Introduction: why performance tuning is crucial for Java™

To create a Java™ program that is functionally perfect yet runs horribly slow is to create a Java program that... runs horribly slow.

Borland® Optimizeit™ Suite provides developers with a complete solution for managing application performance, including minimizing excessive temporary object use.

In light of today’s compressed development cycles, multi-tiered architectures, and complex technologies, many organizations are challenged to get stable enterprise applications out the door in a timely manner. Devoting a small amount of energy throughout the development process to address and correct performance obstacles dramatically lowers the risks and costs associated with poorly performing applications over the life of the code.

Java performance tuning simply means optimizing your code for speed, reliability, scalability, and maintainability. Producing truly scalable, lightening-fast J2SE™ and J2EE™ applications demands clarity of purpose and well-understood programming priorities. A major benefit of adopting regular performance tuning is instantly seeing exactly which parts of your applications represent important bottlenecks and which are behaving efficiently.
Performance tuning: a development best practice

A major strength of Java is its platform-independent byte-code approach and automatic handling of garbage collection. Unlike in C/C++, developers are able to focus on an application’s business requirements and largely are free from platform considerations.

Experienced developers, however, do not focus exclusively on application functionality. The reality is that below this level of abstraction, hard limitations of memory and processing power exist, as do the patterns and constraints of garbage collection, thread scheduling, and a host of other considerations managed by the Java virtual machine (JVM™) and the operating system below it.

Successful developers—and productive development processes—incorporate regular use of performance analysis from the earliest stage of code creation throughout the development process, into QA testing, and beyond. Regular frontline checking and testing of small modules of code by the principal author is a proven way to assure that the Java applications produced will be fast, reliable, and scalable.

Not all code is worth optimizing

The latest generation of application servers bring increased memory and processing power to the party. Yet sheer hardware alone can never overcome truly flawed code. A single buggy line ripples forward and can cause application-wide bottlenecks or can mysteriously trigger disastrous crashes once an application is in production.

The developer’s challenge is that it can be almost impossible to know which part of a J2SE or J2EE application is causing a speed bottleneck or memory issue. The strength of Java and the J2SE and J2EE platforms is the high level of abstraction, re-use of objects, and insulation from layers of processing and system interpretations. But while encapsulation is great for shielding you from vast lower-level complexities, it also leaves you with few clues about where to focus your performance attention.

Tools are needed that extend your intuition and let you effortlessly see and understand how your J2SE or J2EE application behaves when it is actually running. Many years ago Stanford Computer Science Emeritus Professor Donald Knuth famously warned that prematurely optimizing code can lead to wasted effort and poor program architecture. This caution remains true. In fact, the danger is not only in prematurely optimizing, but in misguided optimization in general—focusing on unimportant aspects of the code, or rebuilding the code based on flawed or restricted logic.

With the advent of highly abstracted, object-oriented languages such as Java, Knuth raised the related concern that programmers are in danger of losing touch with the factors determining whether their code will run and scale well: “At first you try to ignore the details of what’s happening at the lower levels. But when you’re debugging, you can’t afford to be too compartmentalized. You can’t afford to only see things at the highest level of abstraction.”

Performance tools allow you to be smart and efficient about optimizing

Knuth advises that developers need insight about what’s going on below the surface if their code is to be scalable, reliable, and fast. The Optimizeit™ Suite of tools are designed to efficiently give developers easy-to-understand, powerful views into the Java virtual machine, granting just this kind of insight.

A fundamental question is: “What are the priority performance issues for this module or application?” Tools like Borland® Optimizeit™ Suite, specifically designed for Java performance tuning, offer an ideal way to answer this question—and be assured that your improvements are informed and efficient. Without tools to help prioritize key Java trouble areas, you are likely to spend time micro-optimizing unimportant sections at the expense of addressing issues that actually drive your application’s overall performance.

The goal ought to be for each member of the development team to be equipped with the tools to be smart about performance tuning each step of the way. Smart performance tuning will take place in the context of an application’s overall business requirements. Some
tiny performance issues simply may not warrant improvement efforts. Other, important optimizing trade-offs will arise only when components are brought together, at which point an understanding of the overall architecture will guide modifications.

**Achieving fast, reliable code**

So what are the causes of poor performance? Three crucial performance issues for Java programs are: excessive use of temporary objects, Java memory leaks, and speed bottlenecks due to poor CPU utilization. This paper focuses on problems due to excessive temporary object allocations.

Tuning your code for speed and performance iteratively, as you develop and bring modules of code together, is the best way to minimize frantic troubleshooting sessions at the end of a project—or, worse, in production, where even small problems are transformed into costly, complex challenges. The tuned applications delivered to QA and to customers will instead be lightweight, stable, scalable, and screamingly fast.

**Address Book J2EE™ application example**

The Address Book J2EE application example is a basic Web application that has been developed for the purpose of this test case. The application allows users to search for contact information stored in an address book. A user can decide to add a person to a list of selected contacts. The list of contacts is available and can be displayed until the user closes the session.

The application uses only servlet components for the sake of simplicity and ease of deployment. The address book data is contained in a XML file, and the Address Book J2EE application example uses the SAX API (Simple API for XML) to retrieve the information from the XML file when needed.

For this example, Address Book WAR files are available for download from http://www.optimizeit.com

Similar Optimizeit Suite performance tuning guides—using the same J2EE application example—takes you through the steps of troubleshooting Java memory leaks and speed bottlenecks due to poor CPU utilization. The Optimizeit™ Profiler (the tool used in each of these examples) integrates easily into your development environment and the intuitive GUI will have you expertly troubleshooting memory leaks and performance bottlenecks in minutes.

**Before you begin**

Before exploring the test case, you will need to complete a few preparation steps. To get ready, make sure you have:

- Installed an application server. If you have not installed an application server, you may download and install the Tomcat server for free from http://jakarta.apache.org
- Integrated Optimizeit™ Profiler with your application server. Appendix A provides a guide for doing this.
- Deployed the Address Book J2EE application example in your application server. Appendix B provides a guide for doing this.
- Launched your application server from within Optimizeit Profiler.
- Attached Optimizeit Profiler to your application server.

Once these steps have been performed, your application server and Optimizeit Profiler will be running. Optimizeit Profiler will be displaying the Heap View (showing a number of red bars). At this point, you should be able to access the Address Book J2EE application example from your browser using a URL similar to http://localhost/addressbook/enter

Note: you may have to modify the URL if your application server uses a port other than the default of 80. If the server uses the port 8080, for example, then the URL to use becomes http://localhost:8080/addressbook/enter
If you have a problem with any of these preparation steps, Appendix C provides a troubleshooting guide.

Finding and fixing excessive temporary object allocations

We start by focusing on memory issues. A common performance problem in Java is excessive use of temporary objects. The good news with Java is that objects are easy to allocate—the bad news is that cleaning up memory when too many temporary objects have been allocated can become a time-consuming and performance-degrading garbage collection task.

While different Java virtual machines use different techniques for garbage collection, no virtual machine can garbage collect instantaneously. The garbage collection process must sort through the heap to find all referenced objects and remove all unreferenced objects. This takes time. The more temporary object allocations there are, the more time it takes.

The reason why garbage collection may appear to have a disproportionate effect on an application’s overall performance is that, for most virtual machines, when the garbage collector runs, all other threads are stopped. The latest generation of the virtual machine, such as Sun® JDK® 1.4, permit the option of running concurrent garbage collection. In any case, if the garbage collector is run often, or needs to run for a long time, an application will run correspondingly slower.

• A key performance objective is therefore to make the job of the garbage collector as easy as possible. Keep in mind that a big heap size will cause garbage collection to take longer.
• Many objects, or complex reference trees will cause garbage collection to take longer.
• Limited free memory available in the heap will cause the garbage collection to run more frequently.

In short, if an application creates a lot of temporary objects, it will use more heap space, causing the garbage collector to run more often and for longer periods. The ratio of overhead-to-data is worst for small objects, so that an application that repeatedly creates many small objects will be garbage collection inefficient.

While some use of temporary objects may be necessary and appropriate, you can often avoid the creation of temporary objects by caching objects. This lightens the garbage collection burden and will improve an application’s overall performance.

Address Book J2EE™ application example—determining which classes create the most instances of temporary object allocation problems

We will now see how the Optimizeit Profiler enables you to track down temporary object allocation problems.

1. Switch to the Optimizeit Profiler. You should see the Heap View (view displaying the horizontal red bars). If you do not see the horizontal red bars, make sure to click the button Show memory profiler on the tool bar, and then the button Show heap. This view shows you all of the classes of your application and displays in real-time how many instances exist in the virtual machine for each class.

2. On the bottom right of the Heap View, there is an option Disable garbage collector.

Select this option. This option disables the garbage collector logically, meaning that the garbage collector still runs as normal in the virtual machine, but the Optimizeit Profiler reports the information as if the garbage collector had been turned off, allowing you to keep track of the compounding accumulation of temporarily created objects.

3. Now click on the Mark current instance count button on the tool bar (button with a pencil). You will note that the “diff” column in the table now reports “None.”
This column will now show a running tally of the number of objects that are created since you put the mark.

4. Go into your browser and open the starting page of the Address Book J2EE application example. In the first name text field, enter Wladimir then click Start Search. After few seconds, the list of people named Wladimir appears in your browser.

5. Return to the Optimizeit Profiler. Look at the value of the diff column. It displays the number of objects created since you selected “mark current instance count,” which is the number of objects created when you performed the search request. You may want to sort by diff by clicking the header of the diff column. You can see that char[] (array of characters) is ranked first, with more than 120,000 instances created since the mark.

6. In the tool bar, click on the i button.

7. This opens the Inspector window. The Inspector shows the options for the current view. In the Inspector, select the Show sizes option. This adds two more columns to the table: “Size” and “Size diff.” The column Size shows how much memory all the instances of the corresponding class use. The column Size diff shows how much memory is used just by the instances created since the last mark. The Size diff column shows that all the arrays of chars created since the mark use more than 5 Mb. So if a single search for a contact created 5 Mb of char arrays, in a real-life situation with many users connected, this is likely to represent a prohibitively large amount of memory.

Determining where the instanciations occur

Our goal now is to figure out how to reduce the number of char arrays created.

1. Select the char[] line in the table and click on the button Show alloc backtrace on the tool bar (note that you can also double-click the char[] line). The Optimizeit Profiler switches to the backtrace mode. This mode shows you where the instances of the selected class (here char[]) were created. The tree is the call graph of the methods responsible for the creation of the char[].

2. First open the Inspector by clicking the i button on the tool bar. In the Inspector, select the option Show allocations since mark. With that option selected, the backtrace mode shows only the information for the objects created since the last mark. We have now isolated the backtraces for the char[] created during the contact search.

3. Drill down the tree on the node with the highest percentage. Your backtrace should follow a similar pattern to the one described below. The method AddressBook.characters() is responsible for more than 75 percent of the char[] allocations.

4. Double-click the line that shows more than 50 percent. This opens the source code viewer, and you will see the relevant line of code highlighted. (If you have not configured your source path correctly, the source code viewer displays that the source file could not be found. In this case, click on the browse button and browse for the source file.) Let’s focus on the line highlighted by the source code viewer

   ```java
   public void characters(char[] ch, int start, int length) {
   nodeValue+=new String(ch,start,length);
   }
   ```

   This method is called by the SAX parser when parsing the XML address book file. SAX (Simple API for XML) is used to parse the XML document. SAX provides an interface that defines a standard
set of parse event listeners that notifies your code when an XML node is parsed. When SAX parses characters, it calls the characters method and provides an array of chars with the characters that have been parsed.

In our Address Book J2EE application example, this method is called when parsing the information contained in a node, like the name of a person or an address. We can not be sure that this method will be called only once for contiguous characters. The parsing of a name, for example, may result in several calls to this method which is why we need to append the given value to the string nodeValue.

If the value retrieved does not match the search criteria, then this value is no longer needed. We can easily improve performance here by using a StringBuffer. The StringBuffer class contains a char array of a fixed length. Rather then creating a String each time characters() is called, we can copy the chars in the StringBuffer, so we always use just one StringBuffer.

Improving the code
This modification is implemented in a new version of the AddressBook class called AddressBook1. The code becomes:

```java
public void characters(char[] ch, int start, int length) {
    nodeValue.append(ch, start, length);
}
```

where nodeValue is now a StringBuffer. The purpose of saving the creation of Strings (and therefore arrays of chars) is so that we:

- Do not have to create a new string to append some characters to the current node value (the insert method of StringBuffer only copies the characters at the specified index in the StringBuffer).
- Do not have to create a new string each time we have finished parsing character data inside an element.

- We save the creation of an extra StringBuffer caused by the + binary string concatenation operator. For example, the code:
  ```java
  myString += "more";
  ```
  is compiled to the equivalent of:
  ```java
  myString=new StringBuffer(myString).append("more").toString();
  ```

Let’s review how we have improved memory usage:

- Access the modified version of the address book at a URL similar to http://localhost/addressbook1/enter
- In the Optimizeit Profiler, put a temporal mark by clicking the button Mark current instance count on the tool bar.
- In the address book page in your browser, enter Wladimir in the first name text field, then click Search. When the request is done, you can see in the Optimizeit Profiler that there are now about 40,000 char arrays created since the mark, which is about 1.7 Mb. We have reduced by a factor of 3 the number of arrays of characters created—and the temporary memory they use.

Important: When you have finished tracking for temporary object allocations, be sure to unselect the option Disable garbage collector at the bottom of the Heap View. This option causes the Optimizet Profiler to retain the information for the garbage collected objects, consuming some processing time and memory.

Conclusion: performance tuning is crucial for Java™
Conceiving, designing, and testing your approach against performance goals as you build means more than just avoiding dog-slow applications and crashes (though that’s always a good start). By being appropriately alert to how your code performs throughout the development process, you avoid expensive, disruptive late-stage fixes. Fast, scalable, high-performance code is
a design imperative from the beginning. It is also a serious, regularly exercised element of the development process for each front-line developer (not a specialized skill for an isolated performance team).

The premise behind getting tools like the Optimizeit Suite in the hands of each developer on the team is simple: Nobody understands a piece of code as well as the person who creates it, and nobody is better positioned than the author to make improvements in the logic and implementation of that code.

This Java performance tuning paper and test case has focused on troubleshooting excessive temporary object allocations. You have seen how to use the Optimizeit Profiler to rapidly pinpoint and fix problems down to the responsible line of code. While this J2EE Address Book application example was small, it could be extended with EJBs that perform the Address Book database transactions, or it could use JMS, JNDI, or JDBC. In each case, the Optimizeit Profiler would provide you with all the necessary information to investigate memory or performance issues across all EJBs, servlets, or JSPs. Similar Optimizeit™ Suite guides are available, using the same Address Book J2EE application example, that describe how to troubleshoot two other important Java performance problem areas: Java memory leaks and speed bottlenecks due to poor CPU utilization.

As each routine or subsystem’s basic functionality is established, the module’s performance can be evaluated within the context of the applications overall business requirements. Clearly not every section of code will need to be labored over with equal intensity (many parts will require no attention at all), and certain key issues will emerge only as pieces of code are combined. Whatever the situation, spotting and assessing the severity of issues throughout the development process avoids delays, problem escalations, and the kind of deep-rooted performance flaws that can sideline an entire project or create business disasters once an application is live with customers in a production environment.

The Borland® Optimizeit™ Suite of performance tools let you keep a tight rein on code, regularly testing where design and compiled code meet hardware and Java virtual machine constraints. Your efforts will stay focused on priority problems, and you will be rewarded with fast, reliable, and truly scalable J2SE and J2EE applications.

**Borland® Optimizeit™ Suite tools make performance tuning easy**

The 80:20 rule fits well for performance tuning. Eighty percent of an application’s performance problems are usually caused by no more than 20 percent of the code. The pitfall that development teams fall into is that they delay the identification of key problem areas until too late, leading to expensive and risky late-stage application redesign. Taking a proactive approach will preempt just this type of rushed, endgame rework. Adopting tools such as the Optimizeit Suite allows everyone in a development team to become more conscious of—and more conscientious about—the application’s speed, scalability, and reliability throughout the development process.
Appendix A

Configuring the Optimizeit™ Profiler with your application server

Integrating the Optimizeit™ Profiler with Tomcat 3.2.1
The Optimizeit Profiler provides a wizard (accessible from the Tools/Application server integration menu) which will automatically perform the integration with Tomcat 3.2.1. Just run the wizard, and it will guide you through the integration process.

Note that you can also use the tutorial available from the Info/Tutorials menu. This tutorial describes how to manually perform the integration.

Starting Tomcat for profiling
Once Tomcat has been integrated with the Optimizeit Profiler, use the script `tomcatWithOptimizeIt.bat` to start Tomcat for profiling. Once Tomcat has started, you can attach from the Optimizeit Profiler GUI: start Optimizeit Profiler, then select the Program/Attach menu. Keep 1470 for the Port Number value then click Attach. You are now profiling in Tomcat.

JRun™ 3.0
Integrating the Optimizeit Profiler with JRun™ 3.0

The Optimizeit Profiler provides a wizard (accessible from the Tools/Application server integration menu) which will automatically perform the integration with JRun 3.0. Just run the wizard; it will guide you through the integration process.

Note that you can also use the tutorial available from the Info/Tutorials menu. This tutorial describes how to manually perform the integration.

Starting the profiling in JRun™ 3.0
Once you have correctly configured JRun 3.0 for profiling with the Optimizeit Profiler, restart the JRun server. From your browser, access the Optimizeit Profiler servlet, with a URL similar to http://localhost/servlet/OptimizeIt. Click on the Start Optimizeit Audit System button on the servlet. The browser should report that the Optimizeit Profiler Audit system is running and waiting on port 1470. At that point, start the Optimizeit Profiler, if it is not already started, then select the Program/Attach menu. Keep 1470 for the Port Number value then click Attach. You are now profiling in JRun.

Integrating the Optimizeit™ Profiler with BEA® WebLogic® 5.1
The Optimizeit Profiler provides a wizard (accessible from the Tools/Application server integration menu) which will automatically perform the integration with WebLogic® 5.1. Just run the wizard; it will guide you through the integration process.

You will have to modify the script used to start WebLogic to add the Xerces(XML) jar file to the CLASSPATH. This is described in the Appendix B: Deploying in WebLogic 5.1.

Starting WebLogic® 5.1 for profiling
Once WebLogic has been integrated with the Optimizeit Profiler, use the script `startWebLogicWithOptimizeIt.bat` to start WebLogic for profiling. Once WebLogic has started, you can attach from the Optimizeit Profiler GUI: start Optimizeit Profiler, then select the Program/Attach menu. Keep 1470 for the fcPort Number value then click Attach. You are now profiling in WebLogic.

Other application servers

Configuring the Optimizeit™ Profiler with other J2EE™ application servers
Many current application servers are fully J2EE compliant and so support the deployment of Web Archive (WAR files). The
Optimizeit Suite works with and supports most of the major application servers. Please consult the documentation of your application server and contact us at www.borland.com/optimizeit for a complete list of application servers or for other information.

Appendix B

Deploying the Address Book J2EE™ application example in your application server

Deploying in Tomcat 3.2.1

1. Copy the files addressbook.war, addressbook1.war, addressbook2.war and addressbook3.war under the directory <Tomcat>/webapps (where <Tomcat> is the directory where you installed Tomcat).

2. Restart the Tomcat server.

The different versions of the Address Book J2EE application example can be accessed with URLs similar to:

http://localhost/addressbook/enter
http://localhost/addressbook1/enter
http://localhost/addressbook2/enter
http://localhost/addressbook3/enter

Note: You may have to modify these URLs if your application server does not use the port 80. If it uses the port 8080, for example, the URLs become: http://localhost:8080/addressbook/enter

Deploying in JRun™ 3.0

You will need the version 3.02 or later of JRun to deploy the Address Book Test Case. Make sure to update JRun to that version or later if you run a version prior to 3.02.

1. Access the JRun Management Console from your browser.
   If you have a default install, the URL http://localhost:8000 will take you there.

2. Login.

3. On the left side of the browser, click on the server where you would like the address book to be deployed.

4. On the right side of the browser, click on the link [WAR deployment]

5. In the Servlet WAR File or Directory field enter the path to the addressbook.war file (you may use the Browse button to browse for this file).

6. In the JRun Server Name list box, make sure to select the server where the Address Book J2EE application example should be deployed.

7. In the Application Name field, enter AddressBook

8. In the Application URL field, enter /addressbook

9. Click the deploy button. After a few seconds, JRun should report that the war file has been successfully deployed.

10. Perform Steps 5 to 9 again with the war files:
    addressbook1.war, addressbook2.war, and
    addressbook3.war. Use the respective Application Names AddressBook1, AddressBook2, and AddressBook3; and use the Application URLs /addressbook1, /addressbook2 and /addressbook3.

11. After successfully deploying the four war files, restart the server. To do this, click the node of the server where you have just deployed on the left side of the browser, then click the restart server button on the right side of the browser.

The different versions of the Address Book J2EE application example can be accessed with URLs similar to:

http://localhost/addressbook/enter
http://localhost/addressbook1/enter
http://localhost/addressbook2/enter
http://localhost/addressbook3/enter
Note: You may have to modify these URLs if your application server does not use the port 80. If it uses the port 8080, for example, the URLs become:
http://localhost:8080/addressbook/enter
http://localhost:8080/addressbook/enter_

**Deploying in WebLogic® 5.1**

1. Create a directory where the first version of the addressbook will be stored (you can choose any directory, for example c:\weblogic51\example\addressbook).

2. From the command line, go into this directory and unjar the addressbook.war, using the command line:

   `jar xvf addressbook.war`

   Note: You need to have jar in your PATH to be able to run this command line. If jar is not found, make sure to add `<JDK>\bin` in your PATH, where JDK is the directory where you installed your JDK.

3. Repeat the previous steps with the files addressbook1.war, addressbook2.war, and addressbook3.war (for example, extract these files in directories:
   c:\weblogic51\example\addressbook1,
   c:\weblogic51\example\addressbook2,
   c:\weblogic51\example\addressbook3)

4. Edit the weblogic.properties file (this file should be located under your WebLogic directory). Append at the end of the file the following lines:

   ```
   weblogic.httpd.webApp.addressbook=c:/weblogic51/examples/addressbook
   weblogic.httpd.webApp.addressbook1=c:/weblogic51/example/addressbook1
   weblogic.httpd.webApp.addressbook2=c:/weblogic51/example/addressbook2
   ```

   Make sure to use / instead of \ in the paths.

   You also need to have the Xerces (XML) package added in your classpath:

   1. Download Xerces if you do not have it from http://xml.apache.org/dist/xerces-j/Xerces-J-bin.1.3.1.zip

   2. Unzip the Xerces-J-bin.1.3.1.zip package on your disk

   Edit the file `startWebLogicWithOptimizeIt.bat` and add the path to the xerces.jar file contained in the package you have just extracted after the line starting with:

   ```
   set JAVACLASSPATH=
   ```
3. Restart the WebLogic server with the script `startWebLogicWithOptimizeIt.bat`. The different versions of the Address Book J2EE application example can be accessed with URLs similar to:

http://localhost/addressbook/enter
http://localhost/addressbook1/enter
http://localhost/addressbook2/enter
http://localhost/addressbook3/enter

Note: You may have to modify these URLs if your application server does not use the port 80. If it uses the port 8080, for example, the URLs become:

http://localhost:8080/addressbook/enter

Appendix C

Troubleshooting

My application server is not described in this document, how should I proceed?

Most of today’s application servers are fully J2EE compliant and will therefore support the deployment of Web Archive (WAR files). Please consult the documentation of your application server.

With JRun 3.0, the browser displays a Null Pointer Exception caused by a Servlet Exception when I access the Address Book page.

You may be running a version of JRun prior to the 3.02 (JRun 3 Service Pack 2). The version of JRun you are running can be checked from the default page of the JRun Management Console. Make sure to upgrade your JRun server to the 3.02 version of JRun or later.

With WebLogic 5.1, the Browser displays “Error 500 Internal Server Error” when I access the Address Book page.

Look at the console where you started WebLogic. If you get the exception `java.lang.NoClassDefFoundError: javax/xml/parsers/ParserConfigurationException` it means the CLASSPATH is not set correctly and does not include the xerces.jar file. Make sure to double check the CLASSPATH in the script that you use to start WebLogic.

My question is not addressed in this document, what should I do next?

Send any question you may have to the Optimizeit technical support team at www.borland.com/optimizeit. Please make sure to describe the issue precisely, including your platform, your application server, and any error message and stack trace generated.