Pro Java EE 5 Performance Management and Optimization

Steven Haines

Apress®

Pro Java EE 5 Performance Management and Optimization

Copyright © 2006 by Steven Haines

ISBN-13: 1-59059-610-2

ISBN-10: 978-1-59059-610-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewers: Mark Gowdy, Dilip Thomas
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editors: Heather Lang, Nicole LeClerc
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Susan Glinert
Proofreader: Liz Welch
Indexer: Broccoli Information Management
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski
Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 6

Performance Tuning Methodology

"I have been reading about performance tuning on our application server vendor's Web site, and it looks so complicated. There are all of these ratios that I need to watch and formulas to apply them to. And which ones are the most important? What's going on with this?"

John was getting frustrated with his tuning efforts. He had his team implementing the proactive performance testing methodology that I helped him with, but the concept of the by-the-book performance tuning was evading him.

"Don't let those ratios fool you—there is a much better approach to performance tuning. Let me ask you, when you take your car in for service, does the service technician plug your car into a computer and tell you what's wrong with it, or does he ask you to describe your problems?"

"Well of course he asks me about my problems, otherwise how would he know where to start looking? A car is a complicated machine," John replied.

"Exactly. There are so many moving parts that you wouldn't want to look at each one. Similarly, when tuning an enterprise application, we want to look at its architecture and common pathways to optimize those pathways. When we step back from application server ratios and focus on what the application does and how it uses the application server, the task becomes much easier." From the look on his face, I could see that he got it. He saw that the focus of tuning should be on the application, not on abstract ratios that he did not understand.

Performance Tuning Overview

Performance tuning is not a black art, but it is something that is not very well understood. When tasked with tuning an enterprise Java environment, you have three options:

- You can read through your application server's tuning documentation.
- You can adopt the brute-force approach of trial and error.
- You can hire professional services.

The problem with the first approach is that the application server vendor documentation is usually bloated and lacks prioritization. It would be nice to have a simple step-by-step list of tasks to perform that realize the most benefit with the least amount of effort and the order in which to perform them, but alas, that does not exist. Furthermore, when consulted on best practices of tuning options, application server vendors typically advise that the optimal configuration depends on your application. This is true, but some general principles can provide a strong starting point from which to begin the tuning process.

The second approach is highly effective, but requires a lot of time and a deep understanding of performance measurements to determine the effect of your changes. Tuning is an iterative process, so some trial and error is required, but it is most effective when you know where to start and where you are going.

The final approach, paying someone else to tune your environment for you, is the most effective, but also the most expensive. This approach has a few drawbacks:

- It is difficult to find someone who knows exactly how to handle this task.
- Unless knowledge transfer is part of the engagement, you are powerless when your application changes; you become dependent on the consultant.
- It is expensive. Talented consultants can cost thousands of dollars per day and expect to provide at least two or three weeks' worth of services.

If you decide to go this route, when you're looking for a reliable and knowledgeable resource, consider the consultant's reputation and referrals. Look for someone who has worked in very complicated environments and in environments that are similar to yours. Furthermore, you want an apples-for-apples tuner: do not hire a WebSphere expert to tune WebLogic. These programs are similar but idiosyncratically different.

In addition, always include knowledge transfer in the engagement statement of work. You do not want to be dependent on someone else for every little change that you make in the future. Keep in mind, though, that it is a good idea to re-engage a proven resource for substantial changes and for new applications. Encourage your team to learn from the consultant, but do not expect them to be fully trained by looking over someone's shoulder for a couple of days.

The cost of a consultant's services may be high, but the cost of application failure is much higher. If you are basing your business and your reputation on an application, then a $50,000 price tag to ensure its performance is not unreasonable. Perform a cost-benefit analysis and a return on investment (ROI) analysis, and see if the numbers work out. If they do, then you can consider hiring a consultant.

One alternative that I have neglected to mention is to befriend an expert in this area and ask him to guide your efforts. And that is the focus of this book and in particular this chapter. In this chapter, I share with you my experience tuning environments ranging from small, isolated applications to huge, mission-critical applications running in a complex shared environment. From these engagements, I have learned what always works and best practice approaches to performance tuning. Each application and environment is different, but in this chapter I show you the best place to start and the 20 percent of tuning effort that will yield 80 percent of your tuning impact. It is not rocket science as long as someone explains it to you.

Load Testing Methodology

Before starting any tuning effort, you need to realize that tuning efforts are only effective for the load that your environment is tuned against. To illustrate this point, consider the patient monitoring system that I have alluded to in earlier chapters. It is database intensive in most of its functionality, and if the load generator does not test the database functionality with enough load, then you can have no confidence that your configuration will meet the demands of users when you roll out the application to production.

With that said, how do you properly design your load tests? Before the application is deployed to a production environment and you can observe real end-user behavior, you have no better option than to take your best guess. “Guess” may not be the most appropriate word to describe this activity, as you’ve spent time up-front constructing detailed use cases. If you did a good job building the use cases, then you know what you expect your users to do, and your guess is based on the distribution of use cases and their scenarios.

In the following sections, we’ll examine how to construct representative load scenarios and then look at the process of applying those load scenarios against your environment.

Load Testing Design

Several times in this book I have emphasized the importance of understanding user patterns and the fact that you can attain this information through access log file analysis or an end-user monitoring device. But thus far I have not mentioned what to do in the case of a new application. When tuning a new application and environment, it is important to follow these three steps:

1. Estimate
2. Validate
3. Reflect

The first step involves estimating what you expect your users to do and how you expect your application to be used. This is where well-defined and thorough use cases really help you. Define load scenarios for each use case scenario and then conduct a meeting with the application business owner and application technical owner to discuss and assign relative weights with which to balance the distribution of each scenario. It is the application business owner’s responsibility to spend significant time interviewing customers to understand the application functionality that users find most important. The application technical owner can then translate business functionality into the application in detailed steps that implement that functionality.

Construct your test plan to exercise the production staging environment with load scripts balanced based off of the results of this meeting. The environment should then be tuned to optimally satisfy this balance of requests.

■ **Note** If your production staging environment does not match production, then there is still value in running a balanced load test; it allows you to derive a correlation between load and resource utilization. For example, if 500 simulated users under this balanced load use 20 database connections, then you can expect 1,000 users to use approximately 40 database connections to satisfy a similar load balance. Unfortunately, linear interpolation is not 100 percent accurate, because increased load also affects finite resources such as CPU that degrade performance rapidly as they approach saturation. But linear interpolation gives you a ballpark estimate or best practice start value from which to further fine-tune. In Chapter 9 I address the factors that limit interpolation algorithms and help you implement the best configurations.

After deploying an application to production and exposing it to end users, the next step is to validate usage patterns against expectations. This is the time to incorporate an access log file analyzer or end-user experience monitor to extract end-user behavior. The first week can be used to perform a sanity-check validation to identify any gross deviations from estimates, but depending on your user load, a month or even a quarter could be required before users become comfortable enough with your application to give you confidence that you have accurately captured their behavior.

User requests that log file analysis or end-user experience monitors reveal need to be reconstructed into use case scenarios and then traced back to initial estimates. If they do match, then your tuning efforts were effective, but if they are dramatically different, then you need to retune the application to the actual user patterns.

Finally, it is important to perform a postmortem analysis and reflect on how estimated user patterns mapped to actual user patterns. This step is typically overlooked, but it is only through this analysis that your estimates will become more accurate in the future. You need to understand where your estimates were flawed and attempt to identify why. In general, your users' behavior is not going to change significantly over time, so your estimates should become more accurate as your application evolves.

Your workload as an enterprise Java administrator should include periodically repeating this procedure of end-user pattern validation. In the early stages of an application, you should perform this validation relatively frequently, such as every month, but as the application matures, you will perform these validation efforts less frequently, such as every quarter or six months. Applications evolve over time, and new features are added to satisfy user feedback; therefore, you cannot neglect even infrequent user pattern validation. For example, I once worked with a customer who deployed a simple Flash game into their production environment that subsequently crashed their production servers. Other procedural issues were at the core of this problem, but the practical application here is that small modifications to a production environment can dramatically affect resource utilization and contention. And, as with this particular customer, the consequences can be catastrophic.

Load Testing Process

If you want your tuning efforts to be as accurate as possible, then ideally you should maintain a production staging environment with the same configuration as your production environment. Unfortunately, most companies cannot justify the additional expense involved in doing so and therefore construct a production staging environment that is a scaled-down version of production. The following are three main strategies used to scale down the production staging environment:

- Scale down the number of machines, but use the same class of machines
- Scale down the class of machines
- Scale down both the number of machines (size of the environment) as well as the class of machines

Unless financial resources dedicated to production staging are plentiful, scaling down the size of an environment is the most effective plan. For example, if your production environment maintains eight servers, then a production staging environment with four servers is perfectly accurate to perform scaled-down tuning against. A scaled-down environment running the same class of machines (with the same CPU, memory, and so forth) is very effective because you can understand how your application should perform on a single server, and depending on the size, you can calculate the percentage of performance lost in interserver communication (such as the overhead required to replicate stateful information across a cluster).

Scaling down classes of machine, on the other hand, can be quite problematic. In many cases, it is necessary—for example, consider a production environment running in a $10 million mainframe. Chances are that this customer is not going to spend an additional $10 million on a testbed. When you scale down classes of machine, then the best your load testing can accomplish is to identify the relative balance of resource utilizations. This information is still interesting because it allows you to extract information about which service requests resolve to database or external resource calls, the relative response times of each service request, relative thread pool utilization, cache utilization, and so on. Most of these values are relative to each other, but as you deploy to a stronger production environment, you can define a relative scale of resources to one another, establishing best "guess" values and scaling resources appropriately.

To perform an accurate load test, you need to quantify your projected user load and configure your load tester to generate a graduated load up to the projected user load. Each step should be graduated with enough granularity so as not to oversaturate the application if a performance problem occurs.

Wait-Based Tuning

I developed the notion of *wait-based tuning* by drawing from two sources:

- Oracle database tuning theory
- IBM WebSphere tuning theory

I owe a debt of thanks to an associate of mine, Dan Wittry, who works in the Oracle tuning realm. Dan explained to me that in previous versions of Oracle, performance tuning was based upon observing various ratios. For example, what is the ratio of queries serviced in memory to those loaded from disk? How far and how frequently is a disk head required to move? The point is that tuning a database was based upon optimizing performance ratios. In newer releases of the Oracle database, the practice has shifted away from ratios and toward the notion of identifying wait points. No longer do we care about the specifics of performance ratio values; we're now concerned with the performance of our queries. Chances are that a database serving content well will maintain superior performance ratios, but the ratios are not the primary focus of the tuning effort—expediting queries is.

After reading through tuning manuals for IBM WebSphere, BEA WebLogic, Oracle Application Server, and JBoss, I understood well the commonalities between their implementations and the similarity between application server tuning and database tuning: a focus on performance ratios. While IBM addressed performance ratios, it traveled down a different path: where in an application can a request wait? IBM identified four main areas:

- Web server
- Web container
- EJB container
- Database connection pools

Furthermore, IBM posed the supposition that the best place for a request to wait is as early in the process as possible. Once you have learned the capacities of each wait zone, then allow only that number of requests to be processed; force all others to wait back at the Web server. In general, a Web server is a fairly light server: it has a very tight server socket listening process that funnels requests into a queue for processing. Threads assigned to that queue examine the request and either forward it to an application server (or other content provider) or return the requested resource. If the environment is at capacity, then it is better for the Web server to accept the burden of holding on to the pending request rather than to force that burden on the application server.

Tuning Theory

IBM's paradigm provides better insight into the actual performance of an application server and makes as much sense as Oracle's notion of wait points. The focus is on maximizing the performance of an application's requests, not on ratios.

Equipped with these theories, I delved a little further into the nature of application requests as they traverse an enterprise Java environment and asked the question, Where in this technology stack can requests wait? Figure 6-1 shows the common path for an application request.

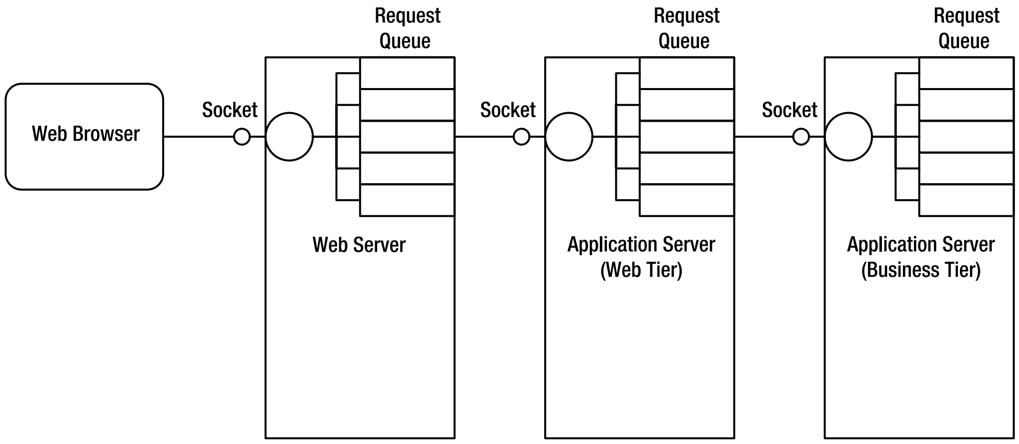


Figure 6-1. *Common path an application request follows through a Java EE stack*

As shown in Figure 6-1, requests travel across the technology stack through request queues. When a browser submits a Web request to a Web server, the Web server receives it through a listening socket and quickly moves the request into a request queue, as only one thread can listen on a single port at any given point in time. When that thread receives the request, its primary responsibility is to return to its port and receive the next connection. If it processed requests serially, then the Web server would be capable of processing only one request at a time—not very impressive. A Web server's listening process would look something like the following:

```
public class WebServer extends Thread {
...
 public void run() {
  ServerSocket  serverSocket = new ServerSocket( 80 );
  while( running ) {
    Socket s = serverSocket.accept();
    Request req = new Request( s );
    addRequestToQueue( req );
  }
}
```

While this is a very simplistic example, it demonstrates that the thread loop is very tight and acts simply as a pass-through to another thread. Each queue has an associated thread pool that waits for requests to be added to the queue to process them. When a request is added to the queue, a thread wakes up, removes the request from the queue, and processes it, for example:

```
public synchronized void addRequestToQueue( Request req ) {
  this.requests.add( req );
  this.requests.notifyAll();
}
```

Threads waiting on the request's object are notified, and the first one there accepts the request for processing. The actions of the thread are dependent on the request (or in the case of separation of business tiers, the request may actually be a remote method invocation). Consider a Web request against an application server. If the Web server and application are separated, then the Web server forwards the request to the application server and the same process repeats. Once the request is in the application server, then the application server needs to determine the appropriate resource to invoke. In this example, it is going to be either a servlet or a JSP file. For the purpose of this discussion, we will consider JSP files to be servlets.

■**Note** JSP files are convenient to build because in simple implementations you are not required to create a `web.xml` file containing `<servlet>` and `<servlet-mapping>` entries. But in the end, a JSP file will become a servlet. The JSP file itself is translated into an associated `.java` servlet file, compiled into a `.class` file, and then loaded into memory to service a request. If you have ever wondered why a JSP file took so much time to respond the first time, it is because it needs to be translated and compiled prior to being loaded into memory. You do have the option to precompile JSP files, which buys you the ease of development of a JSP file and the general performance of a servlet.

The running thread loads the appropriate servlet into memory and invokes its `service()` method. This starts the Java EE application request processing as we tend to think of it. Depending on your use of Java EE components, your next step may be to create a stateless session bean to implement your application's transactional business logic. Rather than your having to create a new stateless session bean for each request, they are pooled; your servlet obtains one from the pool, uses it, and then returns it to the pool. If all of the beans in the pool are in use, then the processing thread must wait for a bean to be returned to the pool.

Most business objects make use of persistent storage, in the form of either a database or a legacy system. It is expensive for a Java application to make a query across a network to persistent storage, so for certain types of objects, the persistence manager implements a cache of frequently accessed objects. The cache is queried, and if the requested object is not found, then the object must be loaded from persistent storage. While caches can provide performance an order of magnitude better than resolving all queries to persistent storage, there is danger in misusing them. Specifically, if a cache is sized too small, then the majority of requests will resolve to querying persistent storage, but we added the overhead of checking the cache for the requested object, selecting an object to be removed from the cache to make room for the new one (typically using a least-recently used algorithm), and adding the new object to the cache. In this case, querying persistent storage would perform much better. The final trade-off is that a large cache requires storage space; if you need to maintain too many objects in a cache to avoid *thrashing* (that is, rapidly adding and removing objects to and from the cache), then you really need to question whether the object should be cached in the first point.

Establishing a connection to persistent storage is an expensive operation. For example, establishing a database connection can take between a half a second and a second and a half on average. Because you do not want your pending request to absorb this overhead on each request, application servers establish these connections on start-up and maintain them in connection pools. When a request needs to query persistent storage, it obtains a connection from the connection pool, uses it, and then returns it to the connection pool. If no connection is available, then the request waits for a connection to be returned to the pool.

Once the request has finished processing its business logic, it needs to be forwarded to a presentation layer before returning to the caller. The most typical presentation layer implementation is to use JavaServer Pages (JSP). As previously mentioned, using JSP can incur the additional overhead of translation to servlet code and compilation, if the JSPs are not precompiled. This up-front performance hit can impact your users and should be addressed, but from a pure tuning perspective, JSP compilation does not impact the *order of magnitude* of the application performance: the impact is observed once, but there is no further impact as the number of users increases.

Observing the scenario we have been discussing, we can identify the following wait points:

- Web server thread pool
- Application server or tier thread pool
- Stateless session bean or business object pool
- Cache management code
- Persistent storage or external dependency connection pool

You can feel free to add to or subtract from this list to satisfy the architecture of your application, but it is a good general framework to start with.

Tuning Backward

The order of wait points is as important as what they are waiting on. IBM's notion of sustaining waiting requests as close to the Web server as possible has been proven to be a highly effective tuning strategy. It is better to queue requests in a business logic–lite tier so as to minimize the impact on the business tier. Furthermore, if the request has already sequestered a Web server thread and it is not ready for processing, then why should we sequester an application server thread and database connection as well? If we do, then we'll add additional burden upon the business tier, and the pending request will not get processed faster anyway.

The approach of forcing requests to wait at the appropriate application point is to open all wait points until the one at the end of the tube saturates. Scale down the saturated resource until it no longer saturates—this is its capacity. For example, if an application server instance can service only 50 database requests per second, then you want to send through only enough requests to generate at most 50 database requests per second; any more requests will simply queue up at the database.

Now that the limiting wait point is identified, tune the application backward by tightening down each wait point to facilitate only the capacity of the limiting wait point. Continue with this exercise until you reach the Web server request queue. If your load is significantly more than the capacity of your limiting wait point, then you might want to configure your Web server to redirect to a "Try your request again later" page.

CASE STUDY: DEPARTMENT OF EDUCATION

I visited the Department of Education for a specific state under the following situation. Once per year the governor of that state announces that school report cards are in and urges parents to log on to the state government's Web site to see how their child's school compared to the rest of the schools in the county and state. The site's traffic patterns are unique: three days per year the load is close to that of Yahoo!, and the rest of the year the site sits virtually idle. My role was to determine the capacity of the application and environment to determine the appropriate cutoff values to set. The customer and I performed an abbreviated version of the tuning exercise described in this section to meet the state's deadline, but in the end it was highly successful.

The point of this story is that once you understand the capabilities of your environment, your Web server can throttle your load to push through only the amount of load you can support. You would rather ask someone to try again later than allow all of your users to suffer, right?

JVM Heap

Although we will discuss heap garbage collection in depth in the next chapter, it is worth noting here that it's probably the biggest wait point in any untuned Java EE environment. The application server and all of its applications run inside of the JVM heap. The Java memory model is such that unreachable objects are eligible to be discarded by a garbage collection process. Garbage collection can run in one of two modes: minor or major. *Minor* collections run relatively quickly and inexpensively, whereas *major* collections typically freeze all code running in the JVM. Furthermore, while minor collections run quickly (usually under a tenth of a second), major collections can take much longer (I have seen some take up to ten seconds to run).

Because all application requests run inside a JVM and are subject to a garbage collection process, your tuning efforts may be fruitless with an untuned heap. In my opinion, tuning memory can have the biggest impact of any tuning effort you perform.

Wait-Based Tuning Conclusions

Rather than focus on performance ratios that infer the health of an application, your tuning efforts should be focused on the application itself. The process is defined as follows:

1. Walk through the application architecture and identify the points where a request could potentially wait.
2. Open all wait points.
3. Generate balanced and representative load against the environment.
4. Identify the limiting wait point's saturation point.
5. Tighten all wait points to facilitate only the maximum load of the limiting wait point.
6. Force all pending requests to wait at the Web server.
7. If the load is too high, then establish a cutoff point where you redirect incoming requests to a "Try your request again later" page. Otherwise, add more resources.

I have effectively implemented this type of tuning exercise at customer sites, and the results have far surpassed my previous efforts of maximizing performance ratios. It is a more difficult and time-consuming exercise, but the results are superior and worth it.

Tuning Example

Tuning a Java EE environment is a difficult exercise to convey in print, but I'll try to illustrate the general process this section by describing the environment I'm tuning at each stage:

- The state of relevant performance metrics (wait points)
- The state of heap garbage collections
- CPU utilization of relevant servers

Figure 6-2 shows a typical medium-sized production environment. The environment consists of two Apache Web servers configured to distribute load across four WebLogic servers running on two separate physical machines and communicating with two clustered Oracle instances.

The environment shown in Figure 6-2 represents the minimum requirements for a high-availability topology: two Web servers (in case one crashes), two physical WebLogic machines, and two databases. In Chapter 8, we'll delve deeper into the benefits of running multiple JVMs on the same physical machine, but for now it suffices to say that doing so can improve performance and resource utilization. The projected utilization for this environment is 2,000 simultaneous users.

Table 6-1 describes the initial configuration for this environment.

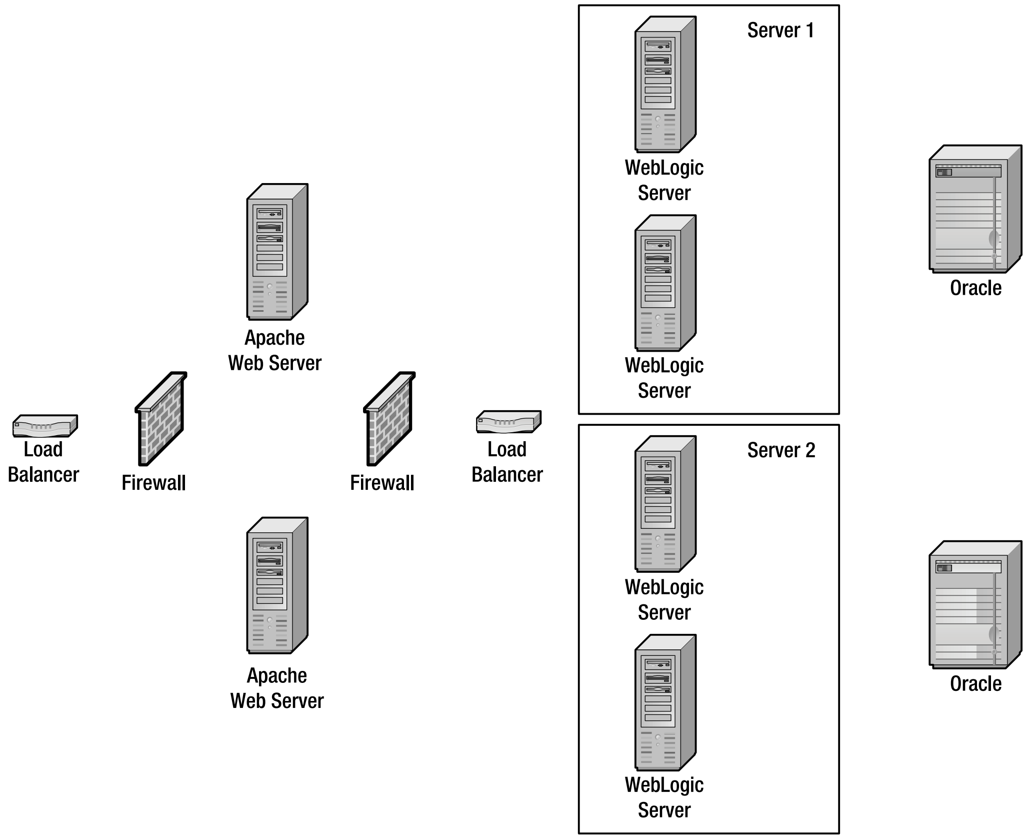


Figure 6-2. *By configuring two physical machines for each logical enterprise tier (Web tier, business/application tier, and database tier), high availability can be more easily attained.*

Table 6-1. *Initial Configuration of the Example Tuning Environment*

Metric	Value
Web server threads	150
Web server CPUs	2
Web server memory	4GB total memory
Application server threads	50
Application server database connections	30
Application server entity bean cache size	500
Application server heap	1GB
Application server CPUs	2
Application server memory	4GB
Database CPUs	2
Database memory	4GB

■Note Each of the values in Table 6-1 is defined for each physical machine. For example, two Web server CPUs means that there are two CPUs per Web server. Figure 6-2 shows two physical Web servers, so four total CPUs are dedicated to the two Web server instances.

The application is an MVC architecture utilizing stateless session beans for business processes and a small set of entity beans for transactional persistence and caching.

I configured my homegrown load tester to climb up to 1,000 users over 20 minutes and then added 50 additional users every 5 minutes. Graduating this load ensures I'm able to identify resource bottlenecks as they occur—increasing the load too rapidly may saturate the environment and mask the true nature of performance problems.

In addition to configuring the environment and the load tester, we can install monitoring software on the operating system of each machine, performance monitoring software on the application server, and tag-and-follow code instrumentation that traces requests from the HTTP server to the application server and finally records the length of calls to the database. We're looking for the following key items here:

- Thread pool utilization and pending requests for each thread pool
- Database connection pool utilization and requests waiting for a connection
- Entity bean caches
- Stateless session bean pools
- Application bottlenecks (to determine if there is anything I can tune in the environment to allow more application throughput)

The sections that follow analyze the testing iterations.

Iteration #1

The first iteration climbed up to 1,200 users before grossly exceeding its SLAs. Table 6-2 lists the observed performance metrics of the sample environment for the first iteration of wait-based performance tuning.

Table 6-2. *Performance of the Sample Environment for Iteration #1*

Metric	Value
Database CPUs	40 percent utilization
Application server CPUs	20 percent utilization
Database connection pool	100 percent utilization
Threads waiting for a connection	10 pending threads
Application server threads	100 percent utilization
Application server pending requests	20 pending requests
Entity bean cache hit ratio	40 percent
Web server thread utilization	50 percent

In this configuration, the Web server is sending significant load to the application server, to the point where the application server uses all of its threads and causes 20 requests to wait. If we look a little deeper, we see that the application server database connection pools are at 100 percent utilization and ten threads are waiting for a database connection; the database connection pool is maxed out and this is what is causing requests to wait.

The next question we need to ask is, Is this the capacity of the database (meaning that we should harden our application servers to limit database load and push pending requests back to the Web server), or does our database have more capacity? In this case, the database CPU is only at 40 percent and can support additional load. Therefore, let's increase the database connection pool size to force more load into the database.

The entity bean cache hit ratio is only 50 percent, meaning that half of our calls to entity beans require a database query. If we can increase this hit ratio (by increasing the size of the cache to service more requests from memory), then we will take additional burden off of the database.

Finally, to validate the effect of these metrics on our application, the tag-and-follow code instrumentation reported two primary wait points in most service requests:

- Application server internal Web server (for example, `HTTP GET /…`)
- JDBC call to `getConnection()`

When requests wait at the application server's internal Web server, or in other words, when there's a significant amount of time between when the application server receives the request and when it invokes the appropriate servlet, then the application server is waiting for an execution thread to process the request. If calls to `getConnection()` take an excessively long time, the request asked the database connection pool for a connection, but all connections were in use, hence it had to wait for a connection to be returned to the pool.

From this analysis, we implement the changes shown in Table 6-3 and test again.

Table 6-3. *New Configuration for the Sample Environment for Iteration #1*

Metric	Value
Database connection pool	60
Application server thread pool	100
Entity bean cache size	1000

The additional database connections should push more load to the database, the increase in application server threads should drive even more load to the database, and the change to the entity bean cache should reduce the number of database calls. Together, these changes should increase the utilization of the application server and database, and hence increase the throughput of the application.

Iteration #2

In the second iteration, the load climbed to 1,800 users before exceeding its SLAs. Table 6-4 lists the observed performance metrics for the second iteration of wait-based performance tuning.

Table 6-4. *Performance of the Sample Environment for Iteration #2*

Metric	Value
Database CPUs	70 percent utilization
Application server CPUs	50 percent utilization
Database connection pool	100 percent utilization
Threads waiting for a connection	10 pending threads
Application server threads	100 percent utilization
Application server pending requests	20 pending requests
Entity bean cache hit ratio	80 percent
Web server thread utilization	75 percent

In this configuration, the Web server continues to send significant load to the application server, and the application server again uses all of its threads and causes ten requests to wait. Looking at the database connection pools, we see that the application server database connection pools are at 100 percent utilization and ten threads are waiting for a database connection.

The database CPU utilization is now up to 70 percent utilization, so there is still some additional capacity we can force out of it. Therefore, let's increase the database connection pool size to force more load into the database.

The entity bean cache hit ratio is at 80 percent, meaning that the majority of calls to entity beans do not require a database query. This is an adequate hit ratio; we may try to increase it later, but it is fine for initial tuning efforts.

The tag-and-follow code instrumentation reported the same two wait points in most service requests:

- Application server internal Web server
- JDBC call to `getConnection()`

From this analysis, we implement the changes shown in Table 6-5 and test again.

Table 6-5. *New Configuration for the Sample Environment for Iteration #2*

Metric	Value
Database connection pool	90
Application server thread pool	120

The additional database connections should push even more load to the database, and the increase in application server threads should drive more load through the application to the database. Our hope is to increase the utilization of both the database and application server.

Iteration #3

In the third iteration, the load climbed to 2,200 users before exceeding its SLAs, and then performance degraded substantially. Table 6-6 lists the observed performance metrics for the third iteration of wait-based performance tuning.

Table 6-6. *Performance of the Sample Environment for Iteration #3*

Metric	Value
Database CPUs	95 percent utilization
Application server CPUs	40 percent utilization
Database connection pool	100 percent utilization
Threads waiting for a connection	20 pending threads
Application server threads	100 percent utilization
Application server pending requests	40 pending requests
Entity bean cache hit ratio	80 percent
Web server thread utilization	85 percent

In this configuration, the Web server continues to send significant load to the application server, and the application server again uses all of its threads and causes 40 requests to wait. Looking at the database connection pools, we see that the application server database connection pools are at 100 percent utilization and 20 threads are waiting for a database connection.

The database CPU utilization is now up to 95 percent, so it has become saturated and cannot effectively process requests. Therefore, let's decrease the database connection pool size to try to bring the CPU utilization down; at 95 percent utilization, all queries are slow.

The entity bean cache hit ratio stayed around 80 percent, meaning that the majority of calls to entity beans do not require a database query.

The tag-and-follow code instrumentation reported the same two wait points in most service requests:

- Application server internal Web server
- JDBC call to `getConnection()`

From this analysis, we implement the changes shown in Table 6-7 and test again.

Table 6-7. *New Configuration for the Sample Environment for Iteration #3*

Metric	Value
Database connection pool	75
Application server thread pool	100

The decrease in database connections should reduce the stress on the database (allowing for better throughput), and the decrease in application server threads should reduce the number of threads waiting for a database connection.

Iteration #4

In the fourth iteration, the load climbed to 2,200 users before exceeding its SLAs, and then performance degraded gradually. Table 6-8 lists the observed performance metrics for the fourth iteration of wait-based performance tuning.

Table 6-8. *Performance of the Sample Environment for Iteration #4*

Metric	Value
Database CPUs	85 percent utilization
Application server CPUs	80 percent utilization
Database connection pool	100 percent utilization
Threads waiting for a connection	5 pending threads
Application server threads	100 percent utilization
Application server pending requests	10 pending requests
Entity bean cache hit ratio	80 percent
Web server thread utilization	85 percent

In this configuration, the Web server continues to send significant load to the application server, and the application server again uses all of its threads and causes ten requests to wait. Looking at the database connection pools, we see that the application server database connection pools are at 100 percent utilization and five threads are waiting for a database connection.

The database CPU utilization is now at 85 percent, which is optimal—the database can efficiently satisfy its requests and at the same time is neither underutilized nor overutilized.

The entity bean cache hit ratio stayed around 80 percent, meaning that the majority of calls to entity beans do not require a database query.

Because we have found the capacity of the database to service the application, it is time to harden the application server thread pools to stop threads from waiting for database connections and then reduce the Web server threads to hold requests at the Web server.

From this analysis, we implement the changes shown in Table 6-9 and test again.

Table 6-9. *New Configuration for the Sample Environment for Iteration #4*

Metric	Value
Web server threads	125
Application server thread pool	90

We hope that decreasing the number of application server threads will allow for the highest application throughput, but reduce the number of threads waiting for database connections. Furthermore, by reducing the number of Web server threads that handle application server requests, we hope to throttle the load at the Web server and avoid application server saturation.

Iteration #5

In the fifth iteration, the load climbed to 2,200 users before exceeding its SLAs, and then performance degraded gradually. Table 6-10 lists the observed performance metrics for the fifth iteration of wait-based performance tuning.

Table 6-10. *Performance of the Sample Environment for Iteration #5*

Metric	Value
Database CPUs	85 percent utilization
Application server CPUs	85 percent utilization
Database connection pool	95 percent utilization
Threads waiting for a connection	0 pending threads
Application server threads	90 percent utilization
Application server pending requests	0 pending requests
Entity bean cache hit ratio	80 percent
Web server thread utilization	85 percent

The environment looks beautiful under this configuration:

- The database CPU is running at 85 percent utilization, which is optimal when the application is under stress.
- The application server CPU is at 85 percent utilization, with 90 percent of its threads in use, indicating it is well utilized and not oversaturated.
- Database connection pool utilization is at 95 percent, with no pending threads, which means we are sending enough threads into the application server to properly utilize the database, but not enough to cause threads to wait.
- The Web server thread utilization is good—well used yet not oversaturated.

Although we did not perform a formal capacity assessment, we can see from this exercise that the approximate capacity of our application running in this environment is 2,200 users. If we need to support additional load, we either add additional hardware or tune the application, the database, or the database queries. Simple changes to the application to reduce database calls can have significant impact on the capacity of the application. For example, if we can reduce the number of queries that a single request spawns from ten to five, then the database's capacity to support the application increases substantially. The load capacity is not necessarily reflective of the performance of the database; rather, it's reflective of the ability of the database to service the application. If the application abuses the database, then performance will suffer.

Application Bottlenecks

At a high level, enterprise Java performance problems can occur in one of four categories:

- Application code
- Platform
- External dependency
- Load

A performance problem can occur inside of application code as the result of slow-running algorithms, memory utilization, or simply the response of the application to increased load or changes to usage patterns. For example, code may run well under low load and a controlled set of data, but when the load increases or the quantity of data increases substantially, then performance problems may manifest. As we saw in the last chapter, using bubble sort to sort 100 objects functions well, but when subjected to 5,000 objects, bubble sort falls apart. This is an application code issue, but it manifests itself only under load.

The platform includes the entire technology stack upon which your Java application runs: the application server, JVM, operating system, hardware, and network. The bottom line is that numerous performance tuning parameters across the technology stack can dramatically affect the performance of your application.

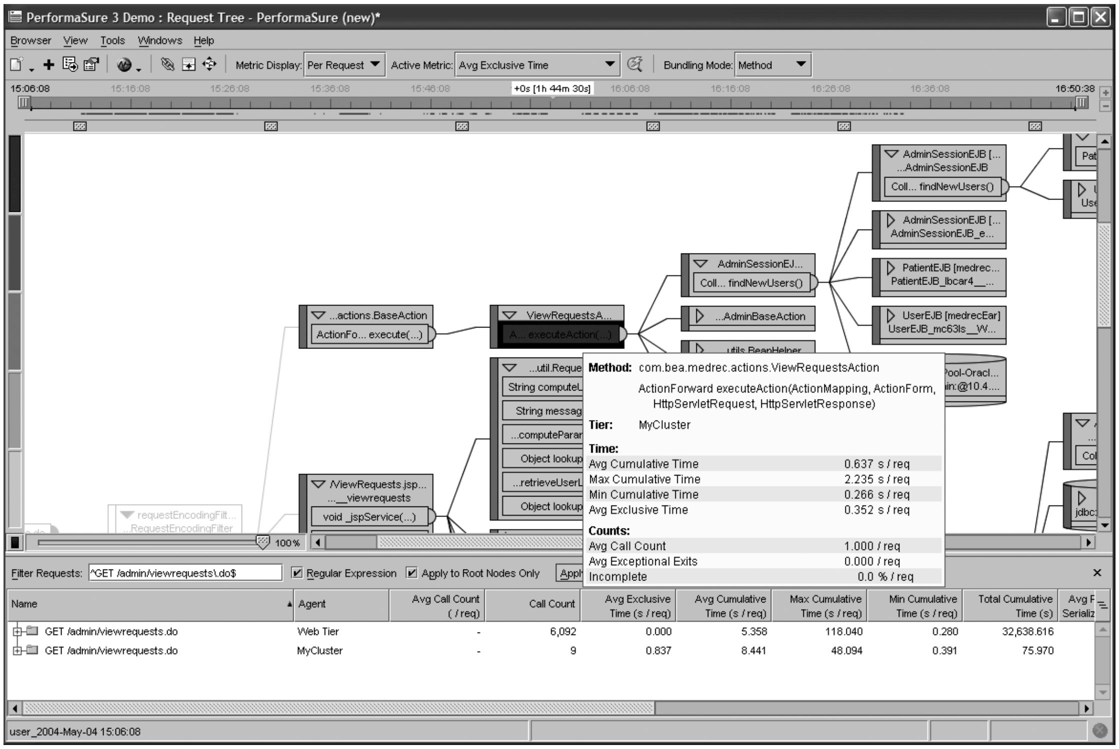
A Java EE application would be fairly useless if it did not interact with some kind of external dependency. External dependencies include databases, legacy systems, and any other system that your application interacts with, such as Web services or proprietary servers. Each external dependency has a communication mechanism (usually a connection pool) that if not configured properly can cause serious performance problems.

When application code is tuned, the platform is properly configured, the external services are running optimally, but the environment is not responding acceptably, then the application business logic simply cannot support the given load. Each Java EE environment has a capacity that it can support before becoming saturated. At this point, the only solution is to add additional hardware resources or restructure the environment. For example, a single Linux box running JBoss has a finite load that it can support, and if it is subjected to more load, then it will undoubtedly fail.

The key to properly diagnosing application bottlenecks is to first triage the performance problem to determine whether the root cause is the application code, platform, external dependency behavior, or load. You accomplish this by implementing a depth of monitoring technologies across your entire application environment, including Web servers, application servers, the database, external dependencies, operating systems, firewalls, load balancers, and network communication pathways. If you can isolate the problem to application code, then the common problems can be categorized as one of the following:

- Poor algorithms
- Memory utilization or object life cycles
- Programmatic contention

Poor algorithms manifest themselves through slow-running methods. The best way to find slow-running methods is by implementing a tag-and-follow monitoring solution. The analysis of an application running with this level of monitoring identifies slow-running service requests, which can be expanded down to the method level to identify offending methods. You are looking for a slow-running method in the call path that is not waiting for any resource (a thread, a database connection, and so on), is not executing a database query, and does not make an external call to another server. In other words, you want to identify methods that are actually using the application server CPU for an inordinate amount of time, as illustrated by Figure 6-3.



Metric Display: Per Request · Active Metric: Avg Exclusive Time · Bundling Mode: Method

15:06:08 · +0s [1h 44m 30s] · 16:50:38

Method: com.bea.medrec.actions.ViewRequestsAction
ActionForward executeAction(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)

Tier: MyCluster

Time:

Avg Cumulative Time	0.637 s / req
Max Cumulative Time	2.235 s / req
Min Cumulative Time	0.266 s / req
Avg Exclusive Time	0.352 s / req

Counts:

Avg Call Count	1.000 / req
Avg Exceptional Exits	0.000 / req
Incomplete	0.0 % / req

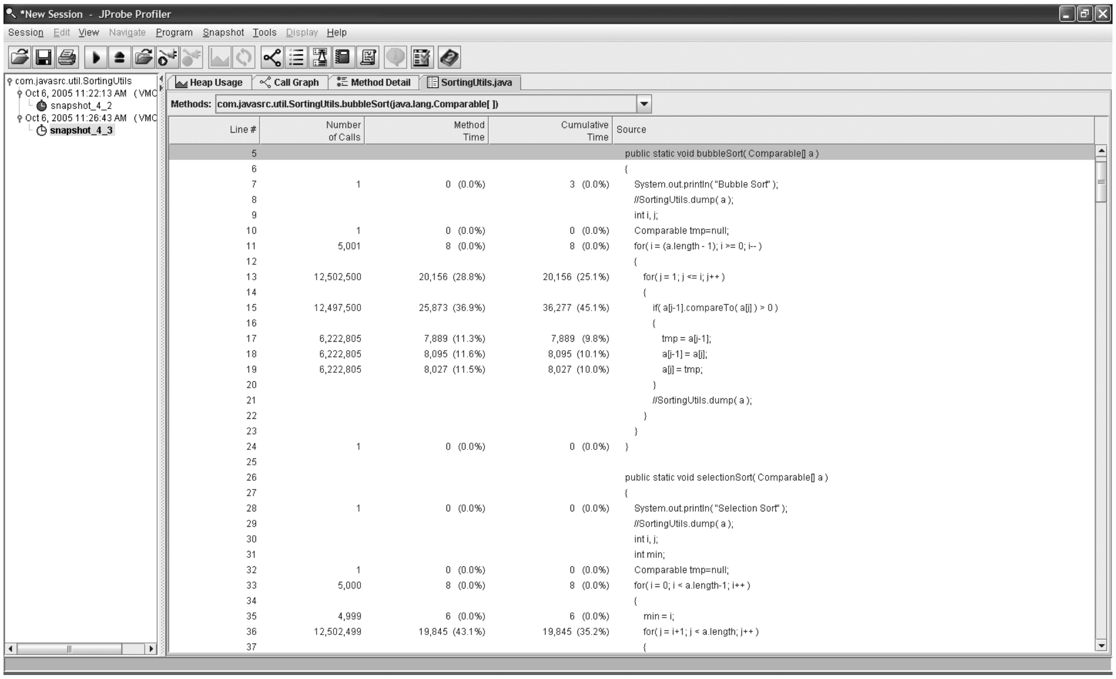
Filter Requests: ^GET /admin/viewrequests\.do$ · ☑ Regular Expression · ☑ Apply to Root Nodes Only

Name	Agent	Avg Call Count (/ req)	Call Count	Avg Exclusive Time (s / req)	Avg Cumulative Time (s / req)	Max Cumulative Time (s / req)	Min Cumulative Time (s / req)	Total Cumulative Time (s)	Avg F Serializ
GET /admin/viewrequests.do	Web Tier	-	6,092	0.000	5.358	118.040	0.280	32,638.616	
GET /admin/viewrequests.do	MyCluster	-	9	0.837	8.441	48.094	0.391	75.970	

user_2004-May-04 15:06:08

Figure 6-3. *The executeAction() method is taking a significant amount of the total processing time for the GET /admin/viewrequests.do service request.*

After slow-running service requests have been identified and it's determined that the root cause is a slow-running method, the resolution plan is to replay the service request inside of a code profiler that will identify the line or lines of code contributing most to the problem. If the performance of the method is unacceptable, then the code needs to be refactored and tuned for better performance. For example, Figure 6-4 shows the code for the bubble sort algorithm running inside of a code profiler, revealing that the poor performance relates to the sheer number of comparison calls (it is making nearly 12.5 million object comparisons).


*New Session - JProbe Profiler
Session Edit View Navigate Program Snapshot Tools Display Help
com.javasrc.util.SortingUtils
Oct 6, 2005 11:22:13 AM (VMC
snapshot_4_2
Oct 6, 2005 11:26:43 AM (VMC
snapshot_4_3
Heap Usage
Call Graph
Method Detail
SortingUtils.java

Methods: com.javasrc.util.SortingUtils.bubbleSort(java.lang.Comparable[ ])

Line #	Number of Calls	Method Time	Cumulative Time	Source
5				public static void bubbleSort(Comparable[] a)
6				{
7	1	0 (0.0%)	3 (0.0%)	System.out.println("Bubble Sort");
8				//SortingUtils.dump(a);
9				int i, j;
10	1	0 (0.0%)	0 (0.0%)	Comparable tmp=null;
11	5,001	8 (0.0%)	8 (0.0%)	for(i = (a.length - 1); i >= 0; i--)
12				{
13	12,502,500	20,156 (28.8%)	20,156 (25.1%)	for(j = 1; j <= i; j++)
14				{
15	12,497,500	25,873 (36.9%)	36,277 (45.1%)	if(a[j-1].compareTo(a[j]) > 0)
16				{
17	6,222,805	7,889 (11.3%)	7,889 (9.8%)	tmp = a[j-1];
18	6,222,805	8,095 (11.6%)	8,095 (10.1%)	a[j-1] = a[j];
19	6,222,805	8,027 (11.5%)	8,027 (10.0%)	a[j] = tmp;
20				}
21				//SortingUtils.dump(a);
22				}
23				}
24	1	0 (0.0%)	0 (0.0%)	}
25				
26				public static void selectionSort(Comparable[] a)
27				{
28	1	0 (0.0%)	0 (0.0%)	System.out.println("Selection Sort");
29				//SortingUtils.dump(a);
30				int i, j;
31				int min;
32	1	0 (0.0%)	0 (0.0%)	Comparable tmp=null;
33	5,000	8 (0.0%)	8 (0.0%)	for(i = 0; i < a.length-1; i++)
34				{
35	4,999	6 (0.0%)	6 (0.0%)	min = i;
36	12,502,499	19,845 (43.1%)	19,845 (35.2%)	for(j = i+1; j < a.length; j++)
37				{

Figure 6-4. *Poor algorithms can be identified by running suspect application code inside a code profiler.*

Memory utilization problems can be difficult to identify, especially if they are subtle. The key to diagnosing an application memory leak is to observe the performance of the heap over a significant period of time, such as several hours to several days depending on how apparent the memory leak is. Objects will be created and destroyed in the heap as a natural result of a running application, but the key indicator that differentiates a potential memory leak from normal behavior is the *valleys* in the heap. If the values of the valleys consistently increase, then there is a portion of memory that the garbage collector is not able to reclaim, which indicates a potential memory leak. Figure 6-5 displays a sample heap exhibiting a potential slow memory leak.

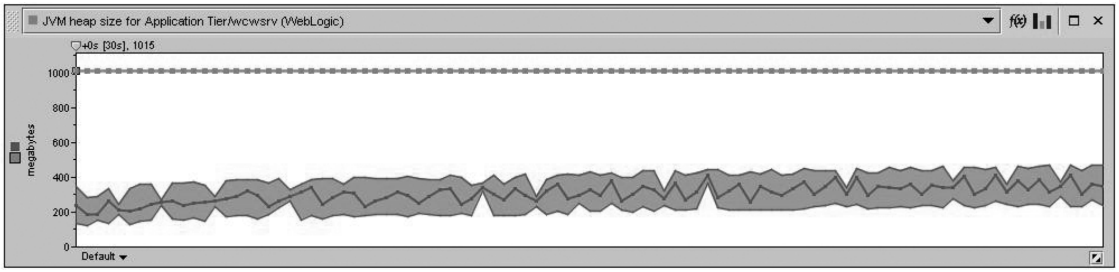


Figure 6-5. *Over time in a heap with a slow memory leak, the number of valleys increases.*

Only by observing the pattern of increasing heap valleys can you discern that you have a memory leak. Eventually this behavior will lead to an application server crash, or if the leak is inside a session object, the garbage collector may be able to reclaim memory before a crash.

Once you have detected a potential memory leak, the next step is to tie back memory utilization to service requests and determine what service request(s) is leaking memory. Without a clear indication of what service requests are leaving additional objects in the heap after they complete, tracking down a memory leak is a nearly insurmountable task because the only way to identify the root cause of a memory leak is to replay each service request inside a memory profiler and manually look at objects left in the heap. Then, with your business domain expertise, you can assess whether these objects do in fact need to be in memory or if they are causing the leak. Without narrowing down memory leaks to a subset of service requests, you are forced to manually examine each service request in your entire application inside a memory profiler. And then, if the memory leak is subtle, you may still miss it. Depending on the size of your application, this task could be impossible.

The two most common culprits of leaking memory are objects for which a reference is inadvertently left inside a collection class and the amount of data stored in HTTP session objects. To avoid these problems, always explicitly remove an object from a collection class rather than nullifying your local reference to it and examine the life cycles of each object stored in your sessions—in other words, properly manage object life cycles. It is also advisable to try to define discrete units of work within the context of a single method, because when the method exits, all method-level variables will be eligible to be reclaimed by the garbage collector.

The final issue that commonly affects application code performance is programmatic contention. Under low load, contention and synchronization are not issues, but as load increases, minor problems can become major problems. If 2,000 users are frequently accessing your application, then it is only a matter of time before thread deadlock or request time-outs occur.

The following steps summarize the process of diagnosing code issues:

1. Triage the problem to determine whether the issue is in the application, platform, or external dependency. Your monitoring tool needs to provide deep monitoring across a breadth of technologies.
2. Through tag-and-follow code instrumentation, identify slow-running service requests and determine if they are slow as a result of an application server resource (a wait point) or a slow-running method. Similarly, if you are detecting memory leaks, identify the offending service requests.
3. Replay offending service requests inside a profiler (code profiler or memory profiler) and find the problematic code.
4. Refactor the code.

Identifying and resolving application bottlenecks is not difficult if you have the appropriate set of monitoring tools to help you pinpoint the problem. Without these tools, you are diagnosing blindly and your chances of success are limited.

Summary

To implement a formal performance tuning methodology, you need to do the following:

- Get to know your users. Discern their usage patterns through either an end-user experience monitor or an access log analyzer.
- Build test scripts that mimic end-user patterns.
- Identify potential wait points in your application technology stack, open all wait points to force load to the limiting resources, and then harden each wait point to allow only enough traffic to effectively use the weakest limiting resource.
- Identify application code issues through tag-and-follow code instrumentation, replay offending requests inside a code or memory profiler, and refactor poorly performing code.

In this chapter, we focused on setting up a proper testing environment and explored the concept of wait-based tuning. In the next chapter, we'll examine tuning the application server engine itself, regardless of the architecture of the applications running in them. We'll look at the infrastructure that each application server must provide in order to satisfy the Java EE 5 specification and how to best tune each.