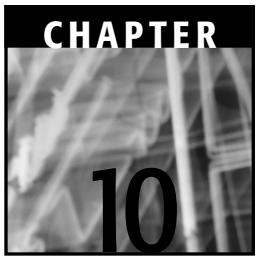

Performance

There is simply no way around the fact that the performance of any real-time Web application is critical to the success or failure of the product. Most user communities today are very unforgiving of applications with substantial page response times. Time is a valuable commodity in today's fast-paced Internet world, so performance is an essential aspect of user acceptance for any software product. Thus, it is critical that performance be considered from the beginning of the software development process. Now, there is a lot of common wisdom on this topic, particularly about the dangers of spending too much time up front on optimization. As in many things, the best answer is to take things in moderation and find a middle ground. Performance should be considered first at the architecture level and then at increasingly lower levels of detail as the iterative software development process continues. To begin, this chapter looks at the overall software development process and how performance engineering fits into the picture.

Overall Performance Approach

A basic development lifecycle with performance engineering integrated into the process is shown in Figure 10.1. Note that this process itself is often performed in an iterative manner that includes both prototypes and multiple production releases.

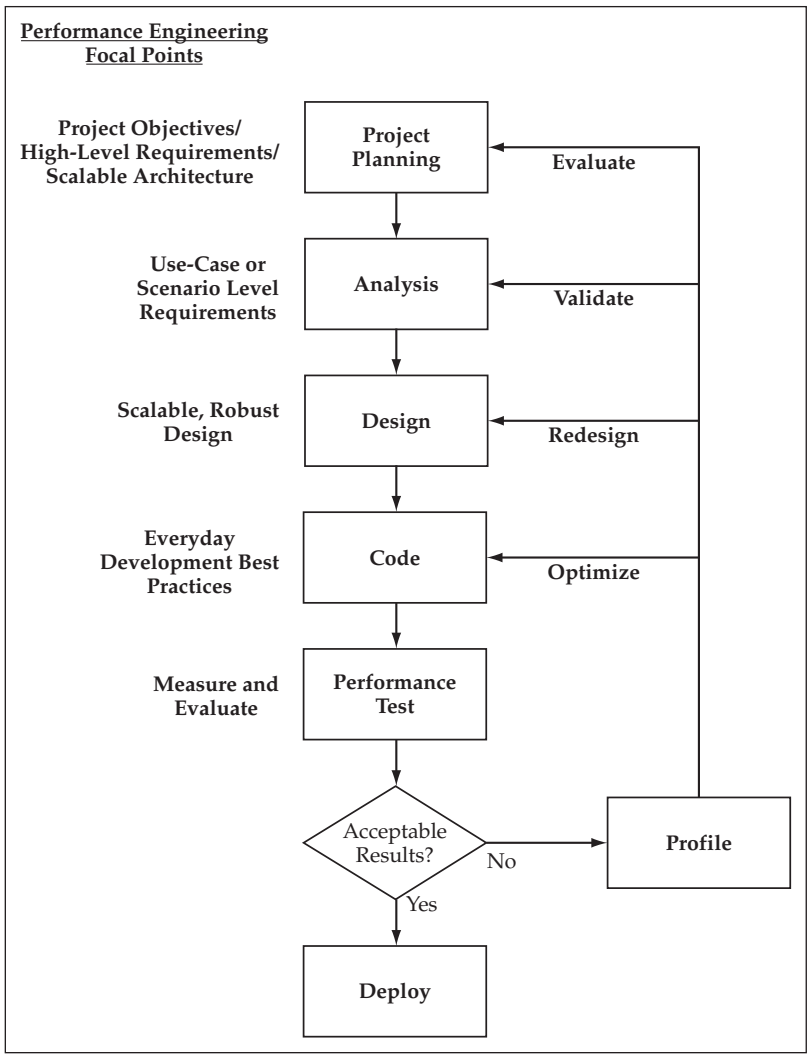

Figure 10.1 A Development Process with Performance Engineering.

It is no surprise to see that the corrective measures after an unacceptable performance test get increasingly more expensive and detrimental as you are required to go farther back in the process. Thus, it is very important to spend some initial time considering performance during the establishment of the overall software architecture. It is much easier to refactor portions of the application code than it is to change the underlying software architecture. As was alluded to earlier, there should still be a balance in terms of how much time and effort is spent on this topic, but the following guidelines usually hold true:

- A scalable, efficient architecture is a must for high-performance applications.
- Lower-level optimizations can always be done later.

As this chapter looks at performance both in the overall software development process and in J2EE technology, more clarity will be brought to these two important points. The next section takes a look at each of the development steps in a bit more detail from the perspective of performance.

Performance Engineering in the Development Process

At the beginning of a software development effort, one of the first steps is to determine the high-level objectives and requirements of the project. In addition to identifying the key functionality provided by the system, the project objectives can include such things as the flexibility of the system or the overall performance requirements. During this time, the overall system architecture is also being developed. For performance-intensive applications and projects with demanding requirements, a scalable architecture is an absolute must. Early architecture reviews cannot ignore the performance aspect of the system.

BEST PRACTICE **A scalable, efficient architecture is essential for high-performance applications. Initial architecture reviews during the early stages of a project can be used to help benchmark and validate high-level performance requirements.**

At this point, you are talking about the high-level software architecture including such things as component interaction, partitioning of functionality, and the use of frameworks and common design patterns. At this point you do not need to spend large amounts of effort or consideration on detailed optimizations such as the use of `StringBuffer` versus `String` or data caching down to the entity level. You are, however, still looking at high-level strategies such as the component implementation model, data caching strategies, and possibilities for asynchronous processing.

The creation of the basic software architecture at this point usually includes some kind of narrow but deep prototype, or proof-of-concept, which executes the communication through all of the layers and primary components of the architecture. This could include a user interface that retrieves data from the database and then sends an update all the way back through the architecture. Some basic load testing can occur at this point to obtain a ballpark estimate of transactions per second, page response time, or some other meaningful unit of measure that can help to frame the discussion on performance. This kind of data can be very helpful in terms of determining the validity of any high-level performance requirements that are being agreed upon during the project's early stages.

Once the individual use cases or scenarios of the system move into the analysis step, specific performance requirements often emerge for different functions and business processes. The analysis step allows you to apply the high-level project objectives against the specific functional requirements in order to derive these lower-level performance requirements for particular functions or pages.

Using the combination of the project objectives, functional requirements, and any case-specific performance requirements, the process moves into the design phase. It is

important that performance be considered at this phase because in many cases, there are trade-offs that must be made between competing objectives on both the business and technical sides of the project. Planning for performance can sometimes require a give and take between business requirements, such as the overall flexibility of the system and technical constraints, such as adherence to pure object-oriented design techniques. Thus, you cannot ignore performance as a consideration during the design phase, yet at the same time, you should not let it drive every decision.

In the coding phase, the everyday coding best practices become a focal point that lead directly into the resulting quality of the product. At this point, common design patterns have been prototyped, optimized to the extent that they can be in limited-use situations, and are being applied to the application functionality. It is the responsibility of the development team to then follow any guidelines set forth, such as the aforementioned use of `StringBuffer` when a large amount of string concatenation is being done to avoid the creation of many small, temporary objects. These are the more minor things that, if done simply out of habit, can all add up to a robust set of application code and the best possible performance results. These types of things can also be caught during code reviews and used as a way to validate and communicate best practices to a development team.

In iterative software development, performance tests are typically run after significant intermediate iterations have been completed or before releases go into production. Testing tools are often used to generate a target number of simulated clients, and the results are measured, again resulting in a set of metrics such as average page response time and transactions per second. If the results are not satisfactory and the root causes are not immediately apparent, profiling tools can be used to determine where the trouble spots are in the code.

NOTE **If your project or organization is on a small budget, there is a nice load-testing tool called OpenSTA available under a GPL (GNU General Public License) license that can be found at http://www.opensta.org. This tool is fairly easy to set up and use to run simulated load tests on Web applications. It may lack all of the features available within some commercial packaged solutions, but it provides almost all of the basic capabilities and reporting functions.**

Even at the end of a development cycle, there are still many lower-level code optimizations that can be done, for example, additional data caching or the use of more efficient Java collections classes. However, major changes to the code involving the component implementation and interaction models are difficult to make unless a modular architecture has already been put in place. Likewise, if the architecture itself is not scalable or efficient for its purposes, you have an entire codebase that may be affected by changes sitting on top of it. If a commonly used pattern in the application is redesigned at this point, it likely has many incarnations across the codebase that need to be changed. Alternatively, if you are talking about something like moving components from Entity Beans to regular Java classes, the migration is much more difficult if you do not have a service layer isolating the business logic from the presentation layer. These types of changes can be costly at this point in the game. Similarly, changes to the application design can have a significant effect. For example, you may have made

much of the business logic of the application configurable through database tables in order to meet a project objective of flexibility. A potential resulting effect of this, in terms of performance, is that the application becomes terribly slow due to the extensive database I/O throughout each transaction. A change to this aspect of the design, such as moving more of the logic back into the application code, could very easily affect the overall flexibility. Now, the role of architecture in this project is not only to provide an efficient foundation to implement these designs but also to allow for a mitigation plan. If you have wrapped access to the configuration data and isolated it to a set of objects, you may be able to cache the data in memory and easily speed up the performance of the application. You may also need to build in a refresh mechanism based on the requirements. In terms of implementing this type of change, it is much less painful to go back and recode a wrapper class than it is to update every business component that used the configuration data. In fact, the foundation logic for the business objects followed this same pattern through the use of the `MetadataManager` and `CacheList` components.

As a last resort, there may be a need to go back and review the specific performance requirements and possibly even validate that the project objectives are in line with what can realistically be done to provide the most value to the user community. To avoid having to go through this, the time and effort spent on performance can be spread a bit more evenly throughout the life of the project in order to mitigate, measure, and meet the performance requirements spelled out for your application.

Measuring Performance

Fully evaluating the performance capabilities and capacity of an application often requires the use of different metrics and different perspectives. Initially, it is usually best to put the focus at the transaction level and measure the individual transaction time or page response time. As the development process continues, the focus expands to include measurements of the transaction throughput, the ability to support concurrent users, and the overall scalability of the application. One of the main challenges in terms of performance in application development is to try to balance these vantage points.

Individual Transaction Response Time

During the early prototyping stages, the first question to ask is, "How fast can I push a single transaction through the system?" This is easy to test, requiring only a simple client program with a timer, yet it provides the basic unit of measure upon which the vast majority of performance metrics will be based. The result of a load test or sizing exercise is usually a multiple of the processing speed of each individual unit of work. Thus, the first area of focus is the basic patterns and architecture components exercised by some basic transactions. If you create efficient patterns going through the core of the user interface and business component models, these basic transactions can be optimized and used as a foundation for the application. Keep in mind, however, that your work does not end here because the next perspective may impact some of the strategies chosen during this first exercise.

Transaction Throughput and Scalability

The second aspect of performance that you want to measure takes the area of focus up a level to the behavior of the application operating under a heavy user load. Scalability is one of the main concerns here that can potentially impact some of the optimizations you want to perform at the individual transaction level. The J2EE component architecture provides a foundation for highly scalable and available applications on which to base your approach. However, there are a couple of things to keep in mind, primarily the memory consumption of the application and the size of the user `HttpSession` object. As an example, you may have a blazing fast page response time for a single user, but that may have been enabled by storing an entire result set from the database in the `HttpSession`. Subsequent page requests can then page through the data without having to go back to the database. If you are in this situation with a large data set, however, you may be able to get only a handful of concurrent users on an individual box because of the memory footprint involved with the application components.

As you look at the transaction throughput with various concurrent user levels, you also want to ask the question, "Does the system performance degrade as I add concurrent users and transactions?" You hope not, as you would like to see a linear response time as you add concurrent users to an application. Once you have hit the maximum number of users by pushing the current hardware to its limit, you would then like to see a linear response time as you add additional hardware. This type of scalability is made possible through the clustering and load balancing of the application components on the Web and EJB tiers. It enables you to add additional hardware and create redundant instances of the application server to meet the demands of your application. The value of the EJB component model is that it provides a standard method of building components to plug into a container and automatically take advantage of these infrastructure services.

Object Instantiation, Garbage Collection, and Scalability

In the Java language, there is also another aspect of code running in a JVM that affects the ideal of linear response time. There are actually two performance hits incurred by the JVM, both associated with instantiating an object in Java:

1. The initial cost of allocating memory to create the object
2. The secondary cost of tracking the object and later running it through the garbage collection (GC) process, potentially multiple times, until it is eventually destroyed and the memory is freed up for other use

Every object that is created in your code must later be checked by the JVM to see if it is being used by another object. This must be done before it can be freed and the memory reallocated for other use. The more objects that are created, the longer this garbage collection process takes, and the less free memory that is available, which then leads to the garbage collection process running more often. You can easily see how this can create a downward spiral that quickly degrades both the transaction throughput and the individual response times.

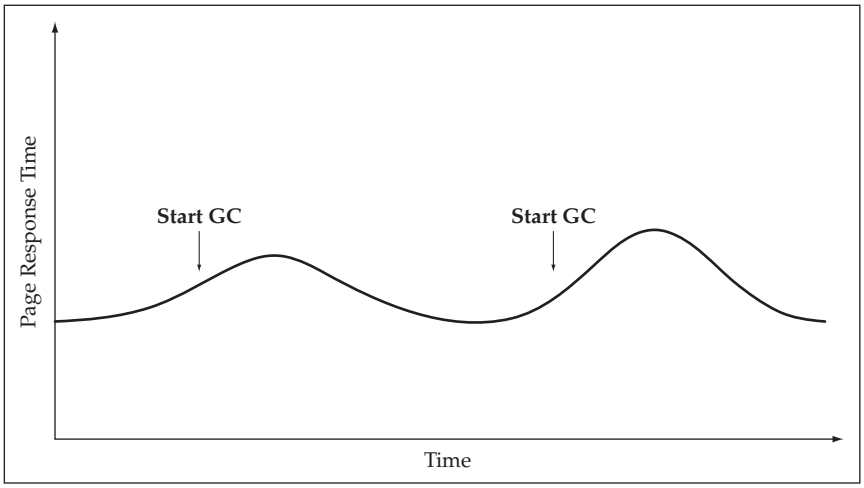

Figure 10.2 Theoretical Response Time with a Single JVM.

NOTE **To quickly see the effects of garbage collection, use the `-verbose:gc` JVM flag. This causes the JVM to write information to the output log showing the time spent in GC, memory used, and memory freed each time GC is run.**

The problem of the downward spiral is magnified if only one JVM is being used because transactions can continue to become backlogged until they eventually start to time out or reach completely unacceptable levels. Figure 10.2 shows a graph to represent the effects of garbage collection on response time for a single JVM under a heavy transaction load.

The secondary cost of object instantiation also prevents you from simply applying the tempting cure of adding more memory to the heap. With a larger heap size, the garbage collection process can become even more cumbersome to manage and then takes away valuable computing cycles that could be used for processing user transactions. Thus, adding more memory works to an extent, but at some point, it may have a marginally negative effect. Once again, the clustering and load-balancing capabilities of the J2EE application server come to the rescue to provide the scalability you need to help maintain a relatively even response time. Because requests are distributed across a cluster of application server instances, you can typically avoid having to use the JVMs that are garbage collecting to process the current transaction. The load-balancing algorithm, of course, is usually not tied directly into the GC status of the JVM, but it does use the law of averages and probabilities to work in your favor. What the clustering also allows you to do is to use a moderately sized memory heap for each JVM instance so that you can find the optimal setting for your application. Tuning this JVM parameter can often have a meaningful affect on the overall performance of an application. Usually it takes a number of trial and error load tests in order to determine the optimal settings for the heap size, although a few general guidelines include setting the minimum size to be half of the maximum size, which usually does not exceed 512 MB. The

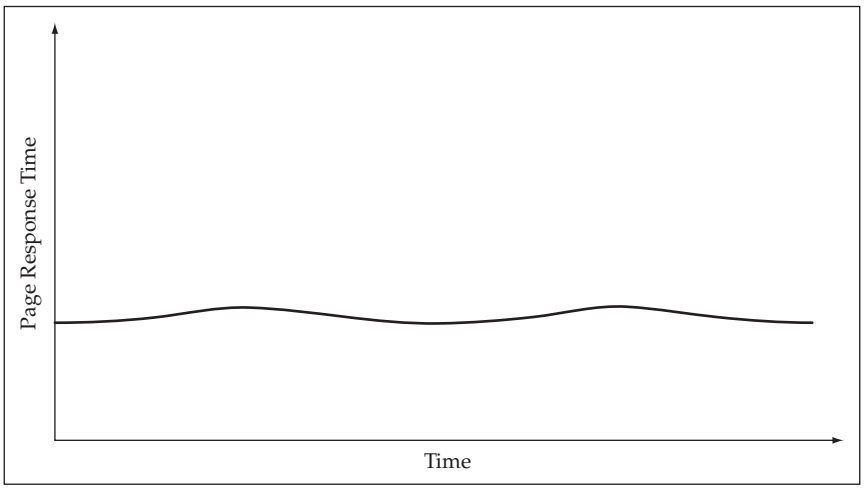

Figure 10.3 Theoretical Response Time with Multiple JVMs.

net result of all of this is a much more even response time and consistent transaction throughput as concurrent user levels increase. Figure 10.3 shows what an improved response time might be for an application clustered across multiple JVMs. Barring other extraneous factors, some minor blips in the curve still appear due to the occasional time periods when a number of the JVMs happen to be collecting garbage at the same time. This is largely unavoidable, but it has a much smaller effect on the overall response curve than in the scenario with a single JVM.

ECperf—An EJB Performance Benchmark

Another performance metric you can use is the ECperf benchmark created through the Java Community Process that is now a part of the J2EE suite of technologies. Its goal is to provide a standard benchmark for the scalability and performance of J2EE application servers and, in particular, the Enterprise JavaBean aspect that serves as the foundation for middle-tier business logic. The focus of the ECperf specification is not the presentation layer or database performance; these aspects are covered by other measures such as the series of TPC benchmarks. The focus of the ECperf tests is to test all aspects of the EJB component architecture including:

- Distributed components and transactions
- High availability and scalability
- Object persistence
- Security and role-based authentication
- Messaging, asynchronous processing, and legacy application integration

The software used for the test is intended to be a nontrivial, real-world example that executes both internal and external business processes, yet it has an understandable

workflow that can be consistently executed in a reasonable amount of time. Four business domains are modeled in the ECperf 1.1 specification as part of a worldwide business case for the tests:

- Manufacturing
- Supplier
- Customer
- Corporate

A number of transactions are defined for each of the domains, each of which is given a method signature to be used by an EJB component in the test. These transactions include such things as `ScheduleWorkOrder` and `CreateLargeOrder` in the manufacturing domain, as well as `NewOrder` and `GetOrderStatus` in the customer domain. Subsequently, two applications are built using these domains. The first is an `OrderEntry Application` that acts on behalf of customers who enter orders, makes changes to them, and can check on their status. The second is a `Manufacturing Application` that manages work orders and production output. The throughput benchmarks are then determined by the activity of these two applications on the system being tested. Reference beans are given for the test, and Entity Beans can be run using either BMP or CMP. The only code changes allowed are for porting BMP code according to regulations set forth in the specification. Deployment descriptors for all of the beans must be used as they are given in order to standardize the transactional behavior as well as the rest of the deployment settings. The reference implementation of these transactions uses stateless and stateful Session Beans as a front to Entity Beans, although the ratio of components is fairly heavily weighted toward Entity Beans.

The primary metric used to capture the result is defined using the term BBops/min, which is the standard for benchmark business operations per minute. This definition includes the number of customer domain transactions plus the number of workorders completed in the manufacturing domain over the given time intervals. This metric must be expressed within either a standard or distributed deployment. In the standard, or centralized deployment, the same application server deployment can be used for all of the domains and can talk to a single database instance containing all of the tables. The distributed version requires separate deployments and different database instances. These two measurements are thus reported as `BBops/min@std` or `BBops/min@Dist`, respectively. For either of these measurements, there is a very helpful aspect built into the specification for technology decision makers, the measure of performance against price, that is, `$ per BBops/min@std`, also commonly referred to as Price/BBops.

The ECperf 1.1 specification also announced that it will be repackaged as SPECjAppServer2001 and reported by the Standard Performance Evaluation Corporation (http://www.spec.org). SPECjAppServer2001 will cover J2EE 1.2 application servers while SPECjAppServer2002 will cover J2EE 1.3 application servers. A good "apples-to-apples" comparison of application servers like this has been a long time coming. The Sun Web site currently refers you to http://ecperf.theserverside.com/ecperf/ for published results. To give you a ballpark idea, there are currently a couple posted results over 16,000 `BBops/min@std` for under $20/BBops.

Performance in J2EE Applications

This section takes a look at various techniques you can use to optimize the architecture, design, and code within your J2EE applications. As a first step, there are key aspects within all Java programs that need to be addressed for their potential impact on application performance. Additionally, there are various performance characteristics associated with J2EE components and technologies that are worth noting. Many solutions involve using enterprise Java services whenever they provide the most benefit, but not as a standard across the board. Using the enterprise components across the board from front to back in the software architecture is a common tendency in building J2EE architectures. A key example of this is the use of Entity Beans. Relatively speaking, Entity Beans are fairly heavyweight components, and thus should not be used to model every business object in an application, particularly if each Entity Bean maps to a row in the database. Doing this can quickly degrade the scalability, and thus the usability, of an application. This goes back to one of the main points, that a scalable architecture is a must for almost any system, and design guidelines must be applied when deciding on the foundation for software components as well as in building the individual components themselves.

Core Aspects of Java Application Performance

Two significant performance aspects to consider for almost all applications are:

- Object instantiation and garbage collection
- Disk and database I/O

Object Instantiation

A key point to take away from the earlier discussion regarding object instantiation and garbage collection is that, to some degree, objects should be instantiated wisely. Each new version of the JVM has seen significant gains in the efficiency of the garbage collection process, but if you can reasonably limit or delay the creation of objects, you can help yourself greatly in terms of performance. This is especially true for larger components that usually encompass the instantiation of many objects. Of course, this does not mean you should go back to doing pure procedural software development and put all of your logic in a single `main` method. This is where performance as a design consideration comes into play. You don't want to sacrifice the potential for reusability and flexibility through solid object-oriented design; thus, you don't let performance drive all of your decisions. Nonetheless, keep it in the back of your mind. And if you aren't quite sure of a potential impact, you can use an abstraction or design pattern to mitigate the concern by providing an escape route to take later. This means that if you have isolated an architecture layer or encapsulated a certain function, it can be changed in one place without great cost or side effects to the remainder of the application.

To maximize the efficiency of time spent considering performance in the design process, consider the following approach. Rather than look at every object in the entire object model, perhaps spend some time concentrating on the two extremes in your implementation: the extremely large objects and components and the extremely small objects. For obvious reasons, large objects and components rapidly increase the memory footprint and can affect the scalability of an application. In the case of larger components, they often spawn the creation of many smaller objects as well. Consider now the case of the very small object, such as the intermediate strings created by the following line of code:

```
String result = value1 + value2 + value3 + value4;
```

This is a commonly referenced example in which, because `String` objects are immutable, you find out that `value1` and `value2` are concatenated to form an intermediate `String` object, which is then concatenated to `value3`, and so on until the final `String` result is created. Even if these strings are only a few characters in size, consider now that each of these small `String` objects has a relatively equal impact on your secondary cost consideration, the object tracking and garbage collection process. An object is still an object, no matter what the size, and the JVM needs to track all of the other objects that reference this one before it can be freed and taken off of the garbage collection list. Thus, all of those little objects, although they do not significantly impact the memory footprint, have an equal effect on slowing down the garbage collection process as it runs periodically throughout the life of the application. For this reason, you want to look at places in the application where lots of small objects are created in order to see if there are other options that can be considered.

In the study of business object components, the concept of lazy instantiation, which delays the creation of an aggregated object until it is requested, was discussed. If strict encapsulation is used where even private methods used a standard `get<Object>` method, you can delay the instantation of the object until it is truly necessary. This concept is particularly important for value objects or other objects used as data transport across a network. This practice minimizes the amount of RMI serialization overhead as well as reducing network traffic.

BEST PRACTICE **Use lazy instantiation to delay object creation until necessary. Pay particular attention to objects that are serialized and sent over RMI.**

Another common use of this concept can be put into practice when lists of objects are used. In many application transactions, a query is executed and a subset of the resulting objects is dealt with in a transactional manner. This concept is particularly important if the business object components are implemented as Entity Beans. For a collection of size `n`, as was discussed in the Business Object Architecture chapter, the potential exists for the (`n + 1`) Entity Bean finder problem, which results in additional database lookups that can be accomplished with a single JDBC query. However, you also want to consider the characteristics of Entity Beans and their effect on the container's performance. Although Entity Beans are fairly heavyweight components, the optimized transaction model is fairly efficient because Entity Bean instances are pooled

and shared by the container for different client transactions. However, once an Entity Bean instance is pulled into a client transaction, it cannot be shared by another client until either the transaction ends or the container passivates the state of that instance for future use. This passivation comes at a cost and additional complexity because the container must activate the instance once again to complete the transaction later in a reliable, safe manner. Considering that the Entity Bean components have a relatively large fixed cost and that there may be many different types in a complex application, you want to size the component pools appropriately and find a balance between resource consumption and large amounts of activation and passivation that can slow down the application server. With all of this being said, if you can avoid using an Entity Bean for pure data retrieval, it is worth doing it. Perhaps not for that individual transaction, but it will aid the scalability and throughput of the overall application under a heavy user load. This comes back to the analysis of performance measurement that first starts at the individual transaction level, but then has to consider the effect on the overall application performance.

This concept is also in line with the idea of using business objects only for transactional updates as opposed to requiring that they be used for data retrieval as well. Thus, if your application deals with a collection of objects, it is perhaps best to first run the query using JDBC, similar to the `ObjectList` utility class. You can then iterate through the collection and instantiate or look up the Entity Bean equivalents when you want to perform a transaction update on a given instance. In the cases in which you do not update the entire collection, you can gain the greatest benefit from this technique. The database lookups for an `n` size collection are then somewhere between 1 and (`n + 1`), depending on the particular circumstances of the transaction. You can also compare this to an aggressive-loading Entity Bean strategy that theoretically limits you to a single database query but then has the overall cost associated with using a significant portion of the free instance pool. In other words, you sacrifice the overall transaction throughput for the benefit of the individual transaction in a heavy user load setting. Note that if the transaction volume is quite sporadic for a given application, an aggressive-loading strategy for Entity Beans may be the better solution because the assumption of fewer concurrent transactions is made; thus the cross-user impact is limited.

Disk and Database I/O

Often, the first thing to look at when tuning an application is the amount of database and disk I/O because of its relative cost compared to regular computational cycles. Thus, look to minimize the amount of database calls and file read/writes in your application. The first strategy to do this is usually to analyze the access patterns and identify redundant or unnecessary database access. In many cases, a significant benefit can be derived from performing this step at the design review and code review stages of a project. Eventually, your application approaches the minimum level of access required, and then you need to look to other techniques to make further improvements, which is where data caching comes into play.

Data caching commonly refers to storing application data in the memory of the JVM, although in general terms, it could also involve storing the data somewhere closer to

the client or in a less costly place than the original source. In a sense, you can refer to data stored in the `HttpSession` of a Web application as being cached if you are not required to go through an EJB component and to the application database to get it. In practice, the `HttpSession` could be implemented by the container through in-memory replication or persistent storage to a separate database, although, in both cases, the access time to get to the data is likely less than it would be to go to the definitive source. Now, of course, the definitive source is just that, and you need to be able to refresh the cache with the updated data if it changes and your application requirements dictate the need, which is often the case. In the Business Object Architecture chapter, a solution for this issue was looked at in the J2EE architecture using JMS as a notification mechanism for caches living within each of the distributed, redundant application server instances. Remember that even this approach has a minor lag time between the update of the definitive source and the notification message being processed by each of the caches. This may still not be acceptable for some mission-critical applications; however it does fit the needs of many application requirements.

The reference architecture uses an XML configuration file for application metadata, and many applications use a set of configuration values coming from a properties file. This type of data is a perfect candidate for caching because it does not change frequently and may not even require a refresh mechanism because changes to this data often require a redeployment of the application.

The use of Entity Beans to cache data should also be addressed here. Whereas Session Beans are used to deal with the state of a particular client at a time, Entity Beans represent an instance of a persistent object across all clients. So how much can you rely on Entity Beans to help with caching? Unfortunately, the benefit is not as great as one might think. Although an instance of an Entity Bean can be shared across clients, the same issue of updates to the definitive source applies here. If you deploy your EJB components to a single instance of an application server, then you can, in fact, take full advantage of this caching. However, most significant deployments wish to use the clustering and load-balancing features of the application servers, so multiple instances are deployed and the cached Entity Bean must consider the possibility of updates by another copy of that Entity Bean in another instance. Thus, in a clustered environment, the `ejbLoad` method must always be invoked at the beginning of a transaction to load the current state and ensure data integrity.

Object Caching

The concept of caching can also be applied to objects that are relatively expensive to instantiate. In a J2EE environment, this can include such objects as the JNDI Initial Context and the EJB Home interfaces. In your own application, you may also have complex components or objects that are expensive to instantiate. Some examples of this might be classes that make use of BeanShell scripts or other external resources that involve I/O, parsing, or other relatively expensive operations. You may want to cache instances of these objects rather than instantiate new ones every time if one of the following requirements can be met:

- Objects can be made thread-safe for access by multiple concurrent clients.
- Objects have an efficient way to clone themselves.

JNDI Objects

Relatively speaking, the JNDI operations can be somewhat expensive for an application. The creation of a `InitialContext` and the subsequent lookups for EJB Home interfaces should be looked at as a performance consideration. If your application does not use a large number of EJB, this may not be worth any further thought. For example, if your business logic is encompassed within Session Beans and you typically have only one EJB lookup in a transaction, it may not be worth the trouble to try and optimize this step. However, if you have a large number of Entity Beans used within a given transaction, it can make a noticeable difference if you can avoid the creation of an `InitialContext` and subsequent JNDI lookup for each component. Caching the JNDI objects should be used with caution, as there are a number of potential impacts to consider. The `InitialContext` object can be created once, such as on the Web tier in the controller servlet's `init` method, and then used for all client requests rather than a new one created for each individual request. In a set of tests with heavy user loads, a single shared `InitialContext` instance did not present any problems; however, you should thoroughly test in your target environment to become comfortable with the approach.

Before looking at the `EJBFactoryImpl` code for an implementation of this solution, you should also consider caching the EJB Home interface objects. This technique can also provide a performance boost in some cases but should be used only after careful consideration. Many application servers provide a Home interface that is aware of the available, redundant application server instances. However, ensure that this is the case for your environment before using this technique. If you are going to reuse an existing Home interface, you don't want one that pins you to a given instance, or you will lose all of your load-balancing and failover capabilities. The other aspect to consider of reusing the Home interface is that problems can result if one or more of the application server instances are brought up or down. A Home interface may become "stale" if the EJB server is restarted, and if instances are added or removed from the cluster, the existing home interface is likely not to be aware of this. In this sense, there also needs to be a refresh capability for the Home interface cache unless it is acceptable to restart the Web tier, or other such client tier, when a change is made to the EJB server configuration. This is likely to be a manual process unless a programmatic management capability can be introduced into the application.

Here are the relevant portions of `EJBFactoryImpl` that use a cached `InitialContext` and cached collection of EJB Home interfaces keyed by the object name. In the examples in this book, this class is always used in the context of an EJB tier underneath a service component deployed as a Session Bean. Thus, note that the `InitialContext` is created without any properties in a static initialization block. In order to be used by remote clients, this class would need to be modified to pass in the provider URL and context factory, but you can see the basic idea from this example. Each time the `findByPrimaryKey` method is invoked, the helper method `getHomeInterface`, which first looks in a collection of Home interfaces to see if the interface was already created and cached, is called. If it is not there, then it is created and stored for future use. This implementation uses a lazy-instantiation approach in which the first time through is a bit slower and then subsequent requests benefit from

the performance improvements. Alternatively, this initial cost could be incurred at server startup time:

```
public class EJBFactoryImpl extends BusinessObjectFactory {

    // Cached initial context
    private static InitialContext jndiContext;

    // Cached set of home interfaces keyed by JNDI name
    private static HashMap homeInterfaces;

    static {
        try {
            // Initialize the context.
            jndiContext = new InitialContext();

            // Initialize the home interface cache.
            homeInterfaces = new HashMap();

        } catch (NamingException ne) {
            ne.printStackTrace();
        }
    }

    /**
     * Helper method to get the EJBHome interface
     */
    private static EJBHome getHomeInterface(String objectName,
                                    BusinessObjectMetadata bom)
        throws BlfException {

        EJBHome home = null;

        try {
            // Check to see if you have already cached this
            // Home interface.
            if (homeInterfaces.containsKey(objectName)) {
                return (EJBHome)
                    homeInterfaces.get(objectName);
            }

            // Get a reference to the bean.
            Object ref = jndiContext.lookup(objectName);

            // Get hold of the Home class.
            Class homeClass =
                Class.forName(bom.getEJBHomeClass());

            // Get a reference from this to the
            // Bean's Home interface.
```

```
                home = (EJBHome)
                    PortableRemoteObject.narrow(ref, homeClass);

                // Cache this Home interface.
                homeInterfaces.put(objectName,home);

            } catch (Exception e) {
                throw new BlfException(e.getMessage());
            }

            return home;
        }

        /*
         * Discover an instance of a business object with the
         * given key object.
         */
        public static Object findByPrimaryKey(String objectName,
                                              Object keyObject)
            throws BlfException {

            // Obtain the business object metadata.
            BusinessObjectMetadata bom =
                MetadataManager.getBusinessObject(objectName);

            // Get the Home interface.
            EJBHome home = getHomeInterface(objectName, bom);

            //
            // Use the Home interface to invoke the finder method...
            //
        }
    }
```

BEST PRACTICE **For increased performance in applications that use a large number of Entity Beans, consider caching the JNDI `InitialContext` and EJB Home interfaces. This optimization should be encapsulated within the EJB business object factory so there is no effect on business object client code. Many application servers provide a Home interface that is aware of the available, redundant application server instances. However, ensure that this is the case for your environment before using this technique so you don't lose the load-balancing and failover capabilities of the application server.**

Entity Beans

Many of the performance characteristics of Entity Beans have already been covered. Although they are fairly heavyweight components, the container pools instances of them, and the regular transaction model can be quite efficient. However, you can get

into trouble when the container is forced to perform large amounts of activation and passivation that can occur under heavy, concurrent usage. There are a number of other things to keep in mind. For example, when using remote interfaces, you want to minimize the amount of remote method invocation and RMI overhead. Thus, you use value objects to communicate data to the Entity Bean. You also want to avoid iterating through collections of Entity Beans through finder methods unless you can mitigate the risks of the (`n + 1`) database lookup problem.

If you are using a Session Bean layer as a front to Entity Beans, similar to the reference architecture and the services layer, you should use local interfaces to access your Entity Beans. This avoids the overhead of RMI and remote method invocations. This forces you to colocate all related Entity Beans in a transaction in a given application server deployment, although this usually does not cause much of a problem unless you have a truly distributed architecture. In many cases, all of the beans are running in a standard centralized deployment for performance reasons and you can do this with ease. At this point, the biggest overhead left for each Entity Bean is the JNDI lookup to access the local interface, and there are options to address this given the earlier discussion of JNDI and object caching.

In many cases, Container-Managed Persistence (CMP) provides the best option in terms of performance for Entity Bean persistence. Bean-Managed Persistence (BMP) does suffer from a serious performance flaw in that a single lookup of an Entity Bean can actually cause two database hits. This problem is similar to the (`n + 1`) problem if considered for a collection of one. The container needs to look up the bean using the primary key after a Home interface method is invoked. Once the component is located and a business method is invoked from the remote or local interface, the `ejbLoad` method, which typically uses application JDBC code to select the remainder of the properties from the database, is called by the container. In the container-managed approach, the container can optimize these steps into one database call. This is a serious consideration for using BMP in your Entity Beans. There are also many other cases in which the container can optimize how persistence is implemented, such as checking for modified fields before executing `ejbStore`. Finally, a major benefit of using Entity Beans is the object persistence service, so carefully consider the benefits of using BMP before taking this approach.

Another factor that can affect the performance of Entity Beans is the transaction isolation setting. The safest option is `TRANSACTION_SERIALIZABLE`, but it is not surprisingly the most expensive. Use the lowest level of isolation that implements the safety required by the application requirements. In many cases, `TRANSACTION_READ_COMMITTED` provides a sufficient level of isolation in that only committed data is accessible by other beans. Transactions should also be kept to the smallest scope possible. However, this can sometimes be difficult to implement using container-managed transactions because you can give each method only a single transaction setting for the entire deployment. Often, methods are used across different contexts in an application, and you would like the setting to be different in various situations. For this, you need to use bean-managed transactions and control this aspect yourself. However, a nice benefit of the Session Bean to Entity Bean pattern is that Entity Beans are usually invoked within a transaction initiated by the Session Bean. In this case, a transaction setting of `TX_SUPPORTS` works in most cases because a transaction will have already been initiated if need be.

Session Beans

Stateless Session Beans are the most efficient type of Enterprise JavaBean. Because the beans are stateless, the container can use a single instance across multiple client threads; thus, there is a minimal cost to using a stateless Session Bean both for the individual transaction and the overall application scalability. Remember that this is not always the case with Entity Beans due to the potential for activation and passivation. The container implementation also has the option to pool instances of stateless Session Beans for maximum efficiency.

A stateful Session Bean is particular to the client that created it. Thus, there is a fixed cost for the individual transaction that uses a stateful Session Bean. Stateful Session Beans are sometimes used as an interface to a remote client that maintains some state about the application. In a Web application, this type of state can usually be stored in the `HttpSession`, although stateful Session Beans are particularly helpful for thick-client Swing front ends. Note that it is important that the client call the `remove` method on the stateful Session Bean when it is done; otherwise the container will passivate it for future use, and this adds to its overall overhead.

BEST PRACTICE **Be sure to remove instances of stateful Session Beans to avoid unnecessary container overhead and processing.**

One thing to note is that some J2EE containers, particularly earlier versions, do not support failover with stateful Session Beans, although the major containers are now doing this. Make sure this is the case in your environment if this is a factor for consideration in your application.

XML

If an application does a large amount of XML parsing, it is important to look at the parsing method being used to do it. Two of the basic parsing options are the Document Object Model (DOM) and the Simple API for XML (SAX). DOM parsers require much more overhead because they parse an entire XML document at once and create an in-memory object representation of the XML tree. This is helpful if the program requires either significant manipulation or the creation of XML documents. However, if your application simply needs to parse through a document once and deal with the data right away, the SAX parser is much more efficient. It reads through a document once and invokes hook methods to process each tag that it comes across in the document. A document handler is written specifically for the application. It is a little more complicated to write because the hook methods are called without much of the XML tag context, such as the name of the parent tag. Thus, it requires the developer to maintain some state in order to correctly process the document if it contains any nested tags. However, the difference in speed can be noticeable for large documents. The reasoning for this goes back to the initial discussion on object creation and garbage collection. A DOM parser creates a large number of objects underneath the covers. The actual number of objects created is a factor of the number of XML nodes because objects are created for each attribute and text node of each element.

Many applications that use XML as a messaging or communications framework will want to manipulate the data in a regular Java object format. There are binding

frameworks such as the Java API for XML Binding (JAXB) that can be used to generate classes that can both extract their data from XML and write out their state as XML. These classes can be quite efficient because they know exactly where in the XML their properties belong and thus can avoid some of the overhead of a generic parsing API. These binding packages create a very powerful framework for exchanging data and dealing with it on both sides of a business service or process.

BEST PRACTICE **If you use XML extensively throughout your application and performance is a concern, choose the most efficient parsing method available to you that meets your requirements. DOM parsers are usually the slowest due to the large number of objects instantiated underneath the covers and their generic nature. If your application simply needs to parse through a document once and deal with the data right away, the SAX parser is much more efficient. Binding frameworks such as JAXB will also be more efficient because they know exactly what they are looking for in the XML or what XML tags they need to create. These types of frameworks are also helpful because they use XML as a data transport but allow programs to access the data through objects.**

Asynchronous Processing

Asynchronous processing is a strategy that can be used in certain circumstances to alleviate performance concerns. There are a limited number of situations for which this approach can be used; however, in the cases in which it is applicable, it can make a noticeable difference. Executing processes in parallel can be considered if any of the following conditions exist:

- Semi-real-time updates fit within the application requirements.
- There are a number of independent external applications to invoke.
- Application data and the relevant units-of-work can be partitioned.

Asynchronous processing can also be used to provide the benefit of perceived performance. For example, if a Web page is waiting on a response from a lengthy transaction, you may want to display the next page prior to the completion of the overall process to give the user the ability to continue work, thus increasing the perceived performance of the application. The next page might include a confirmation message, some intermediate or partial results, or else just a message informing users that they will be notified upon completion of the process, perhaps by email.

For a parallel processing approach to be effective, each asynchronous process needs to be significantly big enough to make the light overhead of a messaging framework, such as JMS, worth the benefit. One interesting thing to note about the J2EE environment is that JMS and Message-Driven EJBs are the only mechanisms provided to perform asynchronous processing. Strictly speaking, the EJB specification prohibits applications from managing their own threads. This makes sense when you think about the responsibilities of an application server. It is managing multiple threads for different types of components, and in order to effectively maximize performance and resource utilization, it requires control of the threads being run on a given machine. Thus, an application component cannot explicitly start a new thread in an object.

However, the Java Message Service provides a mechanism that goes through the container to invoke and start other threads. A message can be sent asynchronously from a client and a component that receives that message can process it in parallel with the execution of the original thread. This strategy is quite easy with the EJB 2.0 specification that provides a third type of Enterprise Bean, the Message-Driven Bean. This is an EJB component that is invoked when a particular type of JMS message is received. Thus, for asynchronous processing, a client can send a JMS message and a defined Message-Driven Bean can be used as a wrapper to invoke additional functionality in parallel.

BEST PRACTICE **Consider the use of asynchronous processing to alleviate performance concerns in applications with semi-real-time updates, multiple external applications that can be invoked in parallel, or work that can be partitioned into segments. Use Message-Driven Beans and JMS to implement parallel processing in a J2EE container. Asynchronous processing can also be used to increase the perceived performance of an application.**

The Web Tier

JavaServer Pages and servlets are extremely efficient in that they are multithreaded components with a very small amount of overhead. These components provide very useful APIs and functions without causing much of an impact to the performance of the application. Unlike EJBs, little or no thought is required in order to use either of these components with regard to performance. The exception to this rule is of course the use of `HttpSession`, something that was alluded to numerous times throughout this book. This state maintenance option can impact the scalability and throughput of an application, so careful attention does need to be paid to its use. Nonetheless, the front end of the J2EE platform provides a very efficient, robust architecture for implementing high-quality Web applications.

Best Practices for J2EE Performance Engineering

A summary of the performance best practices is given in this section.

Considering Performance throughout the Development Process

A scalable, efficient architecture is essential for high-performance applications. Initial architecture reviews during the early stages of a project can be used to help benchmark and validate high-level performance requirements. Lower-level optimizations can be done later in the process. In general, spread the time spent on performance engineering throughout the process rather than wait until the week prior to deployment to run a load test. Remember that performance problems uncovered later in the process become increasingly more expensive to resolve.

Minimizing Object Instantiation Whenever Possible

Use lazy instantiation to delay object creation until necessary. Pay particular attention to objects that are serialized and sent over RMI. If you are invoking a remote Session Bean, try to send only the object data that is required for the component method.

Caching EJB Home Interfaces

For increased performance in applications that use a large number of Entity Beans, consider caching the JNDI `InitialContext` and EJB Home interfaces. This optimization should be encapsulated within the EJB business object factory so that there is no effect on business object client code. Many application servers provide Home interfaces that are aware of the available, redundant application server instances. However, ensure that this is the case for your environment before using this technique so you don't lose the load-balancing and failover capabilities of the application server.

Removing Stateful Session Beans When Finished

Be sure to remove instances of stateful Session Beans when you are done with them to avoid unnecessary container overhead and processing.

Choosing an Efficient XML Parser Based on Your Requirements

The extensive use of XML in an application can have a noticeable effect on application performance. Choose the most efficient parsing method available to you that will meet your requirements. DOM parsers are usually the slowest due to the large number of objects instantiated underneath the covers and their generic nature. If your application simply needs to parse through a document once and deal with the data right away, the SAX parser is much more efficient. Binding frameworks such as JAXB will also be more efficient because they know exactly what they are looking for in the XML or what XML tags they need to create. These types of frameworks are also helpful because they use XML as a data transport, but you can program against the objects that receive the data.

Asynchronous Processing as an Alternative

Asynchronous processing is an option that can be used to alleviate performance concerns in applications with semi-real-time updates, multiple external applications that can be invoked in parallel, or work that can be partitioned into segments. Use Message-Driven Beans and JMS to implement parallel processing in a J2EE container. Asynchronous processing can also be used to increase the perceived performance of an application.

Summary

Performance should be considered throughout the development process. The initial focus is on developing a scalable architecture while lower-level optimizations can be saved until later. A typical approach involves a narrow but deep prototype, or proof-of-concept, which executes the communication through all of the layers and primary components of the architecture. Some basic load testing is done at this point to obtain basic performance metrics that help to validate both the high-level performance requirements and the proposed architecture. Performance should also be considered during the design phase because it often involves trade-offs against flexibility and other requirements. The application architecture and design should help to mitigate performance concerns by providing potential migration paths through the use of isolation and encapsulation. A key example of this concept is the use of a business object factory that provides a placeholder to optimize JNDI lookups without affecting the rest of the application code. Other key factors to consider when looking at J2EE performance include the use of Entity Beans and optimal pool sizes, choice of the right XML parser, and possibilities for asynchronous processing.

This chapter covered best practices for performance engineering in J2EE Web applications. The role of performance in the development process was considered and a number of techniques were discussed for the use of specific technologies such as Entity Beans, Message-Driven Beans, and XML. Whereas this chapter helped make your applications run faster, the next chapter addresses a number of best practices used to speed the development of your applications. These best practices focus on the topic of software reuse.