Charlie Hunt · Binu John
Forewords by James Gosling and Steve Wilson

Java™ Performance

The Java Series

FREE SAMPLE CHAPTER

Java™ Performance

Java™ Performance

Charlie Hunt
Binu John

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Hunt, Charlie, 1962-
Java performance / Charlie Hunt, Binu John.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-13-714252-1
ISBN-10: 0-13-714252-8 (pbk. : alk. paper)
1. Java (Computer program language) 2. Computer programming. I. John, Binu, 1967- II. Title.
QA76.73.J38H845 2012
005.13'3—dc23 2011031889

ISBN-13: 978-0-13-714252-1
ISBN-10: 0-13-714252-8

Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
Third printing, January 2013

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

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

Foreword

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*

Preface

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.

Acknowledgments

Charlie Hunt

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 Doech, 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.

About the Authors

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

6

Java Application Profiling Tips and Tricks

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.

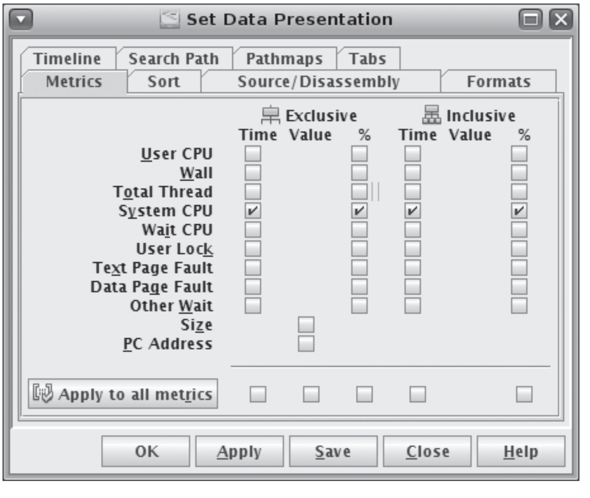
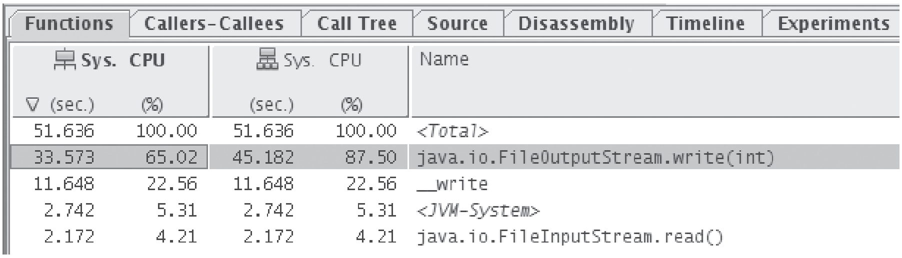


Figure 6-1 Set system CPU data presentation



Functions | Callers-Callees | Call Tree | Source | Disassembly | Timeline | Experiments

Sys. CPU ∇ (sec.)	Sys. CPU (%)	Sys. CPU (sec.)	Sys. CPU (%)	Name
51.636	100.00	51.636	100.00	<Total>
33.573	65.02	45.182	87.50	java.io.FileOutputStream.write(int)
11.648	22.56	11.648	22.56	__write
2.742	5.31	2.742	5.31	<JVM-System>
2.172	4.21	2.172	4.21	java.io.FileInputStream.read()

Figure 6-2 Exclusive system CPU

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:

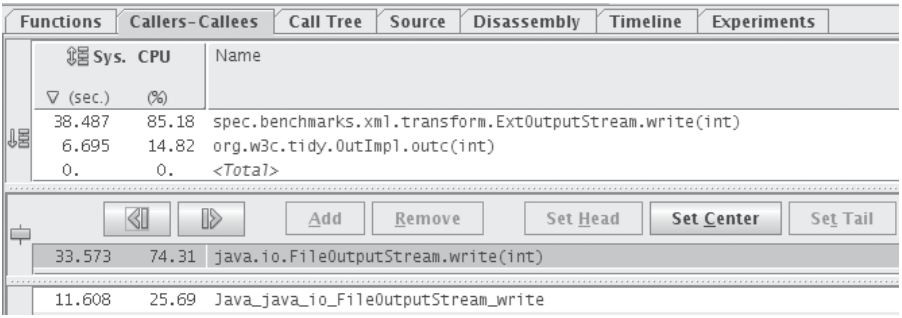


Figure 6-3 FileOutputStream.write(int) callers and callees

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

A look at the implementation of `super.write(b)` shows it is not a call to `FileOutputStream.write(int)`:

```
public void write(int b) throws IOException {
    crc = crc * 33 + b;
}
```

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

```
private FileOutputStream writer;
```

And it is initialized without any type of buffering:

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

`currentFileName` is a field declared as a `String`:

```
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:

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

Then chain a `BufferedOutputStream` and `FileOutputStream` at initialization time:

```
// 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 `BufferOutputStream` has a `write()` method that buffers bytes written to it. This `ExtOutputStream.write(int b)` method is shown here:

```
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:

```
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();
```

```
        }
    }
    public void checkResult(int loopNumber) {
        try {
            writer.flush();
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        check(validiationProperties.getProperty(propertyName));
        outProperties.put(propertyName, "" + getCRC());
        reset();
    }
```

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

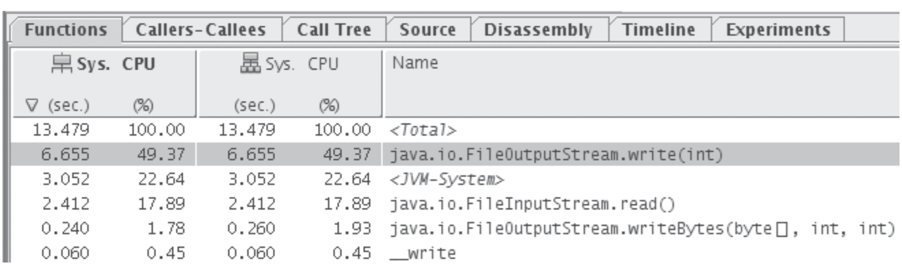
1. A `BufferedOutputStream` can also be allocated with an optional buffered size. The default buffer size, as of Java 6, is 8192. If the application you are profiling is writing a large number of bytes, you might consider specifying an explicit size larger than 8192. If you specify an explicit size, 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 constrast, 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.



Sys. CPU (sec.)	Sys. CPU (%)	Sys. CPU (sec.)	Sys. CPU (%)	Name
13.479	100.00	13.479	100.00	<Total>
6.655	49.37	6.655	49.37	java.io.FileOutputStream.write(int)
3.052	22.64	3.052	22.64	<JVM-System>
2.412	17.89	2.412	17.89	java.io.FileInputStream.read()
0.240	1.78	0.260	1.93	java.io.FileOutputStream.writeBytes(byte[], int, int)
0.060	0.45	0.060	0.45	__write

Figure 6-4 Reduced system CPU utilization

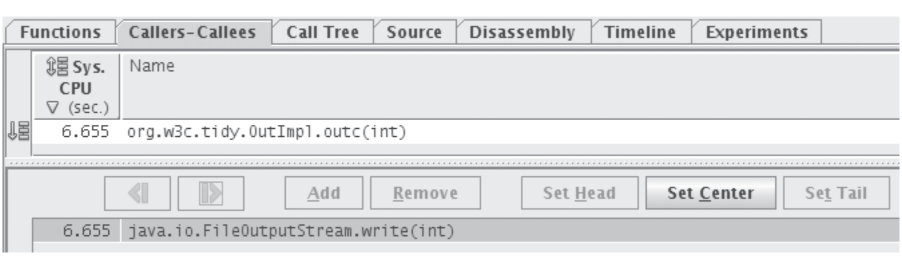


Figure 6-5 Callers-Callees after changes

Comparing the Callers-Callees in Figure 6-5, after the changes to `ExtStream Output`, 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 `Buffered-OutputSteam.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 `ExtStreamOut-put`. 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 `Buff-eredOutputStream.flushBuffer()` is its callee; see Figure 6-7.

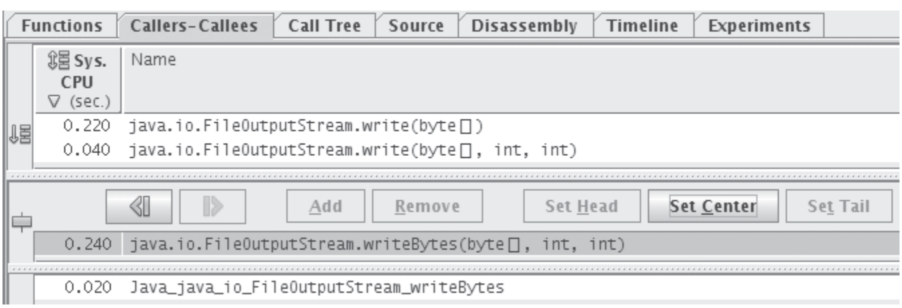


Figure 6-6 Callers-Callees of FileOutputStream.writeBytes(byte[],int,int)

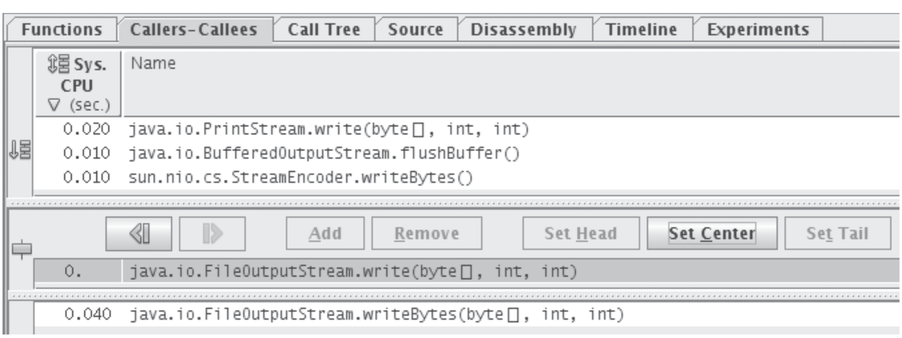


Figure 6-7 Callers-Callees of FileOutputStream.writeBytes(byte[], int, int)

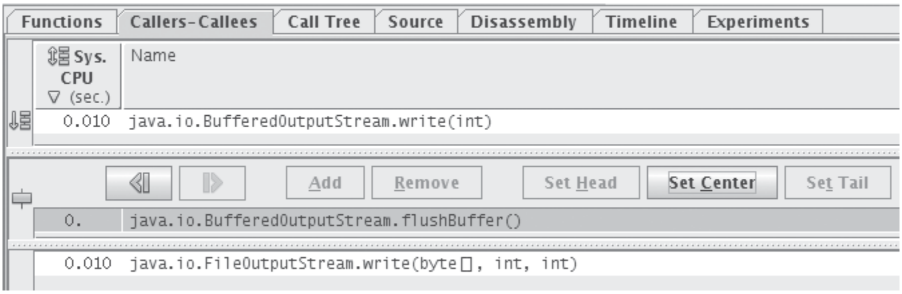


Figure 6-8 Callers-Callees of BufferedOutputStream.flushBuffer()

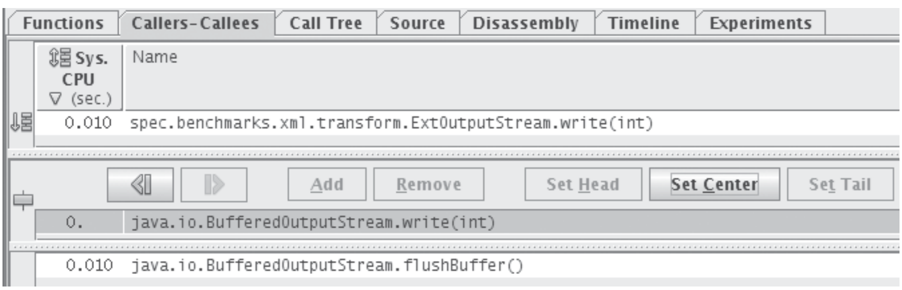


Figure 6-9 Callers-Callees of BufferedOutputStream.write(int)

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).

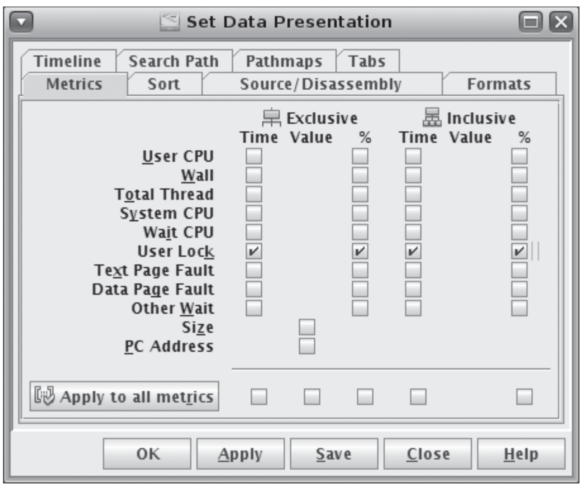
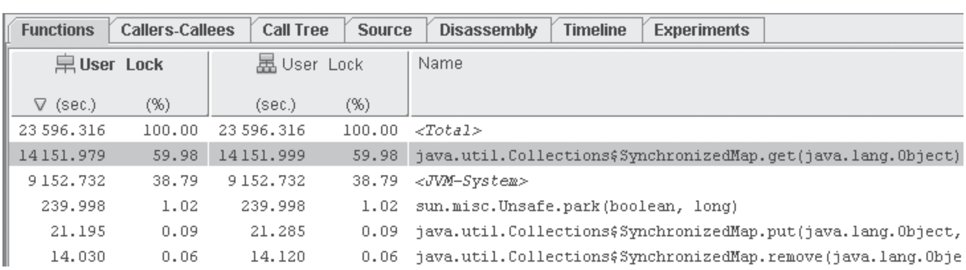


Figure 6-10 Set user lock data presentation



User Lock ▽ (sec.)	User Lock (%)	User Lock (sec.)	User Lock (%)	Name
23 596.316	100.00	23 596.316	100.00	<Total>
14 151.979	59.98	14 151.999	59.98	java.util.Collections$SynchronizedMap.get(java.lang.Object)
9 152.732	38.79	9 152.732	38.79	<JVM-System>
239.998	1.02	239.998	1.02	sun.misc.Unsafe.park(boolean, long)
21.195	0.09	21.285	0.09	java.util.Collections$SynchronizedMap.put(java.lang.Object,
14.030	0.06	14.120	0.06	java.util.Collections$SynchronizedMap.remove(java.lang.Obje

Figure 6-11 Java monitors/locks ordered by exclusive metric

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.

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 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.

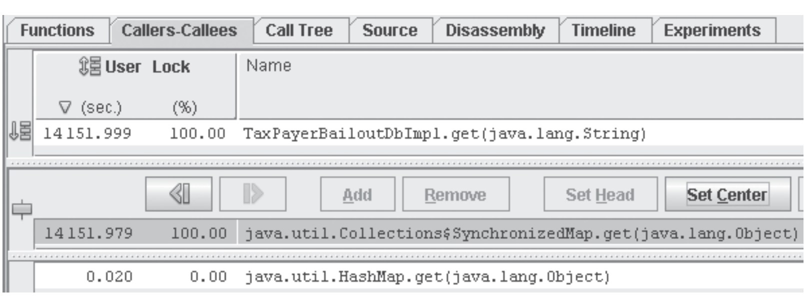


Figure 6-12 Callers-Callees of synchronized HashMap.get()

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)`.

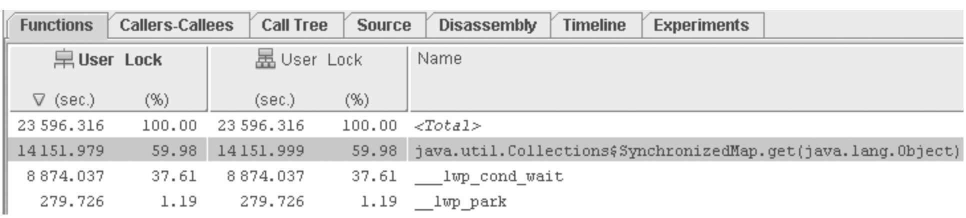


Figure 6-13 Switching from User mode to Expert mode

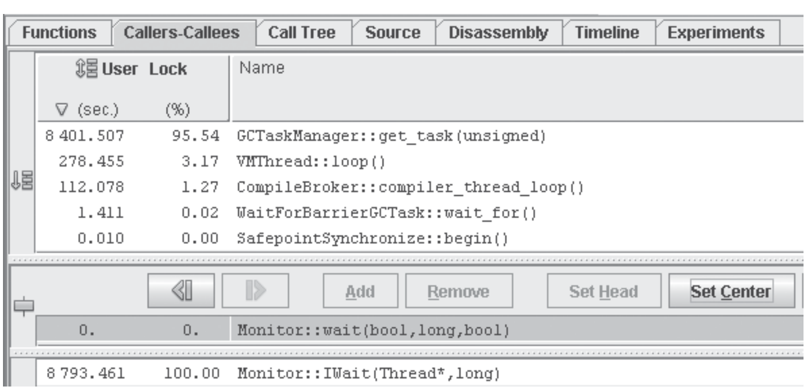


Figure 6-14 Traversing up the call stack of callers of __lwp_cond_wait

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 `VMThread::loop()` method blocks on a queue for work to do on behalf of the Java HotSpot VM. You can think of the `VMThread` as the "kernel thread" of the Java HotSpot VM. The `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 `HashMap` or the use of a `java.util.Hashtable`, the predecessor to the synchronized `HashMap`, is to migrate to using a `java.util.concurrent.ConcurrentHashMap`.[1] Following this practice and executing this program using a `ConcurrentHashMap` instead of a synchronized `HashMap` showed an increase of CPU utilization of 92%. In other words, the previous implementation that used a synchronized `HashMap` had a total CPU utilization of 8% while the `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 `ConcurrentHashMap` implementation increased by a little over 2x to 1,315,000, up from 615,000 with the synchronized `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.

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 `java.util.Random.next(int)` and `java.util.concurrent.atomic.AtomicLong.compareAndSet(long, long)`.

Using the Callers-Callees tab to observe the callers of the `java.util.concurrent.atomic.AtomicLong.compareAndSet(long,log)` method shows `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.

---

1. `java.util.concurrent.ConcurrentHashMap` was introduced in the Java 5 SE class libraries and is available in Java 5 and later Java JDKs/JREs.

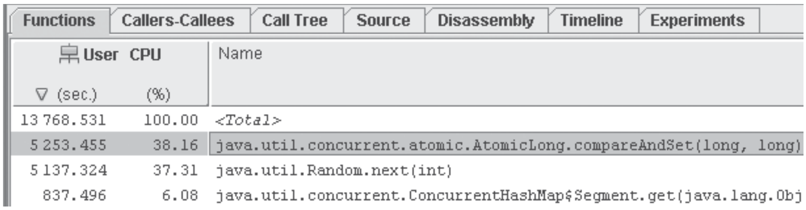


Figure 6-15 Hot methods in the ConcurrentHashMap implementation of the program

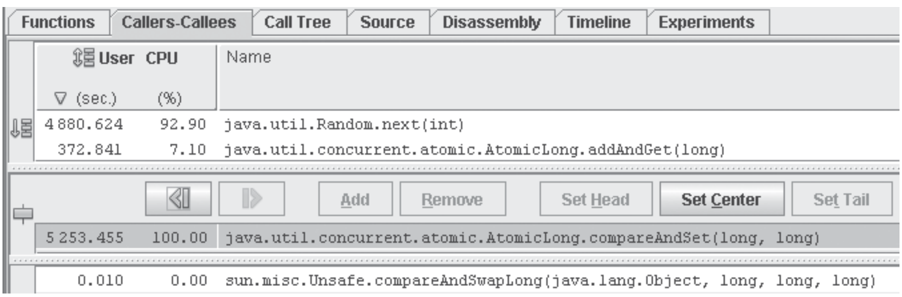


Figure 6-16 Callers of AtomicLong.compareAndSet

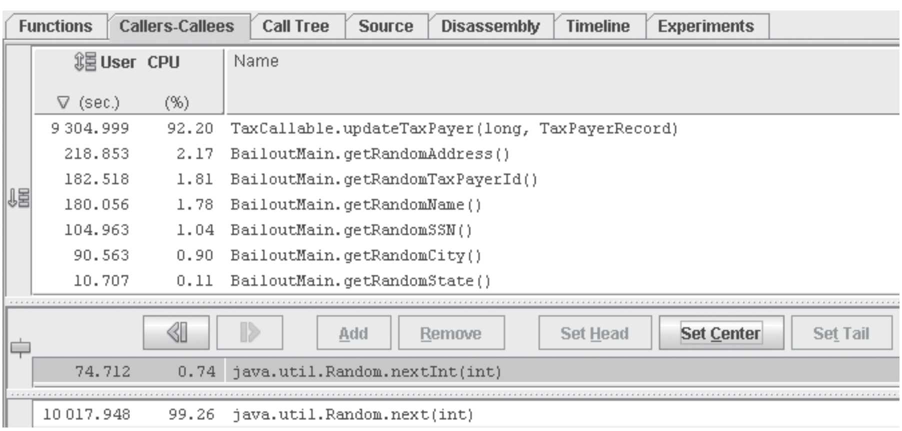


Figure 6-17 Callers and callees of Random.nextInt(int)

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:

```
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)`).

```
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:

```
// 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:

```
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:

```
        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 HotSpot 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:

```
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

Class Name – Live Allocated Objects	Live Bytes ▾	Live Bytes	Live Objects	Allocated Objects
char[]		118,792,912 B (63.6%)	2,460,612 (44.9%)	2,926,057
java.lang.**String**		30,427,464 B (16.3%)	1,267,811 (23.1%)	1,267,815
java.lang.**StringBuilder**		14,524,080 B (7.8%)	907,755 (16.6%)	1,266,550
TaxPayerRecord		8,453,440 B (4.5%)	211,336 (3.9%)	211,336
java.util.**HashMap$Entry**		5,067,552 B (2.7%)	211,148 (3.9%)	211,148
java.util.concurrent.atomic.**AtomicLong**		3,380,416 B (1.8%)	211,276 (3.9%)	211,276
StateAndId		3,378,960 B (1.8%)	211,185 (3.9%)	211,185
java.util.**HashMap$Entry[]**		2,868,112 B (1.5%)	37 (0%)	67
byte[]		11,416 B (0%)	6 (0%)	8

Figure 6-18 Heap profile

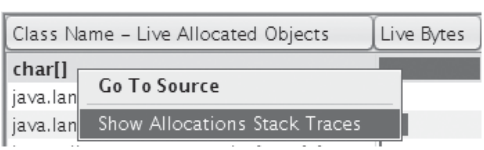


Figure 6-19 Showing allocation stack traces

Method Name – Allocation Call Tree	Live Bytes...	Live Objects ▾	Allocated Objects
char[]		2,460,612 (100%)	2,926,048
java.util.Arrays.**copyOfRange** (char[], int, int)		1,268,072 (51.5%)	1,268,073
java.lang.AbstractStringBuilder.**<init>** (int)		911,056 (37%)	1,266,982
java.util.Arrays.**copyOf** (char[], int)		281,480 (11.4%)	390,988
java.lang.AbstractStringBuilder.**expandCapacity** (int)		281,480 (11.4%)	390,988
java.lang.AbstractStringBuilder.**append** (char)		281,476 (11.4%)	390,984
java.lang.AbstractStringBuilder.**append** (String)		4 (0%)	4

Figure 6-20 char[] allocations from expanding StringBuilders

called from `AbstractStringBuilder.append(char)` and `AbstractString-Builder.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)`.

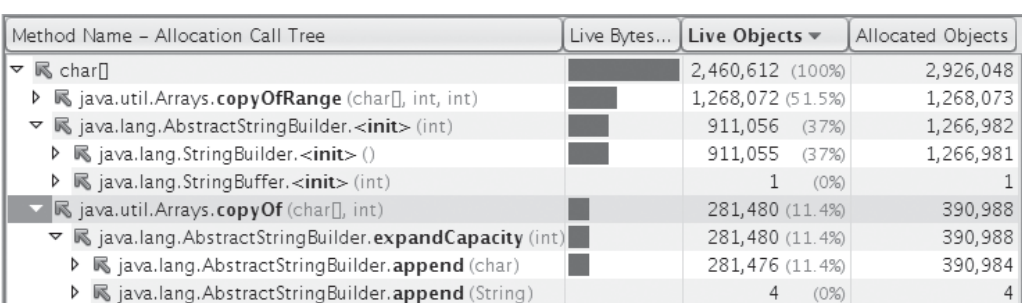
You can also see from Figure 6-20, over 11% of the current live `char[]` objects are from resized `StringBuilder` `char[]`. In addition, there are a total of 2,926,048 `char[]` objects that have been allocated, and of those, 390,988 `char[]` allocations occurred as a result of `StringBuilder` `char[]` 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



Method Name – Allocation Call Tree	Live Bytes...	Live Objects ▾	Allocated Objects
char[]		2,460,612 (100%)	2,926,048
java.util.Arrays.**copyOfRange** (char[], int, int)		1,268,072 (51.5%)	1,268,073
java.lang.AbstractStringBuilder.**<init>** (int)		911,056 (37%)	1,266,982
java.lang.StringBuilder.**<init>** ()		911,055 (37%)	1,266,981
java.lang.StringBuffer.**<init>** (int)		1 (0%)	1
java.util.Arrays.**copyOf** (char[], int)		281,480 (11.4%)	390,988
java.lang.AbstractStringBuilder.**expandCapacity** (int)		281,480 (11.4%)	390,988
java.lang.AbstractStringBuilder.**append** (char)		281,476 (11.4%)	390,984
java.lang.AbstractStringBuilder.**append** (String)		4 (0%)	4

Figure 6-21 Uses of StringBuilder default constructor

```
public static String getRandomTaxPayerId() {
    StringBuilder sb = new StringBuilder(20);
    for (int i = 0; i < 20; i++) {
        int index =
            threadLocalRandom.get().nextInt(alphabet.length);
        sb.append(alphabet[index]);
    }
    return sb.toString();
}

public static String getRandomAddress() {
    StringBuilder sb = new StringBuilder(24);
    int size = threadLocalRandom.get().nextInt(14) + 10;
    for (int i = 0; i < size; i++) {
        if (i < 5) {
            int x = threadLocalRandom.get().nextInt(8);
            sb.append(x + 1);
        }
        int index =
             threadLocalRandom.get().nextInt(alphabet.length);
        char c = alphabet[index];
        if (i == 5) {
            c = Character.toUpperCase(c);
        }
        sb.append(c);
    }
    return sb.toString();
}
```

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:

```
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.

```
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:

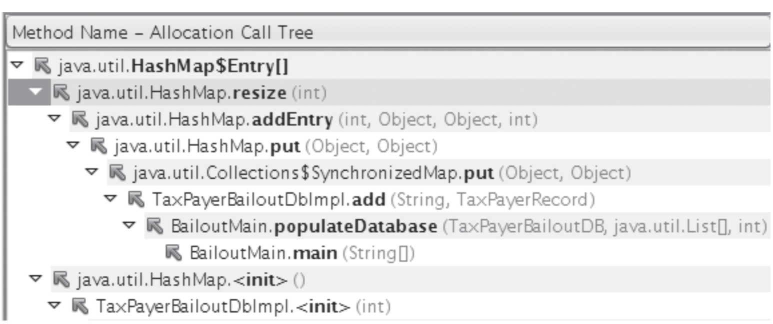


Figure 6-22 HashMap.Entry[] allocation stack traces

```
    private final Map<String, Map<String,TaxPayerRecord>> db;
    private final int dbSize = 2000000;

    public TaxPayerBailoutDbImpl(int dbSize, int numberOfStates) {
        final int outerMapSize = (int) Math.ceil(numberOfStates / .75);
        final int innerMapSize =
                (int) (Math.ceil((dbSize / numberOfStates) / .75));
        db =
           new HashMap<String,Map<String,TaxPayerRecord>>(outerMapSize);
        for (int i = 0; i < numberOfStates; i++) {
            Map<String,TaxPayerRecord> map =
                 Collections.synchronizedMap(
                     new HashMap<String,TaxPayerRecord>(innerMapSize));
            db.put(BailoutMain.states[i], map);
        }
    }
```

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 `HashMaps` 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 HashMaps.

Updating this program variant with the changes described in this section to address both `StringBuilder` and `HashMap` resizing and running on the same UltraSPARC 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 multithreaded 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"):

```
// 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
int numberOfThreads = System.availableProcessors();
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);
```

```
            int index = i % taxPayerIdList.length;
            taxPayerIdList[index].add(key);
        }
        DbInitializerFuture future = new DbInitializerFuture();
        future.addToRecordsCreated(dbSize);
        return future;
    }
```

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.

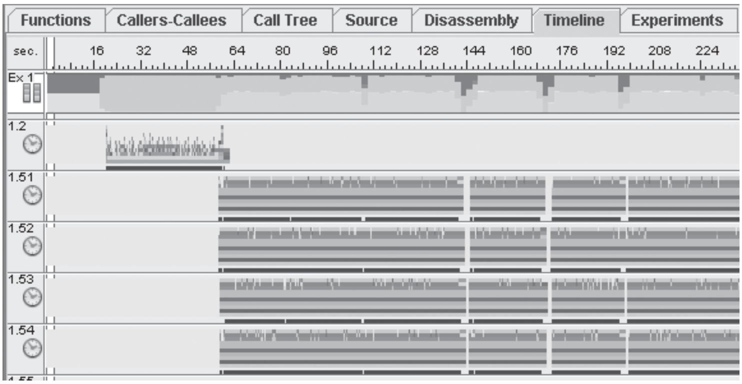


Figure 6-23 Performance analyzer timeline view

Other Useful Analyzer Tips

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

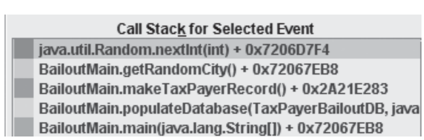


Figure 6-24 Performance analyzer's call stack for selected event panel

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.

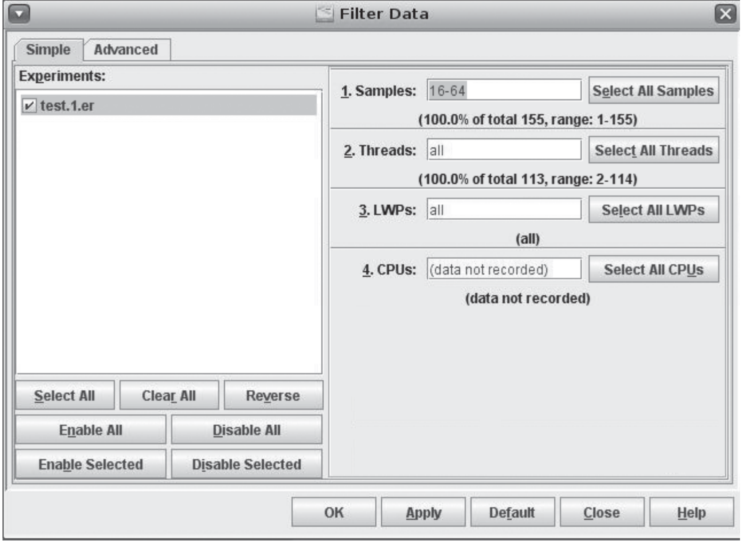


Figure 6-25 Filtering the range of samples to view in performance analyzer

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

Keegan, Patrick, et al., *NetBeans IDE field guide: developing desktop, web, enterprise, and mobile applications,* 2nd Edition. Sun Microsystems, Inc., Santa Clara, CA, 2006.

Oracle Solaris Studio 12.2: Performance Analyzer. Oracle Corporation. http://dlc.sun.com/pdf/821-1379/821-1379.pdf.

JSR-133: Java Memory Model and Thread Specification. JSR-133 Expert Group. http://jcp.org/en/jsr/summary?id=133.

The Java Memory Model. Dr. Bill Pugh. http://www.cs.umd.edu/~pugh/java/memoryModel/.

Index

D

E

I

J

P

Q

R

U

V

W

Y