions between components can produce bottlenecks; processing within components can produce bottlenecks; shared resources introduce potential bottlenecks at the resource acquisition stages (e.g. acquiring locks) and load balancing stages (e.g. optimally allocating CPU time to different threads and processes).

This larger space of potential bottlenecks can make it difficult to identify where the current performance problem is. J2SE tuners can focus on one JVM and two profiles (CPU and heap) to identify the current bottleneck comparatively easily. With J2EE applications, however, bottleneck identification is considerably more challenging. You can use a J2SE profiler to find or infer the bottlenecks of a J2EE application, but to do so you often need to run many more tests in many more configurations than you would like. Performance tuning productivity can decrease dramatically if you only have J2SE profilers available to performance tune a J2EE application.

J2SE profiler disadvantages in J2EE systems
Lets think through what we need for J2EE tuning. Firstly, consider the problems of using a J2SE profiler. Where do you put the profiler: many J2EE apps use more than one JVM. Even in single-JVM server situations, if you put a profiler on that server JVM, it slows down processing by significant amounts: doubling processing time is not unusual and I've even seen servers slowed to a tenth of the un-profiled speed. Profilers also use significant amounts of resources from the machine running the JVM, and this reduces the scalability of the J2EE application. The result is that you may not be able to reach or even approach the expected fully scaled system when running with a J2SE profiler, thus producing tests with limited usefulness. If you cannot test the system at expected maximum scale, you cannot be sure that shared resource conflicts at the higher scales have been optimized or eliminated.

Testing practicalities are also compromised. Consider that when testing your J2EE system with a J2SE profiler, you may need to run multiple tests with different configurations, while the system is possibly running at one tenth of the speed you get in development and scaling is limited to 10% of the expected system. This not only reduces performance testing productivity, but makes testers much more likely to take shortcuts. Some configurations are simply not tested; others have their tests cut short; tests are avoided unless there is a compelling reason to run them; developers cannot afford the time to run performance tests and performance testers cannot afford the time to run even slightly unstable systems, so resulting in conflict rather than co-operation. And, crucially, the limited scalability of the performance testing environment means that you could be missing some important behaviour that only appears at higher user and transaction rates.

Yet another problem of J2SE profilers is that they are essentially designed assuming that they will profile a single-user application, which is fine for most J2SE applications. However, J2EE applications are often multi-user applications, or more generally concurrent-request applications. That is, J2EE applications are normally designed to handle multiple requests invoked and running concurrently. Using a J2SE profiler in a J2EE application often results in no clear correlation between incoming requests and corresponding methods executed or objects created in the application. Although J2SE profilers can distinguish threaded operations, J2EE applications frequently do not have a one-to-one correlation between request invocations and threads, except at the application entry points.

J2EE PERFORMANCE MONITORING REQUIREMENTS
J2SE profilers essentially monitor and log various aspects of the JVM. J2EE performance monitoring needs to monitor and log far more aspects of the overall J2EE system in order to productively identify performance problems. In the last section we identified some of the limitations imposed by using J2SE profilers in J2EE systems. Now we'll use those limitations and other considerations to identify what we want in a J2EE performance monitoring tool.

1. **Monitoring and logging of components and their interfaces.** Clearly, we need a tool that will monitor and log the important aspects of the J2EE system. The tool should be pre-configured to monitor all standard potential performance bottlenecks of a J2EE system, and should be configurable so that application specific functionality can be easily monitored. The potential performance bottlenecks come mainly from three generic locations: processing within components; interfacing between components; and communication between components. Inter-component communication overheads, e.g. network transfers, are distinct from interfacing overheads such as marshalling or other conversions such as SQL request generation.

2. **Low overhead**. The monitoring tool needs to impose only a low overhead on the J2EE system. Less than a 5% overhead is required for the tool to be really useful; targeting a 1% overhead is an ideal. Low overhead performance monitoring lets you run the monitor all the time without worrying about how server behavior is being modified because of profiling overheads. This means that you can leave monitoring switched on at all times, in development systems, in test systems, and in production systems, without any serious performance degradation. This contrasts markedly with J2SE profilers which have such a large overhead that running with a profiler on at all times would kill a project.

3. **Requests mapped to methods**. The monitor should have the ability to correlate incoming requests with subsequently monitored methods, components and communications. It should be possible to easily correlate things like request-to-bean-to-db-queries, so that when it comes to analysing different types of requests, you can identify which requests are causing which different types of bottlenecks. Without this capability you can end up targeting many more bottlenecks than are necessary, or else spending significant time in analysis trying to determine which requests map to which bottlenecks.

4. **Data storage for later analysis**. The performance monitor should store all the data persistently, so that you can decouple analysis from the actual running of the server. It is incredibly annoying to have things happen during a test run, and have no way of later analysing the data because one graph or another only displayed in real-time, with no logged data.

5. **Analysis tools**. Ultimately, the monitor is there to produce data which will be analyzed for problems. The more analysis tools you have to analyze the logged data, the more productive your analysis will be. Analysis is the whole purpose of this monitoring tool, and the procedure becomes much more productive if there are integrated analysis tools to analyze the data.

6. **Scaling capability to scale with the application**. You need the monitoring tool to scale with the application, so that you can use it without worrying about having to restrict tests, and so that you can deploy the monitoring with the application in the production environment. I'll discuss later about why it is essential to carry on monitoring performance in a production environment.

7. **Automatic analysis**. There are two very large components to the effort you need to allocate to performance management. Planning and carrying out tests takes one large amount of effort, and analyzing the results of monitoring the application in tests and after deployment is the second task that takes a significant amount of effort. One major benefit of the J2EE monitoring tool I'm outlining here is that it provides potentially huge productivity benefits compared with J2SE tools, productivity benefits which benefit the first task of planning and carrying out tests. But in the absence of any extra help, the second task (analysis) is not any more productive than with J2SE tools. Additionally, analysis of performance logs is also a more complex skill to acquire. So any help in analysis is a huge benefit. The more a J2EE tool can provide in the way of automatically analyzing its performance logs, the more productive your performance management becomes.

PERFORMANCE MONITORING WITH VERITAS INDEPTH FOR J2EE AND SMARTUNE™ TECHNOLOGY

VERITAS Indepth for J2EE is a server-side tool that monitors J2EE application performance with all the capabilities previously outlined. VERITAS Indepth for J2EE includes the SmarTune automatic analysis tool, which does things such as look for too many calls of various types, or for specific types of calls taking too long. Where SmarTune finds such a performance problem, it automatically analyses those cases to produce plain language advice on what to do to improve the performance of those requests that are producing the problematic performance. Of course, SmarTune doesn't replace the need for analysis to be performed by a person, but for many types of performance bottlenecks SmarTune will quickly identify the problem and suggest a solution, increasing the productivity of the performance tuner.

OPTIMIZING YOUR PROJECT'S CHANCES OF DISCOVERING PERFORMANCE INEFFICIENCIES

Selecting the appropriate J2EE performance monitoring tool, such as VERITAS Indepth for J2EE, is one big step towards optimizing your project's chances of productively eliminating performance inefficiencies. But there are other steps in J2EE performance management, which also help minimize the risks of having performance problems in J2EE projects.

Make a Performance Plan

Its hardly big news that if you plan for something, you're more likely to achieve your goal than if you don't make a plan. Performance planning should be just another aspect of your overall project plan, assigning people, time, money, products, setting performance goals, and so on.

Include a Performance Testing Environment

The performance testing environment is quite simply a testing environment that at every stage of your project aims to come as closely as possible to the expected final deployment. At the design stage, this would typically be a prototype. Even a simple prototype can identify architectural bottlenecks which are much cheaper to fix early on in a project, but very expensive to alter or workaround if left unchecked. Creating or acquiring correctly simulated real-world data is often the most challenging part of setting up a performance testing environment, but is essential if you want to see how the system truly scales.

Model Scaled Behaviour as Early as Possible

By modeling scaled behaviour as early as possible (multiple users, lots of data), you optimize your chances of finding the kinds of performance problems which do not appear at lower scales but can seriously limit the scalability of the application.

Performance Test, Performance Test, Performance Test

Test the performance of your application as often as possible. Again this is not big news. Everyone knows or should know that the more testing you do the more likely you are to reach your goals for QA, and the same is true for performance. So test! But be aware that performance testing is not QA testing. The tools are different, the tests are different and what you are looking for is different. QA looks for failure modes. Performance testing looks to see how close you are to your performance goals. QA flags a failure, sends a report to the developers, and most failures are identified easily, as one or two lines of code being wrong. (I say most, there are always a few bugs that are really difficult to find, and a few which require a large change, but most are simple errors). Performance tests are more complicated, and really quite different. Typically:

- You run your multi-user test for hours,
- then you spend even longer analyzing the performance logs to see where the bottlenecks are,
- then you need to re-run the test in different configurations,
- then you compare the analyses to see how configuration changes affected the performance,
- then you need to rerun the test with specialized logging or profiling in place which makes the test even slower but lets you pinpoint the bottleneck,
- then you have to tune the bottleneck which is another half day at least.

And that is just one round of the ongoing struggle. This long drawn out procedure is one of the reasons why the better your analysis tools, the better and faster your performance tuning goes, hence the importance of tools like VERITAS Indepth for J2EE and features such as SmarTune.

PERFORMANCE TESTING NEVER STOPS FOR J2EE: PLAN FOR THE PRODUCTION ENVIRONMENT

Make sure you plan for resources to carry on performance analysis in the production environment. The production environment is the real test, and if you don't carry on monitoring performance statistics you are losing the most valuable information you can get. And the performance testing is carried out by the users of your application, so it actually becomes cheaper to test! And note that just as performance testing in development is not QA, so production performance monitoring is not bug reporting. In production, bugs are more or less three types. There are your "oh my God, stop production and fix it now!" type of bug, which is probably corrupting data; there's the type of bug that is reported and gets assigned a severity level from high priority down, which will get fixed for the next application version or patch; and there are bugs which are basically feature requests: "it should do this too, but doesn't".

Production performance analysis, though, is exactly the same as it was in development, except that, as I said earlier, the testers are providing performance information for free. But they are only providing the information for free if you are collecting that information. It is important to have performance logging in place with the deployed production application so that all that valuable information is not simply discarded. And since performance analysis is the same for production and development, then if the tool is the same one that was used in development you can use those same skills you acquired during development. And in fact this works both ways: if you start using your performance monitoring tool in production, you can carry both the tool and the skills you've learned through to the next J2EE development.

Because of it's low overhead and high scalability, VERITAS Indepth for J2EE is suitable for both development and production environments, so maximizing both your investment and productivity.

PERFORMANCE TRENDS IN PRODUCTION ARE "EARLY WARNING" SIGNS

The production environment with performance logging in place gives you more than just free info to analyse. You also get performance trends. Basically, all performance aspects monitored produce a normal recurring pattern data. If any data changes from the usual pattern, this signals that there is probably a problem of some sort. Sudden spikes or dips, gradual increases or decreases, they all indicate that something is changing. More often than not, that "something" is a problem that will blow up. But usually the "something" will blow up in the future.

The performance trend is telling you "Hey, look out, something over HERE is not what you expected". Mostly, it's as simple as a gradually depleting resource. CPU, memory, disk, objects in a pool, whatever. Maybe it is simply that the users like the application so much that they are doing more and more with it. Now that's a great trend to be able to report. But even that trend needs capacity planning, or suddenly your great successful application is a victim of its own success, and the users are complaining about how performance has suddenly slowed to a crawl. Like a suddenly popular website. If you have the early warning signs, you can put in more capacity. If you have the late warning signs (the server crashes from being overloaded), you look silly and lose goodwill.

IN SUMMARY, J2EE NEEDS

- A low impact server-side performance monitoring and logging tool
- A tool that can be used both in development and production
- The best analysis tools possible
- The more automated analysis the better
- Lots of performance testing in development
- Performance trends monitored in production

ABOUT THE AUTHOR

Jack Shirazi is the director of JavaPerformanceTuning.com, the website that provides enormous coverage of information on all aspects Java Performance Tuning, and is also the author of the book Java Performance Tuning, published by O'Reilly as one of the books in their very successful Java series. Jack regularly publishes articles about aspects of Java performance, and provides consulting services for customers in need of Java performance management and tuning.