Publications in The Java™ Series are supported, endorsed, and written by the creators of Java at Sun Microsystems, Inc. This series is the official source for expert instruction in Java and provides the complete set of tools you'll need to build effective, robust, and portable applications and applets. The Java™ Series is an indispensable resource for anyone looking for definitive information on Java technology.

Visit Sun Microsystems Press at sun.com/books to view additional titles for developers, programmers, and system administrators working with Java and other Sun technologies.
To the three B’s, Barb, Boyd, and Beau – C.H.
To Rita, Rachael, and Kevin – B.J.
This page intentionally left blank
# Contents

Foreword by James Gosling xi
Foreword by Steve Wilson xiii
Preface xv
Acknowledgments xix
About the Authors xxii

## Chapter 1
Strategies, Approaches, and Methodologies 1
Forces at Play 2
Two Approaches, Top Down and Bottom Up 5
Choosing the Right Platform and Evaluating a System 8
Bibliography 11

## Chapter 2
Operating System Performance Monitoring 13
Definitions 14
CPU Utilization 14
CPU Scheduler Run Queue 28
Memory Utilization 32
Network I/O Utilization 41
Disk I/O Utilization 46
<table>
<thead>
<tr>
<th>Chapter 7</th>
<th>Tuning the JVM, Step by Step</th>
<th>251</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Methodology</td>
<td>252</td>
</tr>
<tr>
<td></td>
<td>Application Systemic Requirements</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td>Rank Systemic Requirements</td>
<td>257</td>
</tr>
<tr>
<td></td>
<td>Choose JVM Deployment Model</td>
<td>258</td>
</tr>
<tr>
<td></td>
<td>Choose JVM Runtime</td>
<td>259</td>
</tr>
<tr>
<td></td>
<td>GC Tuning Fundamentals</td>
<td>262</td>
</tr>
<tr>
<td></td>
<td>Determine Memory Footprint</td>
<td>268</td>
</tr>
<tr>
<td></td>
<td>Tune Latency/Responsiveness</td>
<td>278</td>
</tr>
<tr>
<td></td>
<td>Tune Application Throughput</td>
<td>307</td>
</tr>
<tr>
<td></td>
<td>Edge Cases</td>
<td>316</td>
</tr>
<tr>
<td></td>
<td>Additional Performance Command Line Options</td>
<td>316</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>321</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>Benchmarking Java Applications</td>
<td>323</td>
</tr>
<tr>
<td></td>
<td>Challenges with Benchmarks</td>
<td>324</td>
</tr>
<tr>
<td></td>
<td>Design of Experiments</td>
<td>347</td>
</tr>
<tr>
<td></td>
<td>Use of Statistical Methods</td>
<td>348</td>
</tr>
<tr>
<td></td>
<td>Reference</td>
<td>355</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>355</td>
</tr>
<tr>
<td>Chapter 9</td>
<td>Benchmarking Multitiered Applications</td>
<td>357</td>
</tr>
<tr>
<td></td>
<td>Benchmarking Challenges</td>
<td>357</td>
</tr>
<tr>
<td></td>
<td>Enterprise Benchmark Considerations</td>
<td>360</td>
</tr>
<tr>
<td></td>
<td>Application Server Monitoring</td>
<td>382</td>
</tr>
<tr>
<td></td>
<td>Profiling Enterprise Applications</td>
<td>399</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>401</td>
</tr>
<tr>
<td>Chapter 10</td>
<td>Web Application Performance</td>
<td>403</td>
</tr>
<tr>
<td></td>
<td>Benchmarking Web Applications</td>
<td>404</td>
</tr>
<tr>
<td></td>
<td>Web Container Components</td>
<td>405</td>
</tr>
<tr>
<td></td>
<td>Web Container Monitoring and Performance Tunings</td>
<td>408</td>
</tr>
<tr>
<td></td>
<td>Best Practices</td>
<td>427</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>450</td>
</tr>
<tr>
<td>Chapter 11</td>
<td>Web Services Performance</td>
<td>453</td>
</tr>
<tr>
<td></td>
<td>XML Performance</td>
<td>454</td>
</tr>
</tbody>
</table>
Foreword

Tuning a Java application can be challenging in today’s large-scale mission-critical world. There are issues to be aware of in everything from the structure of your algorithms, to their memory allocation patterns, to the way they do disk and file I/O. Almost always, the hardest part is figuring out where the issues are. Even (perhaps especially) seasoned practitioners find that their intuitions are wrong. Performance-killing gremlins hide in the most unlikely places.

As Wikipedia says, “Science (from Latin: scientia meaning ‘knowledge’) is a systematic enterprise that builds and organizes knowledge in the form of testable explanations and predictions about the world.” Performance tuning must be approached as an experimental science: To do it properly, you have to construct experiments, perform them, and from the result construct hypotheses.

Fortunately, the Java universe is awash in performance monitoring tools. From standalone applications to profilers built into development environments to tools provided by the operating system. They all need to be applied in a cohesive way to tease out the truth from a sea of noise.

This book is the definitive masterclass in performance tuning Java applications. It readably covers a wide variety of tools to monitor and measure performance on a variety of hardware architectures and operating systems. And it covers how to construct experiments, interpret their results, and act on them. If you love all the gory details, this is the book for you.

—James Gosling
This page intentionally left blank
Today, Java is used at the heart of the world’s largest and most critical computing systems. However, when I joined the Java team in 1997 the platform was young and just gaining popularity. People loved the simplicity of the language, the portability of bytecodes, and the safety of garbage collection (versus traditional malloc/free memory management of other systems). However, there was a trade-off for these great features. Java was slow, and this limited the kinds of environments where you could use it.

Over the next few years, we set about trying to fix this. We believed that just because Java applications were portable and safe they didn’t have to be slow. There were two major areas where we focused our attention. The first was to simply make the Java platform faster. Great strides were made in the core VM with advanced Just In Time compilation techniques, parallel garbage collection, and advanced lock management. At the same time the class libraries were tweaked and tuned to make them more efficient. All this led to substantial improvements in the ability to use Java for larger, more critical systems.

The second area of focus for us was to teach people how to write fast software in Java. It turned out that although the syntax of the language looked similar to C, the techniques you needed to write efficient programs were quite different. To that end, Jeff Kessleman and I wrote one of the first books on Java performance, which was published back in 2000. Since then, many books have covered this topic, and experienced developers have learned to avoid some of the most common pitfalls that used to befall Java developers.
After the platform began to get faster, and developers learned some of the tricks of writing faster applications, Java transformed into the enterprise-grade software powerhouse it is today. It began to be used for the largest, most important systems anywhere. However, as this started to happen, people began to realize one part was still missing. This missing piece was observability. When these systems get larger and larger, how do you know if you’re getting all the performance you can get?

In the early days of Java we had primitive profiling tools. While these were useful, they had a huge impact on the runtime performance of the code. Now, modern JVMs come with built-in observability tools that allow you to understand key elements of your system’s performance with almost no performance penalty. This means these tools can be left enabled all the time, and you can check on aspects of your application while it’s running. This again changes the way people can approach performance.

The authors of *Java™ Performance* bring all these concepts together and update them to account for all the work that’s happened in the last decade since Jeff and I published our book. This book you are now reading is the most ambitious book on the topic of Java performance that has ever been written. Inside are a great many techniques for improving the performance of your Java applications. You’ll also come to understand the state of the art in JVM technology from the inside out. Curious about how the latest GC algorithms work? It’s in here! You’ll also learn how to use the latest and greatest observability tools, including those built into the JDK and other important tools bundled into popular operating systems.

It’s exciting to see how all these recent advancements continue to push the platform forward, and I can’t wait to see what comes next.

—Steve Wilson
VP Engineering, Oracle Corporation
Founding member of the Java Performance team
Coauthor of *Java™ Platform Performance: Strategies and Tactics*
Welcome to the definitive reference on Java performance tuning!

This book offers Java performance tuning advice for both Java SE and Java EE applications. More specifically, it offers advice in each of the following areas: performance monitoring, profiling, tuning the Java HotSpot VM (referred to as HotSpot VM hereafter), writing effective benchmarks, and Java EE application performance tuning. Although several Java performance books have been written over the years, few have packed the breadth of information found in this book. For example, the topics covered in this book include items such as an introduction into the inner workings of a modern Java Virtual Machine, garbage collection tuning, tuning Java EE applications, and writing effective benchmarks.

This book can be read from cover to cover to gain an in-depth understanding of many Java performance topics. It can also be used as a task reference where you can pick up the text, go to a specific chapter on a given topic of interest, and find answers.

Readers who are fairly new, or consider themselves a novice in the area of Java performance tuning, will likely benefit the most by reading the first four chapters and then proceeding to the topics or chapters that best address the particular Java performance tuning task they are undertaking. More experienced readers, those who have a fundamental understanding of performance tuning approaches and a basic understanding of the internals of the HotSpot VM along with an understanding of the tools to use for monitoring operating system performance and monitoring JVM performance, will find jumping to the chapters that focus on the performance tuning task at hand to be most useful. However, even those with advanced Java performance skills may find the information in the first four chapters useful.
Reading this book cover to cover is not intended to provide an exact formula to follow, or to provide the full and complete knowledge to turn you into an experienced Java performance tuning expert. Some Java performance issues will require specialized expertise to resolve. Much of performance tuning is an art. The more you work on Java performance issues, the better versed you become. Java performance tuning also continues to evolve. For example, the most common Java performance issues observed five years ago were different from the ones observed today. Modern JVMs continue to evolve by integrating more sophisticated optimizations, runtimes, and garbage collectors. So too do underlying hardware platforms and operating systems evolve. This book provides up-to-date information as of the time of its writing. Reading and understanding the material presented in this book should greatly enhance your Java performance skills. It may also allow you to build a foundation of fundamentals needed to become fluent in the art of Java performance tuning. And once you have a solid foundation of the fundamentals you will be able to evolve your performance tuning skills as hardware platforms, operating systems, and JVMs evolve.

Here’s what you can expect to find in each chapter.

Chapter 1, “Strategies, Approaches, and Methodologies,” presents various different approaches, strategies, and methodologies often used in Java performance tuning efforts. It also proposes a proactive approach to meeting performance and scalability goals for a software application under development through an enhancement to the traditional software development process.

Chapter 2, “Operating System Performance Monitoring,” discusses performance monitoring at the operating system level. It presents which operating system statistics are of interest to monitor along with the tools to use to monitor those statistics. The operating systems of Windows, Linux, and Oracle Solaris are covered in this chapter. The performance statistics to monitor on other Unix-based systems, such as Mac OS X, use similar commands, if not the same commands as Linux or Oracle Solaris.

Chapter 3, “JVM Overview,” provides a high level overview of the HotSpot VM. It provides some of the fundamental concepts of the architecture and workings of a modern Java Virtual Machine. It establishes a foundation for many of the chapters that follow in the book. Not all the information presented in this chapter is required to resolve every Java performance tuning task. Nor is it exhaustive in providing all the necessary background to solve any Java performance issue. However, it does provide sufficient background to address a large majority of Java performance issues that may require some of the concepts of the internal workings and capabilities of a modern Java Virtual Machine. The information in this chapter is applicable to understanding how to tune the HotSpot VM along with understanding the subject matter of Chapter 7 and how to write effective benchmarks, the topics covered in Chapters 8 and 9.

Chapter 4, “JVM Performance Monitoring,” as the title suggests, covers JVM performance monitoring. It presents which JVM statistics are of interest to monitor
along with showing tools that can be used to monitor those statistics. It concludes with suggesting tools that can be extended to integrate both JVM level monitoring statistics along with Java application statistics of interest within the same monitoring tool.

Chapter 5, “Java Application Profiling,” and Chapter 6, “Java Application Profiling Tips and Tricks,” cover profiling. These two chapters can be seen as complementary material to Chapter 2 and Chapter 4, which cover performance monitoring. Performance monitoring is typically used to identify whether a performance issue exists, or provides clues as to where the performance issue exists, that is, in the operating system, JVM, Java application, and so on. Once a performance issue is identified and further isolated with performance monitoring, a profiling activity usually follows. Chapter 5 presents the basics of Java method profiling and Java heap (memory) profiling. This profiling chapter presents free tools for illustrating the concepts behind these types of profiling. The tools shown in this chapter are not intended to suggest they are the only tools that can be used for profiling. Many profiling tools are available both commercially and for free that offer similar capabilities, and some tools offer capabilities beyond what’s covered in Chapter 5. Chapter 6 offers several tips and tricks to resolving some of the more commonly observed patterns in profiles that tend to be indicative of particular types of performance problems. The tips and tricks identified in this chapter are not necessarily an exhaustive list but are ones that have been observed frequently by the authors over the course of years of Java performance tuning activities. The source code in many of the examples illustrated in this chapter can be found in Appendix B.

Chapter 7, “Tuning the JVM, Step by Step,” covers tuning the HotSpot VM. The topics of tuning the HotSpot VM for startup, memory footprint, response time/latency, and throughput are covered in the chapter. Chapter 7 presents a step-by-step approach to tuning the HotSpot VM covering choices such as which JIT compiler to use, which garbage collector to use, and how to size Java heaps, and also provides an indication when the Java application itself may require some rework to meet the performance goals set forth by application stakeholders. Most readers will likely find Chapter 7 to be the most useful and most referenced chapter in this book.

Chapter 8, “Benchmarking Java Applications,” and Chapter 9, “Benchmarking Multi-tiered Applications,” present information on how to write effective benchmarks. Often benchmarks are used to help qualify the performance of a Java application by implementing a smaller subset of a larger application’s functionality. These two chapters also discuss the art of creating effective Java benchmarks. Chapter 8 covers the more general topics associated with writing effective benchmarks such as exploring some of the optimizations performed by a modern JVM. Chapter 8 also includes information on how to incorporate the use of statistical methods to gain confidence in your benchmarking experiments. Chapter 9 focuses more specifically on writing effective Java EE benchmarks.
For readers who have a specific interest in tuning Java EE applications, Chapter 10, “Web Application Performance,” Chapter 11, “Web Services Performance,” and Chapter 12, “Java Persistence and Enterprise Java Beans Performance,” focus specifically on the areas of Web applications, Web services, persistence, and Enterprise Java Bean performance, respectively. These three chapters present in-depth coverage of the performance issues often observed in Java EE applications and provide suggested advice and/or solutions to common Java EE performance issues.

This book also includes two appendixes. Appendix A, “HotSpot VM Command Line Options of Interest,” lists HotSpot VM command line options that are referenced in the book and additional ones that may be of interest when tuning the HotSpot VM. For each command line option, a description of what the command line option does is given along with suggestions on when it is applicable to use them. Appendix B, “Profiling Tips and Tricks Example Source Code,” contains the source code used in Chapter 6’s examples for reducing lock contention, resizing Java collections, and increasing parallelism.
Without the help of so many people this book would not have been possible. First I have to thank my coauthor, Binu John, for his many contributions to this book. Binu wrote all the Java EE material in this book. He is a talented Java performance engineer and a great friend. I also want to thank Greg Doe, our editor, for his patience. It took almost three years to go from a first draft of the book’s chapter outline until we handed over a manuscript. Thank you to Paul Hohensee and Dave Keenan for their insight, encouragement, support, and thorough reviews. To Tony Printezis and Tom Rodriguez, thanks for your contributions on the details of the inner workings of the Java HotSpot VM garbage collectors and JIT compilers. And thanks to all the engineers on the Java HotSpot VM runtime team for having detailed documentation on how various pieces of the HotSpot VM fit together. To both James Gosling and Steve Wilson, thanks for making time to write a foreword. Thanks to Peter Kessler for his thorough review of Chapter 7, “Tuning the JVM, Step by Step.” Thanks to others who contributed to the quality of this book through their insight and reviews: Darryl Gove, Marty Itzkowitz, Geertjan Wielenga, Monica Beckwith, Alejandro Murillo, Jon Masamitsu, Y. Srinivas Ramkakrishna (aka Ramki), Chuck Rasbold, Kirk Pepperdine, Peter Gratzer, Jeanfrancois Arcand, Joe Bologna, Anders Åstrand, Henrik Löf, and Staffan Friberg. Thanks to Paul Ciciora for stating the obvious, “losing the race” (when the CMS garbage collector can’t free enough space to keep up with the young generation promotion rate). Also, thanks to Kirill Soshalskiy, Jerry Driscoll,
both of whom I have worked under during the time of writing this book, and to John Pampuch (Director of VM Technologies at Oracle) for their support. A very special thanks to my wife, Barb, and sons, Beau and Boyd, for putting up with a grumpy writer, especially during those times of “writer’s cramp.”

Binu John

This book has been possible only because of the vision, determination, and perseverance of my coauthor, Charlie Hunt. Not only did he write the sections relating to Java SE but also completed all the additional work necessary to get it ready for publication. I really enjoyed working with him and learned a great deal along the way. Thank you, Charlie. A special thanks goes to Rahul Biswas for providing content relating to EJB and Java persistence and also for his willingness to review multiple drafts and provide valuable feedback. I would like to thank several people who helped improve the quality of the content. Thank you to Scott Oaks and Kim Lichong for their encouragement and valuable insights into various aspects of Java EE performance; Bharath Mundlapudi, Jitendra Kotamraju, and Rama Pulavarthi for their in-depth knowledge of XML and Web services; Mitesh Meswani, Marina Vatkina, and Mahesh Kannan for their help with EJB and Java persistence; and Jeanfrancois Arcand for his explanations, blogs, and comments relating to Web container. I was fortunate to work for managers who were supportive of this work. Thanks to Madhu Konda, Senior Manager during my days at Sun Microsystems; Sef Kloninger, VP of Engineering, Infrastructure, and Operations; and Sridatta Viswanath, Senior VP of Engineering and Operations at Ning, Inc. A special thank you to my children, Rachael and Kevin, and my wonderful wife, Rita, for their support and encouragement during this process.
Charlie Hunt is the JVM Performance Lead Engineer at Oracle. He is responsible for improving the performance of the HotSpot Java Virtual Machine and Java SE class libraries. He has also been involved in improving the performance of both GlassFish Server Open Source Edition and Oracle WebLogic application servers. He wrote his first Java program in 1998 and joined Sun Microsystems, Inc., in 1999 as a Senior Java Architect. He has been working on improving the performance of Java and Java applications ever since. He is a regular speaker on the subject of Java performance at many worldwide conferences including the JavaOne Conference. Charlie holds a Master of Science in Computer Science from the Illinois Institute of Technology and a Bachelor of Science in Computer Science from Iowa State University.

Binu John is a Senior Performance Engineer at Ning, Inc., the world’s largest platform for creating social web sites. In his current role, he is focused on improving the performance and scalability of the Ning platform to support millions of page views per month. Before joining Ning, Binu spent more than a decade working on Java performance at Sun Microsystems, Inc. As a member of the Enterprise Java Performance team, he worked on several open source projects including the GlassFish Server Open Source Edition application server, the Open Source Enterprise Service Bus (Open ESB), and Open MQ JMS product. He has been an active contributor in the development of the various industry standard benchmarks such as SPECjms2007 and SPECjEnterprise2010, has published several performance white papers and has previously contributed to the XMLTest and WSTest benchmark projects at java.net. Binu holds Master of Science degrees in Biomedical Engineering and Computer Science from The University of Iowa.
This page intentionally left blank
Chapter 5, “Java Application Profiling,” presented the basic concepts of using a modern Java profiler such as the Oracle Solaris Studio Performance Analyzer and NetBeans Profiler. It did not, however, show any specific tips and tricks in using the tools to identify performance issues and approaches of how to resolve them. This is the purpose of this chapter. Its intention is to show how to use the tools to identify performance issues and take corrective actions to resolve them. This chapter looks at several of the more common types of performance issues the authors have observed through many years of working as Java performance engineers.

Performance Opportunities

Most Java performance opportunities fall into one or more of the following categories:

- **Using a more efficient algorithm.** The largest gains in the performance of an application come from the use of a more efficient algorithm. The use of a more efficient algorithm allows an application to execute with fewer CPU instructions, also known as a shorter path length. An application that executes with a shorter path length generally executes faster. Many different changes can lead to a shorter path length. At the highest level of the application, using a different data structure or modifying its implementation can lead to a shorter path length. Many applications that suffer application performance issues often use inappropriate data structures. There is no substitute for choosing the
proper data structure and algorithm. As profiles are analyzed, take notice of the data structures and the algorithms used. Optimal performance can be realized when the best data structures and algorithms are utilized.

- **Reduce lock contention.** Contending for access to a shared resource inhibits an application's capability to scale to a large number of software threads and across a large number of CPUs. Changes to an application that allow for less frequent lock contention and less duration of locking allow an application to scale better.

- **Generate more efficient code for a given algorithm.** Clocks per CPU instruction, usually referred to as CPI, for an application is a ratio of the number of CPU clock ticks used per CPU instruction. CPI is a measure of the efficiency of generated code that is produced by a compiler. A change in the application, JVM, or operating system that reduces the CPI for an application will realize an improvement in its performance since it takes advantage of better and more optimized generated code.

There is a subtle difference between path length, which is closely tied to the algorithm choice, and cycles per instruction, CPI, which is the notion of generating more efficient code. In the former, the objective is to produce the shortest sequence of CPU instructions based on the algorithm choice. The latter’s objective is to reduce the number of CPU clocks consumed per CPU instruction, that is, produce the most efficient code from a compiler. To illustrate with an example, suppose a CPU instruction results in a CPU cache miss, such as a load instruction. It may take several hundred CPU clock cycles for that load instruction to complete as a result of the CPU cache miss having to fetch data from memory rather than finding it in a CPU cache. However, if a prefetch instruction was inserted upstream in the sequence of instructions generated by a compiler to prefetch from memory the data being loaded by the load instruction, it is likely the number of clock cycles required to load the data will be less with the additional prefetch instruction since the prefetch can be done in parallel with other CPU instructions ahead of the load instruction. When the load instruction occurs, it can then find the data to be loaded in a CPU cache. However, the path length, the number of CPU instructions executed is longer as a result of the additional prefetch instruction. Therefore, it is possible to increase path length, yet make better use of available CPU cycles.

The following sections present several strategies to consider when analyzing a profile and looking for optimization opportunities. Generally, optimization opportunities for most applications fall into one of the general categories just described.

**System or Kernel CPU Usage**

Chapter 2, “Operating System Performance Monitoring,” suggests one of the statistics to monitor is system or kernel CPU utilization. If CPU clock cycles are spent executing operating system or kernel code, those are CPU clock cycles that cannot
be used to execute your application. Hence, a strategy to improve the performance of an application is to reduce the amount of time it spends consuming system or kernel CPU clock cycles. However, this strategy is not applicable in applications that spend little time executing system or kernel code. Monitoring the operating system for system or kernel CPU utilization provides the data as to whether it makes sense to employ this strategy.

The Oracle Solaris Performance Analyzer collects system or kernel CPU statistics as part of an application profile. This is done by selecting the View > Set Data Presentation menu in Performance Analyzer, choosing the Metrics tab, and setting the options to present system CPU utilization statistics, both inclusive or exclusive. Recall that inclusive metrics include not only the time spent in a given method, but also the time spent in methods it calls. In contrast, exclusive metrics report only the amount of time spent in a given method.

**Tip**

It can be useful to include both inclusive and exclusive metrics when first analyzing a profile. Looking at the inclusive metrics provides a sense of the path the application executes. Looking at the general path an application takes you may identify an opportunity for an alternative algorithm or approach that may offer better performance.

Figure 6-1 shows the Performance Analyzer’s Set Data Presentation form with options selected to present both inclusive and exclusive System CPU metrics. Also notice the options selected report both the raw time value and the percentage of System CPU time.

![Figure 6-1 Set system CPU data presentation](image)
After clicking on the OK button, the Performance Analyzer displays the profile’s System CPU inclusive and exclusive metrics in descending order. The arrow in the metric column header indicates how the data is presented and sorted. In Figure 6-2, the System CPU data is ordered by the exclusive metric (notice the arrow in the exclusive metric header and the icon indicating an exclusive metric).

Figure 6-2 shows a profile from an application that exhibits high system or kernel CPU utilization. You can see this application consumed about 33.5 seconds of System CPU in the java.io.FileOutputStream.write(int) method and about 11.6 seconds in a method called __write(), or about 65% and 22.5%, respectively. You can also get a sense of how significant the improvement can be realized by reducing the System CPU utilization of this application. The ideal situation for an application is to have 0% System CPU utilization. But for some applications that goal is difficult to achieve, especially if there is I/O involved, since I/O operations require a system call. In applications that require I/O, the goal is to reduce the frequency of making a system call. One approach to reduce the call frequency of an I/O system call is buffer the data so that larger chunks of data are read or written during I/O operations.

In the example shown in Figure 6-2, you can see the file write (output) operations are consuming a large amount of time as illustrated by the java.io.FileOutputStream.write(int) and __write() entries. To identify whether the write operations are buffered, you can use the Callers-Callees tab to walk up the call stack to see what methods are calling the FileOutputStream.write(int) method and the __write method. You walk up the call stack by selecting one of the callees from the upper panel and clicking the Set Center button. Figure 6-3 shows the Callers-Callees of the FileOutputStream.write(int) method.

The callers of FileOutputStream.write(int) are ExtOutputStream.write(int) and OutImpl.outc(int). 85.18% of the System CPU attributed to FileOutputStream.write(int) comes from its use in ExtOutputStream.write(int) and 14.82% of it from OutImpl.outc(int). A look at the implementation of ExtOutputStream.write(int) shows:
A look at the implementation of `super.write(b)` shows it is not a call to `FileOutputStream.write(int)`:

```java
public void write(int b) throws IOException {
    super.write(b);
    writer.write((byte)b);
}
```

But the `writer` field in `ExtOutputStream` is declared as a `FileOutputStream`:

```java
private FileOutputStream writer;
```

And it is initialized without any type of buffering:

```java
writer = new FileOutputStream(currentFileName);
```

currentFileName is a field declared as a String:

```java
private String currentFileName;
```

Hence, an optimization to be applied here is to buffer the data being written to `FileOutputStream` in `ExtOutputStream` using a `BufferedOutputStream`. This is done rather quickly and easily by chaining or wrapping the `FileOutputStream` in a `BufferedOutputStream` in an `ExtOutputStream`. Here is a quick listing of the changes required:
Then chain a `BufferedOutputStream` and `FileOutputStream` at initialization time:

```java
// Change FileOutputStream writer to a BufferedOutputStream
// private FileOutputStream writer;
private BufferedOutputStream writer;

// Initialize BufferedOutputStream
// writer = new FileOutputStream(currentFileName);
writer = new BufferedOutputStream(
    new FileOutputStream(currentFileName));
```

Writing to the `BufferedOutputStream`, instead of the `FileOutputStream`, in `ExtOutputStream.write(int b)` does not require any update since `BufferedOutputStream` has a `write()` method that buffers bytes written to it. This `ExtOutputStream.write(int b)` method is shown here:

```java
public void write(int b) throws IOException {
    super.write(b);
    // No update required here, 
    // automatically uses BufferedOutputStream.write()
    writer.write((byte)b);
}
```

The other uses of the `writer` field must be inspected to ensure the use of `BufferedOutputStream` operates as expected. In `ExtStreamOutput`, there are two additional uses of the `writer` field, one in a method called `reset()` and another in `checkResult()`. These two methods are as follows:

```java
public void reset() {
    super.reset();
    try {
        if (diffOutputStream != null) {
            diffOutputStream.flush();
            diffOutputStream.close();
            diffOutputStream = null;
        }
        if (writer != null) {
            writer.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```
The uses of writer as a BufferedOutputStream works as expected. It should be noted that the API specification for BufferedOutputStream.close() indicates it calls the BufferedOutputStream.flush() method and then calls the close() method of its underlying output stream, in this case the FileOutputStream.close() method. As a result, the FileOutputStream is not required to be explicitly closed, nor is the flush() method in ExtOutputStream.checkResult(int) required. A couple of additional enhancements worth consideration are

1. A BufferedOutputStream can also be allocated with an optional buffered size. The default buffer size, as of Java 6, is 8192. If the application you are profiling is writing a large number of bytes, you might consider specifying an explicit size larger than 8192. If you specify an explicit size, consider a size that is a multiple of the operating systems page size since operating systems efficiently fetch memory that are multiples of the operating system page size. On Oracle Solaris, the pagesize command with no arguments reports the default page size. On Linux, the default page size can be obtained using the getconf PAGESIZE command. Windows on x86 and x64 platforms default to a 4K (4096) page size.

2. Change the ExtOutputStream.writer field from an explicit
   BufferedOutputStream type to an OutputStream type, that is,
   OutputStream writer = new BufferedOutputStream(), instead of
   BufferedOutputStream writer = new BufferedOutputStream().
   This allows for additional flexibility in type of OutputStream, for example,
   ByteArrayOutputStream, DataOutputStream, FilterOutputStream,
   FileOutputStream, or BufferedOutputStream.

Looking back at Figure 6-3, a second method calls FileOutputStream.
write(int) called org.w3c.tidy.OutImpl.outc(int), which is a method from a third-party library used in the profiled application. To reduce the amount of system CPU utilization used in a third-party supplied method, the best approach is to file
a bug or enhancement request with the third-party library provider and include the information from the profile. If the source is accessible via an open source license and has acceptable license terms, you may consider further investigating and including additional information in the bug or enhancement request report.

After applying the changes identified in `ExtOutputStream`, using the `BufferedOutputStream` and its default constructor (not including the two additional enhancements just mentioned), and collecting a profile, the amount of system CPU utilization drops substantially. Comparing the profiles in Figure 6-4 to those in Figure 6-2, you can see the amount of inclusive system CPU time spent in `java.io.FileOutputStream` has dropped from 45.182 seconds to 6.655 seconds (exclusive system CPU time is the second column).

Executing this application workload outside the profiler in a performance testing environment prior to making the modifications reports it took this application 427 seconds to run to completion. In contrast, the modified version of the application workload that uses the `BufferOutputStream` in the same performance testing environment reports it runs to completion in 383 seconds. In other words, this application realized about a 10% improvement in its run to completion execution.

In addition, looking at the Callers-Callees tab for `java.io.FileOutputStream.write(int)`, only the call to `org.w3c.tidy.OutImpl.outc(int)` remains as a significant consumer of the `FileOutputStream.write(int)` method. The Callers-Callees of `FileOutputStream.write(int)` are shown in Figure 6-5.

![Figure 6-4 Reduced system CPU utilization](image1)

![Figure 6-5 Callers-Callees after changes](image2)
Comparing the Callers-Callees in Figure 6-5, after the changes to ExtStreamOutput, with the Callers-Callees in Figure 6-3, prior to the changes, you can see the amount of attributable time spent in org.w3c.tidy.OutImpl.outc(int) stays close to the same. This should not be a surprise since the changes made to ExtStreamOutput now use BufferedOutputStream. But recall that the BufferedOutputStream invokes a FileOutputStream method when any of the underlying buffer in the BufferedOutputStream becomes full, the BufferedOutputStream.flush() method is called, or when the BufferedOutputStream.close() method is called. If you look back at Figure 6-4 you see a FileOutputStream.writeBytes(byte[], int, int) method. This is the method that the BufferedOutputStream calls from ExtStreamOutput. Figure 6-6 shows the Callers-Callees tab for the FileOutputStream.writeBytes(byte[], int, int).

Selecting java.io.FileOutputStream.write(byte[], int, int) method from the upper Callee panel and clicking the Set Center button illustrates that BufferedOutputStream.flushBuffer() is its callee; see Figure 6-7.

![Figure 6-6 Callers-Callees of FileOutputStream.writeBytes(byte[], int, int)](image)

![Figure 6-7 Callers-Callees of FileOutputStream.writeBytes(byte[], int, int)](image)
Selecting the `BufferedOutputStream.flushBuffer()` method in the upper Callee panel and clicking the Set Center button shows the callee of `java.io.BufferedOutputStream.flushBuffer()` is `BufferedOutputStream.write(int)`. The Callers-Callees of `BufferedOutputStream.flushBuffer()` are shown in Figure 6-8.

Selecting the `BufferedOutputStream.write(int)` method in the upper Callee panel and clicking the Set Center button shows the callee of `java.io.BufferedOutputStream.write(int)` is `ExtOutputStream.write(int)`, the method that has been modified. The Callers-Callees of `BufferedOutputStream.write(int)` are shown in Figure 6-9.

As mentioned earlier, the next step in reducing System CPU utilization for this application requires a modification to a third-party library, a library that holds the implementation of `org.w3c.tidy.OutImpl.outc(int)`. It may be possible for the maintainers of the third-party library to implement a similar modification to `OutImpl.outc(int) as just described and implemented for ExtOutputStream.write(int)`. However, the performance improvement realized will likely not be as significant since the profile suggests there is more System CPU utilization attributed
to the call path of `ExtOutputStream.write(int)` than to `OutImpl.outc(int)`; refer to Figure 6-3 for attributable System CPU utilization on callers of `FileInputStream.write(int)`. In addition, looking at the amount of System CPU utilization consumed in `OutImpl.outc(int)`, about 6.6 seconds, compared to the total application runtime of 383 seconds is rather small, about 1.5%. Hence, a modification to reduce the amount of System CPU utilization spent in `OutImpl.outc(int)` would likely not yield more than 1% to 2% improvement.

**Tip**

Applications that perform network I/O can employ a similar, general approach to reduce system CPU utilization as that just described in this section. That is, buffer both the data in the input and output stream used to write and read the data.

An additional strategy to reduce system CPU utilization for applications performing large amounts of network I/O is utilizing Java NIO nonblocking data structures. Java NIO was introduced in Java 1.4.2 with many runtime performance improvements added in Java 5 and Java 6. Java NIO nonblocking data structures allow for the ability to read or write as much data as possible in a single call to a network I/O (read or write) operation. Remember that every network I/O call eventually results in the invocation of an operating system’s system call, which consumes system CPU utilization. The challenge with using Java NIO nonblocking data structures is it is more difficult to program than using blocking Java NIO or the older, more traditional Java SE blocking data structures such as `java.net.Socket`. In a Java NIO nonblocking output operation, you can write as many bytes as the operating system allows to be written. But you have to check the return value of the output operation to determine whether all the bytes you asked to be written have indeed been written. In a Java NIO nonblocking input operation, where you read as many bytes as are available, you have to check how many bytes have been read. You also have to implement some complex programming logic to deal with partially read protocol data units, or multiple protocol data units. That is, you may not be able to read enough bytes in a single read operation to construct a meaningful protocol data unit or message. In the case of blocking I/O, you simply wait until you generally read the specified number of bytes that constitute a full protocol data unit or message. Whether to migrate an application to utilize nonblocking network I/O operations should be decided upon by the application’s performance needs. If you want to take advantage of the additional performance promised by using nonblocking Java NIO, you should consider using a general Java NIO framework to make the migration easier. Several popular Java NIO frameworks are available such as Project Grizzly (https://grizzly.dev.java.net) and Apache MINA (http://mina.apache.org).
Another area where high System CPU utilization may show up is in applications experiencing heavy lock contention. Identifying lock contention in a profile and approaches to reduce lock contention are discussed in the next section.

**Lock Contention**

In early JVM releases, it was common to delegate Java monitor operations directly to operating system monitors, or mutex primitives. As a result, a Java application experiencing lock contention would exhibit high values of system CPU utilization since operating system mutex primitives involve system calls. In modern JVMs Java monitors are mostly implemented within the JVM in user code rather than immediately delegating them to operating system locking primitives. This means Java applications can exhibit lock contention yet not consume system CPU. Rather, these applications first consume user CPU utilization when attempting to acquire a lock. Only applications that experience severe lock contention may show high system CPU utilization since modern JVMs tend to delegate to operating system locking primitives as a last resort. A Java application running in a modern JVM that experiences lock contention tends to show symptoms of not scaling to a large number of application threads, CPU cores, or a large number of concurrent users. The challenge is finding the source of the lock contention, that is, where are those Java monitors in the source code and what can be done to reduce the lock contention.

Finding and isolating the location of highly contented Java monitors is one of the strengths of the Oracle Solaris Performance Analyzer. Once a profile has been collected with the Performance Analyzer, finding the highly contented locks is easy. The Performance Analyzer collects Java monitor and lock statistics as part of an application profile. Hence, you can ask the Performance Analyzer to present the Java methods in your application using Java monitors or locks.

**Tip**

You can also view locks used within the JVM with the Performance Analyzer, but that requires setting the presentation view mode to Machine Mode.

By selecting the View > Set Data Presentation menu in Performance Analyzer and choosing the Metrics tab, you can ask the Performance Analyzer to present lock statistics, both inclusive or exclusive. Remember that inclusive lock metrics include not only the lock time spent in a given method but also the lock time spent in methods
it calls. In contrast, exclusive metrics report only the amount of lock time spent in a given method.

Figure 6-10 shows the Performance Analyzer’s Set Data Presentation form with options selected to present both inclusive and exclusive lock information. Also notice the options selected report both the time value and the percentage spent locking.

After clicking OK, the Performance Analyzer displays the profile’s lock inclusive and exclusive metrics in descending order. The arrow in the metric column header indicates how the data is presented. In Figure 6-11, the lock data is ordered by the exclusive metric (notice the arrow in the exclusive metric header and note the icon indicating an exclusive metric).
The screenshot taken in Figure 6-11 is from a simple example program (complete source code for the remaining examples used in this chapter can be found in Appendix B, “Profiling Tips and Tricks Example Source Code”) that uses a java.util.HashMap as a data structure to hold 2 million fictitious tax payer records and performs updates to those records stored in the HashMap. Since this example is multithreaded and the operations performed against the HashMap include adding a new record, removing a new record, updating an existing record, and retrieving a record, the HashMap requires synchronized access, that is, the HashMap is allocated as a synchronized Map using the Collections.synchronizedMap() API. The following list provides more details as to what this example program does:

- Creates 2 million fictitious tax payer records and places them in an in-memory data store, a java.util.HashMap using a tax payer id as the HashMap key and the tax payer’s record as the value.
- Queries the underlying system for the number of available processors using the Java API Runtime.availableProcessors() to determine the number of simultaneous Java threads to execute concurrently.
- Uses the number returned from Runtime.availableProcessors() and creates that many java.util.concurrent.Callable objects to execute concurrently in an allocated java.util.concurrent.ExecutorService pool of Executors.
- All Executors are launched and tax payer records are retrieved, updated, removed, and added concurrently by the Executor threads in the HashMap. Since there is concurrent access to the HashMap through the actions of adding, removing, and updating records, HashMap access must be synchronized. The HashMap is synchronized using the Collections.synchronizedMap() wrapper API at HashMap creation time.

From the preceding description, it should be of little surprise this example program experiences lock contention when a large number of threads are trying to concurrently

Tip
Before blindly looking only at lock metrics in Performance Analyzer, an application should be exhibiting scalability symptoms. The classic scaling symptoms occur when executing an application on a system with a large number of CPUs, CPU cores, or hardware threads does not show an expected scaling in performance throughput relative to a system with a smaller number of CPUs, CPU cores, or hardware threads, or leaves CPU utilization unused. In other words, if an application is not showing scaling issues, then there is no need to investigate an application’s locking activity.
access the same synchronized HashMap. For example, when this program is run on a Sun SPARC Enterprise T5120 Server configured with an UltraSPARC T2 processor, which has 64 virtual processors (the same value as that returned by the Java API `Runtime.availableProcessors()`), the performance throughput reported by the program is about 615,000 operations per second. But only 8% CPU utilization is reported due to heavy lock contention. Oracle Solaris mpstat also reports a large number of voluntary thread context switches. In Chapter 2, the “Memory Utilization” section talks about high values of voluntary thread context switches being a potential indicator of high lock contention. In that section, it is said that the act of parking a thread and awaking a thread after being notified both result in an operating system voluntary context switch. Hence, an application experiencing heavy lock contention also exhibits a high number of voluntary context switches. In short, this application is exhibiting symptoms of lock contention.

Capturing a profile of this example program with the Performance Analyzer and viewing its lock statistics, as Figure 6-11 shows, confirms this program is experiencing heavy lock contention. The application is spending about 59% of the total lock time, about 14,000 seconds, performing a synchronized `HashMap.get()` operation. You can also see about 38% of the total lock time is spent in an entry labeled `<JVM-System>`. You can read more about this in the “Understanding JVM-System Locking” sidebar. You can also see the calls to the `put()` and `remove()` records in the synchronized HashMap as well.

Figure 6-12 shows the Callers-Callees of the `SynchronizedMap.get()` entry. It is indeed called by the `TaxPayerBailoutDBImpl.get()` method, and the `SynchronizedMap.get()` method calls a `HashMap.get()` method.

**Understanding JVM-System Locking**

A JVM-System entry in Performance Analyzer indicates time spent within the JVM internals. In the context of looking at lock contention statistics in Performance Analyzer, this is the amount or percentage of time spent in locks within the internals of the JVM. This may sound alarming when looking at the amount of time spent in the JVM-System in Figure 6-11.
Hence, this requires a little further explanation and clarification. Recall from Chapter 5 that switching from a Data Presentation Format of User mode to either Expert mode or Machine mode shows the internal operations of the JVM and puts them in the JVM-System entry seen in User mode. Also remember that switching to Expert mode or Machine mode also shows highly contended Java monitors as a form of a __lwp_mutex, __lwp_cond_wait, or __lwp_park type of entry and isolates the locking within Java APIs with those found within the JVM. Figure 6-13 shows the same profile but is switched from User mode to Expert mode in the Performance Analyzer.

Comparing Figure 6-11 to Figure 6-13 suggests the JVM-System entry has resolved into __lwp_condition_wait and __lwp_park operations. The sum of the __lwp_condition_wait and __lwp_park are close to what is reported for JVM-System in Figure 6-11. Your initial reaction may be the JVM is also experiencing lock contention. However, selecting the __lwp_cond_wait entry and selecting the Callers-Callees tab and walking up the call stack, the source of the locking activity associated with __lwp_cond_wait, in other words the locking activity associated with the JVM-System entry, is shown in Figure 6-14.

All five of the methods shown in Figure 6-14 are internal JVM methods. Notice that over 95% of the attributable lock time is spent in GCTaskManager::get_task(unsigned).

---

### Figure 6-13 Switching from User mode to Expert mode

### Figure 6-14 Traversing up the call stack of callers of __lwp_cond_wait
This method is part of the garbage collection subsystem of the Java HotSpot VM. This garbage collection method blocks and waits on a queue for work to do on behalf of the garbage collector subsystem. Each of the method names listed in Figure 6-14 represent areas of the Java HotSpot VM that may block and wait for some work to be placed on their respective work queue. For example, the \texttt{VMThread::loop()} method blocks on a queue for work to do on behalf of the Java HotSpot VM. You can think of the \texttt{VMThread} as the “kernel thread” of the Java HotSpot VM. The \texttt{CompilerBroker::compile_thread_loop()} method blocks and waits for work to do on behalf of the JIT compilation subsystem and so on. As a result, the entries reported as the JVM-System entry in User Mode can be ignored as being hot locks in this profile.

Continuing with the example program, the reaction from many Java developers when he or she observes the use of a synchronized \texttt{HashMap} or the use of a \texttt{java.util.Hashtable}, the predecessor to the synchronized \texttt{HashMap}, is to migrate to using a \texttt{java.util.concurrent.ConcurrentHashMap}. Following this practice and executing this program using a \texttt{ConcurrentHashMap} instead of a synchronized \texttt{HashMap} showed an increase of CPU utilization of 92%. In other words, the previous implementation that used a synchronized \texttt{HashMap} had a total CPU utilization of 8% while the \texttt{ConcurrentHashMap} implementation had 100% CPU utilization. In addition, the number of voluntary context switches dropped substantially from several thousand to less than 100. The reported number of operations per second performed with the \texttt{ConcurrentHashMap} implementation increased by a little over 2x to 1,315,000, up from 615,000 with the synchronized \texttt{HashMap}. However, seeing only a 2x performance improvement while utilizing 100% CPU utilization compared to just 8% CPU utilization is not quite what was expected.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Method & CPU Utilization & Context Switches \\
\hline
\texttt{Synchronized HashMap} & 8% & Several thousand \\
\texttt{ConcurrentHashMap} & 100% & Less than 100 \\
\hline
\end{tabular}
\end{table}

\textbf{Tip}

When performance testing, observing an unexpected result or observing a result that looks suspicious is a strong indication to investigate performance results and revisit testing methodology.

Capturing a profile and viewing the results with the Performance Analyzer is in order to investigate what happened. Figure 6-15 shows the hot methods as \texttt{java.util.Random.next(int)} and \texttt{java.util.concurrent.atomic.AtomicLong.compareAndSet(long, long)}.

Using the Callers-Callees tab to observe the callers of the \texttt{java.util.concurrent.atomic.AtomicLong.compareAndSet(long, log)} method shows \texttt{java.util.Random.next(int)} as the most frequent callee. Hence, the two hottest methods in the profile are in the same call stack; see Figure 6-16.

\citeindex{java.util.concurrent.ConcurrentHashMap} was introduced in the Java 5 SE class libraries and is available in Java 5 and later Java JDKs/JREs.
Chapter 6 • Java Application Profiling Tips and Tricks

Figure 6-17 shows the result of traversing further up the call stack of the callers of Random.next(int). Traversing upwards shows Random.next(int) is called by Random.nextInt(int), which is called by a TaxCallable.updateTaxPayer(long, TaxPayerRecord) method and six methods from
the BailoutMain class with the bulk of the attributable time spent in the TaxCallable.updateTaxPayer(long, TaxPayerRecord) method.

The implementation of TaxCallable.updateTaxPayer(long, TaxPayerRecord) is shown here:

```java
final private static Random generator = BailoutMain.random;
// these class fields initialized in TaxCallable constructor
final private TaxPayerBailoutDB db;
private String taxPayerId;
private long nullCounter;
private TaxPayerRecord updateTaxPayer(long iterations,
    TaxPayerRecord tpr) {
    if (iterations % 1001 == 0) {
        tpr = db.get(taxPayerId);
    } else {
        // update a TaxPayer's DB record
        tpr = db.get(taxPayerId);
        if (tpr != null) {
            long tax = generator.nextInt(10) + 15;
            tpr.taxPaid(tax);
        }
    }
    if (tpr == null) {
        nullCounter++;
    }
    return tpr;
}
```

The purpose of TaxCallable.updateTaxPayer(long, TaxPayerRecord) is to update a tax payer's record in a tax payer's database with a tax paid. The amount of tax paid is randomly generated between 15 and 25. This randomly generated tax is implemented with the line of code, long tax = generator.nextInt(10) + 15. generator is a class instance static Random that is assigned the value of BailoutMain.random which is declared in the BailoutMain class as final public static Random random = new Random(Thread.currentThread().getId()). In other words, the BailoutMain.random class instance field is shared across all instances and uses of BailoutMain and TaxCallable. The BailoutMain.random serves several purposes in this application. It generates random fictitious tax payer ids, names, addresses, social security numbers, city names and states which are populated in a tax payer database, a TaxPayerBailoutDB which uses a ConcurrentHashMap in this implementation variant as its storage container. BailoutMain.random is also used, as described earlier, to generate a random tax for a given tax payer.
Since there are multiple instances of `TaxCallable` executing simultaneously in this application, the static `TaxCallable.generator` field is shared across all `TaxCallable` instances. Each of the `TaxCallable` instances execute in different threads, each sharing the same `TaxCallable.generator` field and updating the same tax payer database.

This means all threads executing `TaxCallable.updateTaxPayer(long, TaxPayerRecord)` trying to update the tax payer database must access the same `Random` object instance concurrently. Since the Java HotSpot JDK distributes the Java SE class library source code in a file called `src.zip`, it is possible to view the implementation of `java.util.Random`. A `src.zip` file is found in the JDK root installation directory. Within the `src.zip` file, you can find the `java.util.Random.java` source code. The implementation of the `Random.next(int)` method follows (remember from the Figure 6-17 that `Random.next(int)` is the method that calls the hot method `java.util.concurrent.atomic.AtomicLong.compareAndSet(int,int)`).

```java
private final AtomicLong seed;
private final static long multiplier = 0x5DEECE66DL;
private final static long addend = 0xBL;
private final static long mask = (1L << 48) – 1;
protected int next(int bits) {
    long oldseed, nextseed;
    AtomicLong seed = this.seed;
    do {
        oldseed = seed.get();
        nextseed = (oldseed * multiplier + addend) & mask;
    } while (!seed.compareAndSet(oldseed, nextseed));
    return (int)(nextseed >>> (48 - bits));
}
```

In `Random.next(int)`, there is a `do/while` loop that performs an `AtomicLong.compareAndSet(int,int)` on the old seed and the new seed (this statement is highlighted in the preceding code example in bold). `AtomicLong` is an atomic concurrent data structure. Atomic and concurrent data structures were two of the features added to Java 5. Atomic and concurrent data structures typically rely on some form of a “compare and set” or “compare and swap” type of operation, also commonly referred to as a CAS, pronounced “kazz”.

CAS operations are typically supported through one or more specialized CPU instructions. A CAS operation uses three operands: a memory location, an old value, and a new value. Here is a brief description of how a typical CAS operation works. A CPU atomically updates a memory location (an atomic variable) if the value at that location matches an expected old value. If that property fails to hold, no changes are made. To be more explicit, if the value at that memory location prior to the
CAS operation matches a supplied expected old value, then the memory location is updated with the new value. Some CAS operations return a boolean value indicating whether the memory location was updated with the new value, which means the old value matched the contents of what was found in the memory location. If the old value does not match the contents of the memory location, the memory location is not updated and false is returned.

It is this latter boolean form the `AtomicLong.compareAndSet(int, int)` method uses. Looking at the preceding implementation of the `Random.next(int)` method, the condition in the do/while loop does not exit until the `AtomicLong CAS` operation atomically and successfully sets the `AtomicLong` value to the `nextseed` value. This only occurs if the current value at the `AtomicLong`’s memory location has a value of the `oldseed`. If a large number of threads happen to be executing on the same `Random` object instance and calling `Random.next(int)`, there is a high probability the `AtomicLong.compareAndSet(int, int)` CAS operation will return false since many threads will observe a different `oldseed` value at the `AtomicLong`’s value memory location. As a result, many CPU cycles may be spent spinning in the do/while loop found in `Random.next(int)`. This is what the Performance Analyzer profile suggests is the case.

A solution to this problem is to have each thread have its own `Random` object instance so that each thread is no longer trying to update the same `AtomicLong`’s memory location at the same time. For this program, its functionality does not change with each thread having its own thread local `Random` object instance. This change can be accomplished rather easily by using a `java.lang.ThreadLocal`. For example, in `BailoutMain`, instead of using a static `Random` object, a static `ThreadLocal<Random>` could be used as follows:

```java
// Old implementation using a static Random
//final public static Random random =
//    new Random(Thread.currentThread().getId());

// Replaced with a new ThreadLocal<Random>
final public static ThreadLocal<Random> threadLocalRandom =
    new ThreadLocal<Random>() {
        @Override
        protected Random initialValue() {
            return new Random(Thread.currentThread().getId());
        }
    };
```

Then any reference to or use of `BailoutMain.random` should be replaced with `threadLocalRandom.get()`. A `threadLocalRandom.get()` retrieves a unique `Random` object instance for each thread executing code that used to use `BailoutMain.random`. Making this change allows the `AtomicLong`’s CAS operation
in `Random.next(int)` to succeed quickly since no other thread is sharing the same `Random` object instance. In short, the `do/while` in `Random.next(int)` completes on its first loop iteration execution.

After replacing the `java.util.Random` in `BailoutMain` with a `ThreadLocal<Random>` and re-running the program, there is a remarkable improvement performance. When using the static `Random`, the program reported about 1,315,000 operations per second being executed. With the static `ThreadLocal<Random>` the program reports a little over 32,000,000 operations per second being executed. 32,000,000 operations per second is almost 25x more operations per second higher than the version using the static `Random` object instance. And it is more than 50x faster than the synchronized `HashMap` implementation, which reported 615,000 operations per second.

A question that may be worthy of asking is whether the program that used the synchronized `HashMap`, the initial implementation, could realize a performance improvement by applying the `ThreadLocal<Random>` change. After applying this change, the version of the program that used a synchronized `HashMap` showed little performance improvement, nor did its CPU utilization improve. Its performance improved slightly from 615,000 operations per second to about 620,000 operations per second. This should not be too much of a surprise. Looking back at the profile, the method having the hot lock in the initial version, the one that used a synchronized `HashMap`, and shown in Figure 6-11 and Figure 6-12, reveals the hot lock is on the synchronized `HashMap.get()` method. In other words, the synchronized `HashMap.get()` lock is masking the `Random.next(int)` CAS issue uncovered in the first implementation that used `ConcurrentHashMap`.

One of the lessons to be learned here is that atomic and concurrent data structures may not be the holy grail. Atomic and concurrent data structures rely on a CAS operation, which in general employs a form of synchronization. Situations of high contention around an atomic variable can lead to poor performance or scalability even though a concurrent or lock-free data structure is being used.

Many atomic and concurrent data structures are available in Java SE. They are good choices to use when the need for them exists. But when such a data structure is not available, an alternative is to identify a way to design the application such that the frequency at which multiple threads access the same data and the scope of the data that is accessed is minimized. In other words, try to design the application to minimize the span, size, or amount of data to be synchronized. To illustrate with an example, suppose there was no known implementation of a `ConcurrentHashMap` available in Java, that is, only the synchronized `HashMap` data structure was available. The alternative approach just described suggests the idea to divide the tax payer database into multiple `HashMaps` to lessen the amount or scope of data that needs to be locked. One approach might be to consider a `HashMap` for tax payers in each state. In such an approach, there would be two levels of `Maps`. The first
level Map would find one of the 50 state Maps. Since the first level Map will always contain a mapping of the 50 states, no elements need to be added to it or removed from it. Hence, the first level Map requires no synchronization. However, the second level state maps require synchronized access per state Map since tax payer records can be added, removed, and updated. In other words, the tax payer database would look something like the following:

```java
public class TaxPayerBailoutDbImpl implements TaxPayerBailoutDB {
    private final Map<String, Map<String, TaxPayerRecord>> db;
    public TaxPayerBailoutDbImpl(int dbSize, int states) {
        db = new HashMap<String, Map<String, TaxPayerRecord>>(states);
        for (int i = 0; i < states; i++) {
            Map<String, TaxPayerRecord> map =
                    Collections.synchronizedMap(
                            new HashMap<String, TaxPayerRecord>(dbSize/states));
            db.put(BailoutMain.states[i], map);
        }
    }
    ...
}
```

In the preceding source code listing you can see the first level Map is allocated as a HashMap in the line `db = new HashMap<String, Map<String, TaxPayerRecord>>(dbSize)` and the second level Map, one for each of the 50 states is allocated as a synchronized HashMap in the for loop:

```java
for (int i = 0; i < states; i++) {
    Map<String, TaxPayerRecord> map =
            Collections.synchronizedMap(
                    new HashMap<String, TaxPayerRecord>(dbSize/states));
    db.put(BailoutMain.states[i], map);
}
```

Modifying this example program with the partitioning approach described here shows about 12,000,000 operations per second being performed and a CPU utilization of about 50%. The number of operations per second is not nearly as good as the 32,000,000 observed with a ConcurrentHashMap. But it is a rather large improvement over the single large synchronized HashMap, which yielded about 620,000 operations per second. Given there is unused CPU utilization, it is likely further partitioning could improve the operations per second in this partitioning approach. In general, with the partitioning approach, you trade-off additional CPU cycles for additional path length, that is, more CPU instructions, to reduce the scope of the data that is being locked where CPU cycles are lost blocking and waiting to acquire a lock.
Volatile Usage

JSR-133, which was introduced in Java 5, addressed many issues in the Java Memory Model. This is well documented at http://jcp.org/jsr/detail?id=133 by the JSR-133 Expert Group with further material at http://www.cs.umd.edu/~pugh/java/memoryModel/ maintained by Dr. Bill Pugh. One of the issues addressed with JSR-133 is the use of the Java keyword volatile. Fields in Java objects that are declared as volatile are usually used to communicate state information among threads. The inclusion of JSR-133 into Java 5 and later Java revisions, ensures that a thread that reads a volatile field in an object is guaranteed to have the value that was last written to that volatile field, regardless of the thread that is doing read or write, or the location of where those two threads are executing, that is, different CPU sockets, or CPU cores. The use of a volatile field does limit optimizations a modern JVM’s JIT compiler can perform on such a field. For example, a volatile field must adhere to certain instruction ordering. In short, a volatile field’s value must be kept in sync across all application threads and CPU caches. For instance, when a volatile field’s value is changed by one thread, whose field might be sitting in a CPU cache, any other thread that might have a copy of that volatile field in its CPU cache, a different CPU cache than the other thread that performed the change, must have its CPU cache updated before its thread reads that volatile field found in its local CPU cache, or it must be instructed to retrieve the updated volatile field’s value from memory. To ensure CPU caches are updated, that is, kept in sync, in the presence of volatile fields, a CPU instruction, a memory barrier, often called a membar or fence, is emitted to update CPU caches with a change in a volatile field’s value.

In a highly performance sensitive application having multiple CPU caches, frequent updates to volatile fields can be a performance issue. However, in practice, few Java applications rely on frequent updates to volatile fields. But there are always exceptions to the rule. If you keep in mind that frequent updates, changes, or writes to a volatile field have the potential to be a performance issue (i.e., reads of a volatile field are okay, not a cause for performance concern), you will likely not experience performance issues when using volatile fields.

A profiler, such as the Performance Analyzer, that has the capability to gather CPU cache misses and associate them to Java object field access can help isolate whether the use of a volatile field is a performance issue. If you observe a high number of CPU cache misses on a volatile field and the source code suggests frequent writes to that volatile field, you have an application that is experiencing performance issues as a result of its usage of volatile. The solution to such a situation is to identify ways in which less frequent writes are performed to the volatile field, or refactor the application in a way to avoid the use of the volatile field. Never remove the use of a volatile field if it breaks program correctness or introduces a potential race condition. It is much better to have an underperforming application than it is to have an incorrect implementation, or one that has the potential for a race condition.
Data Structure Resizing

Java applications tend to make high use of Java SE’s StringBuilder or StringBuffer for assembling Strings and also make high use of Java objects that act as containers of data such as the Java SE Collections classes. Both StringBuilder and StringBuffer use an underlying char[] for their data storage. As elements are added to a StringBuilder or StringBuffer, the underlying char[] data storage, may be subject to resizing. As a result of resizing, a new larger char[] array is allocated, the char elements in the old char[] are copied into the new larger char[] array, and the old char[] discarded, that is, available for garbage collection. Similar resizing can also occur in Java SE Collections classes that use an array for their underlying data store.

This section explores ways to identify data structure resizing, in particular StringBuilder, StringBuffer, and Java SE Collections classes resizing.

StringBuilder/StringBuffer Resizing

When a StringBuilder or StringBuffer becomes large enough to exceed the underlying data storage capacity, a new char array of a larger size, 2x larger in the OpenJDK StringBuilder and StringBuffer implementation (used by Java Hot-Spot Java 6 JDK/JRE), is allocated, the old char array elements are copied into the new char array, and the old char array is discarded. A version of the implementation used by StringBuilder and StringBuffer follows:

```java
char[] value;
int count;

public AbstractStringBuilder append(String str) {
    if (str == null) str = "null";
    int len = str.length();
    if (len == 0) return this;
    int newCount = count + len;
    if (newCount > value.length)
        expandCapacity(newCount);
    str.getChars(0, len, value, count);
    count = newCount;
    return this;
}

void expandCapacity(int minimumCapacity) {
    int newCapacity = (value.length + 1) * 2;
    if (newCapacity < 0) {
        newCapacity = Integer.MAX_VALUE;
    } else if (minimumCapacity > newCapacity) {
        newCapacity = minimumCapacity;
    }
    value = Arrays.copyOf(value, newCapacity);
}
```
Continuing with the fictitious tax payer program example from the previous section (full listing of the source code used in this section can be found in Appendix B in the section “First Resizing Variant”), StringBuilder objects are used to assemble random Strings representing tax payer names, addresses, cities, states, social security numbers, and a tax payer id. It also uses the no argument StringBuilder constructor. Hence, the program is likely to be subject to StringBuilder’s underlying char[] being resized. A capture of a memory or heap profile with a profiler such as NetBeans Profiler confirms that is the case. Figure 6-18 shows a heap profile from NetBeans Profiler.

In Figure 6-18, you can see that char[], StringBuilder, and String are the most highly allocated objects and also have the largest amount of live objects. In the NetBeans Profiler, selecting and right-clicking on the char[] class name in the far left column as shown in Figure 6-19 shows the allocation stack traces for all char[] objects.

In the char[] stack allocation traces, shown in Figure 6-20, you can see an entry for java.lang.AbstractStringBuilder.expandCapacity(int), which is
called from `AbstractStringBuilder.append(char)` and `AbstractStringBuilder.append(String)` methods. The `expandCapacity(int)` method calls `java.util.Arrays.copyOf(char[], int)`. Looking back at the previous source code listing, you can see where `AbstractStringBuilder.append(String str)` calls `expandCapacity(int)` and calls `Arrays.copyOf(char[] int)`. You can also see from Figure 6-20, over 11% of the current live `char[]` objects are from resized `StringBuilder char[]`. In addition, there are a total of 2,926,048 `char[]` objects that have been allocated, and of those, 390,988 `char[]` allocations occurred as a result of `StringBuilder char[]` resizing. In other words, about 13% (390,988/2,926,048) of all `char[]` allocations are coming from resized `StringBuilder char[]`s. Eliminating these `char[]` allocations from resizing improves the performance of this program by saving the CPU instructions needed to perform the new `char[]` allocation, copying the characters from the old `char[]` into the new `char[]`, and the CPU instructions required to garbage collect the old discarded `char[]`.

In the Java HotSpot JDK/JRE distributions, both the `StringBuilder` and `StringBuffer` offer no argument constructors that use a default size of 16 for their underlying `char` array data storage. These no argument constructors are being used in this program. This can be seen in the profile by expanding the `java.lang.AbstractStringBuilder.<init>(int)` entry seen in Figure 6-20. The expansion of the `java.lang.AbstractStringBuilder.<init>(int)` entry, shown in Figure 6-21, shows it is called by a no argument `StringBuilder` constructor.

In practice, few `StringBuilder` or `StringBuffer` object instances result in having consumed 16 or fewer `char` array elements; 16 is the default size used with the no argument `StringBuilder` or `StringBuffer` constructor. To avoid `StringBuilder` and `StringBuffer` resizing, use the explicit size `StringBuilder` or `StringBuffer` constructor.

A modification to the example program follows, which now uses explicit sizes for constructing `StringBuilder` objects. A full listing of the modified version can be found in Appendix B in the section “Second Resizing Variant.”

Recent optimizations in Java 6 update releases of the Java HotSpot VM analyze the usage of `StringBuilder` and `StringBuffer` and attempt to determine the

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Allocation Call Tree</th>
<th>Live Bytes...</th>
<th>Live Objects</th>
<th>Allocated Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>char()</td>
<td>↓</td>
<td></td>
<td>2,460,612</td>
<td>2,926,048</td>
</tr>
<tr>
<td>java.util.Arrays.copyOfRange</td>
<td>char[], int, int</td>
<td></td>
<td>1,268,072</td>
<td>1,268,073</td>
</tr>
<tr>
<td>java.lang.AbstractStringBuilder.&lt;init&gt;(int)</td>
<td></td>
<td></td>
<td>911,056</td>
<td>1,266,982</td>
</tr>
<tr>
<td>java.lang.StringBuilder.&lt;init&gt;()</td>
<td></td>
<td></td>
<td>911,055</td>
<td>1,266,981</td>
</tr>
<tr>
<td>java.lang.StringBuffer.&lt;init&gt;(int)</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

![Figure 6-21 Uses of StringBuilder default constructor](image)
optimal char array size to use for a given StringBuilder or StringBuffer object allocation as means to reduce unnecessary char[] object allocations resulting from StringBuilder or StringBuffer expansion.

Measuring the performance impact after addressing StringBuilder and StringBuffer resizing will be done in combination with addressing any Java Collection classes resizing, the topic of the next section.

Java Collections Resizing

The addition of the Java Collections to Java SE offered an enormous boost to developer productivity by providing containers with interfaces allowing the ability to easily switch between alternative concrete implementations. For example, the List interface offers an ArrayList and LinkedList concrete implementation.

Java Collections Definition

As of Java 6, there were 14 interfaces in the Java SE Collections:

Collection, Set, List, SortedSet, NavigableSet, Queue, Deque, BlockingQueue, BlockingDeque, Map, SortedMap, NavigableMap, ConcurrentMap, and ConcurrentNavigableMap
The following is a listing of the most common concrete implementations of the Java SE Collections:

HashMap, HashSet, TreeSet, LinkedHashSet, ArrayList, ArrayDeque, LinkedList, PriorityQueue, TreeMap, LinkedHashMap, Vector, Hashtable, ConcurrentLinkedQueue, LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue, DelayQueue, SynchronousQueue, LinkedBlockingDeque, ConcurrentHashMap, ConcurrentSkipListSet, ConcurrentSkipListMap, WeakHashMap, IdentityHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet, EnumSet, and EnumMap

Some of the Collections’ concrete implementations are subject to potential expensive resizing as the number of elements added to the Collection grows such as ArrayList, Vector, HashMap, and ConcurrentHashMap since their underlying data store is an array. Other Collections such as LinkedList or TreeMap often use one or more object references between the elements stored to chain together the elements managed by the Collection. The former of these, those that use an array for the Collection’s underlying data store, can be subject to performance issues when the underlying data store is resized due to the Collection growing in the number of elements it holds. Although these Collections classes have constructors that take an optional size argument, these constructors are often not used, or the size provided in an application program is not optimal for the Collection’s use.

Tip

It is possible that there exists concrete implementations of Java Collections classes, such as LinkedList and TreeMap, that use arrays as underlying data storage. Those concrete implementations may also be subject to resizing. Collecting a heap profile and looking at collection resizing will show which Java Collections classes are resizing.

As is the case with StringBuilder or StringBuffer, resizing of a Java Collections class that uses an array as its data storage requires additional CPU cycles to allocate a new array, copy the old elements from the old array, and at some point in the future garbage collect the old array. In addition, the resizing can also impact Collection’s field access time, the time it takes to dereference a field, because a new underlying data store, again typically an array, for the Collection’s underlying data store may be allocated in a location in the JVM heap away from the object references stored within the data store and the other fields of the Collection. After a Collection resize occurs, it is possible an access to its resized field can result in CPU cache misses due to the way a modern JVM allocates objects in memory, in particular how those objects are laid out in memory. The way objects and their fields are laid out in memory can vary between JVM implementations. Generally, however, since
an object and its fields tend to be referenced frequently together, an object and its fields laid out in memory within close proximity generally reduce CPU cache misses. Hence, the impact of Collections resizing (this also applies to StringBuffer and StringBuilder resizing) may extend beyond the additional CPU instructions spent to do the resizing and the additional overhead put on the JVM’s memory manager to having a lingering higher field access time due to a change in the layout of the Collection’s fields in memory relative the Collection object instance.

The approach to identifying Java Collections resizing is similar to what was described earlier for identifying StringBuilder and StringBuffer resizing, collecting heap or memory profile with a profiler such as NetBeans Profiler. Looking at the source code for the Java Collection classes helps identify the method names that perform the resizing.

Continuing with the fictitious tax payer program, the program variant in which tax payer records were populated into multiple HashMaps using a tax payer’s state of residence as a key into a second HashMap where a tax payer’s id is used as an index is a good example of where Collections resizing can occur. A full source code listing from this variant can be found in Appendix B in the section “First Resizing Variant.” The source code, found in TaxPayerBailoutDbImpl.java, that allocates the HashMaps follows:

```java
private final Map<String, Map<String, TaxPayerRecord>> db;

public TaxPayerBailoutDbImpl(int numberOfStates) {
    db = new HashMap<String, Map<String, TaxPayerRecord>>() {
        for (int i = 0; i < numberOfStates; i++) {
            Map<String, TaxPayerRecord> map = Collections.synchronizedMap(
                new HashMap<String, TaxPayerRecord>());
            db.put(BailoutMain.states[i], map);
        }
    }
}
```

Here you can see the HashMaps are using a HashMap constructor that takes no arguments. As a result, the HashMap relies on a default size for its underlying mapping array. The following is a portion of OpenJDK’s HashMap.java source code that shows the default size chosen for a HashMap’s underlying data storage.

```java
static final int DEFAULT_INITIAL_CAPACITY = 16;
static final float DEFAULT_LOAD_FACTOR = 0.75f;

public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR;
    threshold = (int)(DEFAULT_INITIAL_CAPACITY * DEFAULT_LOAD_FACTOR);
    table = new Entry[DEFAULT_INITIAL_CAPACITY];
    init();
}
void init() {
}
```
Two factors decide when the data storage for a HashMap is resized: the capacity of the data storage and the load factor. The capacity is the size of the underlying data storage. That's the HashMap.Entry[]’s size. And the load factor is a measure of how full the HashMap is allowed to reach before the HashMap’s data storage, the Entry[], is resized. A HashMap resize results in a new Entry[] being allocated, twice as large as the previous Entry[], the entries in the Entry[] are rehashed and put in the Entry[]. The CPU instructions required to resize a HashMap are greater than what is required by StringBuilder or StringBuffer resizing due to the rehashing of the Entry[] elements.

In Figure 6-18, you can see a row for java.util.HashMap$Entry[]. For this entry you can see there are 67 allocated objects, and 37 of them are live at the time of the profile snapshot. This suggests that 37/67, about 55%, are still live. That also suggests 45% of those Entry[] objects that had been allocated have been garbage collected. In other words, the HashMaps are experiencing resizing. Notice that the total bytes consumed by HashMap.Entry[] objects is much less than those consumed by char[] objects. This suggests the impact of eliding the HashMap resizing is likely to be less than the impact realized from eliding the StringBuilder resizing.

Figure 6-22 shows the allocation stack traces for HashMap.Entry[]. Here you can see some of those HashMap.Entry[] allocations result from a HashMap.resize(int) method call. In addition, you can see the no argument HashMap constructor is being used, which also allocates a HashMap.Entry[].

Since this example program populates 50 different HashMaps with a total of 2,000,000 fictitious records, each of those 50 HashMaps hold about 2,000,000 / 50 = 40,000 records. Obviously, 40,000 is much greater than the default size of 16 used by the no argument HashMap constructor. Using the default load factor of .75, and the fact that each of the 50 HashMap holds 40,000 records, you can determine a size for the HashMaps so they will not resize (40,000 / .75 = ~ 53,334). Or simply passing the total number of records to store divided by the number of states, divided by the default load factor, i.e., (2,000,000 / 50) / .75, to the HashMap constructor that holds the records. Following is the modified source code for TaxPayerBailoutDbImpl.java that elides HashMap resizing:

```
Figure 6-22 HashMap.Entry[] allocation stack traces
```
In this example program, both `StringBuilder` and `HashMap` resizing occur during the initialization phase of the program, the phase of the program that populates a `Map` of `Maps` with fictitious, randomly generated tax payer records. Hence, to measure the performance impact of eliding the `StringBuilder` and `HashMap` resizing, the initialization phase of this program has been instrumented with a time stamp at the beginning of the program and after the `Map` of `Maps` has been populated. A modified version of this example program, one that uses the no argument `HashMap` constructor, calculates and reports the time it takes to populate the `HashMap`s with 2,000,000 records, can be found in Appendix B in the section “First Resizing Variant.”

When this variant of the program is run on a Sun SPARC Enterprise T5120 Server configured with 64 virtual processors (the same value as that returned by the Java API `Runtime.availableProcessors()`), the amount of time it takes to complete the initialization phase is 48.286 seconds.

### Tip

Since the populating of records is single threaded and the Sun SPARC Enterprise T5120 Server has a 1.2GHz clock rate, a processor with a smaller number of cores with a higher clock rate will likely report a shorter duration time needed to populate the 2,000,000 records in the `HashMap`s.

Updating this program variant with the changes described in this section to address both `StringBuilder` and `HashMap` resizing and running on the same Ultra-SPARC T5120 system with the same JVM command line options reports it takes 46.019 seconds to complete its initialization phase. That’s about a 5% improvement in elapsed time. The source code for this variant can be found in Appendix B in the section “Second Resizing Variant.”
Applying the data resizing strategy reduces the application’s path length, the total number of CPU instructions required to execute the program, and potentially more efficient use of CPU cycles due to fewer possibilities of CPU cache misses as a result of frequently accessed data structure fields being laid out in memory next to each other.

You may have noticed that the initialization phase in this program is single threaded. But the system it is being executed on has a CPU that is multicore and multithreaded per core. The Sun SPARC Enterprise T5120 Server this program is executing on has 8 cores, and 8 hardware threads per core. It is a chip multithreading type of CPU chip, CMT for short. In other words, 8 cores and 8 hardware threads per core means it has 64 virtual processors. That also means the Java API, System.availableProcessors(), returns a value of 64. A next step to improve the performance of the initialization phase of this program is to refactor it to utilize all of those 64 virtual processors. This is the topic of the next section.

Increasing Parallelism

Modern CPU architectures have brought multiple cores and multiple hardware execution threads to developer desktops. This means there are more CPU resources available to do additional work. However, to take advantage of those additional CPU resources, programs executed on them must be able to do work in parallel. In other words, those programs need to be constructed or designed in a multithreaded manner to take advantage of the additional hardware threads.

Java applications that are single threaded cannot take advantage of additional hardware threads on modern CPU architectures. Those applications must be refactored to be multithreaded to do their work in parallel. In addition, many Java applications have single-threaded phases, or operations, especially initialization or startup phases. Therefore, many Java applications can improve initialization or startup performance by doing tasks in parallel, that is, making use of multiple threads at the same time.

The example program used in the previous sections “Lock Contention” and “Data Structure Resizing” has a single-threaded initialization phase where random fictitious tax payer records are created and added to a Java Map. This single-threaded initialization phase could be refactored to being multithreaded. The single-threaded form, as it was run in the “Lock Contention” and “Data Structure Resizing” sections, when run on the same Sun SPARC Enterprise T5120 Server, reports it takes about 45 to 48 seconds for the initialization phase to complete. Since there are 64 virtual processors on an a Sun SPARC Enterprise T5120 Server, 63 of those 64 virtual processors are idle doing little or no work during the initialization phase. Therefore, if the initialization phase could be refactored to utilize those additional 63 virtual processors, the elapsed time it takes to execute the initialization phase should be significantly less.
The key to being able to refactor single-threaded phases of a program to be multi-threaded is constrained by the program’s logic. If there is a loop of execution involved, and much of the work performed within that loop is independent of what happens within each loop iteration, it may be a good candidate to be refactored into a multithreaded version. In the case of the fictitious tax payer program, Map records are added to a ConcurrentMap. Since a ConcurrentMap can handle multiple threads adding records to it and the records can be created independently of each other, the work performed in the single-threaded loop can be broken up and spread among multiple threads. With a Sun SPARC Enterprise T5120 Server that has 64 virtual processors, the work that is being done in the single-threaded loop could be spread across those 64 virtual processors.

Here is the core part of the single-threaded loop logic (full implementation can be found in Appendix B in the section “Increasing Parallelism Single-Threaded Implementation”):

```java
// allocate the database
TaxPayerBailoutDB db = new TaxPayerBailoutDbImpl(dbSize);
// allocate list to hold tax payer names
List<String>[] taxPayerList = new ArrayList[numberOfThreads];
for (int i = 0; i < numberOfThreads; i++) {
    taxPayerList[i] = new ArrayList<String>(taxPayerListSize);
}
// populate the database and tax payer list with random records
populateDatabase(db, taxPayerList, dbSize);
...
private static void populateDatabase(TaxPayerBailoutDB db,
        List<String>[] taxPayerIdList,
        int dbSize) {
    for (int i = 0; i < dbSize; i++) {
        // make random tax payer id and record
        String key = getRandomTaxPayerId();
        TaxPayerRecord tpr = makeTaxPayerRecord();
        // add tax payer id & record to database
        db.add(key, tpr);
        // add tax payer id to to tax payer list
        int index = i % taxPayerIdList.length;
        taxPayerIdList[index].add(key);
    }
}
```

The core part of refactoring the for/loop to be multithreaded results in creating a Runnable, or Callable, along with an ExecutorService to execute the Runnables or Callables in addition to ensuring the implementation of a TaxPayerBailoutDB and taxPayerIdList are thread safe. That is, the data they hold will not be corrupted as a result of having multiple threads writing data to them simultaneously. Following are segments of source code that contain the most relevant parts to the multithreaded refactoring (full implementation can be found in Appendix B in the section “Increasing Parallelism Multithreaded Implementation”):
// allocate the database
TaxPayerBailoutDB db = new TaxPayerBailoutDbImpl(dbSize);
List<String>[] taxPayerList = new List[numberOfThreads];
for (int i = 0; i < numberOfThreads; i++) {
    taxPayerList[i] =
        Collections.synchronizedList(
            new ArrayList<String>(taxPayerListSize));
}

// create a pool of executors to execute some Callables
ExecutorService pool = Executors.newFixedThreadPool(numberOfThreads);
Callable<DbInitializerFuture>[] dbCallables =
    new DbInitializer[numberOfThreads];
for (int i = 0; i < dbCallables.length; i++) {
    dbCallables[i] =
        new DbInitializer(db, taxPayerList, dbSize/numberOfThreads);
}

// start all db initializer threads running
Set<Future<DbInitializerFuture>> dbSet =
    new HashSet<Future<DbInitializerFuture>>();
for (int i = 0; i < dbCallables.length; i++) {
    Callable<DbInitializerFuture> callable = dbCallables[i];
    Future<DbInitializerFuture> future = pool.submit(callable);
    dbSet.add(future);
}

// A Callable that will execute multi-threaded db initialization
public class DbInitializer implements Callable<DbInitializerFuture> {
    private TaxPayerBailoutDB db;
    private List<String>[] taxPayerList;
    private int recordsToCreate;
    public DbInitializer(TaxPayerBailoutDB db,
                         List<String>[] taxPayerList,
                         int recordsToCreate) {
        this.db = db;
        this.taxPayerList = taxPayerList;
        this.recordsToCreate = recordsToCreate;
    }

    @Override
    public DbInitializerFuture call() throws Exception {
        return BailoutMain.populateDatabase(db, taxPayerList,
                                             recordsToCreate);
    }
}

static DbInitializerFuture populateDatabase(TaxPayerBailoutDB db,
                                             List<String>[] taxPayerIdList,
                                             int dbSize) {
    for (int i = 0; i < dbSize; i++) {
        String key = getRandomTaxPayerId();
        TaxPayerRecord tpr = makeTaxPayerRecord();
        db.add(key, tpr);
    }
}
After applying the refactoring to make the initialization phase multithreaded by dividing up the number of records to be added to the Map to run in 64 threads rather than 1 thread, the time it takes to perform the initialization phase drops from about 45 seconds to about 3 seconds on the Sun SPARC Enterprise T5120 Server. A higher clock rate dual or quad core desktop system may not observe as much of an improvement. For example, the author’s dual core desktop system realized about a 4 second improvement, 16 seconds down to about 12. The larger the number of virtual processors that additional parallel work can be spread among, the greater the potential performance improvement.

This simple example illustrates the potential benefit of being able to take advantage of additional virtual processors on a system that may be idle for some phase of an application by making that phase multithreaded.

**High CPU Utilization**

Sometimes an application simply cannot meet service level performance or scalability agreements even though performance efforts have reduced system CPU utilization, have addressed lock contention, and other optimization opportunities have been addressed. In such cases, doing an analysis of the program logic and the algorithms used is the direction to take. Method profilers such as the Performance Analyzer or NetBeans Profilers do a good job at collecting information about where in general an application spends most of its time.

The Performance Analyzer’s Call Tree tab is good at providing an application’s hottest use case by showing the call stack trees. This information can be leveraged to answer questions in a more abstract way, such as how long does it take the application to perform a unit of work, or perform a transaction, use case, and so on so long as the person looking at the profile has sufficient understanding of the implementation to be able to map a method entry point as the beginning of a unit of work, beginning of a transaction, use case, and so on. Being able to analyze the profile in this way provides the opportunity to step back, look at a higher level, and ask questions such as whether the algorithms and data structures being used are the most optimal or are there any alternative algorithms or data structures that might yield better performance or scalability. Often the tendency when analyzing profiles is to focus primarily on the methods that consume the most time in an exclusive metric kind of way, that is, focusing only on the contents of a method rather than at a higher level unit of work, transaction, use case, and so on.
Another useful strategy to employ when using the Performance Analyzer is to look at the Timeline view in the Performance Analyzer GUI (see Figure 6-23).

The Timeline view provides a listing of all threads, one in each row of the listing, that executed during the time when the profile was collected. At the top of the Timeline view is a timeline of seconds that have passed since the initiation of the collection of the profile. If the recording of the profiling data is enabled at Java application launch time, then the timeline contains data since the launching of the Java application. For each horizontal row, a thread within the application, a unique color is used to distinguish the method the application was executing in at the time of the sample. Selecting a thread, one of the rows within a colored area shows the call stack, their method names in the Call Stack for Selected Event panel, executing at the time the sample was taken. Figure 6-24 is a screenshot of the Call Stack for Selected Event panel for the selected thread, thread 1.2 in Figure 6-23.

Hence, by looking at the timeline, you can determine which threads are executing in the program at any particular point in time. This can be useful when looking for opportunities to multithread single-threaded phases or operations in an application. Figure 6-23, shows the single-threaded program variant presented in the “Increasing Parallelism” section earlier in the chapter. In Figure 6-23, you can see from the timeline, from about 16 seconds to a little past 64 seconds, the thread labeled as Thread 1.2, is the only thread that appears to be executing. The timeline
in Figure 6-23, suggests the program may be executing its initialization or beginning phase as a single threaded. Figure 6-24 shows a Call Stack for the Selected Event after clicking in the region of Thread 1.2 between the timeline of 16 seconds and 64 seconds. Figure 6-24 shows the call stack that’s being executed during the selected thread and selected timeline sample. As you can see in Figure 6-24, a method by the name `BailoutMain.populateDatabase()` is being called. This is the method identified in the “Increasing Parallelism” section earlier in the chapter as one that could be multithreaded. Hence, this illustrates how you can use the Performance Analyzer to identify areas or phases of an application that could benefit from parallelism.

Another useful tip when using the Timeline view is make note of the range of seconds for some time period of interest that has caught your attention in the timeline. Then use the filtering capability to narrow the profile data loaded by the Analyzer GUI. After applying the filter, the Functions and Callers-Callees views show data only for the filtered range. In other words, filtering allows you to focus exclusively on the profile data collected within the period of interest. To illustrate with an example, in Figure 6-23, Thread 1.2 between 16 and 64 seconds is the only thread executing. To narrow the focus of the collected profile data to that particular time range, the Analyzer can be configured to load only the profile data between 16 and 64 seconds using the View > Filter Data menu and specifying 16-64 samples in the Filter Data form’s Samples field as shown in Figure 6-25.

Filtering allows for the ability to eliminate data collected outside an area of interest, which leads to more accurate analysis since only the data of interest is being presented.

![Figure 6-25 Filtering the range of samples to view in performance analyzer](image)

Figure 6-25 Filtering the range of samples to view in performance analyzer
There are many additional features of the Performance Analyzer, but this chapter presents those likely to be the most useful when profiling and analyzing Java applications. Additional details on using Performance Analyzer for profiling Java applications, including the Java EE application, can be found at the Performance Analyzer product Web site: http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html.

Bibliography


Index

: (colon), keyword delimiter, 182
* (asterisk), wildcard character, 44
\ (backslash), line termination character, 181
- (dash) option, 181
! (exclamation point) keyword, 182
% (percent sign) keyword, 182
+ (plus sign) keyword, 182
. (period) keyword, 182
32-bit runtime environment vs. 64-bit, 260–261

A (alpha), 351–353
A keyword, 182
-A option, collect tool, 163
Acceptor threads, monitoring and tuning, 414–417
acceptor-thread property, 415
Access logging, best practices, 446–450
Accessing XML documents, 455, 458–459
Adaptive heap sizing
  description, 104–105
  disabling, 105, 309–311
  enabling/disabling, 558
  HotSpot VM, 104–105, 558
  policy, printing, 563
  throughput, tuning, 309–311
Adaptive tuning. See HotSpot VM adaptive tuning.
Administration console, monitoring server applications, 383–384
Aggressive optimization, 568–569
Aging statistics, 145–146
Algorithms, increasing efficiency, 211–212
Allocated objects, profiling, 205
Allocation, HotSpot VM garbage collectors, 91
Allocations tracked, specifying, 204
Alpha (α), 351–353
Analyzer, definition, 158
APIs. See also JPA (Java Persistence API).
  DOM, 459–460
  JAXB (Java API for XML Binding), 454, 469–470
  JAXP (Java API for XML Processing), 454, 457
  showing/hiding, 168
  System.currentTimeMillis API, 328–329
  System.nanoTime API, 328–329
  for XML documents, selecting, 468–471
Application performance
  ideal CPU utilization, 15
  improving with network I/O utilization, 45
Application server monitoring
  disk I/O, 395–398
  external systems, 392–395
  with GlassFish
    administration console, 383–384
    asadmin CLI, 386–388
    JConsole, 384–386
    overview, 382
    VisualVM, 384–386
  monitoring resource pools, 398–399
  overview, 382
  subsystems
    JVM, 388–389
    network I/O, 390–392
Application server monitoring (continued)
  thread dumps, 389–390
  tuning resource pools, 398–399
Application threads, isolating, 25, 27
Applications. See also Benchmarking multitiered
applications; Benchmarking Web
applications; Java applications.
  concurrent run time, printing, 564
developing. See Software development.
  startup time, decreasing, 68
  stop time, printing, 563
Archiving artifacts, 163
asadmin CLI, monitoring server applications,
386–388
Asterisk (*), wildcard character, 44
Asynchronous benchmarks, 381
Asynchronous requests, benchmarking, 360
Attach Mode, specifying, 193–194
Attributed time, definition, 158
Availability
  performance metrics, calculating, 365–366
  service, benchmarking, 359
tuning the JVM, 255–256
Average age, profiling, 206
Averages, calculating, 349

B
Backedge counters, 95–96
Backslash (\), line termination character, 181
Bandwidth, monitoring, 44
Barriers, memory, 234
Bean caches, monitoring and tuning, 514–520
Bean pools, monitoring and tuning, 514–520
Benchmarking. See also Experiments; Statistics.
  compilation activity, eliminating, 333–334
deoptimization, 340–345
  EJB best practices, 522
  elapsed time, calculating, 328–329
  garbage collection pauses, 327–328
  inlining methods, 335–339
  micro-benchmarks, creating, 345–346
  optimizing away dead code, 329–335
  warm-ups, 324–327, 333–334
Web services, 473–476
Benchmarking multitiered applications. See also
Applications.
  challenges
    asynchronous requests, 360
    external dependencies, 360
    firewalls, 360
    nature of enterprise applications, 358
    payload sizes, 359
    secure interactions, 359
    service availability, 359
    session maintenance, 359
    user scaling, 358

variety of client types, 359
vertical and horizontal scaling, 358, 377
enterprise considerations
  availability metrics, calculating, 365–366
cycle time, 365
  injection rate, 365
  Markov chains, 362–366
  micro-benchmarks, developing, 361–362
  system boundaries, defining, 360–361
  think time, 364
  user interaction modeling, 362–366
Little's Law verification, 372–374
maximum number of concurrent clients,
372–374
performance metrics, calculating
  availability, 365–366
  page view, 366–367
  requests, 366
  response time, 368–369
  round-trip time, 366
  think time, 366
  throughput, 369–370
  user transactions, 366, 367–368
running the benchmark
  asynchronously, 381
  isolating the SUT, 378–379
  ramp down time, 380
  ramp up time, 380
  repeatability, 380–381
  resource monitoring, 379–380
  statistical methods, 381–382
  steady state time, 380
scalability
  analysis, 377–378
  hybrid, 377
  user scaling, 358
  vertical and horizontal scaling, 358, 377
scalability analysis, 377–378
scaling the benchmark, 370–372
SUT (System Under Test), isolating, 360–361,
378–379
think time
  benchmarking, 374–377
  calculating, 366
definition, 366
  enterprise considerations, 364
Benchmarking Web applications
See also Applications, 446–450
best practices
  access logging, 446–450
  accessing JavaBean components, 434–436
  bean, locating or instantiating, 432–434
  compression, 440–443
  content caching, 439–443
  context listeners, 427–429
  distributed caches, 439–443
  EL (expression language), 434–436
  HTTP compression, 436–438
<table>
<thead>
<tr>
<th>Index</th>
<th>671</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Server File Cache, 445–450</td>
<td>servlets, 427–438</td>
</tr>
<tr>
<td>JSP, 427–438</td>
<td>session persistence, 443–445</td>
</tr>
<tr>
<td>JSP include mechanism, 429–430</td>
<td>trimming whitespaces, 430–431</td>
</tr>
<tr>
<td>log file aggregation, 450</td>
<td>tuning the file cache, 446</td>
</tr>
<tr>
<td>overview, 427</td>
<td>JPA (Java Persistence API)</td>
</tr>
<tr>
<td>serialization, 440–443</td>
<td>bulk updates, 548–549</td>
</tr>
<tr>
<td>servlets, 427–438</td>
<td>connection pooling, 546–548</td>
</tr>
<tr>
<td>session persistence, 443–445</td>
<td>database locking strategies, 549</td>
</tr>
<tr>
<td>trimming whitespaces, 430–431</td>
<td>dynamic queries, 541</td>
</tr>
<tr>
<td>tuning the file cache, 446</td>
<td>inheritance, 550</td>
</tr>
<tr>
<td>overview, 404–405</td>
<td>JPA Query Language queries, 540–543</td>
</tr>
<tr>
<td>Web container components, GlassFish</td>
<td>named native queries, 541</td>
</tr>
<tr>
<td>Coyote connector, 407</td>
<td>named queries, 541</td>
</tr>
<tr>
<td>GlassFish, 406–407</td>
<td>native queries, 542</td>
</tr>
<tr>
<td>Grizzly connector, 406–407</td>
<td>query results cache, 543–544</td>
</tr>
<tr>
<td>HTTP connector, 406–407</td>
<td>reads without transactions, 550</td>
</tr>
<tr>
<td>overview, 405–406</td>
<td>Web service performance</td>
</tr>
<tr>
<td>servlet engines, 407–408</td>
<td>binary payload, 486–495</td>
</tr>
<tr>
<td>Web container monitoring and tuning</td>
<td>catalog file locations, 502–503</td>
</tr>
<tr>
<td>configuration settings, 408–409</td>
<td>client performance, 502–503</td>
</tr>
<tr>
<td>development mode, 408–409</td>
<td>Fast Infoset, 499–501</td>
</tr>
<tr>
<td>garbage collection, 411</td>
<td>HTTP compression, 501–502</td>
</tr>
<tr>
<td>HTTP service, 412</td>
<td>MTOM (Message Transmission Optimization Mechanism), 487–495</td>
</tr>
<tr>
<td>JTT compiler tuning, 410</td>
<td>overview, 486</td>
</tr>
<tr>
<td>JVM tuning, 410–412</td>
<td>Provider interface, 495–498</td>
</tr>
<tr>
<td>overview, 408</td>
<td>SOAP messages, 499–501</td>
</tr>
<tr>
<td>page freshness, checking, 409</td>
<td>XML documents, 492</td>
</tr>
<tr>
<td>production mode, 408–409</td>
<td>XML documents as attachments, 492–495</td>
</tr>
<tr>
<td>security manager, 409–410</td>
<td>Best practices, EJB (Enterprise JavaBeans)</td>
</tr>
<tr>
<td>Web container monitoring and tuning, HTTP</td>
<td>beans, locating or instantiating, 432–434</td>
</tr>
<tr>
<td>listener</td>
<td>benchmarking, 522</td>
</tr>
<tr>
<td>acceptor threads, 414–417</td>
<td>EJB 2.1</td>
</tr>
<tr>
<td>connection queues, 414–417</td>
<td>cache static resource references, 524–526</td>
</tr>
<tr>
<td>elements to be monitored, 412</td>
<td>coarse-grained access, 529–530</td>
</tr>
<tr>
<td>individual applications, 420–427</td>
<td>control serialization, 523–524</td>
</tr>
<tr>
<td>keep alive, 414–417</td>
<td>database locking strategies, 532–533</td>
</tr>
<tr>
<td>request processing, 418–420</td>
<td>EJB Query Language, 533–535</td>
</tr>
<tr>
<td>request response codes, 419</td>
<td>lazy loading, 530–532</td>
</tr>
<tr>
<td>thread pools, 412–414</td>
<td>local vs. remote interfaces, 526–528</td>
</tr>
<tr>
<td>Best practices</td>
<td>optimistic locking, 532–533</td>
</tr>
<tr>
<td>benchmarking Web applications</td>
<td>pessimistic locking, 532–533</td>
</tr>
<tr>
<td>access logging, 446–450</td>
<td>prefetching, 530–532</td>
</tr>
<tr>
<td>accessing JavaBean components, 434–436</td>
<td>read-only entity beans, 535–536</td>
</tr>
<tr>
<td>bean, locating or instantiating, 432–434</td>
<td>Session Façade pattern, 529–530</td>
</tr>
<tr>
<td>compression, 440–443</td>
<td>transaction attributes, choosing, 523</td>
</tr>
<tr>
<td>content caching, 439–443</td>
<td>transactions, container managed vs. bean managed, 522–523</td>
</tr>
<tr>
<td>context listeners, 427–429</td>
<td>EJB 3.0</td>
</tr>
<tr>
<td>distributed caches, 439–443</td>
<td>business method interceptors, 537–540</td>
</tr>
<tr>
<td>EL (expression language), 434–436</td>
<td>compatibility with EJB 2.1, 536–537</td>
</tr>
<tr>
<td>HTTP compression, 436–438</td>
<td>Biased locking, enabling, 569</td>
</tr>
<tr>
<td>HTTP Server File Cache, 445–450</td>
<td>Binary heap dumps, 140</td>
</tr>
<tr>
<td>JSP, 427–438</td>
<td>Binary XML payload, Web service performance</td>
</tr>
<tr>
<td>JSP include mechanism, 429–430</td>
<td>best practices, 486–495</td>
</tr>
<tr>
<td>log file aggregation, 450</td>
<td>Blocked thread state, 74</td>
</tr>
<tr>
<td>object size vs. cost, 444</td>
<td>Blocking vs. nonblocking sockets, 45</td>
</tr>
</tbody>
</table>
Bootstrap class loader, 65
Bottom up software development, 7–8
buffer-size-bytes property, 415
Bulk updates, JPA best practices, 548–549
Bump-the-pointer technique, 85
Business interface, 506–507
Business method interceptors, 537–540
Bytecode analysis, JIT compilers, 96–97
Bytecode verification, 66–67

C
C++ heap management, 76–77
Cache static resource references, 524–526
calibrate.sh script, 196–197
Call stack trees, displaying, 246
Call stacks, attributed time, 174–175
Call Tree tab, 169–171, 246
Call trees, 157–158, 170–171
Caller-callee relationships, 158, 172–174
Callers-callees, monitoring System CPU usage, 218–221
callers-callees command, 184
Callers-Callees tab, 169–170, 172–174
Callers-Callees tables, printing, 184–185
Card tables, 82–83
Catalog file locations, Web service performance
best practices, 502–503
Catalog resolvers, 463–464
Catching exceptions, 70–71
checkInterval property, 409
Chrome, Developer Tools for, 363
Class data sharing, 65, 67–68
Class Hierarchy Analysis, 94–95
Class level interceptor methods, 538–539
Class loader. See also HotSpot VM Runtime, class
loading.
delegetion, 65
time, monitoring, 144
Class metadata, 66
Classes. See also specific classes.
uninitialized, 98
unloaded, 98
Client JIT, 97
Client performance, Web service performance best
practices, 502–503
Client runtime environment vs. server, 260
Client types, benchmarking, 359
Clock cycles. See CPU, cycles.
Clock cycles per CPU instruction (CPI), 15, 211–212
cmometrics command, 186
CMS (Concurrent Mark-Sweep GC)
collection cycle, initiating, 298–303
concurrent collection, enabling, 561
incremental mode, 561
incremental pacing, 562
overview, 88–90
pause time tuning, 305–306
remarks, scavenging before, 560
sample output, 113–114
throughput, tuning, 307–308
tuning latency/responsiveness, 287–289, 298–303
Coarse-grained access, EJB best practices, 529–530
collect tool, 158, 162–164
Colon (:), keyword delimiter, 182
Command line flags, printing, 571
Command line names, printing, 572
-command option, 181
Common subexpression elimination, 93
Compilation
activity, eliminating, 333–334
JIT compilers, 93
policy, JIT compilers, 95–96
Compile time, monitoring, 144
Compiler structure, JIT compilers, 93
compressableMimeType property, 438
Compressed oops, 57, 554
Compression
best practices, 440–443
GlassFish server, 436–438
HTTP, 436–438
Compression property, 438
compressionMinSize property, 438
Concurrent collection, enabling, 560
Concurrent garbage collection, sample output, 115–117
Concurrent marking phase, 88
Concurrent Mark-Sweep GC (CMS). See CMS
(Concurrent Mark-Sweep GC).
Concurrent mode failure, 117
Concurrent permanent generation garbage
collection, 304–305
Concurrent sweeping phase, 89
CONDVAR_WAIT statement, 74
Confidence intervals, calculating, 350–351
Configuring remote systems for profiling, 196–197
Connection pooling, JPA best practices, 546–548
Connection queues, monitoring and tuning, 414–417
Constant folding, 93
Contended operations, 71–72
Content caching, best practices, 439–443
Context listeners, best practices, 427–429
Context switching, monitoring, 37
Control flow representation, JIT compilers, 98–100
Control serialization, EJB best practices, 523–524
Copying collectors, 85
corestat tool
aggregating instruction counts, 52
downloading, 52
monitoring CPU utilization, SPARC T-series, 52
count5xx-count attribute, 419
count200-count attribute, 419
count302-count attribute, 419
count304-count attribute, 419
count404-count attribute, 419
countconnections-count attribute, 417
counthits-count attribute, 417
countoverflows-count attribute, 416
countqueued-count attribute, 416
countqueued*minuteaverage-count attribute, 416
countrefusals-count attribute, 417
countrequests-count attribute, 419
counttimeouts-count attribute, 417
Coyote connector, 407
CPI (clock cycles per CPU instruction), 15, 211–212
CPU
architecture, choosing, 9–10. See also specific architectures.
cache efficiency, 57
counters, collecting, 163–164
cycles
CPI (clock cycles per CPU instruction), 15
IPC (CPU instructions per clock cycle), 15
monitoring, 14–16. See also Monitoring CPU utilization.
monitoring context switching, 39
stalls, 15
waiting for data, 15
performance counters
listing, 50
monitoring, 49–50
scheduler’s run queue, monitoring
Linux, 31–32
overview, 28–29
Solaris, 31
Windows, 29–31
utilization. See also Monitoring CPU utilization.
application performance, ideal situation for, 15
definition, 15
high, identifying, 246
scalability, ideal situation for, 15
system CPU, 15
user CPU, 15
CPU instructions per clock cycle (IPC), 15
cpubar tool. See also icobar tool.
monitoring CPU utilization, 21–24
monitoring memory utilization, 34–35
monitoring run queue depth, 31
cpustat tool
listing CPU performance counters, 50
monitoring CPU performance counters, 49–50
monitoring CPU utilization, SPARC T-series, 52
monitoring instructions per hardware thread, 52
cputrack tool
listing CPU performance counters, 50
monitoring CPU performance counters, 49–50
Criteria for performance, 2–3. See also Metrics.
csingle command, 185–186
currentthreadsbusy-count attribute, 414
D
-d option, collect tool, 163
-d64 option, 554
Dash (-) option, 181
Data structure resizing
identifying, 235
Java collections, 238
overview, 235
StringBuffer, 235–238
StringBuilder, 235–238
Database locking strategies
EJB best practices, 532–533
JPA best practices, 549
Data-fetching strategy, JPA best practices, 544–546
Date and time stamps
monitoring garbage collection, 117–119
printing, 266, 562
dateTime schema, effects on Web service performance, 481–482
Dead code, optimizing away, 329–335
Deadlocks, 80
Debug VM, 69
Debugging
alternative interface, enabling, 568
log files, dumping, 79
threads, 74–75
VMError class, 79
-XX:OnError, 79
Default interceptor methods, 538
DefaultServlet servlet engine, 408
DefNew garbage collector, 111, 264
Degrees of freedom, 351–353
Deoptimization, 95, 96–97, 340–345
Deployment model, choosing
multiple JVM deployment, 258–259
overview, 259
single JVM deployment, 258
Destroying threads, 73–74
DestroyJavaVM method, 62–63
DetachCurrentThread method, 60
Development mode, Web containers, 408–409
Disassembly tab, 169–170
Disk I/O utilization. See Monitoring disk I/O.
Disks, formatting, 49
Distributed caches, best practices, 439–443
DocumentBuilder class, creating, 455–456
DocumentBuilderFactory class, 456
DOM APIs
modifying XML documents, 459–460
XML document performance, 469–470
DTD (document type definition), external subsets, 462–464
Dynamic queries, JPA best practices, 541
E

E keyword, 182
EclipseLink session cache, monitoring and tuning, 519–520
Eden space
description, 83–85
size, compared to survivor space, 290–291, 556
utilization, monitoring, 143, 144
Edge cases, tuning, 316
EJB (Enterprise JavaBeans). See also NetBeans.
Business interface, 506–507
components, 505–506
Home interface, 506–507
message driven beans, 505–506
optimistic locking, 521
persistent entities, 505–506
programming model, 506–507
session beans, 505–506
stateful session beans, 506
stateless session beans, 506
transaction isolation levels, 521–522
EJB (Enterprise JavaBeans), best practices
beans, locating or instantiating, 432–434
benchmarking, 522
EJB 2.1
cache static resource references, 524–526
course-grained access, 529–530
control serialization, 523–524
database locking strategies, 532–533
EJB Query Language, 533–535
lazy loading, 530–532
local vs. remote interfaces, 526–528
optimistic locking, 532–533
pessimistic locking, 532–533
prefetching, 530–532
read-only entity beans, 535–536
Session Façade pattern, 529–530
transaction attributes, choosing, 523
transactions, container managed vs. bean managed, 522–523
EJB 3.0
business method interceptors, 537–540
compatibility with EJB 2.1, 536–537
EJB container, monitoring and tuning
bean caches, 514–520
bean pools, 514–520
EclipseLink session cache, 519–520
entity bean caches, 516
invocation patterns, 512
overview, 511
Ready Cache, 516–517
stateful session bean caches, 516
thread pool, 512–514
Transactional Cache, 516–517
EJB Query Language, best practices, 533–535
EL (expression language), best practices, 434–436
Elapsed time
calculating, 328–329
monitoring garbage collection, 114
Endpoint implementation, effects on Web service performance, 483–484
Entering a Java monitor, 71–72
Enterprise applications, profiling, 399–400
Entity bean caches, monitoring and tuning, 516
Entity resolvers, 462–464
Ergonomics
defaults, printing, 102–103
definition, 100
Java 1.4.2 defaults, 101
Java 5 defaults, 101–103
Java 6 Update 18 defaults, 103–104
server-class machines, 101–103
er_print tool. See also Printing, experiment profiles.
: (colon), keyword delimiter, 182
\ (backslash), line termination character, 181
- (dash) option, 181
! (exclamation point) keyword, 182
% (percent sign) keyword, 182
+ (plus sign) keyword, 182
. (period) keyword, 182
A keyword, 182
abbreviations, 181
callers-callees command, 184
cmetrics command, 186
-c-command option, 181
csingle command, 185–186
definition, 158
e keyword, 182
er_print_metric_list command, 183
filters command, 186–187
i keyword, 182
limit command, 183–184
lock keyword, 182
metric keywords, 182–184
outfile command, 187
-script option, 181
scripting, 180, 187–189
sort command, 183
splitting commands, 181
syntax, 180–181
system keyword, 182
user keyword, 182
-v option, 181
viewmode command, 187
er_print_metric_list command, 183
Error checking, XML documents, 460
Error handling, 568
Escape analysis, enabling, 569
Even Faster Web Sites, 404
Event tab, 168–169
Exception handling, 70–71
Exclamation point (!) keyword, 182
Exclusive time
definition, 158, 160
displaying, 176
Exiting a Java monitor, 71–72
Experiment files
  creating, 163
  opening, 168
  specifying a directory for, 163
Experiments. See also Benchmarking; Monitoring;
  Performance Analyzer, experiments;
  Profiles; Profiling; Tuning.
  definition, 158
  designing, 347–348
Experiments tab, 170
Expert mode, 178
Explicit garbage collection
  monitoring, 121
  tuning latency/responsiveness, 303–304
Expression language (EL), best practices, 434–436
External dependencies, benchmarking, 360

F
Factory lookup, 456–457
Factory objects, reusing, 457
Fast allocation, HotSpot VM garbage collectors, 85
Fast Infoset Web service performance best
  practices, 499–501
Fast-path code, synchronization, 72
Fatal error handling, 78–80
FetchType, 544–546
File cache tuning, best practices, 446
Filtering data
  data presentation, 168, 179–180, 248–249
  printing experiment profiles, 186–187
Filters, definition, 158
  filters command, 186–187
Firebug plug-in, 363
Firewalls, benchmarking, 360
Footprint. See Memory footprint.
  format command, 49
Fragmentation issues, garbage collection, 90
Full garbage collection. See also Major garbage
  collection.
  definition, 85
  monitoring, 109–110, 112–113
  sample output, 112
  scavenging young generation space, 110, 561
  tuning latency/responsiveness, 286
Full Identity Map option, 509
Functions tab, 169–170, 171–174

G
G1 GC, 90–91
Garbage collection. See also HotSpot VM garbage
  collectors.
  definition, 159
GlassFish server, 411–412
  logging results, 562
  monitoring. See Monitoring garbage collection.
old generation, enabling, 558
pauses, benchmarking, 327–328
pausing for swapping, 32
stop-the-world, 76, 558
tuning latency/responsiveness, activities
  affecting, 278–279
Web containers, 411
Garbage collection reports
  adaptive size policy, 563
  application concurrent run time, 564
  application stop time, 563
  date and time stamps, printing, 562
detailed, enabling, 562
  enabling, 562
  safepoint statistics, 564
tenuring statistics, 563
Garbage collection threads, 75
Garbage collectors. See also HotSpot VM garbage
  collectors.
tuning
  choosing, 261–262
  command line options, 263–267
date stamp, printing, 266
directing output to a file, 264
latency, 262
logging, 263–267
memory footprint, 262
OutOfMemoryError, 273–274
performance attributes, 262–263
  principles of, 263
  safepoint pauses, 267
  statistics, printing, 264
  throughput, 262, 308–311
time stamp, printing, 264, 266
Garbage-First GC, 90–91
GC time, monitoring, 144
GCHisto tool, 121–125
Generational garbage collection. See HotSpot VM
  garbage collectors, generational.
Generations, NetBeans Profiler, 206–207
getElementsByTagName method, 459
getElementsByTagNameNS method, 459
GlassFish server
  access logging, 447
  application server monitoring
    administration console, 383–384
    asadmin CLI, 386–388
    JConsole, 384–386
    overview, 382
    VisualVM, 384–386
  benchmarking Web container components
    Coyote connector, 407
    GlassFish, 406–407
    Grizzly connector, 406–407
    HTTP connector, 406–407
    servlet engines, 407–408
  compression, 436–438
dynamic JSP modification, 408
garbage collection, 411–412
GlassFish server (continued)
  maximum connections, 407
  monitoring Java applications, 150–151
  monitoring server applications
    administration console, 383–384
    asadmin CLI, 386–388
    JConsole, 384–386
    overview, 382
    VisualVM, 384–386
  RMI server, 411–412
  security manager, 410
Web container components
  Coyote connector, 407
  GlassFish, 406–407
  Grizzly connector, 406–407
  HTTP connector, 406–407
  overview, 405–406
  servlet engines, 407–408
Web containers
  Coyote connector, 407
  development mode, 408–409
  GlassFish, 406–407
  Grizzly connector, 406–407
  HTTP connector, 406–407
  production mode, 408–409
  servlet engines, 407–408
GlassFish Server Open Source Edition. See GlassFish server.
GlassFish Web container
  development mode, 408–409
  production mode, 408–409
  GMT, adjusting to local time, 118–119
  GNOME System Monitor, monitoring CPU utilization, 20–21
  gnome-system-monitor command, 20–21
Graph coloring, 94
Graphs panel, 144–145
Grizzly connector, 406–407

H
  -h option, collect tool, 163–164
  Handler performance, effects on Web service performance, 484–486
  Hard Cache Weak Identity Map option, 510
  Hardware threads, SPARC T-series processor, 9–10
  hasAttributes method, 459
Heap
  aggressive options, 569
  definition, 159
  initial size, configuring, 275–277
  in JConsole. See Memory pools.
  layout, 268–272
  live data size, calculating, 274–275
  management, C++, 76–77
  memory, JConsole, 129
  profiling. See Memory profiles.
  size, specifying, 554–555
  size, starting point, 272–274
space, on NUMA systems, 571
space limitation, 57
splitting for garbage collection, 81
utilization, monitoring, 111–113, 114
Heap dumps
  analyzing with NetBeans Profiler, 209
directory path, specifying, 567–568
enabling on OutOfMemoryError, 567
on OutOfMemoryError, 567
specifying a location for, 80
Heap sizing, adaptive
description, 104–105
disabling, 105, 309–311
enabling/disabling, 558
HotSpot VM, 104–105, 558
policy, printing, 563
throughput, tuning, 309–311
High Performance Web Sites, 404
Histogram panel, 145–146
Home interface, 506–507
Horizontal scaling, 358, 377. See also Scaling.
Hot locks, isolating, 39–40
HotSpot VM. See also JVM (Java Virtual Machine), tuning.
  64-bit version, loading, 554
architectural overview
  32-bit vs. 64-bit versions, 57
  compressed oops, 57
  CPU cache efficiency, 57
  garbage collectors, 57
  high level architecture, 56–58
  Java heap space limitation, 57
  memory address limitation, 57
  platforms supporting, 58
  register spilling, 58
debg VM, 69, 337
launching, 60–62
lock optimization, 37
shutting down, 62–63
HotSpot VM, optimization aggressive, 568–569
for client applications, 553
for server applications, 553
HotSpot VM adaptive tuning
  adaptive heap sizing
    enabling/disabling, 558
    overview, 104–105
ergonomics
  defaults, printing, 102–103
  definition, 100
  Java 1.4.2 defaults, 101
  Java 5 defaults, 101–103
  Java 6 Update 18 defaults, 103–104
server-class machines, 101–103
heap sizing, disabling, 105
overview, 100
HotSpot VM garbage collectors
  allocation, 91
  bump-the-pointer technique, 85
creating work for, 91
fast allocation, 85
history of, 92
live data size, 91
monitoring. See Monitoring garbage collection.
overview, 80–81
reference updates in old generation, 91
TLABs (Thread-Local Allocation Buffers), 85
HotSpot VM garbage collectors, generational
card tables, 82–83
copying collectors, 85
full garbage collection, 85. See also Major
garbage collection.
gerational, 81–83
major garbage collection, 81. See also Full
garbage collection; Old generation
garbage collection.
minor collection. See also Young generation
garbage collection.
definition, 81
process flow, 84–85
reducing runtime, 82–83
old generation space, 81
permanent generation, 81
premature promotion, 85
promotion, 81
promotion failure, 85
splitting the heap, 81
tenure, 81
weak generational hypothesis, 81
write barriers, 83
young generation collection
definition, 81
eden space, 83–85
layout, 83–85
survivor spaces, 84–85
HotSpot VM garbage collectors, types of. See also
specific types.
CMS (Concurrent Mark-Sweep GC), 88–90
comparison chart, 91
G1 GC, 90–91
Garbage-First GC, 90–91
mark-compact, 86–87
Mostly-Concurrent GC
concurrent marking phase, 88
concurrent sweeping phase, 89
definition, 88
disadvantages of, 89–90
enabling, 559
fragmentation issues, 90
initial mark, 88
phases of, 88–89
pre-cleaning phase, 89
remark pause, 88–89
Parallel GC, 87–88
Parallel Old GC, 87–88
Serial GC, 86–87, 92
sliding compacting mark-sweep, 86–87
Throughput GC. See Parallel GC.
Train GC, 92
HotSpot VM JIT compilers. See JIT compilers.
HotSpot VM Runtime
application startup time, decreasing, 68
bytecode verification, 66–67
C++ heap management, 76–77
class data sharing, 67–68
class loading
bootstrap class loader, 65
class data sharing, 65
class loader delegation, 65
class metadata, 66
definition, 63
initialization phase, 64
internal data, 66
link phase, 64
load class phase, 64
monitoring, 147–150
phases, 64–65
reasons for, 64
safepoints, 66
type safety, 65–66
command line options, 58–59
developer command line options, 59
exception handling, 70–71
fatal error handling, 78–80
interpreter, 69–70
JNI (Java Native Interface), 77–78
memory footprint cost, reducing, 68
nonstandard command line options, 59
overview, 58
standard command line options, 59
synchronization
biased state, 72
concurrency, 71
contended operations, 71–72
entering a Java monitor, 71–72
exiting a Java monitor, 71–72
fast-path code, 72
inflated state, 72
Java monitors, 71–72
mark word, 72
mutual exclusion, 71
neutral state, 72
owning Java monitors, 71–72
races, avoiding, 71–72
slow-path code, 72
stack-loaded state, 72
states, 72
synchronized blocks, 71
uncontended operations, 71
thread management
blocked thread state, 74
CONDVAR_WAIT statement, 74
creating threads, 73–74
deadlocks, 80
debugging, 74–75
destroying threads, 73–74
HotSpot VM Runtime (continued)
garbage collection threads, 75
internal VM threads, 75
JIT compiler threads, 75
MONITOR_WAIT statement, 74
new thread state, 74
OBJECT_WAIT statement, 75
overview, 72
periodic task threads, 75
safepoints, 75–76
signal dispatcher thread, 75
thread in Java state, 74
thread in Java vm state, 74
thread states, 74–75
threading model, 72–73
VM operations, 75–76
VM threads, 75
type inference, 67
type verification, 67
VM life cycle, 59–61
HTTP compression
best practices, 436–438
Web service performance, best practices, 501–502
HTTP connector, 406–407
HTTP listener, monitoring and tuning
acceptor threads, 414–417
connection queues, 414–417
elements to be monitored, 412
individual applications, 420–427
keep alive, 414–417
request processing, 418–420
request response codes, 419
thread pools, 412–414
HTTP Server File Cache, best practices, 445–450
HTTP service, Web containers, 412
Hybrid scalability, 377
Hypothesis tests, 351–354

I
i keyword, 182
Identity transformation, 93
include file directive, 429
Inclusive time
definition, 158, 160
displaying, 160
Inflated state, synchronization, 72
Inheritance, JPA best practices, 550
init method, 427–429
Initial mark, 88
Initialization phase, 64
Injection rate, benchmarking, 365
Inlined methods
benchamking, 335–339
maximum bytecode size, 567
printing, 566–567
Inlined of functions, 93
Instrumentation, definition, 159
Intermediate representation (IR), 93
Internal class loader data, 66
Internal VM threads, 75
Interpreter
adaptive optimization, 70
overview, 69
vs. switch statements, 69
Invocation counters, 95
Involuntary context switching, monitoring, 40–41
I/O, monitoring System CPU usage, 214–218, 221–222
iobar tool, Solaris, 46–47. See also cpubar tool.
iosnoop.d script, 47–48
iostat tool, 46–47
iostat tool, 46–47. See also prstat tool; top tool.
IPC (CPU instructions per clock cycle), 15
IR (intermediate representation), 93
Iteration splitting, 99–100

J
Java API for XML Binding (JAXB), 454, 469–470
Java API for XML Processing (JAXP), 454, 457
Java applications. See also Applications.
listing, 134
monitoring
GlassFish server, 150–151
jstack output, example, 151–153
overview, 150–151
quick lock contention, 151–153
Java collections
overview, 238–243
resizing, 238
Java heap. See Heap.
Java HotSpot VM. See HotSpot VM.
Java monitors, synchronization, 71–72
Java Native Interface (JNI), 77–78
Java Persistence API (JPA). See JPA (Java Persistence API).
Java Virtual Machine (JVM). See JVM (Java Virtual Machine), tuning.
JavaBean components, accessing with best practices, 434–436
java.util.Random, lock contention
hottest methods, displaying, 228–229
replacing with ThreadLocal<Random>, 232
sample code, 593–603, 603–613, 613–624, 624–635
source code, 230
javaw command, 60
javaws command, 60
JAXB (Java API for XML Binding), 454, 469–470
JAXP (Java API for XML Processing), 454, 457
JAX-WS RI (JAX-WS Reference Implementation)
stack, 471–473
JConsole. See also VisualGC; VisualVM.
heap memory, 129
local monitoring, 127
memory, monitoring, 128–130
memory metrics, 129–130
memory pools, mapping to HotSpot VM spaces, 129
monitoring server applications, 384–386
overview, 125–127
remote monitoring, 127–128
tabs, 128–130
JIT compiler reports
inlined methods, 566–567
optimization decisions, 567
optimized methods, 565–566
JIT compilers
backedge counters, 95–96
batch, 564–565
bytecode analysis, 96–97
class files, 93
Class Hierarchy Analysis, 94–95
Client JIT, 97
common subexpression elimination, 93
compilation, 93
compilation policy, 95–96
compiler structure, 93
constant folding, 93
control flow representation, 98–100
defaults for server-class machines, 101–102
deoptimization, 95, 96–97
future enhancements, 100
graph coloring, 94
in HotSpot VM, 70
identity transformation, 93
inline methods, maximum bytecode size, 567
inlining of functions, 93
invocation counters, 95
IR (intermediate representation), 93
linear scan register allocation, 94
loop optimization, 99–100
machine representation, 93–94
metadata for compiled code, 96–97
method counters, 95
Method Liveness, 96–97
methodDataOop object, 98
monitoring, 146–147
OopMaps tables, 97
optimizations, 93–94
OSRs (On Stack Replacements), 95
overridden methods, detecting, 94–95
overview, 92–94
program dependence graphs, 98–100
register allocation, 94
register tables, 97
running in background, 564–565
Server JIT, 97–98
SSA (single static assignment), 93, 98–100
stack location tables, 97
superword, 99–100
threads, 75
tiered compilation, 565
tuning Web containers, 410
uncommon traps, 96–97, 98–100
uninitialized classes, 98
unloaded classes, 98
JMeter tool, 363
JMX applications, configuring, 135–137
JNI (Java Native Interface), 77–78
JNI_CreateJavaVM method, 61–62
JOINED inheritance, 550
JPA (Java Persistence API)
best practices
bulk updates, 548–549
connection pooling, 546–548
database locking strategies, 549
data-fetching strategy, 544–546
dynamic queries, 541
inclusion, 550
JPA Query Language queries, 540–543
named native queries, 541
named queries, 541
native queries, 542
query results cache, 543–544
reads without transactions, 550
L2 (level two) cache
configuring, 509–511
default type, 511
Full Identity Map option, 509
Hard Cache Weak Identity Map option, 510
No Identity Map option, 510
options, 509–511
overview, 508
size, vs. performance, 508
Soft Cache Weak Identity Map option, 510
Soft Identity Map option, 509
Weak Identity Map option, 509
overview, 507
JPA Query Language queries, best practices, 540–543
JSP best practices, 427–438
jsp:include page action, 429
jspInit method, 428–429
JspServlet servlet engine, 408
jsp:useBean action, 432–434
JSR-133, 234
jstack command
monitoring CPU utilization, 27–28
monitoring thread dumps, 390
output, example, 151–153
jstat command, 389
jstatd daemon, 133–134
jvisualvm program, 191
JVM (Java Virtual Machine), tuning. See also
HotSpot VM.
application systemic requirements
availability, 255–256
latency, 256
manageability, 256
memory footprint, 256–257
overview, 255
responsiveness, 256
JVM (Java Virtual Machine), tuning. See also HotSpot VM. (continued)
  startup time, 256–257
  throughput, 256
  application throughput
    adaptive sizing, disabling, 309–311
    CMS, 307–308
    deploying on NUMA systems, 315
    garbage collectors, 308–311
    overview, 307
    parallel GC threads, 314–315
    survivor spaces, 311–314
  assumptions, 254
  command line options, latest optimizations, 317
  deployment model, choosing
    multiple JVM deployment, 258–259
    overview, 259
    single JVM deployment, 258
  edge cases, 316
  garbage collectors
    choosing, 261–262
    command line options, 263–267
    date stamp, printing, 266
    directing output to a file, 264
    latency, 262
    logging, 263–267
    memory footprint, 262
    OutOfMemoryError, 273–274
    performance attributes, 262–263
    principles of, 263
    safepoint pauses, 267
    statistics, printing, 264
    throughput, 262
    time stamp, printing, 264, 266
  latency/responsiveness
    CMS (Concurrent Mark-Sweep GC), 287–289
    CMS collection cycle, initiating, 298–303
    CMS pause time tuning, 305–306
    concurrent permanent generation garbage collection, 304–305
    explicit garbage collection, 303–304
    full garbage collections, 286
    garbage collection activities affecting, 278–279
    inputs, 279–280
    overview, 278–279
    promotion, 291–293
    survivor spaces, 289–291
    survivor spaces, occupancy, 298
    survivor spaces, sizing, 294–303
    tenuring threshold, 291–294
    young generation size, refining, 280–283
    overview, 252–255
  ranking systemic requirements, 257–258
  runtime environment, choosing
    32-bit vs. 64-bit, 260–261
    client vs. server, 260
    tiered, 260
  testing infrastructure requirements, 255
  Web containers, 410–412

K
  Keep alive, monitoring and tuning, 414–417
  Kernel CPU. See System CPU.
  Kernel statistics, 49
  Kernel thread queue depths, monitoring, 21–24
  Kesselman, Jeff, 2–5
  kstat tool, 49

L
  L2 (level two) cache
    configuring, 509–511
    default type, 511
    Full Identity Map option, 509
    Hard Cache Weak Identity Map option, 510
    No Identity Map option, 510
    options, 509–511
    overview, 508
    size, vs. performance, 508
    Soft Cache Weak Identity Map option, 510
    Soft Identity Map option, 509
    Weak Identity Map option, 509
  Latency/responsiveness
    tuning garbage collectors, 262
    tuning the JVM
      CMS (Concurrent Mark-Sweep GC), 287–289
      CMS collection cycle, initiating, 298–303
      CMS pause time tuning, 305–306
      concurrent permanent generation garbage collection, 304–305
      description, 256
      explicit garbage collection, 303–304
      full garbage collections, 286
      garbage collection activities affecting, 278–279
      inputs, 279–280
      old generation size, refining, 283–287
      overview, 278–279
      promotion, 291–293
      survivor spaces, 289–291
      survivor spaces, occupancy, 298
      survivor spaces, sizing, 294–303
      tenuring threshold, 291–294
      young generation size, refining, 280–283
  Lazy loading, EJB best practices, 530–532
  limit command, 183–184
  Linear scan register allocation, 94
  Link phase, 64
  Little’s Law verification, 372–374
  Live bytes, profiling, 205
  Live data size, HotSpot VM garbage collectors, 91
  Live HTTP Headers, 363
  Live objects, profiling, 205
Live Results control, 199
Load class phase, 64
Local vs. remote interfaces, EJB best practices, 526–528
Lock contention
  finding, 222–225
  isolating, 222–225
  overview, 222–225
  reducing, 212
  scaling symptoms, 224
  User Lock metric, 176–177
Lock contention, monitoring
  hot locks, isolating, 39–40
  HotSpot VM, 37
  Linux, 38–39
  Solaris, 36–38
  Windows, 39
Lock contention, sample code
  ConcurrentHashMap, 583–593
  java.util.Random, 593–603, 603–613, 613–624, 624–635
  parallelism
    multithreaded, 657–668
    single-threaded, 647–657
  partitioned database, 624–635
  resizing variant, 624–635, 636–647
  synchronized HashMap, 573–583, 603–613
Lock keyword, 182
Locking, JVM-System
  ConcurrentHashMap, 227–233
  overview, 225–233
  synchronized HashMap, 225–233
Log files
  aggregation, best practices, 450
  dumping, 79
  garbage collection, specifying, 119
  loading multiple, 124–125
Logging
  best practices, 396
  garbage collection results, 562
  garbage collectors, 263–267
  GlassFish server, 447
Long latency CPU events, SPARC T-series
  processor, 11
Loops
  iteration splitting, 99–100
  optimizing, 99–100
  range check elimination, 99–100
  superword, 99–100
  unrolling, 99–100
  unswitching, 99–100

M
Machine mode, 178
Machine representation of code, 93–94
Major garbage collection, 81, 109–110. See also
  Full garbage collection; Old generation garbage collection.
Manageability, tuning the JVM, 256
Mark word, 72
Mark-compact garbage collectors, 86–87
Markov chains, benchmarking, 362–366
Marshal XML documents. See Parse/unmarshall;
  Serialize/marshall.
max-connections-count property, 415–416
Maximum number of concurrent clients, 414
benchmarking, 372–374
maxthreads-count attribute, 414
Members, memory, 234
Memory
  address limitations, 57
  barriers, 234
  fences, 234
  footprint cost, reducing, 68
  members, 234
  metrics, 129–130
  OutOfMemoryError, 78–80
  scan rate, monitoring, 21–24
  volatile usage, 234
Memory footprint
  garbage collectors, 262
  tuning the JVM, 256–257
Memory footprint, determining
  application total memory, determining, 277
  constraints, 268
  heap
    initial size, configuring, 275–277
    layout, 268–272
    live data size, calculating, 274–275
    size, starting point, 272–274
  old generation space, 269–272
  overview, 268
  permanent generation space, 269–272
  young generation space, 269–272
Memory leaks
  definition, 159
  NetBeans Profiler, 206–207, 208
Memory pages
  large, enabling, 570
  touching, enabling, 570–571
Memory paging, monitoring, 21–24
Memory pools, mapping to HotSpot VM spaces, 129
Memory profiles, NetBeans Profiler, 202–205
Memory utilization. See also Monitoring memory utilization.
  freeing memory, See Garbage collection.
  monitoring, 23–24
  swap space, 32. See also Swapping memory.
Message driven beans, 505–506
Message size, effects on Web service performance, 477–479
Message Transmission Optimization Mechanism (MTOM), best practices, 487–495
Metadata for compiled code, 96–97
Method counters, 95
Method level interceptor methods, 539
Method Liveness, 96–97
Method profiles. See NetBeans Profiler, method profiles.
methodDataOop object, 98
Methods
  overridden, detecting, 94–95
  showing/hiding, 168
Metric keywords, 182–184
Metrics, profiling, 175–176. See also Criteria for performance; Performance Analyzer, metrics.
Micro-benchmarks. See also Benchmarking.
  creating, 345–346
  developing, 361–362
Minor garbage collection. See also Young generation garbage collection.
  definition, 81
  monitoring, 109–110
  process flow, 84–85
  reducing runtime, 82–83
  sample output, 113–114
Modes, experiment data
  Expert, 178
  Machine, 178
  User, 177–178
Modify Profiling control, 199
Modifying XML documents
  attributes, checking for and retrieving, 459
  definition, 455
  description, 459–460
  DOM APIs, 459–460
  error checking, 460
  node expansion, deferring, 460
  nodes, creating, renaming and moving, 459
Monitor contention, 177
Monitoring. See also Experiments; Profiling; Tuning.
  application servers. See Application server monitoring.
  definition, 14, 108
  JIT compilers, 146–147
  JVM, 388–389
  local applications, 127
  memory, 128–130
  network I/O, 390–392
  remote applications, 127–128, 133–137
  resource pools, 398–399
  thread dumps, 389–390
Monitoring CPU scheduler’s run queue
  Linux, 31–32
  overview, 28–29
  Solaris, 31
  Windows, 29–31
Monitoring CPU utilization. See also CPU, utilization.
  Linux
    application threads, isolating, 25, 27
    memory utilization, 26–27
    printing statistics, 26–27
  reporting intervals, setting, 24–26
Linux tools
  command line tools, 24–28
  GNOME System Monitor, 20–21
  mpstat tool, 25–26
  top tool, 26
  vmstat tool, 24–25
  xosview tool, 21
overview, 14–16
Solaris
  application threads, isolating, 25, 27
  kernel thread queue depths, 21–24
  memory paging, 21–24
  memory scan rate, 21–24
  memory utilization, 23–24
  printing statistics, 26–27
  process thread stack dumps, 27
  reporting intervals, setting, 24–26
  thread ids, converting to hexadecimal, 27–28
Solaris tools
  command line tools, 24–28
  cpubar, 21–24
  GNOME System Monitor, 21
  jstack, 27–28
  mpstat, 25–26
  prstat, 26–27
  pstack, 27
  vmstat, 24–25
SPARC T-series systems
  overview, 50
  stalls, 50–51
Windows
  Performance Manager, 16–19
  Task Manager, 16–19
  typeperf tool, 19–20
Monitoring disk I/O
  benchmarking, 395–398
  disk cache, enabling, 48–49
Linux, 46
  patterns, 48
  process ids, 47–48
  seek times, 48
  service times, 48
  servicing I/O events, 48
Solaris, 46
  user ids, 47–48
Windows, 46
Monitoring EJB containers
  bean caches, 514–520
  bean pools, 514–520
  EclipseLink session cache, 519–520
  entity bean caches, 516
  invocation patterns, 512
  overview, 511
  Ready Cache, 516–517
  stateful session bean caches, 516
  thread pool, 512–514
  Transactional Cache, 516–517
Monitoring garbage collection

  CPU usage, 114–115
  data of interest, 109
  enabling/disabling, 110
  full collections, 109–110
  GCHisto tool, 121–125
  graphical tools, 125. See also specific tools.
  major collections, 109–110
  minor collections, 109–110
  offline analysis, 121–125
  overhead, 122–123
  overview, 108–109
  pause times, 122–124
  stop-the-world pauses, 122
  types of collections, 109–110
  young generation collections, 109–110

Monitoring garbage collection, reporting concurrent mode failure, 117

  CPU usage, 114–115
  date and time stamps, 117–119
  elapsed time, 114
  explicit collection, 121
  full garbage collection, 112–113
  Java heap utilization, 111–113, 114
  log files, specifying, 119
  offline analysis, 119
  old generation space
    calculating, 112–113, 114
    reducing, 116–117
  permanent generation space, 113
  premature promotion, 117
  recommended command line options, 121
  runtime between safepoint operations, 119–120
  sample output
call to System.gc, 121
  CMS (Concurrent Mark-Sweep GC), 113–114
  concurrent garbage collection, 115–117
  full garbage collection, 112
  minor garbage collection, 113–114
  runtime between safepoint operations, 119–120
  from -XX:+PrintGCApplicationConcurrentTime option, 120
  from -XX:+PrintGCApplicationStopDuration option, 120
  from -XX:+PrintGCDetails option, 110–111
  from -XX:+PrintGCTimeStamps option, 118–119
  tenuring distribution, 117
  -verbose option, 110

Monitoring HTTP listener

  acceptor threads, 414–417
  connection queues, 414–417
  elements to be monitored, 412
  individual applications, 420–427
  keep alive, 414–417
  request processing, 418–420
  request response codes, 419
  thread pools, 412–414

Monitoring Java applications

  GlassFish server, 150–151
  jstack output, example, 151–153
  overview, 150–151
  quick lock contention, 151–153

Monitoring memory utilization. See also Memory utilization.

  involuntary context switching, 40–41
  Linux, 35–36
  lock contention
    hot locks, isolating, 39–40
    HotSpot VM, 37
    Linux, 38–39
    Solaris, 36–38
    Windows, 39
    Solaris, 34–35
    Windows, 33–34
  Monitoring network I/O. See also Network I/O utilization.
    Linux, 43
    Solaris, 42–43
    Windows, 44–45
  Monitoring Web containers
    configuration settings, 408–409
    development mode, 408–409
    garbage collection, 411
    HTTP service, 412
    JIT compiler tuning, 410
    JVM tuning, 410–412
    overview, 408
    page freshness, checking, 409
    production mode, 408–409
    security manager, 409–410
  MONITOR_WAIT statement, 74

Mostly-Concurrent GC

  concurrent marking phase, 88
  concurrent sweeping phase, 89
  definition, 88
  disadvantages of, 89–90
  fragmentation issues, 90
  initial mark, 88
  phases of, 88–89
  pre-cleaning phase, 89
  remark pause, 88–89

mpstat tool, Linux

  lock contention, 37–38
  monitoring CPU utilization, 25–26

mpstat tool, Solaris

  monitoring context switching, 37–38
  monitoring CPU utilization, 25–26
  monitoring involuntary context switching, 40–41
  monitoring lock contention, 37–38
  monitoring thread migrations, 41
  reporting CPU utilization for SPARC T-series, 51–52
MTOM (Message Transmission Optimization Mechanism), best practices, 487–495
Multithreaded reference processing, enabling, 561
Multithreaded young generation garbage collection, 111, 559

N
Named native queries, JPA best practices, 541
Named queries, JPA best practices, 541
Native queries, JPA best practices, 542
NetBeans Profiler
allocations tracked, specifying, 204
downloading, 190–191
features, 190
generations, 206–207
heap dumps, analyzing, 209
installing, 190–191
memory leaks, 206–207, 208
memory profiles, 202–205
overview, 189–190
results
allocated objects, 205
average age, 206
discarding, 199
displaying, 199
generations, 206
live bytes, 205
live objects, 205
taking snapshots, 199, 207–208
supported platforms, 190
terminology, 159
vs. VisualVM, 189
NetBeans Profiler, method profiles. See also Profilers.
Attach Mode, specifying, 193–194
calibrating the target JVM, 196–197
configuring the remote system, 196–197
terminology, 159
controls, 198–199
local vs. remote, specifying, 193–195
remote profiling pack, generating, 194, 196
results
displaying, 201
taking a snapshot, 201–202
sample rate, reducing, 193
starting a session, 191–198
status, 198–199
telemetry, 200–201
views, 200
Network I/O
monitoring, 390–392
System CPU usage, monitoring, 221–222
Network I/O utilization. See also Monitoring network I/O.
bandwidth, 44
blocking vs. nonblocking sockets, 45
improving application performance, 45
overview, 41–42
Neutral state, synchronization, 72
nicstat tool, 42–43
NIO nonblocking data structures, 221–222
No Identity Map option, 510
Nodes, XML documents
creating, 459
expansion, deferring, 460
moving, 459
renaming, 459
Nonstandard command line options, 59
Null hypothesis, 351–353
NUMA (Non-Uniform Memory Architecture) systems
deploying applications on, 315
heap space, 571
numberofavailablethreads-count attribute, 513–514
numberofworkitemsinqueue-current attribute, 513–514

O
-o option, collect tool, 163
Object size vs. cost, best practices, 444
OBJECT_WAIT statement, 75
Offline analysis, garbage collection, 119, 121–125
Old generation garbage collection, enabling, 558
Old generation space
calculating, 112–113, 114
definition, 81
memory footprint, 269–272
size, refining, 283–287
triggering CMS garbage collection, 559–560
utilization, monitoring, 144–145
On Stack Replacements (OSRs), 95
oops (ordinary object pointers), 57, 554
Optimistic locking, 521, 532–533
Optimization decisions, printing, 567
Optimizations, JIT compilers, 93–94
Optimized methods, printing, 565–566
Optimizing away dead code, 329–335
Optimizing loops, 99–100
Oracle Solaris. See Solaris.
Oracle Solaris Studio Performance Analyzer. See Performance Analyzer.
Ordinary object pointers (oops), 57, 554
OSRs (On Stack Replacements), 95
outfile command, 187
OutOfMemoryError
error handling, 78–80
heap dumps, enabling, 567
running commands on error, 568
tuning garbage collectors, 273–274
Overhead
definition, 157
reducing, 91–92
Owning Java monitors, 71–72
er_print tool, 158, 180–189
exiting, 168
experiment files
creating, 163
opening, 168
specifying a directory for, 163
filters, definition, 158
installing, 161–162
modes
Expert, 178
Machine, 178
User, 177–178
new windows, creating, 168
overview, 156–157
printing data, 168
product Web page, 159
supported platforms, 160–161
System CPU time, printing, 182
tabs
Call Tree, 169–171, 246
Callers-Callee, 172–174
Callers-Callees, 169–170
Disassembly, 169–170
Event, 168–169
Experiments, 170
Functions, 169–170, 171–174
Source, 169–170
Summary, 168–169
Timeline, 170, 246–248
terminology, 158
toolbar, 168
User CPU time, printing, 182
viewing mode, switching, 168
Performance Analyzer, experiments. See also
Experiments.
archiving artifacts, 163
call stacks, attributed time, 174–175
collecting data, 162–166, 168
combining, 168
CPU counters, collecting, 163–164
data presentation
APIs, showing/hiding, 168
filtering data, 168, 179–180, 248–249
by function name, 178
lock contention, 176–177
by method name, 177–178
methods, showing/hiding, 168
metrics, 175–176
monitor contention, 177
definition, 158
dropping results from, 168
metrics of interest, 176
printing, 180–189
printing experiment profiles. See also er_print tool.
Callers-Callees tables, 184–185
directory output to a file, 187
filtering, 186–187
limiting methods printed, 183–184
Performance Analyzer, experiments. See also Experiments. (continued)
metrics, specifying, 182–184
scripting, 180, 187–189
splitting commands, 181
System CPU time, 182
User CPU time, 182
view mode, specifying, 187
profiling interval, specifying, 163
toggling data collection on/off, 163
viewing, 166–175
Performance Analyzer, metrics
adding/removing, 175
exclusive time
definition, 158, 160
displaying, 176
inclusive time
definition, 158, 160
displaying, 160
System CPU, 158, 176
User CPU, 158, 176
User Lock, 176
Performance counters, CPU, 49–50
Performance Manager, monitoring
CPU utilization, 16–19
lock contention, 39
memory utilization, 33–34
run queue depth, 29–31
Period (.) keyword, 182
Periodic task threads, 75
Permanent generation garbage collection, 560
Permanent generation space
definition, 81
memory footprint, 269–272
monitoring, 113
size
specifying, 556
triggering CMS garbage collection, 560
utilization, monitoring, 145
Persistent entities, 505–506
Pessimistic locking, EJB best practices, 532–533
pidstat tool, Linux
monitoring involuntary context switching, 41
monitoring lock contention, 38–39
ping utility, 390–391
Platforms, choosing, 9–10
Plus sign (+) keyword, 182
Pre-cleaning phase, 89
Prefetching, EJB best practices, 530–532
Premature promotion, garbage collection, 85
Printing
Callers-Callees tables, 184–185
CPU utilization statistics, 26–27
data, 168
experiment profiles. See also er_print tool.
Callers-Callees tables, 184–185
directory output to a file, 187
filtering, 186–187
limiting methods printed, 183–184
metrics, specifying, 182–184
scripting, 180, 187–189
sorting, 183
splitting commands, 181
System CPU time, 182
User CPU time, 182
view mode, specifying, 187
optimized methods, 325, 565–566
Process thread stack dumps, monitoring, 27
Product Web page, 159
Production mode, Web containers, 408–409
Profilers, 157.
See also NetBeans Profiler;
Performance Analyzer.
Profiles, 157. See also Experiments.
Profiling. See also Experiments; Monitoring;
Tuning.
definition, 14, 108
enterprise applications, 399–400
memory, 156
method, 156
with VisualVM
capabilities, 131, 138
pausing, 138–139
remote, 138–139
Program dependence graphs, 98–100
Programs, developing. See Software development.
Promotion
garbage collection
definition, 81
failure, 85
premature, 117
tuning latency/responsiveness, 291–293
Provider interface, 495–498
prstat tool, Solaris. See also iotop tool.
involuntary context switching, 40–41
monitoring CPU utilization, 26–27
pstack tool, 27
p-value, 353
Q
Query results cache, JPA best practices, 543–544
Quick lock contention, monitoring, 151–153
R
Races, avoiding, 71–72. See also Synchronization.
Ramp down time, 380
Ramp up time, 380
Range check elimination, 99–100
READ_COMMITTED isolation level, 521
Read-only entity beans, EJB best practices,
535–536
Reads without transactions, 550
READ_UNCOMMITTED isolation level, 521
Ready Cache, 516–517
Reference updates in old generation, 91
Register allocation, 94
Register tables, 97
Rejecting a true null hypothesis, 353
Remark pause, 88–89
Remote profiling pack, generating, 194, 196
Repeatability, benchmarking, 380–381
REPEATABLE_READ isolation level, 521
Reporting intervals, setting, 24–26
Request processing, monitoring and tuning, 418–420
Request response codes, monitoring and tuning, 419
Requests, calculating performance metrics, 366
ReRun Last Profiling control, 199
Reset Collected Results control, 199
Resource monitoring, benchmarking, 379–380
Resource pools, monitoring and tuning, 398–399
Response time
  calculating performance metrics, 368–369
  Web services metric, 476
Responsiveness, tuning the JVM, 256. See also Latency/responsiveness.
Results of profiling
  displaying, 201
  method profiles, 201–202
  NetBeans Profiler
    allocated objects, 205
    average age, 206
    discarding, 199
    displaying, 199
    generations, 206
    live bytes, 205
    live objects, 205
    taking snapshots, 199, 201–202, 207–208
RMI server, 411–412
Root method, definition, 159
Round-trip time, calculating performance metrics, 366
Run GC control, 199
Run queue. See CPU, scheduler’s run queue.
Runtime. See HotSpot VM Runtime.
Runtime environment, choosing, 260–261

S

Safe points
  class loaders, 66
  HotSpot VM Runtime
    class loading, 66
    initiating, 76
    thread management, 75–76
    operations, monitoring runtime between, 119–120
    pauses, tuning garbage collectors, 267
    statistics, printing, 564
    VM operations, 75–76
Sample rate, reducing, 193
sar tool, 49
SAX performance, XML documents, 469–470
SAXParser, creating, 455–456
SAXParserFactory class, 456
Scalability
  analysis, 377–378
  ideal CPU utilization, 15
Scaling
  benchmarks, 370–372
  user, 358
  vertical and horizontal, 358, 377
Scavenging young generation space, 110, 306, 561
Schema caching, 461–462
Schema types, effects on Web service performance, 479–483
-script option, 181
Scripting, er_print tools, 180, 187–189
Secure interactions, benchmarking, 359
Security manager, 409–410
Security policies, VisualVM, 133
Self time, 159
Serial GC, 86–87, 92
SERIALIZABLE isolation level, 521
Serialization, best practices, 440–443
Serialize/marshall XML documents. See also Parse/unmarshall XML documents.
  definition, 455
  description, 460
Server JIT, 97–98
-server option, 553
Server runtime environment vs. client, 260
Server-class machines, JIT defaults for, 101–102
Servers, monitoring. See Application server monitoring.
Service availability, benchmarking, 359
Service Oriented Architecture (SOA). See Web services; XML documents.
Servlets, best practices, 427–438
Session beans, 505–506
Session Façade pattern, 529–530
Session maintenance, benchmarking, 359
Session persistence, best practices, 443–445
setStrictErrorChecking attribute, 460
Single static assignment (SSA), 93, 98–100
SINGLE_TABLE inheritance, 550
Single-threaded young generation garbage collection, 111
SJSXSP performance, 469–470
Sliding compacting mark-sweep garbage collection, 86–87
Slow-path code, synchronization, 72
Snapshots. See also Thread dumps.
  NetBeans Profiler, 199, 201–202, 207–208
  NetBeans Profiler results, 199, 207–208
  Take Snapshot control, 199
  VisualVM applications, 132, 139–140
Snapshots of applications
  saving, 139–140
  taking, 132
  viewing, 139–140
SOA (Service Oriented Architecture). See Web services; XML documents.

SOAP messages, Web service performance best practices, 499–501

Soft Cache Weak Identity Map option, 510
Soft Identity Map option, 509
Software development
  bottom up approach, 7–8
  phases of, 2–5. See also specific phases.
  process overview, 3
  top down approach, 6–7
Solaris Performance Analyzer. See Performance Analyzer.
Solaris Performance Tools CD 3.0, 47
Solaris Studio Performance Analyzer. See Performance Analyzer.
sort command, 183
Source tab, 169–170
Space utilization, monitoring, 142–143
Spaces panel, 142–143
SPARC T-series processor
  evaluating performance, 10–11
  hardware threads, 9–10
  long latency CPU events, 11
  monitoring CPU utilization, 52
  multiprocessing, 9–10
  multithreading, 9–10
  Solaris Internals wiki, 51
  thread context switches, 9–10
SSA (single static assignment), 93, 98–100
Stack-loaded state. synchronization, 72
Stalls
  CPU cycles, 15
  SPARC T-series systems, 50–51
Standard command line options, 59
Standard deviations, calculating, 349
Startup time, tuning the JVM, 256–257
Stateful session bean caches, monitoring and tuning, 516
Stateful session beans, 506
Stateless session beans, 506
States, synchronization, 72
Statistics. See also Benchmarking; Experiments.
  \( \alpha \) (alpha), 351–353
  aging, 145–146
  averages, calculating, 349
  benchmarking, 381–382
  confidence intervals, calculating, 350–351
  degrees of freedom, 351–353
  guidelines for using, 354–355
  hypothesis tests, 351–354
kstat tool, 49
null hypothesis, 351–353
performance, collecting, 49
plotting performance, 144–145
printing
  CPU utilization, 26–27
  monitoring CPU utilization, 26–27
  safepoint, 564
tuning garbage collectors, 264
p-value, 353
rejecting a true null hypothesis, 353
safepoint, garbage collection reports, 564
sar tool, 49
standard deviations, calculating, 349
tenuring, garbage collection reports, 563
t-statistics, 351–353
Type I Errors, 353
Steady state time, benchmarking, 380
Stop control, 199
Stop-the-world garbage collection, 76, 558
Stop-the-world pauses, monitoring, 122
StringBuffer, resizing, 235–238
StringBuilder, resizing, 235–238
Studio Performance Analyzer. See Performance Analyzer.
Summary tab, 168–169
Sun Microsystems. See Solaris; SPARC T-series processor.
Superword, 99–100
Supported platforms, 160–161
Survivor spaces
  description, 84–85
  occupancy, 298
  overflows, 145
  size
    after garbage collection, 557–558
    changing, 294–303
    compared to eden space, 290–291, 556
    initial ratio, specifying, 557
    sizing, 294–303
    throughput, tuning, 311–314
    tuning latency/responsiveness, 289–291
    utilization, monitoring, 143, 144
SUT (System Under Test), isolating, 360–361, 378–379
Swapping memory, 32–36
Sweeping, enabling, 560
Synchronization
  biased state, 72
  concurrency, 71
  contended operations, 71–72
  entering a Java monitor, 71–72
  exiting a Java monitor, 71–72
  fast-path code, 72
  inflated state, 72
  Java monitors, 71–72
  mark word, 72
  mutual exclusion, 71
  neutral state, 72
  owning Java monitors, 71–72
  races, avoiding, 71–72
  slow-path code, 72
  stack-loaded state, 72
  states, 72
  synchronized blocks, 71
  uncontended operations, 71
Synchronized blocks, 71
Synchronized HashMap
  lock contention, sample code, 573–583, 603–613
  locking, JVM-System, 225–233
System boundaries, defining for benchmarking, 360–361
System CPU. See also CPU, utilization.
  definition, 15
  profiling, 158, 176
  time, printing, 182
  usage, monitoring
callers-callees, 218–221
I/O, 214–218
network I/O, 221–222
NIO nonblocking data structures, 221–222
overview, 212–222
system keyword, 182
System Under Test (SUT), isolating, 360–361, 378–379
System.currentTimeMillis API, 328–329
System.gc
  full garbage collection, disabling, 110, 561
  invoking CMS cycle vs. stop-the-world, 561
  sample output, 121
  unloading classes, 561
System.nanoTime API, 328–329
T
Take Snapshot control, 199
Task Manager
  monitoring CPU utilization, 16–19
  monitoring involuntary context switching, 41
Telemetry, 200–201
TemplateTable data structure, 69
Tenure, 81
Tenuring
  distribution, monitoring, 117
  maximum threshold, setting, 559
  monitoring, 145–146
  statistics, printing, 563
  threshold, 291–294
Terminology, 158
Thick clients, Web services, 474–476
Thin clients, Web services, 475–476
Think time
  benchmarking, 364, 374–377
  calculating, 366
definition, 366
  enterprise considerations, 364
  performance metrics, calculating, 366
32-bit runtime environment vs. 64-bit runtime environment, 260–261
Thread dump analysis, unanticipated file interactions, 397
Thread dumps. See also Snapshots.
  monitoring, 389–390
  VisualVM, 138
Thread ids, converting to hexadecimal, 27–28
Thread in Java state, 74
Thread in vm state, 74
Thread management
  blocked thread state, 74
  CONDVAR_WAIT statement, 74
  creating threads, 73–74
deadlocks, 80
debugging, 74–75
destroying threads, 73–74
garbage collection threads, 75
internal VM threads, 75
JIT compiler threads, 75
MONITOR_WAIT statement, 74
new thread state, 74
OBJECT_WAIT statement, 75
overview, 72
periodic task threads, 75
safepoints, 75–76
signal dispatcher thread, 75
thread in Java state, 74
thread in Java vm state, 74
thread states, 74–75
threading model, 72–73
VM operations, 75–76
VM threads, 75
Thread pools, monitoring and tuning, 412–414, 512–514
Thread safety, parsing/unmarshalling XML documents, 457
Thread states, 74–75
Threading model, 72–73
Threads control, 200
Throughput
  metric, Web services, 476
  performance metrics, calculating, 369–370
tuning
  adaptive sizing, disabling, 309–311
  CMS, 307–308
  deploying on NUMA systems, 315
garbage collectors, 262, 308–311
  JVM, 256
  overview, 307
  parallel GC threads, 314–315
  survivor spaces, 311–314
Throughput GC. See Parallel GC.
Throwing exceptions, 70–71
Tiered runtime environment, 260
Time and date stamp, printing, 264, 266
Time stamps. See Date and time stamps.
Timeline tab, 170, 246–248
TLABs (Thread-Local Allocation Buffers), 85
Toolbar, 168
Top down software development, 6–7
top tool, 26. See also iotop tool.
Train GC, 92
Transactional Cache, 516–517
Transactions
  attributes, choosing, 523
  container managed vs. bean managed, 522–523
  isolation levels, 521–522
Trimming whitespaces, best practices, 430–431

t-statistics, 351–353

Tuning. See also Experiments.

declaration, 14, 108. See also Monitoring; Profiling.

the file cache, best practices, 446
resource pools, 398–399

Tuning EJB container

bean caches, 514–520
bean pools, 514–520
EclipseLink session cache, 519–520
entity bean caches, 516
invocation patterns, 512
overview, 511
Ready Cache, 516–517
stateful session bean caches, 516

thread pool, 512–514

Transactional Cache, 516–517

Tuning HTTP listener

acceptor threads, 414–417
connection queues, 414–417
elements to be monitored, 412
individual applications, 420–427
keep alive, 414–417
request processing, 418–420
request response codes, 419
thread pools, 412–414

Tuning the JVM

application systemic requirements
availability, 255–256
latency, 256
manageability, 256
memory footprint, 256–257
overview, 255
responsiveness, 256
startup time, 256–257
throughput, 256

application throughput
adaptive sizing, disabling, 309–311
CMS, 307–308
deploying on NUMA systems, 315
garbage collectors, 308–311
overview, 307
parallel GC threads, 314–315

survivor spaces, 311–314
assumptions, 254
deployment model, choosing
multiple JVM deployment, 258–259
overview, 259
single JVM deployment, 258
edge cases, 316
garbage collectors
choosing, 261–262
command line options, 263–267
date stamp, printing, 266
directing output to a file, 264

delay, 262
logging, 263–267
memory footprint, 262

OutofMemoryError, 273–274
performance attributes, 262–263
principles of, 263

safepoint pauses, 267
statistics, printing, 264
throughput, 262
time stamp, printing, 264, 266

latency/responsiveness
CMS (Concurrent Mark-Sweep GC), 287–289
CMS collection cycle, initiating, 298–303
CMS pause time tuning, 305–306
concurrent permanent generation garbage collection, 304–305
explicit garbage collection, 303–304
full garbage collections, 286
garbage collection activities affecting,
278–279
inputs, 279–280
old generation size, refining, 283–287
overview, 278–279
promotion, 291–293
survivor spaces, 289–291
survivor spaces, occupancy, 298
survivor spaces, sizing, 294–303


tenuring threshold, 291–294
young generation size, refining, 280–283
overview, 252–255
ranking systemic requirements, 257–258
runtime environment, choosing
32-bit vs. 64-bit, 260–261
client vs. server, 260
tiered, 260
testing infrastructure requirements, 255
work flow, 253

Tuning the JVM, command line options
biased locking, 318–319
escape analysis, 317–318
garbage collection read/write barriers, eliminating, 318
large pages
Linux, 320–321
Solaris, 319–320
window, 321
object explosion, 317
scalar replacement, 318
synchronization, eliminating, 318
thread stack allocation, 318

Tuning the JVM, determining memory footprint
application total memory, determining, 277
constraints, 268
heap
initial size, configuring, 275–277
layout, 268–272
live data size, calculating, 274–275
size, starting point, 272–274
old generation space, 269–272
overview, 268
permanent generation space, 269–272
young generation space, 269–272

Tuning Web containers
configuration settings, 408–409
development mode, 408–409
garbage collection, 411
HTTP service, 412
JIT compiler tuning, 410
JVM tuning, 410–412
overview, 408
page freshness, checking, 409
production mode, 408–409
security manager, 409–410

Type I Errors, 353
Type safety, class loaders, 65–66

V

-Uncommon traps, 96–97, 98–100
-Uncontended operations, 71
-Uninitialized classes, 98
-Unloaded classes, 98
-Unloading classes, System.gc, 561
-Unrolling loops, 99–100
-Unswitching loops, 99–100

User CPU. See also CPU, utilization.
description, 15
profiling, 158, 176
time, printing, 182

User interaction modeling, benchmarking,
362–366

user keyword, 182
User Lock, 176
User mode, 177–178
User scaling, 358. See also Scaling.
User transactions, calculating performance
metrics, 366, 367–368

W

Waiting for data. See Stalls.
Warm-ups, benchmarking, 324–327, 333–334
Weak generational hypothesis, 81
Web containers
  components, GlassFish
    Coyote connector, 407
    GlassFish, 406–407
    Grizzly connector, 406–407
    HTTP connector, 406–407
    servlet engines, 407–408
  monitoring and tuning
    configuration settings, 408–409
    development mode, 408–409
    garbage collection, 411
    HTTP service, 412
    JIT compiler tuning, 410
    JVM tuning, 410–412
    overview, 408
    page freshness, checking, 409
    production mode, 408–409
    security manager, 409–410
  monitoring and tuning HTTP listener
    acceptor threads, 414–417
    connection queues, 414–417
    elements to be monitored, 412
    individual applications, 420–427
    keep alive, 414–417
    request processing, 418–420
    request response codes, 419
    thread pools, 412–414
Web pages, checking freshness, 409
Web service performance
  best practices
    binary payload, 486–495
    catalog file locations, 502–503
    client performance, 502–503
    Fast Infoset, 499–501
    HTTP compression, 501–502
    MTOM (Message Transmission Optimization Mechanism), 487–495
    overview, 486
    Provider interface, 495–498
    SOAP messages, 499–501
    XML documents, 492
    XML documents as attachments, 492–495
  factors affecting
    dateTime schema, 481–482
    endpoint implementation, 483–484
    handler performance, 484–486
    message size, 477–479
    schema types, 479–483
Web services. See also XML documents.
  benchmark metrics, 476
  benchmarking, 473–476
  response time metric, 476
  thick clients, 474–476
  thin clients, 475–476
  throughput metric, 476
Whitespaces, trimming, 430–431
Woodstox performance, 469–470
Write barriers, 83

X
- -Xbatch, 564–565
- -Xcheck:jni, 568
- -Xcheck:jni method, 78
- -Xloggc, 264, 267, 562
- -Xloggc option, 119
XML documents. See also Web services.
  APIs, selecting, 468–471
  catalog resolvers, 463–464
  DOM performance, 469–470
  encoding in binary format, 499–501
  entity resolvers, 462–464
  external DTD subsets, 462–464
  JAXB (Java API for XML Binding), 454, 469–470
  JAXP (Java API for XML Processing), 454, 457
  JAX-WS RI (JAX-WS Reference Implementation) stack, 471–473
  parsing performance, comparisons, 469–470
  partial processing, 465–468
  resolving external entities, 462–464
  SAX performance, 469–470
  schema caching, 461–462
  sending as attachments, 492–495
  SJSXMP performance, 469–470
  validation, 460–462
  Web service performance, best practices, 492–495
  Woodstox performance, 469–470
XML documents, processing life cycle
  access
    definition, 455
    description, 458–459
  modify
    attributes, checking for and retrieving, 459
    definition, 455
    description, 459–460
    DOM APIs, 459–460
    error checking, 460
    node expansion, deferring, 460
    nodes, creating, renaming and moving, 459
  overview, 454–455
parse/unmarshal
  definition, 455
  description, 455–458
  DocumentBuilder, creating, 455–456
  factory lookup, 456–457
  Factory objects, reusing, 457
  parser, creating, 455–456
  SAXParser, creating, 455–456
  thread safety, 457
  XMLStreamReader, creating, 455–456
serialize/marshal
  definition, 455
  description, 460
XMLInputFactory class, 456
XMLStreamReader, creating, 455–456
- -Xmn, 270, 555
Index

-XX:+PrintCommandLineFlags, 102–103
-XX:+PrintGCDateStamps, 264
-XX:+PrintGCDetails, 264
-XX:+PrintGCTimeStamps, 264
-XX:ScavengeBeforeFullGC, 110
-XX:UseAdaptiveSizePolicy, 309–311, 558
-XX:+AggressiveHeap, 569
-XX:+AggressiveOpts, 317, 568
-XX:+AlwaysPreTouch, 570–571
-XX:+BackgroundCompilation, 564
-XX:+CMSClassUnloadingEnabled, 560
-XX:+CMSIncrementalMode, 561
-XX:+CMSIncrementalPacing, 562
-XX:CMSInitiatingOccupancyFraction, 299–300, 559–560
-XX:CMSInitiatingPermOccupancyFraction, 305, 560
-XX:+CMSPermGenSweepingEnabled, 560
-XX:+CMSScavengeBeforeRemark, 306, 560
-XX:+DisableExplicitGC, 412, 561
-XX:+DoEscapeAnalysis, 317–318, 569
-XX:ErrorFile, 79
-XX:+ExplicitGCInvokesConcurrent, 561
-XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses, 561
-XX:+HeapDumpOnOutOfMemoryError, 567
-XX:HeapDumpPath, 567
-XX:InitialHeapSize, 272
-XX:InitialSurvivorRatio, 557
-XX:LargePageSizeInBytes, 570
-XX:+MaxHeapSize, 272
-XX:+MaxInlineSize, 567
-XX:+MaxNewSize, 270, 555
-XX:+MaxPermSize, 270–271, 276, 556
-XX:+MaxTenuringThreshold, 292–293, 559
-XX:+NewRatio, 555
-XX:+NewSize, 269–270, 555
-XX:+OnError, 79, 568
-XX:+OnOutOfMemoryError, 568
-XX:ParallelGCThreads, 305–306, 559
-XX:+ParallelRefProcEnabled, 561
-XX:PermSize, 270, 276, 556
-XX:+PrintAdaptiveSizePolicy, 310, 563
-XX:+PrintCommandLineFlags, 272, 571
-XX:+PrintCompilation, 325, 565–566
-XX:+PrintFlagsFinal, 572
-XX:+PrintGC, 562
-XX:+PrintGCApplicationConcurrentTime, 120, 267, 564
-XX:+PrintGCApplicationStoppedTime, 120, 267, 564
-XX:+PrintGCDateStamps, 267
-XX:+PrintGCDateStamps, 562
-XX:+PrintGCDetails, 110–111, 267, 389, 562
-XX:+PrintGCTimeStamps
date and time stamps, 118–119
description, 562
garbage collection logging, 267
garbage collection reporting, 117–119
monitoring the JVM, 389
-XX:+PrintInlining, 566–567
-XX:+PrintOptoAssembly, 567
-XX:+PrintSafepointStatistics, 267, 564
-XX:+PrintTenuringDistribution, 293–294, 563
-XX:+ScavengeBeforeFullGC, 561
-XX:+ShowMessageBoxOnError, 568
-XX:SurvivorRatio, 290–291, 556
-XX:+TargetSurvivorRatio, 298, 557–558
-XX:+TieredCompilation, 565
-XX:+UseBiasedLocking, 318–319, 569
-XX:+UseCMSInitiatingOccupancyOnly, 300, 560
-XX:+UseCompressedOops, 554
-XX:+UseConcMarkSweepGC, 559
-XX:+UseLargePages, 319–321, 570
-XX:+UseNUMA, 571
-XX:+UseParallelGC, 272, 558
-XX:+UseParallelOldGC, 272, 558
-XX:+UseParNewGC, 292, 559
-XX:+UseSerialGC, 558

Y

-y option, collect tool, 163

Young generation garbage collection. See also
Minor garbage collection.
definition, 81
DefNew collector, 111
eden space, 83–85
layout, 83–85
monitoring, 109–110
multithreaded, 111
ParNew collector, 111, 559
single-threaded, 111
survivor spaces, 84–85

Young generation space
memory footprint, 269–272
time
size
compared to old generation space, 555
refining, 280–283
specifying, 555