Performance Tuning

Agenda

- Introduction
- Profiling Tools
- JVM Improvements
- Object Creation
- Strings
- **Exceptions, Assertions, Casts, and Variables**
- **Loops, Switches, and Recursion**
- **I/O, Logging, and Console Output**
- **Sorting**
- **Threading**
- **Appropriate Data Structures and Algorithms**

Agenda

- **Distributed Computing**
- **When to Optimize**
- **Underlying Operating System and Network Improvements**
- **J2EE Performance Tuning**
- **Tuning JDBC**
- **Tuning Servlets and JSPs**
- **Tuning EJBs**

Object Creation

Object Creation

- Java manages Memory for Objects.
  - We do not worry about memory allocation/de allocation.
- However, Java does not prevent you from using excessive amounts of memory nor from cycling through too much memory (e.g., creating and dereferencing many objects).
- Contrary to popular opinion, you can get memory leaks (or, more accurately, object retention) by holding onto objects without releasing references.
  - This stops the garbage collector from reclaiming those objects, resulting in increasing amounts of memory being used.
  - In addition, Java does not provide for large numbers of objects to be created simultaneously (as you could do in C by allocating a large buffer), which eliminates one powerful technique for optimizing object creation.

Object Creation-Does it cost performance?

- Creating objects costs time and CPU effort for an application.
- Garbage collection and memory recycling cost more time and CPU effort.
- The difference in object usage between two algorithms can make a huge difference.
  - String and StringBuffer
- These can be an order of magnitude faster than some of the conversions supplied with Java.
- A significant portion of the speedup is obtained by avoiding extra temporary objects used and discarded during the data conversions.

General guidelines for using object memory efficiently

- Avoid creating objects in frequently used routines.
  - Because these routines are called frequently, you will likely be creating objects frequently, and consequently adding heavily to the overall burden of object cycling.
  - By rewriting such routines to avoid creating objects, possibly by passing in reusable objects as parameters, you can decrease object cycling.
- When multiple instances of a class need access to a particular object in a variable local to those instances, it is better to make that variable static rather than have each instance hold a separate reference. This reduces the space taken by each object (one fewer instance variable) and can also reduce the number of objects created if each instance creates a separate object to populate that instance variable.
- Reuse exception instances when you do not specifically require a stack trace

General guidelines for using object memory efficiently

- Try to presize any collection object to be as big as it will need to be.
- It is better for the object to be slightly bigger than necessary than to be smaller than it needs to be.
- This recommendation really applies to collections that implement size increases in such a way that objects are discarded.
  - For example, Vector grows by creating a new larger internal array object, copying all the elements from the old array, and discarding it.
  - Most collection implementations have similar implementations for growing the collection beyond its current capacity, so presizing a collection to its largest potential size reduces the number of objects discarded.

General guidelines for using object memory efficiently

- When multiple instances of a class need access to a particular object in a variable local to those instances, it is better to make that variable static rather than have each instance hold a separate reference.
- This reduces the space taken by each object (one fewer instance variable) and can also reduce the number of objects created if each instance creates a separate object to populate that instance variable.
- Reuse exception instances when you do not specifically require a stack trace.

Object-Creation Statistics

- Objects need to be created before they can be used, and garbage-collected when they are finished with.
- The more objects you use, the heavier this garbage-cycling impact becomes.

Object Reuse

- Objects are expensive to create.
- Where it is reasonable to reuse the same object, you should do so.
- You need to be aware of when not to call new.
  - This can be particularly important for objects that are constantly used and discarded: for example, in graphics processing, objects such as Rectangles, Points, Colors, and Fonts are used and discarded all the time. Recycling these types of objects can certainly improve performance.
- Recycling can also apply to the internal elements of structures.
  - For example, a linked list has nodes added to it as it grows, and as it shrinks, the nodes are discarded. Holding onto the discarded nodes is an obvious way to recycle these objects and reduce the cost of object creation

HOW do you reuse objects?

- **Pool Management.**
- **ThreadLocals.**
- **Reusable Parameters.**
- **Canonicalizing Objects.**
  - **String canonicalization.**
  - **Changeable objects.**
  - **Weak references.**
  - **Enumerating constants.**

Pool Management

- For better reuse of container classes like Vector and HashTable you may use an Object pool.
- In their HotSpot FAQ, Sun engineering states that pooling should definitely no longer be used because it actually gives worse performance with the latest HotSpot engines.
- This is rather a sweeping statement.
- Object pools are still useful even with HotSpot, but presumably not as often as before.
- Certainly for shared resources pooling will always be an option if the overhead associated with creating a shareable resource is expensive.
- Various recent tests have shown that the efficiency of pooling objects compared to creating and disposing of objects is highly dependent on the size and complexity of the objects.

ThreadLocals

- When you can guarantee you need only one object per thread, but any one thread must consistently use the same object.
  - Singletons that maintain some state information are a prime example of this sort of object.
- In this case, you might want to use a ThreadLocal object.
- ThreadLocals have accessors that return an object local to the current thread.

Weak references

- If Collection Object grows too large overtime then use Weakreference.
- If you need to maintain one or more pools of objects with a large number of objects being held, you may start coming up against memory limits of the VM.
  - In this case, you should consider using WeakReference objects to hold onto your pool elements.
  - Objects referred to by WeakReferences can be automatically garbage-collected if memory gets low enough.
- J2SE comes with a java.util.WeakHashMap class that implements a hash table with keys held by weak references

Strings

Strings

- Strings have a special status in Java.
- They are the only objects with:
  - Their own operators (+ and +=)
  - A literal form (characters surrounded by double quotes, e.g., "hello")
  - Their own externally accessible collection in the VM and class files (i.e., string pools, which provide uniqueness of String objects if the string sequence can be determined at compile time)
- Strings are immutable and have a special relationship with StringBuffer objects.
  - A String cannot be altered once created. Applying a method that looks like it changes the String (such as String.trim( )) doesn't actually do so; instead, it returns an altered copy of the String.
  - Strings are also final, and so cannot be subclassed.
  - These points have advantages and disadvantages so far as performance is concerned. For fast string manipulation, the inability to subclass String or access the internal char array can be a serious problem

The Performance Effects of Strings

- Advantages:
  - Compilation creates unique strings. At compile time, strings are resolved as far as possible.
  - Because String objects are immutable, a substring operation doesn't need to copy the entire underlying sequence of characters.
  - Instead, a substring can use the same char array as the original string and simply refer to a different start point and endpoint in the char array.
  - This means that substring operations are efficient, being both fast and conserving of memory; the extra object is just a wrapper on the same underlying char array with different pointers into that array

The Performance Effects of Strings

- Advantages:
  - Strings have strong support for internationalization. It would take a large effort to reproduce the internationalization support for an alternative class.

The Performance Effects of Strings

- Disadvantages:
  - Not being able to subclass String means that it is not possible to add behavior to String for your own needs.
  - The previous point means that all access must be through the restricted set of currently available String methods, imposing extra overhead.
  - The only way to increase the number of methods allowing efficient manipulation of String characters is to copy the characters into your own array and manipulate them directly, in which case String is imposing an extra step and extra objects you may not need.
  - char arrays are faster to process directly.

- For optimized use of Strings, you should know the difference between compile-time resolution of Strings and runtime creation.
- At compile time, Strings are resolved to eliminate the concatenation operator if possible. For example, the line:

```
String s = "hi " + "Mr. " + " " + "Buddy";
```

is compiled as if it read:

```
String s = "hi Mr. Buddy";
```

Loops

- Make the loop do as little as possible.
  - Remove from the loop any execution code that does not need to be executed on each pass.
  - Move any code that is repeatedly executed with the same result, and assign that code to a temporary variable before the loop ("code motion").
  - Avoid method calls in loops when possible, even if this requires rewriting or inlining.
  - Multiple access or update to the same array element should be done on a temporary variable and assigned back to the array element when the loop is finished.
  - Avoid using a method call in the loop termination test.
  - Use int data types preferentially, especially for the loop variable.

- Use System.arraycopy( ) for copying arrays.
- Try to use the fastest tests in loops.
- Convert equality comparisons to identity comparisons when possible.
- Phrase multiple boolean tests in one expression so that they "short circuit" as soon as possible.
- Eliminate unneeded temporary variables from loops.
- Try unrolling the loop to various degrees to see if this improves speed.

- Rewrite any switch statements to use a contiguous range of case values.
- Identify whether a recursive method can be made faster.

Tuning Threads

- Include multithreading at the design stage.
  - Parallelize tasks with threads to speed up calculations.
  - Run slow operations in their own threads to avoid slowing down the main thread.
  - Keep the interface in a separate thread from other work so that the application feels more responsive.
  - Avoid designs and implementations that force points of serialized execution.
  - Use multiple resolution strategies racing in different threads to get quicker answers.

Tuning JMS

- Remember the following points to ensure optimal JMS performance:
  - Close resources (e.g., connections, session objects, producers, and consumers) when you finish with them.
  - Start the consumer before the producer so the initial messages do not need to queue when waiting for the consumer.
  - Nontransactional sessions are faster than transactional ones. If you have transactional sessions, try to separate nontransactional messages and use nontransactional sessions for them.

- Avoid locking more resources than necessary.
  - Avoid synchronizing methods of stateless objects.
  - Build classes with synchronized wrappers, and use synchronized versions except when unsynchronized versions are definitely sufficient.
  - Selectively unwrap synchronized wrapped classes to eliminate identified bottlenecks.
  - Avoid synchronized blocks by using thread-specific data structures, combining data only when necessary.
  - Use atomic assignment where applicable.
- Load-balance the application by distributing tasks among multiple threads, using a queue and thread-balancing mechanism for distributing tasks among task-processing threads.
  - Use thread pools to reuse threads if many threads are needed or if threads are needed for very short tasks.
  - Use a thread pool manager to limit the number of concurrent threads used.

Tuning JMS

- Nonpersistent messages are faster than persistent messages.
- Longer messages take longer to deliver and process. You could compress message bodies or eliminate nonessential content to keep the size down.
- The redelivery count should be specified to avoid indefinitely redelivered messages. A higher redelivery delay and lower redelivery limit reduces overhead.
- Set the Delivery TimeToLive value as low as is feasible (the default is for messages to never expire).
- A smaller Delivery capacity increases message throughput. Since fewer messages can sit in the Delivery queue, they have to be moved along more quickly. However, if the capacity is too small, efficiency is reduced because producers have to delay sending messages until the Delivery queue has the spare capacity to accept them.

Tuning Servlets and JSP

- Don't use SingleThreadModel. Make the servlet thread-safe, but try to minimize the amount of time spent in synchronized code while still maintaining a thread-safe servlet.
- Use as many servlet threads as necessary to handle the request throughput. Use resource pools to distribute resources among the servlet threads.
- The amount of data sent in the first network packet is crucial to optimal performance. Send the response headers and the first load of data as a single packet instead of two separate packets.

- Use StringBuffers or other efficient String or byte array-building mechanisms. Avoid generating intermediate Strings and other objects whenever possible. Avoid the + and += concatenation operators.
- Use the browser's caching mechanism to have pages reread by correctly implementing the getLastModified( ) method.
- Precalculate all static formatting for generated HTML pages. High-volume web applications prerender pages that are the same for all users.
- Use the include directive rather than the include action.
- Minimize the useBean action's scope to page where possible.

- Remember that redirects (using sendRedirect( )) are slower than forwards (<jsp:forward ...>).
- Avoid creating HttpSession objects if not needed, and time out HttpSessions when they are needed.
- "Context" has a much wider scope than "session." Use HttpSession methods for session resources.
- Avoid having HttpSession objects serialized by the servlet container. Remove HttpSession objects explicitly with HttpSession.invalidate( ) when the session is finished, such as when the user logs out.

- Implement the HttpSessionBindingListener for any resources that need to be cleaned up when sessions terminate, and explicitly release resources in the valueUnbound( ) method.
- The servlet init( ) and destroy( ) or jspInit( ) and jspDestroy( ) methods are ideal for creating and destroying limited and expensive resources, such as cached objects and database connections.
- Compress output if the browser supports displaying compressed pages.
- Avoid reverse DNS lookups.
- Precompile your JSPs.
- Remember that servlet filters have overhead associated with the filter mechanism.
- Validate data at the client if it can be done efficiently.
- Increase server TCP/IP listen queues.
- Disable autoreloading features that periodically reload servlets and JSPs.
- Tune the pool sizes in the server.
- Transform your data to minimize the costs of searching it.
- **Disable Access Logging**

Tuning EJB

- The performance of EJB-based J2EE systems overwhelmingly depends on getting the design right. Use performance-optimizing design patterns: Value Object, Page-by-Page Iterator, ValueListHandler, Data Access Object, Fast Lane Reader, Service Locator, Verified Service Locator, EJBHomeFactory, Session Façade, CompositeEntity, Factory, Builder, Director, Recycler, Optimistic Locking, Reactor, Front Controller, Proxy, Decorator, and Message Façade.
- Explicitly remove beans from the container when a session is expired. Leaving beans too long will get them serialized by the container, which can dramatically decrease performance.

- Coarse-grained EJBs are faster. Remote EJB calls should be combined to reduce the required remote invocations.
- Design the application to access entity beans from session beans.
- Collocated EJBs should be defined as Local EJBs (from EJB 2.0), collocated within an application server that can optimize local EJB communications, or built as normal JavaBeans and then wrapped in an EJB to provide one coarse-grained EJB (CompositeEntity design pattern).
- EJBs should not be simple wrappers on database data rows; they should have business logic. To simply access data, use JDBC directly.

- Stateless session beans are faster than stateful session beans. If you have stateful beans in your design, convert them to stateless session beans by adding parameters that hold the extra state to the bean methods.
- Optimize read-only EJBs to use their own design, their own application server, read-only transactions, and their own optimal configuration.
- Cache JNDI lookups.
- Use container-managed persistence (CMP) by default. Profile the application to determine which beans cause bottlenecks from their persistency, and implement bean-managed persistence (BMP) for those beans.
- Use the Data Access Object design pattern to abstract your BMP implementations so you can take advantage of optimizations possible when dealing with multiple beans or database-specific features.

- Minimize the time spent in any transaction, but don't shorten transactions so much that you are unnecessarily increasing the total number of transactions. Combine transactions that are close in time to minimize overall transaction time. This may require controlling the transaction manually (i.e., turning off auto-commit for JDBC transactions or using TX_REQUIRED for EJBs).
- J2EE transactions are defined with several isolation modes. Choose the lowest-cost transaction isolation level that avoids corrupting the data. Transaction levels in order of increasing cost are: TRANSACTION_READ_UNCOMMITTED, TRANSACTION_READ_COMMITTED, TRANSACTION_REPEATABLE_READ, and TRANSACTION_SERIALIZABLE.

- Don't leave transactions open, relying on the user to close them. There will inevitably be times when the user does not close the transaction, and the consequent long transaction will decrease the performance of the system significantly.
- Bulk or batch updates are usually more efficiently performed in larger transactions.
- Lock only where the design absolutely requires it.
- Beans.instantiate( ) incurs a filesystem check to create new bean instances in some application servers. Use the Factory pattern with new to avoid the filesystem check.
- Tune the message-driven beans' pool size to optimize the concurrent processing of messages.
- Use initialization and finalization methods to cache bean-specific resources. Good initialization locations are setSessionContext( ), ejbCreate( ), setEntityContext( ), and setMesssageDrivenContext( ); good finalization locations are ejbRemove( ) and unSetEntityContext( ).
- Tune the application server's JVM heap, pool sizes, and cache sizes.

Tuning Web Services

- Web services performance is affected primarily by these characteristics of the XML documents that are sent to Web services:
- **Size** refers to the length of data elements in the XML document.
- **Complexity** refers to number of elements that the XML document contains.
- **Level of nesting** refers to objects or collections of objects that are defined within other objects in the XML document, as in this example:

```
<primaryObject>
    <groupObject>
        <singleObject/>
        <singleObject/>
    </groupObject>
</primaryObject>
```

- In addition, a Web services engine contains three major pressure points that define the performance of Web services:
- **Parsing** (Input): When a request is received, the Web services engine parses the input. There are two major performance components in parsing:
  - scanner
  - symbol or name identification
- **XML-to-object deserialization** (Input): As the document is parsed the XML input is deserialized and converted into business objects that are presented as business object parameters to the Web services. The Web services provider, which is a JavaBean provider, is not aware of its participation in a Web service.
- **Object-to-XML serialization** (Output): After the request is processed, reply is serialized into an XML document. Large documents or complex objects can affect output serialization.

- Design Web Service applications for course-grained service with moderate size payloads.
- Choose correct service (RPC or Document) and encoding (Encoded or Literal) style.
- Control serializer overheads and namespaces declarations to achieve better performance.
- Use MTOM/XOP or Fast Infoset to optimizing the format of a SOAP message.
- Carefully design SOAP attachments and security implementations to minimize negative performance.
- Consider using an asynchronous messaging model for applications with:
- Slow and unreliable transport.
- Complex and long-running process.
- Use replication and caching of data and schema definitions to improve performance by minimizing network overhead.
- Use XML compression where the XML compression overhead is less than network latency.