

Optimizing Linux® Performance: A Hands-On Guide to Linux® Performance Tools

By Phillip G. Ezolt

.....
Publisher: **Pearson PTR**
Pub Date: **March 14, 2005**
Print ISBN: **0-13-148682-9**
Pages: **384**

[Table of Contents](#) | [Index](#)

Overview

The first comprehensive, expert guide for end-to-end Linux application optimization. Learn to choose the right tools—and use them to improve application performance is more crucial than ever—and in today's complex production environments, it's tougher to ensure, code access, plus an exceptional array of optimization tools. But the tools are scattered across the Internet. Many are poorly documented, and many are not Linux-specific. Now, one of those experts has written the definitive Linux tuning primer for every professional: *Optimizing Linux® Performance: A Hands-On Guide to Linux® Performance Tools*. This book covers each of today's most important Linux optimization tools, showing how they fit into a proven methodology for perfecting overall system performance. You'll learn how to pinpoint exact lines of source code that are impacting performance. He teaches sysadmins and application developers how to find solutions more quickly. You'll discover how to: Identify bottlenecks even if you're not familiar with the underlying system. Find the meaning of the events you're measuring. Optimize system CPU, user CPU, memory, network I/O, and disk I/O—and understand how to apply these optimizations to your own environment. Install and use oprofile, the advanced system profiler. Get a clear and practical introduction to all the principles and strategies you'll need. If you're migrating to Linux, you'll quickly master Linux. If you have a Linux background or environment, this book can help you improve the performance of all your Linux applications—increasing business productivity.



Optimizing Linux® Performance: A Hands-On Guide to Linux® Performance Tools

By Phillip G. Ezolt

.....
Publisher: **Pearson PTR**
Pub Date: **March 14, 2005**
Print ISBN: **0-13-148682-9**
Pages: **384**

[Table of Contents](#) | [Index](#)

Copyright

Hewlett-Packard® Professional Books

Preface

Why Is Performance Important?

Linux: Strengths and Weakness

How Can This Book Help You?

Why Learn How to Use Performance Tools?

Can I Tune for Performance?

Who Should Read This Book?

How Is This Book Organized?

Acknowledgments

About the Author

Chapter 1. Performance Hunting Tips

Section 1.1. General Tips

Section 1.2. Outline of a Performance Investigation

Section 1.3. Chapter Summary

Chapter 2. Performance Tools: System CPU

Section 2.1. CPU Performance Statistics

Section 2.2. Linux Performance Tools: CPU

Section 2.3. Chapter Summary

Chapter 3. Performance Tools: System Memory

Section 3.1. Memory Performance Statistics

Section 3.2. Linux Performance Tools: CPU and Memory

Section 3.3. Chapter Summary

Chapter 4. Performance Tools: Process-Specific CPU

Section 4.1. Process Performance Statistics

Section 4.2. The Tools

Section 4.3. Chapter Summary

Chapter 5. Performance Tools: Process-Specific Memory

Section 5.1. Linux Memory Subsystem

Section 5.2. Memory Performance Tools

Section 5.3. Chapter Summary

Chapter 6. Performance Tools: Disk I/O

Section 6.1. Introduction to Disk I/O

Section 6.2. Disk I/O Performance Tools

Section 6.3. What's Missing?

Section 6.4. Chapter Summary

Chapter 7. Performance Tools: Network

Section 7.1. Introduction to Network I/O

Section 7.2. Network Performance Tools

Section 7.3. Chapter Summary

Chapter 8. Utility Tools: Performance Tool Helpers

Section 8.1. Performance Tool Helpers

Section 8.2. Tools

Section 8.3. Chapter Summary

Chapter 9. Using Performance Tools to Find Problems

Section 9.1. Not Always a Silver Bullet

Section 9.2. Starting the Hunt

Section 9.3. Optimizing an Application

Section 9.4. Optimizing a System

Section 9.5. Optimizing Process CPU Usage

Section 9.6. Optimizing Memory Usage

Section 9.7. Optimizing Disk I/O Usage

Section 9.8. Optimizing Network I/O Usage

Section 9.9. The End

Section 9.10. Chapter Summary

Chapter 10. Performance Hunt 1: A CPU-Bound Application (GIMP)

Section 10.1. CPU-Bound Application

Section 10.2. Identify a Problem

Section 10.3. Find a Baseline/Set a Goal

Section 10.4. Configure the Application for the Performance Hunt

Section 10.5. Install and Configure Performance Tools

Section 10.6. Run Application and Performance Tools

Section 10.7. Analyze the Results

Section 10.8. Jump to the Web

Section 10.9. Increase the Image Cache

Section 10.10. Hitting a (Tiled) Wall

Section 10.11. Solving the Problem

Section 10.12. Verify Correctness?

Section 10.13. Next Steps

Section 10.14. Chapter Summary

Chapter 11. Performance Hunt 2: A Latency-Sensitive Application (nautilus)

Section 11.1. A Latency-Sensitive Application

Section 11.2. Identify a Problem

Section 11.3. Find a Baseline/Set a Goal

Section 11.4. Configure the Application for the Performance Hunt

Section 11.5. Install and Configure Performance Tools

Section 11.6. Run Application and Performance Tools

Section 11.7. Compile and Examine the Source

Section 11.8. Using gdb to Generate Call Traces

Section 11.9. Finding the Time Differences

Section 11.10. Trying a Possible Solution

Section 11.11. Chapter Summary

Chapter 12. Performance Hunt 3: The System-Wide Slowdown (prelink)

Section 12.1. Investigating a System-Wide Slowdown

Section 12.2. Identify a Problem

Section 12.3. Find a Baseline/Set a Goal

Section 12.4. Configure the Application for the Performance Hunt

Section 12.5. Install and Configure Performance Tools

Section 12.6. Run Application and Performance Tools

Section 12.7. Simulating a Solution

Section 12.8. Reporting the Problem

Section 12.9. Testing the Solution

Section 12.10. Chapter Summary

Chapter 13. Performance Tools: What's Next?

Section 13.1. The State of Linux Tools

Section 13.2. What Tools Does Linux Still Need?

Section 13.3. Performance Tuning on Linux

Section 13.4. Chapter Summary

Appendix A. Performance Tool Locations

Appendix B. Installing oprofile

B.1 Fedora Core 2 (FC2)

B.2 Enterprise Linux 3 (EL3)

B.3 SUSE 9.1

Index



< Day Day Up >



Copyright

www.hp.com/hpbooks

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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 U. S., please contact:

International Sales

international@pearsoned.com

Visit us on the Web: www.phptr.com

Library of Congress Number: 2004117118

Copyright © 2005 Hewlett-Packard Development Company, L.P.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.

Rights and Contracts Department

One Lake Street

Upper Saddle River, NJ 07458

Text printed in the United States on recycled paper at RR Donnelley & Sons Company in Crawfordsville, IN

First printing, March 2005

Dedication

This book is dedicated to my wife Sarah, (the best in the world), who gave up so many weekends to make this book possible. Thank you, Thank you, Thank you!

 PREV

< Day Day Up >

NEXT 

Hewlett-Packard® Professional Books

HP-UX

Cooper/Moore

HP-UX 11i Internals

Fernandez

Configuring CDE

Keenan

HP-UX CSE: Official Study Guide and Desk Reference

Madell

Disk and File Management Tasks on HP-UX

Olker

Optimizing NFS Performance

Poniatowski

HP-UX 11i Virtual Partitions

Poniatowski

HP-UX 11i System Administration Handbook and Toolkit, Second Edition

Poniatowski

The HP-UX 11.x System Administration Handbook and Toolkit

Poniatowski

HP-UX 11.x System Administration "How To" Book

Poniatowski

HP-UX 10.x System Administration "How To" Book

Poniatowski

HP-UX System Administration Handbook and Toolkit

Poniatowski

Learning the HP-UX Operating System

Rehman

HP-UX CSA: Official Study Guide and Desk Reference

Sauers/Ruemmler/Weygant

HP-UX 11i Tuning and Performance

Weygant

Clusters for High Availability, Second Edition

Wong

HP-UX 11i Security

UNIX, LINUX

Mosberger/Eranian

IA-64 Linux Kernel

Poniatowski

Linux on HP Integrity Servers

Poniatowski

UNIX User's Handbook, Second Edition

 PREV

< Day Day Up >

NEXT 

Preface

Why Is Performance Important?

Linux: Strengths and Weakness

How Can This Book Help You?

Why Learn How to Use Performance Tools?

Can I Tune for Performance?

Who Should Read This Book?

How Is This Book Organized?

Why Is Performance Important?

If you have ever sat waiting for a computer to do something, (while pounding on your desk, cursing, and wondering, "What is *taking* so long?"), you know why it is important to have a fast and well-tuned computer system. Although not all performance problems can be easily solved, understanding *why* things are slow can mean the difference between fixing the problem in software, upgrading the slow hardware, or simply throwing the whole computer out the window. Fortunately, most operating systems, Linux in particular, provide the tools to figure out why the machine runs slowly. By using a few basic tools, you can determine where the system is slowing down and fix the parts that are running inefficiently.

Although a slow system is particularly annoying to end users, application developers have an even more important reason to performance tune their applications: An efficient application runs on more systems. If you write sluggish applications that need a zippy computer, you eliminate customers who have slower computers. After all, not everyone has the latest hardware. A well-tuned application is usable by more customers, resulting in a bigger potential user base. In addition, if potential customers must choose between two different applications with similar functionality, they often choose the one that runs faster or is more efficient. Finally, a long-lived application likely goes through several rounds of optimization to cope with different customer demands, so it is crucial to know how to track down performance problems.

If you are a system administrator, you have a responsibility to the users of the system to make sure that it runs at an adequate performance level. If the system runs slowly, users complain. If you can determine the problem and fix it quickly, they stop complaining. As a bonus, if you can solve their problem by tuning the application or operating system (and thus keep them from having to buy new hardware), you make company bean counters happy. Knowing how to effectively use performance tools can mean the difference between spending days or spending hours on a performance problem.

Linux: Strengths and Weakness

If you use Linux, maintain it, and develop on it, you are in a strange but good situation. You have unprecedented access to source code, developers, and mailing lists, which often document design decisions years after they are made. Linux is an excellent environment in which to find and fix performance problems. This contrasts a proprietary environment, where it can be difficult to get direct access to software developers, may be hard to find written discussions about most design decisions, and is nearly impossible to access source code. In addition to this productive environment, Linux also has powerful performance tools that enable you to find and fix performance problems. These tools rival their proprietary counterparts.

Even with these impressive benefits, the Linux ecosystem still has challenges to overcome. Linux performance tools are scattered everywhere. Different groups with different aims develop the tools, and as a result, the tools are not necessarily in a centralized location. Some tools are included in standard Linux distributions, such as Red Hat, SUSE, and Debian; others are scattered throughout the Internet. If you're trying to solve a performance problem, you first have to know that the tools you need exist, and then figure out where to find them. Because no single Linux performance tool solves every type of performance problem, you also must figure out how to use them jointly to determine what is broken. This can be a bit of an art, but becomes easier with experience. Although most of the general strategies can be documented, Linux does not have any guide that tells you how to aggregate performance tools to actually solve a problem. Most of the tools or subsystems have information about tuning the particular subsystem, but not how to use them with other tools. Many performance problems span several areas of the system, and unless you know how to use the tools collectively, you will not be able to fix the problem.

How Can This Book Help You?

You will learn many things from this book, including the following:

- What the various performance tools measure
- How to use each tool
- How to combine the tools to solve a performance problem
- How to start with a poorly performing system and pinpoint the problem
- How the methods are used to solve real-world problems (case studies)

Using the methods in this book, you can make a well-organized and diagnosed problem description that you can pass on to the original developers. If you're lucky, they will solve the problem for you.

Why Learn How to Use Performance Tools?

Why should anyone put effort into tuning a system or application?

- A well-tuned system can do more work with fewer resources.
- A well-tuned application can run on older hardware.
- A well-tuned desktop can save users time.
- A well-tuned server provides a higher service quality for more users.

If you know how to effectively diagnose performance problems, you can take a targeted approach to solving the problem instead of just taking a shot in the dark and hoping that it works. If you are an application developer, this means that you can quickly figure out what piece of code is causing the problem. If you are a system administrator, it means that you can figure out what part of the system needs to be tuned, or upgraded, without wasting time unsuccessfully trying different solutions. If you are an end user, you can figure out which applications are lagging and report the problem to the developers (or update your hardware, if necessary).

Linux has reached a crossroads. Most of the functionality for a highly productive system is already complete. The next evolutionary step is for Linux and its applications to be tuned to compete with and surpass the performance of other operating systems. Some of this performance optimization has already begun. For example, the SAMBA, Apache, and TUX Web server projects have, through significant time investments, tuned and optimized the system and code. Other performance optimizations—such as the Native POSIX Thread Library (NPTL), which dramatically improves threading performance; and object prelinking, which improves application startup time—are just starting to be integrated into Linux. Linux is ripe for performance improvements.

Can I Tune for Performance?

The best thing about performance optimization is that you do not need to know the details of the entire application or system to effectively fix performance problems. Performance optimization requires a complementary set of skills to those of a typical application developer.

You need to be observant and persistent. It takes more of a detective than a programmer to hunt down and eliminate performance problems. It is exhilarating to find and fix these. When you start, performance is terrible. However, you track down the cause, rip it out by the roots, and, if you're lucky, the system then runs twice as fast. *Voilà!*

To get to the *voilà*, you must understand the powerful but sometimes confusing world of Linux performance tools. This takes some work, but in the end, it is worth it. The tools can show you aspects of your application and system that you never expected to see.

Who Should Read This Book?

This book helps Linux software developers, system administrators, and end users to use the Linux performance tools to find the performance problems in a given system. Beginning performance investigators learn the basics of performance investigation and analysis. Medium to advanced performance investigators, especially those with performance experience on other proprietary operating systems, learn about the Linux equivalents of commands from other systems with which they may already be familiar.

Software developers learn how to pinpoint the exact line of source code that causes a performance problem. System administrators who are performance tuning a system learn about the tools that show why a system is slowing down, and they can then use that information to tune the system. Finally, although not the primary focus of the book, end users learn the basic skills necessary to figure out which applications are consuming all the system resources.

How Is This Book Organized?

This book teaches an audience of various levels of experience to find and fix performance problems. To accomplish this, the chapters are presented so that you can pick and choose to read different parts of the book without reading the entire book straight through.

[Chapter 1](#) is devoted to the basic methods of performance problem hunting. It contains a series of non-Linux-specific tips and suggestions that prove useful for tracking down performance issues. These guidelines are general suggestions for performance problem hunting and can be applied to tracking down performance issues on any type of computer system.

[Chapters 2](#) through [8](#) (the bulk of this book) cover the various tools available to measure different performance statistics on a Linux system. These chapters explain what various tools measure, how they are invoked, and provide an example of each tool being used. Each chapter demonstrates tools that measure aspects of different Linux subsystems, such as system CPU, user CPU, memory, network I/O, and disk I/O. If a tool measures aspects of more than one subsystem, it is presented in more than one chapter. Each chapter describes multiple tools, but only the appropriate tool options for a particular subsystem are presented in a given chapter. The descriptions follow this format:

1. *Introduction*— This section explains what the tool is meant to measure and how it operates.
2. *Performance tool options*— This section does not just rehash the tool's documentation. Instead, it explains which options are relevant to the current topic and what those options mean. For example, some performance tool man pages identify the events that a tool measures but do not explain what the events mean. This section explains the meaning of the events and how they are relevant to the current subsystem.
3. *Example*— This section provides one or more examples of the tool being used to measure performance statistics. This section shows the tool being invoked and any output that it generates.

[Chapter 9](#) is Linux specific and contains a series of steps to use when confronted with a slow-performing Linux system. It explains how to use the previously described Linux performance tools in concert to pinpoint the cause of the performance problem. This chapter is the most useful if you want to start with a misbehaving Linux system and just diagnose the problem without necessarily understanding the details of the tools. [Chapters 10](#) through [12](#) present case studies in which the methodologies and tools previously described are used together to solve real-world problems. The case studies highlight Linux performance tools used to find and fix different types of performance problems: a CPU-bound application, a latency-sensitive application, and an I/O bound application.

[Chapter 13](#) overviews the performance tools and the opportunities Linux has for improvement.

This book also has two appendixes. [Appendix A](#) contains a table of the performance tools discussed in this book and includes a URL to the latest version of each tool. [Appendix A](#) also identifies which Linux distributions support each particular tool. Finally, [Appendix B](#) contains information that explains how to install `oprofile`, which is a very powerful but hard-to-install tool on a few major Linux distributions.

Acknowledgments

First, I want to thank to the good people at Prentice Hall, including Jill Harry, Brenda Mulligan, Gina Kanouse, and Keith Cline.

Second, I want to thank all the people who reviewed the initial book proposal and added valuable technical reviews and suggestions, including Karel Baloun, Joe Brazeal, Bill Carr, Jonathan Corbet, Matthew Crosby, Robert Husted, Paul Lussier, Scott Mann, Bret Strong, and George Vish II. I also want to thank all the people who taught me what I know about performance and let me optimize Linux even though the value of Linux optimization was uncertain at the time, including John Henning, Greg Tarsa, Dave Stanley, Greg Gaertner, Bill Carr, and the whole BPE tools group (which supported and encouraged my work on Linux).

In addition, I want to thank the good folks of SPEC who took me in and taught me why benchmarks, when done well, help the entire industry. I especially want to thank Kaivalya Dixit, whose passion and integrity for benchmarking will be sorely missed.

Thanks also to all the people who helped me keep my sanity with many games of Carcassonne and Settlers of Catan, including Sarah Ezolt, Dave and Yoko Mitzel, Tim and Maureen Chorma, Ionel and Marina Vasilescu, Joe Doucette, and Jim Zawisza.

Finally, I want to thank my family, including Sasha and Mischief, who remind me that we always have time for a walk or to chase dental floss; Ron and Joni Elias, who cheer me on; Russell, Carol, and Tracy Ezolt, who gave their support and encouragement as I worked on this; and to my wife, Sarah, who is the most understanding and supportive person you can imagine.

About the Author

PHIL EZOLT discovered Linux more than nine years ago when pursuing an undergraduate electrical and computer engineering degree from Carnegie Mellon. For six years, Phil ported and designed Linux performance tools for Compaq's Alpha performance group and represented Compaq in the SPEC CPU subcommittee. During that time, he improved Linux performance on the industry standard SPEC Web and CPU benchmarks using the performance tools he ported and developed. Phil is currently part of a Hewlett-Packard team developing Sepia, a visualization technology that uses Linux XC clusters, OpenGL, and off-the-shelf graphics cards to parallelize real-time high-end visualizations. He is also currently pursuing a master's degree in computer science from Harvard University and is a big fan of German-style board games.

Chapter 1. Performance Hunting Tips

Solving performance problems without foresight and planning is painful. You waste time and are constantly frustrated as the cause of the problem repeatedly slips through your fingers. By following the right set of procedures, you can transform a frustrating performance hunt into an interesting detective story. Each piece of information leads you closer to the true culprit. People can not always be trusted. The evidence will be your only friend. As you investigate the problem, it will take unusual twists and turns, and the information that you discovered in the beginning of the hunt may turn out to be what helps you figure out the problem in the end. The best part is that you will have a thrill of adrenaline and sense of accomplishment when you finally nab the "bad guy" and fix the problem.

If you have never investigated a performance problem, the first steps can be overwhelming. However, by following a few obvious and nonobvious tips, you can save time and be well on your way to finding the cause of a performance problem. The goal of this chapter is to provide you with a series of tips and guidelines to help you hunt a performance problem. These tips show you how to avoid some of the common traps when investigating what is wrong with your system or application. Most of these tips were hard-learned lessons that resulted from wasted time and frustrating dead ends. These tips help you solve your performance problem quickly and efficiently.

After reading this chapter, you should be able to

- Avoid repeating the work of others.
- Avoid repeating your own work.
- Avoid false leads that result from gathering misleading information.
- Create a useful reference document about your investigation.

Although no performance investigation is flawless (you will almost always say, "If only I would have thought of that *first*"), these tips help you to avoid some of the common mistakes of a performance investigation.

1.1. General Tips

1.1.1. Take Copious Notes (Save Everything)

Probably the *most* important thing that you can do when investigating a performance problem is to record every output that you see, every command that you execute, and every piece of information that you research. A well-organized set of notes allows you to test a theory about the cause of a performance problem by simply looking at your notes rather than rerunning tests. This saves a huge amount of time. Write it down to create a permanent record.

When starting a performance investigation, I usually create a directory for the investigation, open a new "Notes" file in GNU emacs, and start to record information about the system. I then store performance results in this directory and store interesting and related pieces of information in the Notes file. I suggest that you add the following to your performance investigation file and directory:

- *Record the hardware/software configuration*— This involves recording information about the hardware configuration (amount of memory and type of CPU, network, and disk subsystem) as well as the software environment (the OS and software versions and the relevant configuration files). This information may seem easy to reproduce later, but when tracking down a problem, you may significantly change a system's configuration. Careful and meticulous notes can be used to figure out the system's configuration during a particular test.

Example: Save the output of `cat /proc/pci`, `dmesg`, and `uname -a` for each test.

- *Save and organize performance results*— It can be valuable to review performance results a long time after you run them. Record the results of a test with the configuration of the system. This allows you to compare how different configurations affect the performance results. It would be possible just to rerun the test if needed, but usually testing a configuration is a time-consuming process. It is more efficient just to keep your notes well organized and avoid repeating work.
- *Write down the command-line invocations*— As you run performance tools, you will often create complicated and complex command lines that measure the exact areas of the system that interest you. If you want to rerun a test, or run the same test on a different application, reproducing these command lines can be annoying and hard to do right on the first try. It is better just to record exactly what you typed. You can then reproduce the exact command line for a future test, and when reviewing past results, you can also see exactly what you measured. The Linux command `script` (described in detail in [Chapter 8](#), "Utility Tools: Performance Tool Helpers") or "cut and paste" from a terminal is a good way to do this.
- *Record research information and URLs*— As you investigate a performance problem, it is important to record relevant information you found on the Internet, through e-mail, or through personal interactions. If you find a Web site that seems relevant, cut and paste the text into your notes. (Web sites can disappear.) However, also save the URL, because you might need to review the page later or the page may point to information that becomes important later in an investigation.

As you collect and record all this information, you may wonder why it is worth the effort. Some information may seem useless or misleading now, but it might be useful in the future. (A good performance investigation is like a good detective show: Although the clues are confusing at first, everything becomes clear in the end.) Keep the following in mind as you investigate a problem:

The implications of results may be fuzzy— It is not always clear what a performance tool is telling you. Sometimes, you need more information to understand the implications of a particular result. At a later point, you might look back at seemingly useless test results in a new light. The old information may actually disprove or prove a particular theory about the nature of the performance problem.

All information is useful information (which is why you save it)— It might not be immediately clear

 PREV

< Day Day Up >

NEXT 

1.2. Outline of a Performance Investigation

This section outlines a series of essential steps as you begin a performance investigation. Because the ultimate goal is to fix the problem, the best idea is to research the problem before you even touch a performance tool. Following a particular protocol is an efficient way to solve your problem without wasting valuable time.

1.2.1. Finding a Metric, Baseline, and Target

The first step in a performance investigation is to determine the current performance and figure out how much it needs to be improved. If your system is significantly underperforming, you may decide that it is worth the time to investigate. However, if the system is performing close to its peak values, it might not be worth an investigation. Figuring out the peak performance values helps you to set reasonable performance expectations and gives you a performance goal, so that you know when to stop optimizing. You can always tweak the system just a little more, and with no performance target, you can waste a lot of time squeezing out that extra percent of performance, even though you may not actually need it.

1.2.1.1 Establish a Metric

To figure out when you have finished, you must create or use an already established metric of your system's performance. A metric is an objective measurement that indicates how the system is performing. For example, if you are optimizing a Web server, you could choose "serviced Web requests per second." If you do not have an objective way to measure the performance, it can be nearly impossible to determine whether you are making any progress as you tune the system.

1.2.1.2 Establish a Baseline

After you figure out how you are going to measure the performance of a particular system or application, it is important to determine your current performance levels. Run the application and record its performance before any tuning or optimization; this is called the baseline value, and it is the starting point for the performance investigation.

1.2.1.3 Establish a Target

After you pick a metric and baseline for the performance, it is important to pick a target. This target guides you to the end of the performance hunt. You can indefinitely tweak a system, and you can always get it just a little better with more and more time. If you pick your target, you will know when you have finished. To pick a reasonable goal, the following are good starting points:

- *Find others with a similar configuration and ask for their performance measurements*— This is an ideal situation. If you can find someone with a similar system that performs better, not only will you be able to pick a target for your system, you may also be able to work with that person to determine why your configuration is slower and how your configurations differ. Using another system as a reference can prove immensely useful when investigating a problem.
- *Find results of industry standard benchmarks*— Many Web sites compare benchmark results of various aspects of computing systems. Some of the benchmark results can be achieved only with a heroic effort, so they might not represent realistic use. However, many benchmark sites have the configuration used for particular results. These configurations can provide clues to help you tune the system.
- *Use your hardware with a different OS or application*— It may be possible to run a different application on your system with a similar function. For example, if you have two different Web servers, and one performs slowly, try a different one to see whether it performs any better. Alternatively, try running the same application on a different operating system. If the system performs better in either of these cases, you know that your original application has room for improvement.

 PREV

< Day Day Up >

NEXT 

1.3. Chapter Summary

Hunting a performance problem should be a satisfying and exciting process. If you have a good method in place to research and analyze, it will be repaid back many times as you hunt the problem. First, determine whether other people have had similar problems; if they have, try their solutions. Be skeptical of what they tell you, but look for others with experience of a similar problem. Create a reasonable metric and target for your performance hunt; the metric enables you to know when you have finished. Automate performance tests. Be sure to save test results and configuration information when you generate them so that you can review the results later. Keep your results organized and record any research and other information that you find that relates to your problem. Finally, periodically review your notes to find information that you might have missed the first time. If you follow these guidelines, you will have a clear goal and a clear procedure to investigate the problem.

This chapter provided a basic background for a performance investigation, and the following chapters cover the Linux-specific performance tools themselves. You learn how to use the tools, what type of information they can provide, and how to use them in combination to find performance problems on a particular system.

Chapter 2. Performance Tools: System CPU

This chapter overviews the system-wide Linux performance tools. These tools are your first line of defense when tracking a performance problem. They can show you how the overall system is performing and which areas are misbehaving. This chapter discusses the statistics that these tools can measure and how to use the individual tools to gather those statistics. After reading this chapter, you should

- Understand the basic metrics of system-wide performance, including CPU usage
- Understand which tools can retrieve these system-wide performance metrics

2.1. CPU Performance Statistics

Each system-wide Linux performance tool provides different ways to extract similar statistics. Although no tool displays all the statistics, some of the tools display the same statistics. Rather than describe the meaning of the statistics multiple times (once for each tool), we review them once before all the tools are described.

2.1.1. Run Queue Statistics

In Linux, a process can be either runnable or blocked waiting for an event to complete. A blocked process may be waiting for data from an I/O device or the results of a system call. If a process is runnable, that means that it is competing for the CPU time with the other processes that are also runnable. A runnable process is not necessarily using the CPU, but when the Linux scheduler is deciding which process to run next, it picks from the list of runnable processes. When these processes are runnable, but waiting to use the processor, they form a line called the run queue. The longer the run queue, the more processes wait in line.

The performance tools commonly show the number of processes that are runnable and the number of processes that are blocked waiting for I/O. Another common system statistic is that of load average. The load on a system is the total amount of running and runnable process. For example, if two processes were running and three were available to run, the system's load would be five. The load average is the amount of load over a given amount of time. Typically, the load average is taken over 1 minute, 5 minutes, and 15 minutes. This enables you to see how the load changes over time.

2.1.2. Context Switches

Most modern processors can run only one process or thread at a time. Although some processors, such as hyperthreaded processors, can actually run more than one process simultaneously, Linux treats them as multiple single-threaded processors. To create the illusion that a given single processor runs multiple tasks simultaneously, the Linux kernel constantly switches between different processes. The switch between different processes is called a context switch, because when it happens, the CPU saves all the context information from the old process and retrieves all the context information for the new process. The context contains a large amount of information that Linux tracks for each process, including, among others, which instruction the process is executing, which memory it has allocated, and which files the process has open. Switching these contexts can involve moving a large amount of information, and a context switch can be quite expensive. It is a good idea to minimize the number of context switches if possible.

To avoid context switches, it is important to know how they can happen. First, context switches can result from kernel scheduling. To guarantee that each process receives a fair share of processor time, the kernel periodically interrupts the running process and, if appropriate, the kernel scheduler decides to start another process rather than let the current process continue executing. It is possible that your system will context switch every time this periodic interrupt or timer occurs. The number of timer interrupts per second varies per architecture and kernel version. One easy way to check how often the interrupt fires is to use the `/proc/interrupts` file to determine the number of interrupts that have occurred over a known amount of time. This is demonstrated in [Listing 2.1](#).

Listing 2.1.

```
root@localhost asm-i386]# cat /proc/interrupts | grep timer
; sleep 10 ; cat /proc/interrupts | grep timer
0:    24060043          XT-PIC  timer
0:    24070093          XT-PIC  timer
```

In [Listing 2.1](#), we ask the kernel to show us how many times the timer has fired, wait 10 seconds, and then ask again. That means that on this machine, the timer fires at a rate of (24 070 093 - 24 060 043)

 PREV

< Day Day Up >

NEXT 

2.2. Linux Performance Tools: CPU

Here begins our discussion of performance tools that enable you to extract information previously described.

2.2.1. vmstat (Virtual Memory Statistics)

`vmstat` stands for virtual memory statistics, which indicates that it will give you information about the virtual memory system performance of your system. Fortunately, it actually does much more than that. `vmstat` is a great command to get a rough idea of how your system performs as a whole. It tells you

- How many processes are running
- How the CPU is being used
- How many interrupts the CPU receives
- How many context switches the scheduler performs

It is an excellent tool to use to get a rough idea of how the system performs.

2.2.1.1 CPU Performance-Related Options

`vmstat` can be invoked with the following command line:

```
vmstat [-n] [-s] [delay [count]]
```

`vmstat` can be run in two modes: sample mode and average mode. If no parameters are specified, `vmstat` runs in average mode, where `vmstat` displays the average value for all the statistics since system boot. However, if a delay is specified, the first sample will be the average since system boot, but after that `vmstat` samples the system every delay seconds and prints out the statistics. [Table 2-1](#) describes the options that `vmstat` accepts.

Table 2-1. `vmstat` Command-Line Options

Option	Explanation
<code>-n</code>	By default, <code>vmstat</code> periodically prints out the column headers for each performance statistic. This option disables that feature so that after the initial header, only performance data displays. This proves helpful if you want to import the output of <code>vmstat</code> into a spreadsheet.
<code>-s</code>	This displays a one-shot details output of system statistics that <code>vmstat</code> gathers. The statistics are the totals since the system booted.
<code>delay</code>	This is the amount of time between <code>vmstat</code> samples.

`vmstat` provides a variety of different output statistics that enable you to track different aspects of the system performance. [Table 2-2](#) describes those related to CPU performance. The next chapter covers those related to memory performance.

Table 2-2. CPU-Specific `vmstat` Output

Column	Explanation
--------	-------------

 PREV

< Day Day Up >

NEXT 

2.3. Chapter Summary

This chapter focused on the system-wide performance metric of CPU usage. These metrics mainly demonstrate how the operating system and machine are behaving, rather than a particular application.

This chapter demonstrated how performance tools, such as `sar` and `vmstat`, can be used to extract this system-wide performance information from a running system. These tools are the first line of defense when diagnosing a system problem. They help to determine how the system is behaving and which subsystem or application may be particularly stressed. The next chapter focuses on the system-wide performance tools that enable you to analyze the memory usage of the entire system.

Chapter 3. Performance Tools: System Memory

This chapter overviews the system-wide Linux memory performance tools. This chapter discusses the memory statistics that these tools can measure and how to use the individual tools to gather those statistics. After reading this chapter, you should

- Understand the basic metrics of system-wide performance, including memory usage
- Understand which tools can retrieve system-wide memory performance metrics

3.1. Memory Performance Statistics

Each system-wide Linux performance tool provides different ways to extract similar statistics. Although no tool displays all the statistics, some of the tools display the same statistics. The beginning of this chapter reviews the details of these statistics, and those descriptions are then referenced as the tools are described.

3.1.1. Memory Subsystem and Performance

In modern processors, saving information to and retrieving information from the memory subsystem usually takes longer than the CPU executing code and manipulating that information. The CPU usually spends a significant amount of time idle, waiting for instructions and data to be retrieved from memory before it can execute them or operate based on them. Processors have various levels of cache that compensate for the slow memory performance. Tools such as `oprofile` can show where various processor cache misses can occur.

3.1.2. Memory Subsystem (Virtual Memory)

Any given Linux system has a certain amount of RAM or physical memory. When addressing this physical memory, Linux breaks it up into chunks or "pages" of memory. When allocating or moving around memory, Linux operates on page-sized pieces rather than individual bytes. When reporting some memory statistics, the Linux kernel reports the number of pages per second, and this value can vary depending on the architecture it is running on. [Listing 3.1](#) creates a small application that displays the number of bytes per page for the current architecture.

Listing 3.1.

```
#include <unistd.h>
int main(int argc, char *argv[])
{
    printf("System page size: %d\n", getpagesize());
}
```

On the IA32 architecture, the page size is 4KB. In rare cases, these page-sized chunks of memory can cause too much overhead to track, so the kernel manipulates memory in much bigger chunks, known as HugePages. These are on the order of 2048KB rather than 4KB and greatly reduce the overhead for managing very large amounts of memory. Certain applications, such as Oracle, use these huge pages to load an enormous amount of data in memory while minimizing the overhead that the Linux kernel needs to manage it. If HugePages are not completely filled with data, these can waste a significant amount of memory. A half-filled normal page wastes 2KB of memory, whereas a half-filled HugePage can waste 1,024KB of memory.

The Linux kernel can take a scattered collection of these physical pages and present to applications a well laid-out virtual memory space.

3.1.2.1 Swap (Not Enough Physical Memory)

All systems have a fixed amount of physical memory in the form of RAM chips. The Linux kernel allows applications to run even if they require more memory than available with the physical memory. The Linux kernel uses the hard drive as a temporary memory. This hard drive space is called swap space.

Although swap is an excellent way to allow processes to run, it is terribly slow. It can be up to 1,000 times slower for an application to use swap rather than physical memory. If a system is performing poorly, it usually proves helpful to determine how much swap the system is using.

3.1.2.2 Buffers and Cache (Too Much Physical Memory)

 PREV

< Day Day Up >

NEXT 

3.2. Linux Performance Tools: CPU and Memory

Here begins our discussion of performance tools that enable you to extract the memory performance information described previously.

3.2.1. vmstat (Virtual Memory Statistics) II

As you have seen before, `vmstat` can provide information about many different performance aspects of a system—although its primary purpose, as shown next, is to provide information about virtual memory system performance. In addition to the CPU performance statistics described in the previous chapter, it can also tell you the following:

- How much swap is being used
- How the physical memory is being used
- How much memory is free

As you can see, `vmstat` provides (via the statistics it displays) a wealth of information about the health and performance of the system in a single line of text.

3.2.1.1 System-Wide Memory-Related Options

In addition to the CPU statistics `vmstat` can provide, you can invoke `vmstat` with the following command-line options when investigating memory statistics:

```
vmstat [-a] [-s] [-m]
```

As before, you can run `vmstat` in two modes: sample mode and average mode. The added command-line options enable you to get the performance statistics about how the Linux kernel is using memory. [Table 3-1](#) describes the options that `vmstat` accepts.

Table 3-1. `vmstat` Command-Line Options


Option	Explanation
<code>-a</code>	This changes the default output of memory statistics to indicate the active/inactive amount of memory rather than information about buffer and cache usage.
<code>-s</code> (<code>procps</code> 3.2 or greater)	This prints out the <code>vm</code> table. This is a grab bag of different statistics about the system since it has booted. It cannot be run in sample mode. It contains both memory and CPU statistics.
<code>-m</code> (<code>procps</code> 3.2 or greater)	This prints out the kernel's slab info. This is the same information that can be retrieved by typing <code>cat /proc/slabinfo</code> . This describes in detail how the kernel's memory is allocated and can be helpful to determine what area of the kernel is consuming the most memory.

[Table 3-2](#) provides a list of the memory statistics that `vmstat` can provide. As with the CPU statistics, when run in normal mode, the first line that `vmstat` provides is the average values for all the rate statistics (`so` and `si`) and the instantaneous value for all the numeric statistics (`swpd`, `free`, `buff`, `cache`, `active`, and `inactive`).

Table 3-2. Memory-Specific `vmstat` Output Statistics

 PREV

< Day Day Up >

NEXT 

3.3. Chapter Summary

This chapter focused on system-wide memory performance metrics. These metrics mainly demonstrate how the operating system is using memory rather than a particular application.

This chapter demonstrated how performance tools such as `sar` and `vmstat` can be used to extract this system-wide memory performance information from a running system. The output of these tools indicates how the system as a whole is using available memory. The next chapter describes the tools available to investigate a single process's CPU usage.

Chapter 4. Performance Tools: Process-Specific CPU

After using the system-wide performance tools to figure out which process is slowing down the system, you must apply the process-specific performance tools to figure out how the process is behaving. Linux provides a rich set of tools to track the important statistics of a process and application's performance.

After reading this chapter, you should be able to

- Determine whether an application's runtime is spent in the kernel or application.
- Determine what library and system calls an application is making and how long they are taking.
- Profile an application to figure out what source lines and functions are taking the longest time to complete.

4.1. Process Performance Statistics

The tools to analyze the performance of applications are varied and have existed in one form or another since the early days of UNIX. It is critical to understand how an application is interacting with the operating system, CPU, and memory system to understand its performance. Most applications are not self-contained and make many calls to the Linux kernel and different libraries. These calls to the Linux kernel (or system calls) may be as simple as "what's my PID?" or as complex as "read 12 blocks of data from the disk." Different systems calls will have different performance implications. Correspondingly, the library calls may be as simple as memory allocation or as complex as graphics window creation. These library calls may also have different performance characteristics.

4.1.1. Kernel Time Versus User Time

The most basic split of where an application may spend its time is between kernel and user time. Kernel time is the time spent in the Linux kernel, and user time is the amount of time spent in application or library code. Linux has tools such `time` and `ps` that can indicate (appropriately enough) whether an application is spending its time in application or kernel code. It also has commands such as `oprofile` and `strace` that enable you to trace which kernel calls are made on the behalf of the process, as well as how long each of those calls took to complete.

4.1.2. Library Time Versus Application Time

Any application with even a minor amount of complexity relies on system libraries to perform complex actions. These libraries may cause performance problems, so it is important to be able to see how much time an application spends in a particular library. Although it might not always be practical to modify the source code of the libraries directly to fix a problem, it may be possible to change the application code to call different or fewer library functions. The `ltrace` command and `oprofile` suite provide a way to analyze the performance of libraries when they are used by applications. Tools built in to the Linux loader, `ld`, helps you determine whether the use of many libraries slows down an application's start time.

4.1.3. Subdividing Application Time

When the application is known to be the bottleneck, Linux provides tools that enable you to profile an application to figure out where time is spent within an application. Tools such as `gprof` and `oprofile` can generate profiles of an application that pin down exactly which source line is causing large amounts of time to be spent.

4.2. The Tools

Linux has a variety of tools to help you determine which pieces of an application are the primary users of the CPU. This section describes these tools.

4.2.1. `time`

The `time` command performs a basic function when testing a command's performance, yet it is often the first place to turn. The `time` command acts as a stopwatch and times how long a command takes to execute. It measures three types of time. First, it measures the real or elapsed time, which is the amount of time between when the program started and finished execution. Next, it measures the user time, which is the amount of time that the CPU spent executing application code on behalf of the program. Finally, `time` measures system time, which is the amount of time the CPU spent executing system or kernel code on behalf of the application.

4.2.1.1 CPU Performance-Related Options

The `time` command (see [Table 4-1](#)) is invoked in the following manner:

```
time [-v] application
```

Table 4-1. `time` Command-Line Options

Option	Explanation
--------	-------------

<code>-v</code>	This option presents a verbose display of the program's time and statistics. Some statistics are zeroed out, but more statistics are valid with Linux kernel v2.6 than with Linux kernel v2.4.
-----------------	--

Most of the valid statistics are present in both the standard and verbose mode, but the verbose mode provides a better description for each statistic.

The `application` is timed, and information about its CPU usage is displayed on standard output after it has completed.

[Table 4-2](#) describes the valid output statistic that the `time` command provides. The rest are not measured and always display zero.

Table 4-2. CPU-Specific `time` Output

Column	Explanation
User time (seconds)	This is the number of seconds of CPU spent by the application.
System time (seconds)	This is the number of seconds spent in the Linux kernel on behalf of the application.
Elapsed (wall-clock) time (h:mm:ss or m:ss)	This is the amount of time elapsed (in wall-clock time) between when the application was launched and when it completed.
Percent of CPU this job got	This is the percentage of the CPU that the process consumed as it was running.

 PREV

< Day Day Up >

NEXT 

4.3. Chapter Summary

This chapter covered how to track the CPU performance bottlenecks of individual processes. You learned to determine how an application was spending its time by attributing the time spent to the Linux kernel, system libraries, or even to the application itself. You also learned how to figure out which calls were made to the kernel and system libraries and how long each took to complete. Finally, you learned how to profile an application and determine the particular line of source code that was spending a large amount of time. After mastering these tools, you can start with an application that hogs the CPU and use these tools to find the exact functions that are spending all the time.

Subsequent chapters investigate how to find bottlenecks that are not CPU bound. In particular, you learn about the tools used to find I/O bottlenecks, such as a saturated disk or an overloaded network.

Chapter 5. Performance Tools: Process-Specific Memory

This chapter covers tools that enable you to diagnose an application's interaction with the memory subsystem as managed by the Linux kernel and the CPU. Because different layers of the memory subsystem have orders of magnitude differences in performance, fixing an application to efficiently use the memory subsystem can have a dramatic influence on an application's performance.

After reading this chapter, you should be able to

- Determine how much memory an application is using (`ps`, `/proc`).
- Determine which functions of an application are allocating memory (`mempref`).
- Profile the memory usage of an application using both software simulation (`kcachegrind`, `cachegrind`) and hardware performance counters (`oprofile`).
- Determine which processes are creating and using shared memory (`ipcs`).

5.1. Linux Memory Subsystem

When diagnosing memory performance problems, it may become necessary to observe how an application performs at various levels within the memory subsystem. At the top level, the operating system decides how the swap and physical memory are being used. It decides what pieces of an application's address space will be in physical memory, which is called the resident set. Other memory used by the application but not part of the resident set will be swapped to disk. The application decides how much memory it will request from the operating system, and this is called the virtual set. The application can allocate this explicitly by calling `malloc` or implicitly by using a large amount of stack or using a large number of libraries. The application can also allocate shared memory that can be used by itself and other applications. The `ps` performance tool is useful for tracking the virtual and resident set size. The `memprowf` performance tool is useful for tracking which code in an application is allocating memory. The `ipcs` tool is useful for tracking shared memory usage.

When an application is using physical memory, it begins to interact with the CPU's cache subsystem. Modern CPUs have multiple levels of cache. The fastest cache is closest to the CPU (also called L1 or Level 1 cache) and is the smallest in size. Suppose, for instance, that the CPU has only two levels of cache: L1 and L2. When the CPU requests a piece of memory, the processor checks to see whether it is already in the L1 cache. If it is, the CPU uses it. If it was not in the L1 cache, the processor generates a L1 cache miss. It then checks in the L2 cache; if the data is in the L2 cache, it is used. If the data is not in the L2 cache, an L2 cache miss occurs, and the processor must go to physical memory to retrieve the information. Ultimately, it would be best if the processor never goes to physical memory (because it finds the data in the L1 or even L2 cache). Smart cache use—rearranging an application's data structures and reducing code size, for example—may make it possible to reduce the number of caches misses and increase performance. `cachegrind` and `oprofile` are great tools to find information about how an application is using the cache and about which functions and data structures are causing cache misses.

5.2. Memory Performance Tools

This section examines the various memory performance tools that enable you to investigate how a given application is using the memory subsystem, including the amount and different types of memory that a process is using, where it is being allocated, and how effectively the process is using the processor's cache.

5.2.1. ps

`ps` is an excellent command to track a process's dynamic memory usage. In addition to the CPU statistics already mentioned, `ps` gives detailed information about the amount of memory that the application is using and how that memory usage affects the system.

5.2.1.1 Memory Performance-Related Options

`ps` has many different options and can retrieve many different statistics about the state of a running application. As you saw in the previous chapter, `ps` can retrieve information about the CPU that a process is spending, but it also can retrieve information about the amount and type of memory that a process is using. It can be invoked with the following command line:

```
ps [-o vsz,rss,tsiz,dsiz,majflt,minflt,pmem,command] <PID>
```

Table 5-1 describes the different types of memory statistics that `ps` can display for a given PID.

Table 5-1. `ps` Command-Line Options

Option	Explanation
<code>-o <statistic></code>	Enables you to specify exactly what process statistics you want to track. The different statistics are specified in a comma-separated list with no spaces.
<code>vsz</code>	Statistic: The virtual set size is the amount of virtual memory that the application is using. Because Linux only allocated physical memory when an application tries to use it, this value may be much greater than the amount of physical memory the application is using.
<code>rss</code>	Statistic: The resident set size is the amount of physical memory the application is currently using.
<code>tsiz</code>	Statistic: Text size is the virtual size of the program code. Once again, this isn't the physical size but rather the virtual size; however, it is a good indication of the size of the program.
<code>dsiz</code>	Statistic: Data size is the virtual size of the program's data usage. This is a good indication of the size of the data structures and stack of the application.
<code>majflt</code>	Statistic: Major faults are the number of page faults that caused Linux to read a page from disk on behalf of the process. This may happen if the process accessed a piece of data or instruction that remained on the disk and Linux loaded it seamlessly for the application.
<code>minflt</code>	Statistic: Minor faults are the number of faults that Linux could fulfill without resorting to a disk read. This might happen if the application touches a piece of memory that has been allocated by the Linux kernel. In this case, it is not necessary to go to disk, because the kernel can just pick a free piece of memory and assign it to the application.

 PREV

< Day Day Up >

NEXT 

5.3. Chapter Summary

This chapter presented the various Linux tools that are available to diagnose memory-performance problems. It demonstrated tools that show how much memory an application is consuming (`ps`, `/proc`) and which functions within the application are allocating that memory (`mempref`). It also covered tools that can monitor the effectiveness of the processor and system cache and memory subsystem (`cachegrind`, `kcachegrind` and `oprofile`). Finally, it described a tool that monitors shared memory usage (`ipcs`). Used together, these tools can track every allocation of memory, the size of these allocations, the functional locations of the allocations in the applications, and how effectively the application is using the memory subsystem when accessing these allocations.

The next chapter moves away from memory to investigate disk I/O bottlenecks.

Chapter 6. Performance Tools: Disk I/O

This chapter covers performance tools that help you gauge disk I/O subsystem usage. These tools can show which disks or partitions are being used, how much I/O each disk is processing, and how long I/O requests issued to these disks are waiting to be processed.

After reading this chapter, you should be able to

- Determine the amount of total amount and type (read/write) of disk I/O on a system (`vmstat`).
- Determine which devices are servicing most of the disk I/O (`vmstat`, `iostat`, `sar`).
- Determine how effectively a particular disk is fielding I/O requests (`iostat`).
- Determine which processes are using a given set of files (`lssof`).

6.1. Introduction to Disk I/O

Before diving into performance tools, it is necessary to understand how the Linux disk I/O system is structured. Most modern Linux systems have one or more disk drives. If they are IDE drives, they are usually named `hda`, `hdb`, `hdc`, and so on; whereas SCSI drives are usually named `sda`, `sdb`, `sdc`, and so on. A disk is typically split into multiple partitions, where the name of the partition's device is created by adding the partition number to the end of the base device name. For example, the second partition on the first IDE hard drive in the system is usually labeled `/dev/hda2`. Each individual partition usually contains either a file system or a swap partition. These partitions are mounted into the Linux root file system, as specified in `/etc/fstab`. These mounted file systems contain the files that applications read to and write from.

When an application does a read or write, the Linux kernel may have a copy of the file stored into its cache or buffers and returns the requested information without ever accessing the disk. If the Linux kernel does not have a copy of the data stored in memory, however, it adds a request to the disk's I/O queue. If the Linux kernel notices that multiple requests are asking for contiguous locations on the disk, it merges them into a single big request. This merging increases overall disk performance by eliminating the seek time for the second request. When the request has been placed in the disk queue, if the disk is not currently busy, it starts to service the I/O request. If the disk is busy, the request waits in the queue until the drive is available, and then it is serviced.

6.2. Disk I/O Performance Tools

This section examines the various disk I/O performance tools that enable you to investigate how a given application is using the disk I/O subsystem, including how heavily each disk is being used, how well the kernel's disk cache is working, and which files a particular application has "open."

6.2.1. `vmstat` (ii)

As you saw in [Chapter 2, "Performance Tools: System CPU,"](#) `vmstat` is a great tool to give an overall view of how the system is performing. In addition to CPU and memory statistics, `vmstat` can provide a system-wide view of I/O performance.

6.2.1.1 Disk I/O Performance-Related Options and Outputs

While using `vmstat` to retrieve disk I/O statistics from the system, you must invoke it as follows:

```
vmstat [-D] [-d] [-p partition] [interval [count]]
```

[Table 6-1](#) describes the other command-line parameters that influence the disk I/O statistics that `vmstat` will display.

Table 6-1. `vmstat` Command-Line Options

Option	Explanation
<code>-D</code>	This displays Linux I/O subsystem total statistics. This option can give you a good idea of how your I/O subsystem is being used, but it won't give statistics on individual disks. The statistics given are the totals since system boot, rather than just those that occurred between this sample and the previous sample.
<code>-d</code>	This option displays individual disk statistics at a rate of one sample per <code>interval</code> . The statistics are the totals since system boot, rather than just those that occurred between this sample and the previous sample.
<code>-p partition</code>	This displays performance statistics about the given partition at a rate of one sample per <code>interval</code> . The statistics are the totals since system boot, rather than just those that occurred between this sample and the previous sample.
<code>interval</code>	The length of time between samples.
<code>count</code>	The total number of samples to take.

If you run `vmstat` without any parameters other than `[interval]` and `[count]`, it shows you the default output. This output contains three columns relevant to disk I/O performance: `bo`, `bi`, and `wa`. These statistics are described in [Table 6-2](#).

Table 6-2. `vmstat` I/O Statistics

Statistic	Explanation
<code>bo</code>	This indicates the number of total blocks written to disk in the previous interval. (In <code>vmstat</code> , block size for a disk is typically 1,024 bytes.)

 PREV

< Day Day Up >

NEXT 

6.3. What's Missing?

All the disk I/O tools on Linux provide information about the utilization of a particular disk or partition. Unfortunately, after you determine that a particular disk is a bottleneck, there are no tools that enable you to figure out which process is causing all the I/O traffic.

Usually a system administrator has a good idea about what application uses the disk, but not always. Many times, for example, I have been using my Linux system when the disks started grinding for apparently no reason. I can usually run `top` and look for a process that might be causing the problem. By eliminating processes that I believe are not doing I/O, I can usually find the culprit. However, this requires knowledge of what the various applications are supposed to do. It is also error prone, because the guess about which processes are not causing the problem might be wrong. In addition, for a system with many users or many running applications, it is not always practical or easy to determine which application might be causing the problem. Other UNIXes support the `inblk` and `oublk` parameters to `ps`, which show you the amount of disk I/O issued on behalf of a particular process. Currently, the Linux kernel does not track the I/O of a process, so the `ps` tool has no way to gather this information.

You can use `lsof` to determine which processes are accessing files on a particular partition. After you list all PIDs accessing the files, you can then attach to each of the PIDs with `strace` and figure out which one is doing a significant amount of I/O. Although this method works, it is really a Band-Aid solution, because the number of processes accessing a partition could be large and it is time-consuming to attach and analyze the system calls of each process. This may also miss short-lived processes, and may unacceptably slow down processes when they are being traced.

This is an area where the Linux kernel could be improved. The ability to quickly track which processes are generating I/O would allow for much quicker diagnosis of I/O performance-related problems.

6.4. Chapter Summary

This chapter presented the Linux disk I/O performance tools used to extract information about system-wide (`vmstat`), device-specific (`vmstat`, `iostat`, `sar`), and file-specific (`lsof`) disk I/O usage. It explained the different types of I/O statistics and how to extract these statistics from Linux using the I/O performance tools. It also discussed some of the significant limitations of the current tools and areas for future growth.

The next chapter examines the tools that enable you to determine the cause of network bottlenecks.

Chapter 7. Performance Tools: Network

This chapter introduces some of the network performance tools available on Linux. We primarily focus on the tools that analyze the network traffic on a single box rather than network-wide management tools. Although network performance evaluation usually does not make sense in total isolation (that is, nodes do not normally talk to themselves), it is valuable to investigate the behavior of a single system on the network to identify local configuration and application problems. In addition, understanding the characteristics of network traffic on a single system can help to locate other problem systems, or local hardware and applications errors that slow down network performance.

After reading this chapter, you should be able to

- Determine the speed and duplex settings of the Ethernet devices in the system (`mii-tool`, `ethtool`).
- Determine the amount of network traffic flowing over each Ethernet interface (`ifconfig`, `sar`, `gkrellm`, `iptraf`, `netstat`, `etherape`).
- Determine the types of IP traffic flowing in to and out of the system (`gkrellm`, `iptraf`, `netstat`, `etherape`).
- Determine the amount of each type of IP traffic flowing in to and out of the system (`gkrellm`, `iptraf`, `etherape`).
- Determine which applications are generating IP traffic (`netstat`).

7.1. Introduction to Network I/O

Network traffic in Linux and every other major operating system is abstracted as a series of hardware and software layers. The link, or lowest, layer contains network hardware such as Ethernet devices. When moving network traffic, this layer does not distinguish types of traffic but just transmits and receives data (or frames) as fast as possible.

Stacked above the link layer is a network layer. This layer uses the Internet Protocol (IP) and Internet Control Message Protocol (ICMP) to address and route packets of data from machine to machine. IP/ICMP make their best-effort attempt to pass the packets between machines, but they make no guarantees about whether a packet actually arrives at its destination.

Stacked above the network layer is the transport layer, which defines the Transport Control Protocol (TCP) and User Datagram Protocol (UDP). TCP is a reliable protocol that guarantees that a message is either delivered over the network or generates an error if the message is not delivered. TCP's sibling protocol, UDP, is an unreliable protocol that deliberately (to achieve the highest data rates) does not guarantee message delivery. UDP and TCP add the concept of a "service" to IP. UDP and TCP receive messages on numbered "ports." By convention, each type of network service is assigned a different number. For example, Hypertext Transfer Protocol (HTTP) is typically port 80, Secure Shell (SSH) is typically port 22, and File Transport Protocol (FTP) is typically port 21. In a Linux system, the file `/etc/services` defines all the ports and the types of service they provide.

The final layer is the application layer. It includes all the different applications that use the layers below to transmit packets over the network. These include applications such as Web servers, SSH clients, or even peer-to-peer (P2P) file-sharing clients such as bittorrent.

The lowest three layers (link, network, and transport) are implemented or controlled within the Linux kernel. The kernel provides statistics about how each layer is performing, including information about the bandwidth usage and error count as data flows through each of the layers. The tools covered in this chapter enable you to extract and view those statistics.

7.1.1. Network Traffic in the Link Layer

At the lowest levels of the network stack, Linux can detect the rate at which data traffic is flowing through the link layer. The link layer, which is typically Ethernet, sends information into the network as a series of frames. Even though the layers above may have pieces of information much larger than the frame size, the link layer breaks everything up into frames to send them over the network. This maximum size of data in a frame is known as the maximum transfer unit (MTU). You can use network configuration tools such as `ip` or `ifconfig` to set the MTU. For Ethernet, the maximum size is commonly 1,500 bytes, although some hardware supports jumbo frames up to 9,000 bytes. The size of the MTU has a direct impact on the efficiency of the network. Each frame in the link layer has a small header, so using a large MTU increases the ratio of user data to overhead (header). When using a large MTU, however, each frame of data has a higher chance of being corrupted or dropped. For clean physical links, a high MTU usually leads to better performance because it requires less overhead; for noisy links, however, a smaller MTU may actually enhance performance because less data has to be re-sent when a single frame is corrupted.

At the physical layer, frames flow over the physical network; the Linux kernel collects a number of different statistics about the number and types of frames:

- *Transmitted/received*— If the frame successfully flowed in to or out of the machine, it is counted as a transmitted or received frame.
- *Errors*— Frames with errors (possibly because of a bad network cable or duplex mismatch).
- *Dropped*— Frames that were discarded (most likely because of low amounts of memory or buffers).

 PREV

< Day Day Up >

NEXT 

7.2. Network Performance Tools

This section describes the Linux network performance tools available to diagnose performance problems. We start with the tools to determine the lowest level of network performance (physical statistics) and add tools that can investigate the layers above that.

7.2.1. mii-tool (Media-Independent Interface Tool)

`mii-tool` is an Ethernet-specific hardware tool primarily used to configure an Ethernet device, but it can also provide information about the current configuration. This information, such as the link speed and duplex setting, can be useful when tracking down the cause of an under-performing network device.

7.2.1.1 Network I/O Performance-Related Options

`mii-tool` requires root access to be used. It is invoked with the following command line:

```
mii-tool [-v] [device]
```

`mii-tool` prints the Ethernet settings for the given device. If no devices are specified, `mii-tool` displays information about all the available Ethernet devices. If the `-v` option is used, `mii-tool` displays verbose statistics about the offered and negotiated network capabilities.

7.2.1.2 Example Usage

[Listing 7.1](#) shows the configuration of `eth0` on the system. The first line tells us that the Ethernet device is currently using a 100BASE-T full-duplex connection. The next few lines describe the capabilities of the network card in the machine and the capabilities that the card has detected of the network device on the other end of the wire.

Listing 7.1.

```
[root@nohs linux-2.6.8-1.521]# /sbin/mii-tool -v eth0
eth0: negotiated 100baseTx-FD, link ok
  product info: vendor 00:00:00, model 0 rev 0
  basic mode:   autonegotiation enabled
  basic status: autonegotiation complete, link ok
  capabilities: 100baseTx-FD 100baseTx-HD 10baseT-FD 10baseT-HD
  advertising: 100baseTx-FD 100baseTx-HD 10baseT-FD 10baseT-HD
flow-control
  link partner: 100baseTx-FD 100baseTx-HD 10baseT-FD 10baseT-HD
```

`mii-tool` provides low-level information about how the physical level of the ethernet device is configured.

7.2.2. ethtool

`ethtool` provides similar capabilities to `mii-tool` for configuration and display of statistics for Ethernet devices. However, `ethtool` is the more powerful tool and contains more configuration options and device statistics.

7.2.2.1 Network I/O Performance-Related Options

`ethtool` requires root access to be used. It is invoked with the following command line:

```
ethtool [device]
```

 PREV

< Day Day Up >

NEXT 

7.3. Chapter Summary

This chapter provided information about how to use the Linux network performance tools to monitor the network traffic flowing through a system all the way from the low-level network interfaces to high-level applications. It introduced tools to query the current physical link settings (`mii-tool`, `ethtool`) and tools that monitor the amount and types of packets flowing through the low-level interfaces (`ifconfig`, `ip`, `sar`, `gkrellm`, `iptraf`, `netstat`, `etherape`). It then presented tools that display the different types of IP traffic (`gkrellm`, `iptraf`, `netstat`, `etherape`) and the amounts of each type of traffic (`gkrellm`, `iptraf`, `etherape`). This chapter then presented a tool (`netstat`) that maps the IP socket usage to the process that is receiving/sending each type of traffic. Finally, a network-visualization tool was presented that visualizes the relationship between the type and amount of data flowing through a network and which nodes it is flowing between (`etherape`).

The next chapters describe some of the common Linux tools that make using performance tools easier. They are not performance tools themselves, but they make using the performance tools more palatable. They can also help to visualize and analyze the results of the tools, as well as automate some of the more repetitive tasks.

Chapter 8. Utility Tools: Performance Tool Helpers

This chapter provides information about the utilities available on a Linux system that can enhance the effectiveness and usability of the performance tools. The utility tools are not performance tools themselves, but when used with the performance tools, they can help automate tedious tasks, analyze performance statistics, and create performance tool-friendly applications.

After reading this chapter, you should be able to

- Automate the display and collection of periodic performance data (`bash`, `watch`).
- Record all commands and output displayed during a performance investigation (`tee`, `script`).
- Import, analyze, and graph performance data (`gnnumeric`).
- Determine the libraries that an application is using (`ldd`).
- Determine which functions are part of which libraries (`objdump`).
- Investigate runtime characteristics of an application (`gdb`).
- Create performance tool/debugging-friendly applications (`gcc`).

8.1. Performance Tool Helpers

Linux has a rich heritage of tools that can be used together and become greater than the sum of the parts. Performance tools are no different. Although performance tools are useful on their own, combining them with other Linux tools can significantly boost their effectiveness and ease of use.

8.1.1. Automating and Recording Commands

As mentioned in an earlier chapter, one of the most valuable steps in a performance investigation is to save the commands that are issued and results that are generated during a performance investigation. This allows you to review them later and look for new insights. To help with this, Linux provides the `tee` command, which enables you to save tool output to a file, and the `script` command, which records every key press and every output displayed on the screen. This information can be saved and reviewed later or used to create a script to automate test execution.

It is important to automate commands because it reduces the chance of errors and enables you to think about the problem without having to remember all the details. Both the `bash` shell and the `watch` command enable you to periodically and automatically execute long and complicated command lines after typing them once. After you have the command line correct, `bash` and `watch` can periodically execute the command without the need to retype it.

8.1.2. Graphing and Analyzing Performance Statistics

In addition to the tools for recording and automation, Linux provides powerful analysis tools that can help you understand the implications of performance statistics. Whereas most performance tools generate performance statistics as text output, it is not always easy to see patterns and trends over time. Linux provides the powerful `gnnumeric` spreadsheet, which can import, analyze, and graph performance data. When you graph the data, the cause of a performance problem may become apparent, or it may at least open up new areas of investigation.

8.1.3. Investigating the Libraries That an Application Uses

Linux also provides tools that enable you to determine which libraries an application relies on, as well as tools that display all the functions that a given library provides. The `ldd` command provides the list of all the shared libraries that a particular application is using. This can prove helpful if you are trying to track the number and location of the libraries that an application uses. Linux also provides the `objdump` command, which enables you to search through a given library or application to display all the functions that it provides. By combining the `ldd` and `objdump` commands, you can take the output of `ltrace`, which only provides the names of the functions that an application calls, and determine which library a given function is part of.

8.1.4. Creating and Debugging Applications

Finally, Linux also provides tools that enable you to create performance-tool-friendly applications, in addition to tools that enable you to interactively debug and investigate the attributes of running applications. The GNU compiler collection (`gcc`) can insert debugging information into applications that aid `oprofile` in finding the exact line and source file of a specific performance problem. In addition, the GNU debugger (`gdb`) can also be used to find information about running applications not available by default to various performance tools.

 PREV

< Day Day Up >

NEXT 

8.2. Tools

Used together, the following tools can greatly enhance the effectiveness and ease of use of the performance tools described in previous chapters.

8.2.1. bash

`bash` is the default Linux command-line shell, and you most likely use it every time you interact with the Linux command line. `bash` has a powerful scripting language that is typically used to create shell scripts. However, the scripting language can also be called from the command line and enables you to easily automate some of the more tedious tasks during a performance investigation.

8.2.1.1 Performance-Related Options

`bash` provides a series of commands that can be used together to periodically run a particular command. Most Linux users have `bash` as their default shell, so just logging in to a machine or opening a terminal brings up a `bash` prompt. If you are not using `bash`, you can invoke it by typing `bash`.

After you have a `bash` command prompt, you can enter a series of `bash` scripting commands to automate the continuous execution of a particular command. This feature proves most useful when you need to periodically extract performance statistics using a particular command. These scripting options are described in [Table 8-1](#).

Table 8-1. `bash` Runtime Scripting Options

Option	Explanation
<code>while condition</code>	This executes a loop until the condition is false.
<code>do</code>	This indicates the start of a loop.
<code>done</code>	This indicates the end of a loop.

`bash` is infinitely flexible and is documented in the `bash` man page. Although `bash`'s complexity can be overwhelming, it is not necessary to master it all to put `bash` immediately to use.

8.2.1.2 Example Usage

Although some performance tools, such as `vmstat` and `sar`, periodically display updated performance statistics, other commands, such as `ps` and `ifconfig`, do not. `bash` can call commands such as `ps` and `ifconfig` to periodically display their statistics. For example, in [Listing 8.1](#), we ask `bash` to do something in a `while` loop based on the condition `TRue`. Because the `TRue` command is always true, the `while` loop will never exit. Next, the commands that will be executed after each iteration start after the `do` command. These commands ask `bash` to sleep for one second and then run `ifconfig` to extract performance information about the `eth0` controller. However, because we are only interested in the received packets, we `grep` output of `ifconfig` for the string `"RX packets"`. Finally, we issue the `done` command to tell `bash` we are done with the loop. Because the `TRue` command always returns true, this entire loop will run forever unless we interrupt it with a `<Ctrl-C>`.

Listing 8.1.

```
[ezolt@wintermute tmp]$ while true; do sleep 1; /sbin/ifconfig
eth0 | grep
"RX packets" ; done;
```

 PREV

< Day Day Up >

NEXT 

8.3. Chapter Summary

This chapter provided a grab bag of Linux utility tools that are useful when investigating a performance problem. It introduced tools such as `bash`, `watch`, `tee`, and `script`, which automate the display and collection of performance data. It also introduced `gnnumeric`, a tool that can both graph and analyze the results of text-based performance tools. It then investigated `ldd` and `objdump`, which can be used to track down which library a function is part of. It then described `gdb`, a tool that can investigate the execution and runtime information of currently running applications. Finally, this chapter described `gcc`, a tool that can produce binaries with symbolic debugging information that helps other performance tools, such as `oprofile`, to map events back to a specific source line.

In the upcoming chapters, we put together all the tools presented so far and solve some real-life performance problems.

Chapter 9. Using Performance Tools to Find Problems

This chapter contains a method for using the previously presented performance tools together to narrow down the cause of a performance problem.

After reading this chapter, you should be able to

- Start with a misbehaving system and use the Linux performance tools to track down the misbehaving kernel functions or applications.
- Start with a misbehaving application and use the Linux performance tools to track down the misbehaving functions or source lines.
- Track down excess usage of the CPU, memory, disk I/O, and network.

9.1. Not Always a Silver Bullet

This chapter assumes that it is possible to solve a performance problem by changing software. Tuning an application or system to achieve a target performance goal is not always possible. If tuning fails, it may require a hardware upgrade or change. If the capacity of the system is maxed out, performance tuning only helps to a certain extent.

For example, it may be necessary (or even cheaper) to just upgrade the amount of system memory rather than track down which applications are using system memory, and then tune them so that they reduce their usage. The decision to just upgrade the system hardware rather than track down and tune a particular performance problem depends on the problem and is a value judgment of the individual investigating it. It really depends on which option is cheaper, either time-wise (to investigate the problem) or money-wise (to buy new hardware). Ultimately, in some situations, tuning will be the preferred or only option, so that is what this chapter describes.

9.2. Starting the Hunt

After you decide to start optimizing something on Linux, you first have to decide what you are going to optimize. The method used in this chapter covers some of the more common performance problems and an example shows you how to use the previously presented tools together to solve a problem. The next series of sections helps guide you in your discovery of the cause of a performance problem. In many sections you are asked to run various performance tools and jump to different sections in this chapter based on the results. This helps to pinpoint the source of the problem.

As stated in previous chapters, it is a good idea to save the results of each test that you perform. This enables you to review the results later and even to send the results to someone else if the investigation is inconclusive.

Let's get started.

When investigating a problem, it is best to start with a system that has as little unrelated programs running as possible, so close or kill any unneeded applications or processes. A clean system helps eliminate the potentially confusing interference caused by any extraneous applications.

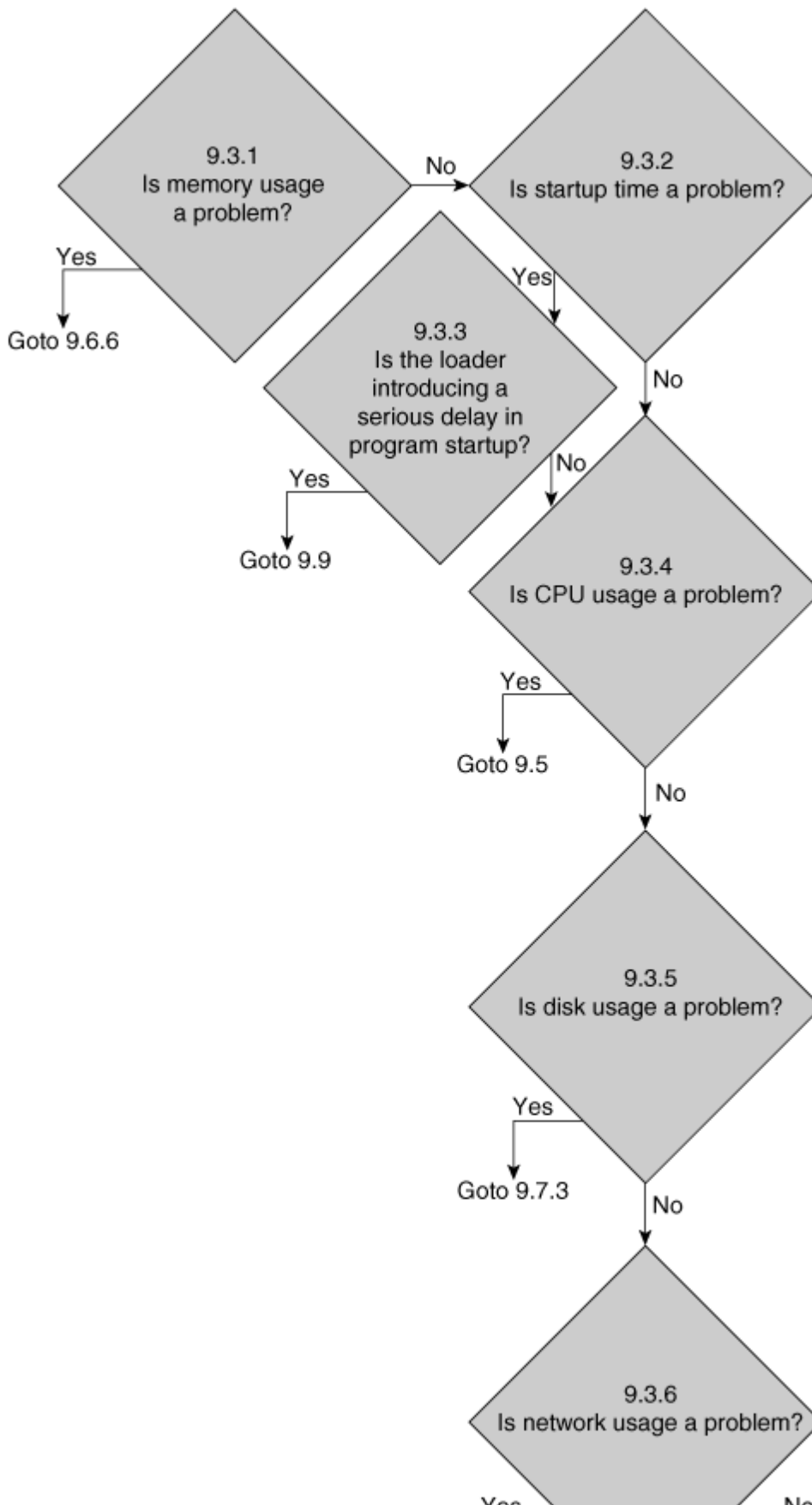
If you have a specific application or program that is not performing as it should, jump to [Section 9.3](#). If no particular application is sluggish and, instead, the entire Linux system is not performing as it should, jump to [Section 9.4](#).

9.3. Optimizing an Application

When optimizing an application, several areas of the application's execution may present a problem. This section directs you to the proper section based on the problem that you are seeing.

Figure 9-1 shows the steps that we will take to optimize the application.

Figure 9-1.



 PREV

< Day Day Up >

NEXT 

9.4. Optimizing a System

Sometimes, it is important to approach a misbehaving system and figure out exactly what is slowing everything down.

Because we are investigating a system-wide problem, the cause can be anywhere from user applications to system libraries to the Linux kernel. Fortunately, with Linux, unlike many other operating systems, you can get the source for most if not all applications on the system. If necessary, you can fix the problem and submit the fix to the maintainers of that particular piece. In the worst case, you can run a fixed version locally. This is the power of open-source software.

Figure 9-2 shows a flowchart of how we will diagnose a system-wide performance problem.

Figure 9-2.

Go to [Section 9.4.1](#) to begin the investigation.

9.4.1. Is the System CPU-Bound?

Use `top`, `procinfo`, or `mpstat` and determine where the system is spending its time. If the entire system is spending less than 5 percent of the total time in idle and wait modes, your system is CPU-bound. Proceed to [Section 9.4.3](#). Otherwise, proceed to [Section 9.4.2](#).

9.4.2. Is a Single Processor CPU-Bound?

Although the system as a whole may not be CPU-bound, in a symmetric multiprocessing (SMP) or hyperthreaded system, an individual processor may be CPU-bound.

Use `top` or `mpstat` to determine whether an individual CPU has less than 5 percent in idle and wait modes. If it does, one or more CPU is CPU-bound; in this case, go to [Section 9.4.4](#).

Otherwise, nothing is CPU-bound. Go to [Section 9.4.7](#).

9.4.3. Are One or More Processes Using Most of the System CPU?

The next step is to figure out whether any particular application or group of applications is using the CPU. The easiest way to do this is to run `top`. By default, `top` sorts the processes that use the CPU in descending order. `top` reports CPU usage for a process as the sum of the user and system time spent on behalf of that process. For example, if an application spends 20 percent of the CPU in user space code, and 30 percent of the CPU in system code, `top` will report that the process has consumed 50 percent of the CPU. Sum up the CPU time of all the processes. If that time is significantly less than the system-wide system plus user time, the kernel is doing significant work that is *not* on the behalf of applications. Go to [Section 9.4.5](#).

Otherwise, go to [Section 9.5.1](#) once for each process to determine where it is spending its time.

9.4.4. Are One or More Processes Using Most of an Individual CPU?

The next step is to figure out whether any particular application or group of applications is using the individual CPUs. The easiest way to do this is to run `top`. By default, `top` sorts the processes that use the CPU in descending order. When reporting CPU usage for a process, `top` shows the total CPU and system time that the application uses. For example, if an application spends 20 percent of the CPU in user space code, and 30 percent of the CPU in system code, `top` will report that the application has consumed 50 percent of the CPU.

First, run `top`, and then add the last CPU to the fields that `top` displays. Turn on Irix mode so that `top`

 PREV

< Day Day Up >

NEXT 

9.5. Optimizing Process CPU Usage

When a particular process or application has been determined to be a CPU bottleneck, it is necessary to determine where (and why) it is spending its time.

Figure 9-3 shows the method for investigating a processs CPU usage.

Figure 9-3.

Go to [Section 9.5.1](#) to begin the investigation.

9.5.1. Is the Process Spending Time in User or Kernel Space?

You can use the `time` command to determine whether an application is spending its time in kernel or user mode. `oprofile` can also be used to determine where time is spent. By profiling per process, it is possible to see whether a process is spending its time in the kernel or user space.

If the application is spending a significant amount of time in kernel space (greater than 25 percent), go to [Section 9.5.2](#). Otherwise, go to [Section 9.5.3](#).

9.5.2. Which System Calls Is the Process Making, and How Long Do They Take to Complete?

Next, run `strace` to see which system calls are made and how long they take to complete. You can also run `oprofile` to see which kernel functions are being called.

It may be possible to increase performance by minimizing the number of system calls made or by changing which systems calls are made on behalf of the program. Some of the system's calls may be unexpected and a result of the application's calls to various libraries. You can run `ltrace` and `strace` to help determine why they are being made.

Now that the problem has been identified, it is up to you to fix it. Go to [Section 9.9](#).

9.5.3. In Which Functions Does the Process Spend Time?

Next, run `oprofile` on the application using the cycle event to determine which functions are using all the CPU cycles (that is, which functions are spending all the application time).

Keep in mind that although `oprofile` shows you how much time was spent in a process, when profiling at the function level, it is not clear whether a particular function is hot because it is called very often or whether it just takes a long time to complete.

One way to determine which case is true is to acquire a source-level annotation from `oprofile` and look for instructions/source lines that should have little overhead (such as assignments). The number of samples that they have will approximate the number of times that the function was called relative to other high-cost source lines. Again, this is only approximate because `oprofile` samples only the CPU, and out-of-order processors can misattribute some cycles.

It is also helpful to get a call graph of the functions to determine how the hot functions are being called. To do this, go to [Section 9.5.4](#).

9.5.4. What Is the Call Tree to the Hot Functions?

Next, you can figure out how and why the time-consuming functions are being called. Running the application with `gprof` can show the call tree for each function. If the time-consuming functions are in a library, you can use `ltrace` to see which functions. Finally, you can use newer versions of `oprofile`

 PREV

< Day Day Up >

NEXT 

9.6. Optimizing Memory Usage

Often, it is common that a program that uses a large amount of memory can cause other performance problems to occur, such as cache misses, translation lookaside buffer (TLB) misses, and swapping.

Figure 9-4 shows the flowchart of decisions that we will make to figure out how the system memory is being used.

Figure 9-4.

Go to [Section 9.6.1](#) to start the investigation.

9.6.1. Is the Kernel Memory Usage Increasing?

To track down what is using the system's memory, you first have to determine whether the kernel itself is allocating memory. Run `slabtop` and see whether the total size of the kernel's memory is increasing. If it is increasing, jump to [Section 9.6.2](#).

If the kernel's memory usage is not increasing, it may be a particular process causing the increase. To track down which process is responsible for the increase in memory usage, go to [Section 9.6.3](#).

9.6.2. What Type of Memory Is the Kernel Using?

If the kernel's memory usage is increasing, once again run `slabtop` to determine what type of memory the kernel is allocating. The name of the slab can give some indication about why that memory is being allocated. You can find more details on each slab name in the kernel source and through Web searches. By just searching the kernel source for the name of that slab and determining which files it is used in, it may become clear why it is allocated. After you determine which subsystem is allocating all that memory, try to tune the amount of maximum memory that the particular subsystem can consume, or reduce the usage of that subsystem.

Go to [Section 9.9](#).

9.6.3. Is a Particular Process's Resident Set Size Increasing?

Next, you can use `top` or `ps` to see whether a particular process's resident set size is increasing. It is easiest to add the `rss` field to the output of `top` and sort by memory usage. If a particular process is increasingly using more memory, we need to figure out what type of memory it is using. To figure out what type of memory the application is using, go to [Section 9.6.6](#). If no particular process is using more memory, go to [Section 9.6.4](#).

9.6.4. Is Shared Memory Usage Increasing?

Use `ipcs` to determine whether the amount of shared memory being used is increasing. If it is, go to [Section 9.6.5](#) to determine which processes are using the memory. Otherwise, you have a system memory leak not covered in this book. Go to [Section 9.9](#).

9.6.5. Which Processes Are Using the Shared Memory?

Use `ipcs` to determine which processes are using and allocating the shared memory. After the processes that use the shared memory have been identified, investigate the individual processes to determine why the memory is being used for each. For example, look in the application's source code for calls to `shmget` (to allocate shared memory) or `shmat` (to attach to it). Read the application's documentation and look for options that explain and can reduce the application's use of shared memory.

Try to reduce shared memory usage and go to [Section 9.9](#).

 PREV

< Day Day Up >

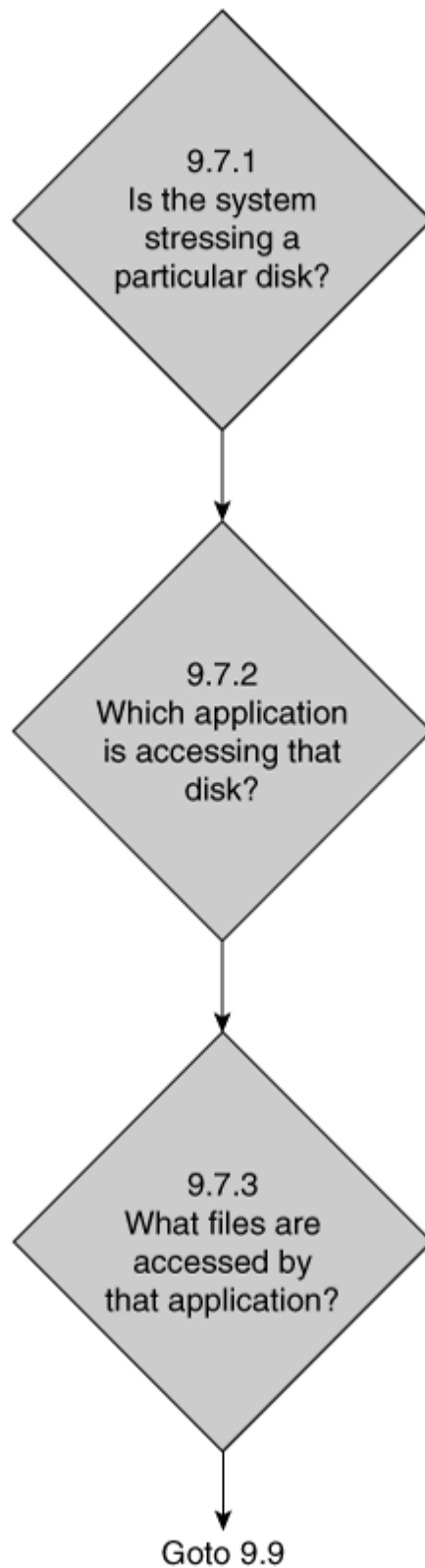
NEXT 

9.7. Optimizing Disk I/O Usage

When you determine that disk I/O is a problem, it can be helpful to determine which application is causing the I/O.

Figure 9-5 shows the steps we take to determine the cause of disk I/O usage.

Figure 9-5.



To begin the investigation, jump to [Section 9.7.1](#)

9.7.1. Is the System Stressing a Particular Disk?

 PREV

< Day Day Up >

NEXT 

9.8. Optimizing Network I/O Usage

When you know that a network problem is happening, Linux provides a set of tools to determine which applications are involved. However, when you are connected to external machines, the fix to a network problem is not always within your control.

Figure 9-6 shows the steps that we take to investigate a network performance problem.

Figure 9-6.

To start the investigation, continue to [Section 9.8.1](#).

9.8.1. Is Any Network Device Sending/Receiving Near the Theoretical Limit?

The first thing to do is to use `ethtool` to determine what hardware speed each Ethernet device is set to. If you record this information, you then investigate whether any of the network devices are saturated. Ethernet devices and/or switches can be easily mis-configured, and `ethtool` shows what speed each device believes that it is operating at. After you determine the theoretical limit of each of the Ethernet devices, use `iptraf` (of even `ifconfig`) to determine the amount of traffic that is flowing over each interface. If any of the network devices appear to be saturated, go to [Section 9.8.3](#); otherwise, go to [Section 9.8.2](#).

9.8.2. Is Any Network Device Generating a Large Number of Errors?

Network traffic can also appear to be slow because of a high number of network errors. Use `ifconfig` to determine whether any of the interfaces are generating a large number of errors. A large number of errors can be the result of a mismatched Ethernet card / Ethernet switch setting. Contact your network administrator, search the Web for people with similar problems, or e-mail questions to one of the Linux networking newsgroups.

Go to [Section 9.9](#).

9.8.3. What Type of Traffic Is Running on That Device?

If a particular device is servicing a large amount of data, use `iptraf` to track down what types of traffic that device is sending and receiving. When you know the type of traffic that the device is handling, advance to [Section 9.8.4](#).

9.8.4. Is a Particular Process Responsible for That Traffic?

Next, we want to determine whether a particular process is responsible for that traffic. Use `netstat` with the `-p` switch to see whether any process is handling the type of traffic that is flowing over the network port.

If an application is responsible, go to [Section 9.8.6](#). If none are responsible, go to [Section 9.8.5](#).

9.8.5. What Remote System Is Sending the Traffic?

If no application is responsible for this traffic, some system on the network may be bombarding your system with unwanted traffic. To determine which system is sending all this traffic, use `iptraf` or `etherape`.

If it is possible, contact the owner of this system and try to figure out why this is happening. If the owner is unreachable, it might be possible to set up `ipfilters` within the Linux kernel to always drop this particular traffic. or to set up a firewall between the remote machine and the local machine to intercept

 PREV

< Day Day Up >

NEXT 

9.9. The End

When you finally arrive here, your problem may or may not be solved, but you will have a lot of information characterizing it. Search the Web and newsgroups for people with similar problems. E-mail them and developers to see how they resolved it. Try a solution and see whether the system's or application's behavior has changed. Every time you try a new solution, jump to [Section 9.2](#) to diagnose the system again, because the application's behavior may change with every fix.

9.10. Chapter Summary

This chapter provided a method for using the Linux performance tools together to track down different types of performance problems. Although it is not possible to capture every type of performance problem that could go wrong, this methodology helps you find some of the more common problems. In addition, if the problem that you face is not covered here, the data that you collect will still be useful because it might open up different areas of investigation.

The next few chapters show this method being used to find performance problems on a Linux system.

Chapter 10. Performance Hunt 1: A CPU-Bound Application (GIMP)

This chapter contains an example of how to use the Linux performance tools to find and fix performance problems in a CPU-bound application.

After reading this chapter, you should be able to

- Figure out which source lines are using all the CPU in a CPU-bound application.
- Use `ltrace` and `oprofile` to figure out how often an application is calling various internal and external functions.
- Look for patterns in the applications source, and search online for information about how an application behaves and possible solutions.
- Use this chapter as a template for tracking down a CPU-related performance problem.

10.1. CPU-Bound Application

In this chapter, we investigate an application that is CPU-bound. It is important to be able to optimize a CPU-bound application because it is one of the most common performance problems.

It is also usually the final frontier for a heavily tuned application.

As the disk and network bottlenecks are removed, the application becomes CPU-bound. In addition, it is often easier to buy faster disks or more memory than to upgrade a CPU, so if a process is CPU-bound, it is an important skill to be able to hunt down and fix a CPU performance problem rather than just buy a new system.

10.2. Identify a Problem

The first step in a performance hunt is to identify a problem to investigate. In this case, I chose to investigate a performance problem that crops up when using GIMP, an open-source image-manipulation program. GIMP can slice and dice various aspects of an image, but it also has a powerful set of filters that can warp and change an image in a variety of ways. These filters can change the appearance of the image based on some complicated algorithms. Typically, the filters take a long time to complete and are very CPU-intensive. One of the filters in particular, Van Gogh (LIC), takes an input image and modifies it so that it looks like a painting done in the style of Van Gogh. This filter takes a particularly long amount of time to complete. When running, the filter uses nearly 100 percent of CPU and takes several minutes to complete. The amount of time to complete depends on the size of the image, the machine's CPU speed, and the values of the parameters passed into the filter. In this chapter, we investigate why this filter is so slow using Linux performance tools and see whether there is any way to speed it up.

10.3. Find a Baseline/Set a Goal

This first step in any performance hunt is to determine the current state of the problem. In the case of the GIMP filter, we need to figure out how much time it takes to run on a particular image. This is our baseline time. Once we have this baseline time, we can then try an optimization and see whether it decreases the execution time. Sometimes, it can be tricky to time how long something takes to execute. It is not as easy as using a stopwatch because the operating system may be scheduling other tasks at the same time that our particularly CPU-intensive job is running. In this case, if other jobs are running in addition to the CPU-intensive one, the amount of wall-clock time could be much greater than the amount of CPU time that the process actually uses. In this case, we are lucky; by looking at `top` as the filter is running, we can see that the `lic` process is taking up most of the CPU usage, as shown in [Listing 10.1](#).

Listing 10.1.

```
[ezolt@localhost ktracer]$ top
```

```
top - 08:24:48 up 7 days, 9:08, 6 users, load average: 1.04, 0.64, 0.76
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	
32744	ezolt	25	0	53696	45m	11m	R	89.6	14.6	0:16.00	lic
2067	root	15	0	69252	21m	17m	S	6.0	6.8	161:56.22	X
32738	ezolt	15	0	35292	27m	14m	S	2.3	8.7	0:05.08	

gimp

From this, we can deduce that GIMP actually spawns a separate process to run when running the filter. So when the filter is running, we can then use `ps` to track how much CPU time the process is using and when it has finished. When we have the PID of the filter using `top`, we can run the loop in [Listing 10.2](#) and ask `ps` to periodically observe how much CPU time the filter is using.

Listing 10.2.

```
while true ; do sleep 1 ; ps 32744; done
```

PID	TTY	STAT	TIME	COMMAND
32744	pts/0	R	2:46	/usr/local/lib/gimp/2.0/plugin-ins/lic -gimp 8 6 -run 0


Note

If you need to time an application but do not have a stopwatch, you can use `time` and `cat` as a simple stopwatch. Just type `time cat` when you want to start timing, and then press `<Ctrl-D>` when you are finished. `time` shows you how much time has passed.

When running the `lic` filter on the reference image (which is a fetching picture of my basement) and using the `ps` method just mentioned to time the filter, we can see from [Listing 10.2](#) that it takes 2 minutes and 46 seconds to run on the entire image. This time is our baseline time. Now that we know the amount of time that the filter takes to run out of the box, we can set our goal for the performance hunt. It is not always clear how to set a reasonable goal for a performance investigation. A reasonable value for a goal can depend on several factors, including the amount of tuning that has already been done on the

 PREV

< Day Day Up >

NEXT 

10.4. Configure the Application for the Performance Hunt

The next step in our investigation is to set up the application for the performance hunt by recompiling the application with symbols. Symbols allow the performance tools (such as `oprofile`) to investigate which functions and source lines are responsible for all the CPU time that is being spent.

For the GIMP, we download the latest GIMP tarball from its Web site, and then recompile it. In the case of GIMP and much open-source software, the first step in recompilation is running the `configure` command, which generates the makefiles that will be used to build the application. The `configure` command passes any flags present in the `CFLAGS` environmental variable into the makefile. In this case, because we want the GIMP to be built with symbols, we set the `CFLAGS` variable to contain `-g3`. This causes symbols to be included in the binaries that are built. This command is shown in [Listing 10.3](#) and overrides the current value of the `CFLAGS` environmental variable and sets it to `-g3`.

Listing 10.3.

```
[root@localhost gimp-2.0.3]# env CFLAGS=-g3 ./configure
```

We then make and install the version of GIMP with all the symbols included, and when we run this version, the performance tools will tell us where time is being spent.

10.5. Install and Configure Performance Tools

The next step in the hunt is to install the performance tools if they are not already installed. Although this might seem like an easy thing to do, it often involves chasing down custom-made packages for a distribution or even recompiling the tools from scratch. In this case, we are going to use `oprofile` on Fedora Core 2, so we have to track down both the `oprofile` kernel module, which in Fedora's case, is only included in the symmetric multiprocessing (SMP) kernels and the `oprofile` package. It also may be interesting to use the `ltrace` performance tool to see which library functions are called and how often they are being called. Fortunately, `ltrace` is included in Fedora Core 2, so we do not have to track it down.

10.6. Run Application and Performance Tools

Next, we run the application and take measurements using the performance tools. Because the `lic` filter is called directly from the GIMP, we have to use tools that can attach and monitor an already running process.

In the case of `oprofile`, we can start `oprofile`, run the filter, and then stop `oprofile` after the filter has been completed. Because the `lic` filter takes up approximately 90 percent of the CPU when running, the system-wide samples that `oprofile` collects will be mainly relevant for the `lic` filter. When `lic` starts to run, we start `oprofile` in another window; when `lic` finishes in that other window, we stop `oprofile`. The starting and stopping of `oprofile` is shown in [Listing 10.4](#).

Listing 10.4.

```
[root@localhost ezolt]# opcontrol --start
Profiler running.
[root@localhost ezolt]# opcontrol --dump
[root@localhost ezolt]# opcontrol --stop
Stopping profiling.
```

`ltrace` must be run a little differently. After the filter has been started, `ltrace` can be attached to the running process. Unlike `oprofile`, attaching `ltrace` to a process brings the entire process to a crawl. This can inaccurately inflate the amount of time taken for each library call; however, it provides information about the number of times each call is made. [Listing 10.5](#) shows a listing from `ltrace`.

Listing 10.5.

```
[ezolt@localhost ktracer]$ ltrace -p 32744 -c
```

% time	seconds	usecs/call	calls	function
43.61	156.419150	254	614050	rint
16.04	57.522749	281	204684	gimp_rgb_to_hsl
14.92	53.513609	261	204684	g_rand_double_range
13.88	49.793988	243	204684	gimp_rgba_set_uchar
11.55	41.426779	202	204684	
gimp_pixel_rgn_get_pixel				
0.00	0.006287	6287	1	gtk_widget_destroy
0.00	0.003702	3702	1	g_rand_new
0.00	0.003633	3633	1	gimp_progress_init
0.00	0.001915	1915	1	gimp_drawable_get
0.00	0.001271	1271	1	
gimp_drawable_mask_bounds				
0.00	0.000208	208	1	g_malloc
0.00	0.000110	110	1	gettext
0.00	0.000096	96	1	gimp_pixel_rgn_init

100.00	358.693497		1432794	total

To get the full number of library calls, it is possible to let `ltrace` run until completion; however, it takes a really long time, so in this case, we pressed `<Ctrl-C>` after a long period of time had elapsed. This will not always work, because an application may go through different stages of execution, and if you stop it early, you may not have a complete picture of what functions the application is calling. However, this short sample will at least give us a starting point for analysis.

 PREV

< Day Day Up >

NEXT 

10.7. Analyze the Results

Now that we have used `oprofile` to collect information about where time is spent when the filter is running, we have to analyze the results to look for ways to change its execution and increase performance.

First, we use `oprofile` to look at how the entire system was spending time. This is shown in [Listing 10.6](#).

Listing 10.6.

```
[root@localhost ezolt]# oprofile -f | less

CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
      TIMER:0 |
samples |      % |
-----|-----|
  69896 36.9285 /usr/local/lib/libgimp-2.0.so.0.0.3
  44237 23.3719 /usr/local/lib/libgimpcolor-2.0.so.0.0.3
  28386 14.9973 /usr/local/lib/gimp/2.0/plug-ins/lic
  16133  8.5236 /usr/lib/libglib-2.0.so.0.400.0
.....
```

As [Listing 10.6](#) shows, 75 percent of the CPU time was spent in the `lic` process or GIMP-related libraries. Most likely, these libraries are called by the `lic` process, a fact that we can confirm by combining the information that `ltrace` gives us with the information from `oprofile`. [Listing 10.7](#) shows the library calls made for a small portion of the run of the filter.

Listing 10.7.

```
[ezolt@localhost ktracer]$ ltrace -p 32744 -c
% time      seconds  usecs/call   calls      function
-----|-----|-----|-----|-----|
 46.13   101.947798      272    374307  rint
 15.72    34.745099      278    124862  g_rand_double_range
 14.77    32.645236      261    124862
gimp_pixel_rgn_get_pixel
 13.01    28.743856      230    124862  gimp_rgba_set_uchar
 10.36    22.905472      183    124862  gimp_rgb_to_hsl
  0.00     0.006832     6832         1  gtk_widget_destroy
  0.00     0.003976     3976         1  gimp_progress_init
  0.00     0.003631     3631         1  g_rand_new
  0.00     0.001992     1992         1  gimp_drawable_get
  0.00     0.001802     1802         1
gimp_drawable_mask_bounds
  0.00     0.000184      184         1  g_malloc
  0.00     0.000118      118         1  gettext
  0.00     0.000100      100         1  gimp_pixel_rgn_init
-----|-----|-----|-----|-----|
100.00   221.006096                873763  total
```

Next, we investigate the information that `oprofile` gives us about where CPU time is being spent in each of the libraries, and see whether the hot functions in the libraries are the same as those that the filter calls. For each of the three top CPU-using images, we ask `oprofile` to give us more details about which functions in the library are spending all the time. The results are shown in [Listing 10.8](#) for the

 PREV

< Day Day Up >

NEXT 

10.8. Jump to the Web

Now that we have found which GIMP functions are used for much of the time, we have to figure out exactly what these functions are and possibly optimize their use.

First, we search the Web for `pixel_rgn_get_pixel` and try to determine what it does. After a few false starts, the following link and information revealed in [Listing 10.11](#) confirm our suspicions about what `pixel_rgn_get_pixel` does.

Listing 10.11.

```
"There are calls for pixel_rgn_get_ pixel, row, col, and rect,
which grab
data from the image and dump it into a buffer that you've
pre-allocated.
And there are set calls to match. Look for "Pixel Regions" in
gimp.h."
(from
http://gimp-plug-ins.sourceforge.net/doc/Writing/html/sect-
image.html )
```

In addition, the information in [Listing 10.12](#) suggests that it is a good idea to avoid using `pixel_rgn_get_` calls.

Listing 10.12.

```
"Note that these calls are relatively slow, they can easily be
the
slowest thing in your plug-in. Do not get (or set) pixels one
at a time
using pixel_rgn_[get|set]_pixel if there is any other way. "
(from
http://www.home.unix-ag.org/simon/gimp/guadec2002/gimp-
plugin/html/imagedata.html)
```

In addition, the Web search yields information about the `gimp_rgb_set_uchar` function by simply turning up the source for the function. As shown in [Listing 10.13](#), this call just packs the red, green, and blue values into a `GimpRGB` structure that represents a single color.

Listing 10.13.

```
void
gimp_rgb_set_uchar (GimpRGB *rgb,
                   guchar r,
                   guchar g,
                   guchar b)
{
    g_return_if_fail (rgb != NULL);

    rgb->r = (gdouble) r / 255.0;
    rgb->g = (gdouble) g / 255.0;
    rgb->b = (gdouble) b / 255.0;
}
```

Information gleaned from the Web confirms our suspicion: The `pixel_rgn_get_ pixel` function is a

 PREV

< Day Day Up >

NEXT 

10.9. Increase the Image Cache

The Web sites explain that GIMP manages images in a slightly counterintuitive fashion. Instead of storing the image in a big array, GIMP instead breaks the image up into a series of tiles. These tiles are 64x64 wide. When a filter wants to access a particular pixel of the image, GIMP loads the appropriate tile, and then finds and returns the pixel value. Each call to retrieve a particular pixel can be slow. If this process is done repeatedly for each pixel, this can dramatically slow down performance as GIMP reloads the tile that it will use to retrieve the pixel values. Fortunately, GIMP provides a way to cache the old tile values and use the cache values rather than reload the tiles at each time. This should increase performance. The amount of cache that GIMP provides can be controlled by using the `gimp_tile_cache_ntiles` call. This call is currently used inside the `lic` and sets the cache to twice as many tiles as the image is wide.

Even though this might seem like enough cache, the GIMP might possibly still need more. The simple way to test this is to increase the cache to a very large value and see whether that improves performance. So, in this case, we increase the amount of cache to 10 times the amount that is normally used. After increasing this value and rerunning the filter, we receive a time of 2 minutes and 40 seconds. This is an increase of 6 seconds, but we have not reached our goal of 2 minutes and 30 seconds. This says that we must look in other areas to increase the performance.

10.10. Hitting a (Tiled) Wall

In addition to using the tile cache, the Web pages suggest a better way to increase the performance of `get_pixel`. By accessing the pixel information directly (without a call to `gimp_pixel_rgn_get_pixel`), it is possible to dramatically increase the performance of the pixel access.

GIMP can provide a way for the filter programmer to directly access the tiles of an image. The filter can then access the image data as if it were accessing a data array, instead of requiring a call into a GIMP library. However, there is a catch. When you have direct access to the pixel data, it is only for the current tile. GIMP will then iterate over all the tiles in the image, allowing you to ultimately have access to all the pixels in the image, but you cannot access them all simultaneously. It is only possible to look at the pixels from a single tile, and this is incompatible with how `lic` accesses data. When the `lic` filter is generating a new pixel at a particular location, it calculates its new value based on the values of the pixels that surround it. Therefore, when generating new pixels on the edge of a tile, the `lic` filter requires pixel data from all the pixels around it. Unfortunately, these pixels may be on the previous tile or the next tile in the image. Because this pixel information is not available, the image filter will not work with this optimized access method.

10.11. Solving the Problem

Because we have determined that the reading of pixel values is taking a significant amount of time, there is yet another solution that may solve the problem. We have to start looking at how the filter runs. As it generates the new image, it repeatedly asks for the same pixel. Because the new pixel value is based on the pixels that surround it, during the course of running the filter on the image, each pixel can be accessed by each of its nine neighbors. This means that each pixel in the image will be read by each of its neighbors and, as a result, it is read at least nine times.

Because the calls to the GIMP library are expensive, we would only like to do them once for each pixel rather than nine times. It is possible to optimize access to the image by reading the entire image into a local array when the filter starts up, and then accessing this local array as the filter runs, rather than calling the GIMP library routines each time we want to access the data. This method should significantly reduce the overhead for looking up the pixel data. Instead of a couple of function calls for each data access, we just access our local array. On filter initialization, the array is allocated with `malloc` and filled with the pixel data. This is shown in [Listing 10.14](#).

Listing 10.14.

```
int g_image_width, g_image_height;
GimpRGB *g_cached_image;

void cache_image(GimpPixelRgn *src_rgn,int width,int height)
{
    static guchar data[4];
    int x,y;
    GimpRGB *current_pixel;

    g_image_width = width;
    g_image_height = height;

    g_cached_image = malloc(sizeof(GimpRGB)*width*height);
    current_pixel = g_cached_image;

    /* Malloc */
    for (y = 0; y < height; y++)
    {
        for (x = 0; x < width; x++)
        {
            gimp_pixel_rgn_get_pixel (src_rgn, data, x, y);
            gimp_rgba_set_uchar (current_pixel, data[0],
data[1], data[2],
data[3]);
            current_pixel++;
        }
    }
}
```


In addition, the `peek` routine has been rewritten just to access this local array rather than call into the GIMP library functions. This is shown in [Listing 10.15](#).

Listing 10.15.

```
static void peek (GimpPixelRgn *src_rgn,
                gint          x,
                gint          y,
                GimpRGB       *color)
```

 PREV

< Day Day Up >

NEXT 

10.12. Verify Correctness?

After we have an optimization that has significantly reduced the filter's runtime, it is necessary to verify that the output image it produces is the same for both the optimized and the unoptimized filter. After loading up the original reference image and comparing it to the newly generated image, I used GIMP to take the difference of the two images. If the reference and optimized image are identical, all the pixels should be zero (black). However, the different image was not perfectly black. Visually, it looked black, but upon closer inspection (using the GIMP color picker), some of the pixels were nonzero. This means that the reference and optimized images are different.

This would normally be a cause for concern, because this might indicate that optimization changed the behavior of the filter. However, a closer examination of the filter's source code showed several places where random noise was used to slightly jitter the image before the filter was run. Any two runs of the filter would be different, so the optimization was likely not to blame. Because the differences between the two images were so visually small, we can assume that the optimization did not introduce any problems.

10.13. Next Steps

We exceeded our goal of 10 percent performance increase in the `lic` filter, so in that sense, we are done with the optimization process. However, if we want to continue to increase performance, we have to reprofile the filter when using the new optimizations. It is important to reprofile the application after each performance optimization is applied and to not rely on old profiles when continuing to optimize the application. The application's runtime behavior can change dramatically after each optimization. If you do not profile after every optimization, you run the risk of chasing a performance problem that no longer exists.

10.14. Chapter Summary

In this chapter, we determined why an application (the GIMP filter `lic`) was CPU-bound. We figured out the base runtime of the application, set a goal for optimization, and saved a reference image to verify that our optimizations did not change the behavior of the application. We used the Linux CPU performance tools (`oprofile` and `ltrace`) to investigate exactly why the application was CPU-bound. We then used the Web to understand how the application worked and to figure out different ways to optimize it. We tried a few different optimizations, but ultimately, we chose the classic performance trade-off of increased memory usage for reduced CPU usage.

We beat our optimization goal, and then verified that our optimization did not change the output of the application.

Whereas this chapter focused on optimizing a single application's runtime, the next chapter presents a performance hunt that concentrates on reducing the amount of latency when interacting with X Windows. Reducing latency can be tricky, because a single event often sets off a nonobvious set of other events. The hard part is figuring out what events are being called and how long each of them are taking.

Chapter 11. Performance Hunt 2: A Latency-Sensitive Application (nautilus)

This chapter contains an example of how to use the Linux performance tools to find and fix a performance problem in a latency-sensitive application.

After reading this chapter, you should be able to

- Use `ltrace` and `oprofile` to figure out where latency is being generated in a latency-sensitive application.
- Use `gdb` to generate a stack trace for each call to a "hot" function.
- Use performance tools to determine where time is spent for an application that uses many different shared libraries.
- Use this chapter as a template to find the cause of high latency in a latency-sensitive application.

11.1. A Latency-Sensitive Application

In this chapter, we investigate an application that is sensitive to slow latency. Latency can be thought of as the time it takes for an application to respond to different external or internal events. An application with a latency performance problem often does not hog the CPU for long periods of time; instead, it only uses a small amount of CPU time to respond to different events. However, the response to the particular events is not swift enough. When fixing a latency performance problem, we need to reduce the latency in response to the various events and figure out what parts of the application are slowing down the response. As you will see, tracking down a latency problem requires a slightly different tactic than tracking down a CPU-intensive problem.

11.2. Identify a Problem

As with the performance problem in the preceding chapter, we have to define what we will investigate and try to overcome it. In this case, we will optimize the time to open a pop-up menu when using the nautilus file manager for the GNOME desktop. In nautilus, pop-up menus are opened by right-clicking anywhere in a nautilus file management window. In this particular case, we will be investigating the performance of the pop-up menus that appear when we right-click the background of an open window rather than when we right-click a particular file or folder.

Why should we optimize this? Even though the amount of time to open a pop-up may be less than a second, it is still slow enough that users can perceive the lag between when they right-click the mouse and when the menu shows up. This sluggish pop-up gives the GNOME user the impression that the computer is running slowly. People notice a slight delay, and it can make interaction with nautilus annoying or give the impression that the desktop is slow.

This particular performance problem is different from the GIMP problem of the preceding chapter. First, the core components of the desktop (in this case, GNOME) are typically more complicated and interlocked than a typical desktop application. The components typically rely on a variety of subsystems and shared libraries to do their work. Whereas the GIMP was a relatively self-contained application, making it easier to profile and recompile when necessary, the GNOME desktop is made up of many different interlocking components. The components may require multiple processes and shared libraries, each performing a different task on behalf of the desktop. nautilus, in particular, is linked to 72 different shared libraries. Tracking down exactly which piece of code is spending time, how much it is spending, and why it is spending it, can be a daunting task.

The significant second difference of this performance investigation from the GIMP investigation is that the times we are trying to reduce are on the order of milliseconds rather than seconds or minutes. When the times are so small, it can be difficult to make sure that the profiling data that you are capturing is actually the result of the event that you are trying to measure rather than just the noise around trying to stop and start the profiling tools. However, this short time period also makes it practical to trace all aspects of what the application does for the interesting period of time.

11.3. Find a Baseline/Set a Goal

As with the previous hunt, the first step is to determine the current state of the problem. To make our lives a little easier, and to avoid some of the profiling problems mentioned in the preceding section, we are going to cheat a little and make the pop-up menu problem look more similar to the long-running CPU-intensive tasks that we measured before. The amount of time that it takes for a single pop-up to appear is in the millisecond range, which makes it hard to accurately measure it with our performance profiling tools. As mentioned previously, it will be difficult to start them and stop them in the proper amount of time and guarantee that we are only measuring what we are interested in (that is, the CPU time spent to open up the actual menu). Here is where we cheat. Instead of opening up the menu just one time, we will open up the menu 100 times in rapid succession. This way, the total amount of time spent opening menus will increase by a factor of 100. This enables us to use our profiling tools to capture information about how the menu is executing.

Because right-clicking 100 times would be tedious, and a human (unless very well trained) could not reliably open up a pop-up menu 100 times in a repeatable manner, we must automate it. To reliably open up the pop-up menu 100 times, we rely on the `xautomation` package. The `xautomation` package is available at <http://hoopajoo.net/projects/xautomation.html>. It can send arbitrary X Window events to the X server, mimicking a user. After downloading the `xautomation` tar file, unzipping and compiling it, we can use it to automate the right mouse click.

Unlike with the GIMP, we cannot simply measure the amount of CPU time used by `nautilus` to evaluate the time needed to create 100 pop-up menus. This is mainly because `nautilus` does not start immediately before a menu is opened and end immediately after. We are going to use wall-clock time to see how much time it takes to complete this task. This requires that the system not have any other things running while we run the test.

[Listing 11.1](#) shows the shell script of `xautomation` commands that are used to open 100 pop-up menus in the `nautilus` file browser. When we run the test, we have to make sure that we have oriented the `nautilus` window so that none of the clicks actually opens a pop-up menu on a folder, and that instead all the pop-ups occur on the background. This is important because the code paths for the different pop-up menus could be radically different.

Listing 11.1.

```
#!/bin/bash
for i in `seq 1 100`;
do
    echo $i
    ./xte 'mousemove 100 100' 'mouseclick 3' 'mouseclick 3'
    ./xte 'mousemove 200 100' 'mouseclick 3' 'mouseclick 3'
done
```

The commands in [Listing 11.1](#) move the cursor to position (100,100) on the X screen, and click the right mouse button (button 3). This brings up a menu. Then they click the right mouse button again, and this closes the menu. They then move to X position (100,100), and repeat the process.

Next, we use `time` to see how much the script of these 100 iterations takes to complete. This is our baseline time. When we do our optimizations, we will check them against this time to see whether they have improved. This baseline time for the stock Fedora 2 version of `nautilus` on my laptop is 26.5 seconds.

Finally, we have to pick a goal for our optimization path. One easy way to do this is to find an application that already has fast pop-up menus and see how long it takes for it to bring up a pop-up menu 100 times. A perfect example of this is `xterm`, which has nice snappy menus. Although the menus are not as complicated as those in `nautilus`, they should at least be considered an upper bound on how

 PREV

< Day Day Up >

NEXT 

11.4. Configure the Application for the Performance Hunt

The next step in the investigation is to set up the application for the performance hunt. Whereas with GIMP we recompiled the application immediately, we are going to take a different approach with nautilus. It may be hard to figure out exactly which pieces need to be recompiled because it relies on so many different shared libraries. Instead of recompiling, we are going to download and install the debugging information for each of the applications and libraries. For Fedora and Enterprise Linux, Red Hat provides a set of `debuginfo` rpms that contain all the symbol information and sources that were generated by the compiler when the application was compiled. Each binary package or library has a corresponding `debuginfo` rpm that contains the debugging information. This allows Red Hat to ship the binaries without the disk-space-consuming debugging information. However, it allows developers, or those investigating performance problems, to download the appropriate `debuginfo` packages and use them. In this case, Red Hat's version of `oprofile` will also recognize the `debuginfo` packages and pick up the symbols when profiling both an application, such as `nautilus`, and a library, such as `gtk`. In this case, we are going to download the `debuginfo` for `gtk`, `nautilus`, `glib`, and the kernel. If `oprofile` finds a library that contributes a significant amount of cycles, but does not allow you to analyze the libraries (`opreport` prints out "no symbols"); this indicates that no debugging information is installed for the library. We can download and install the appropriate `debuginfo` package for the library, and then `oprofile` will have access to the debugging information and will then be able to map the events back to the original functions and source lines.

11.5. Install and Configure Performance Tools

The next step in the hunt is to install the performance tools we need to investigate the problem. As we did in the performance hunt for the GIMP, we will install both `oprofile` and `ltrace`. In this case, we will also download and install `gdb` (if it is not already installed). `gdb` enables us to look at some of the dynamic aspects of the running application.

11.6. Run Application and Performance Tools

Next, we run the application and take measurements using the performance tools. Because we already suspect that a complex interaction of many different processes and libraries might be the cause of the problem, we are going to start with `oprofile` and see what it has to say.

Because we only want `oprofile` to measure events that occur while we are opening the pop-up menus, we are going to use the command line shown in [Listing 11.3](#) to start and stop the profiling immediately before and immediately after we run our script (named `script.sh`) that opens and closes 100 pop-up menus.

Listing 11.3.

```
opcontrol -start ; ./script.sh ; opcontrol -stop
```

Running `opreport` after that profiling information has been collected gives us the information shown in [Listing 11.4](#).

Listing 11.4.

```
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
      TIMER:0 |
samples |      % |
-----|-----|
  3134 27.1460 /usr/lib/libgobject-2.0.so.0.400.0
  1840 15.9376 /usr/lib/libglib-2.0.so.0.400.0
  1303 11.2863 /lib/tls/libc-2.3.3.so
  1048  9.0775 /lib/tls/libpthread-0.61.so
   900  7.7956 /usr/lib/libgtk-x11-2.0.so.0.400.0
   810  7.0160 /usr/X11R6/bin/Xorg
   719  6.2278 /usr/lib/libgdk-x11-2.0.so.0.400.0
   334  2.8930 /usr/lib/libpango-1.0.so.0.399.1
   308  2.6678 /lib/ld-2.3.3.so
   298  2.5812 /usr/X11R6/lib/libX11.so.6.2
   228  1.9749 /usr/lib/libbonoboui-2.so.0.0.0
   152  1.3166 /usr/X11R6/lib/libXft.so.2.1.2
```

As you can see, time is spent in many different libraries. Unfortunately, it is not at all clear which application is responsible for making those calls. In particular, we have no idea which processes have called the `libgobject` library. Fortunately, `oprofile` provides a way to record the shared libraries' functions that an application uses during a run. [Listing 11.5](#) shows how to configure `oprofile`'s sample collection to separate the samples by library, which means that `oprofile` will attribute the samples collected in shared libraries to the programs that called them.

Listing 11.5.

```
opcontrol -p library; opcontrol ---reset
```

After we rerun our test (using the commands in [Listing 11.3](#)), `opreport` splits up the library samples per application, as shown in [Listing 11.6](#).

Listing 11.6.

```
[root@localhost ~]# opreport --output=...
```


 PREV

< Day Day Up >

NEXT 

11.7. Compile and Examine the Source

So now that we have some idea about which calls the application is taking all of the time, we will download the source and compile it. Until now, all of our analysis was possible using the binary packages that Red Hat provides. However, now we need to dive into the source code to examine why the hot functions are called and then, when we figure out why, make changes in the source to alleviate the performance problem. As we did for GIMP, when we recompile, we generate debugging symbols by setting `CFLAGS` to `-g` before we call the configure script.

In this case, we downloaded and installed Red Hat's source `rpm` for `nautilus`, which places the source of `nautilus` in `/usr/src/redhat/SOURCES/`. By using Red Hat's source package, we have the exact source and patches that Red Hat used to create the binary in the package. It is important to investigate the source that was used to create the binaries that we have been investigating, because another version may have different performance characteristics. After we extract the source, we can begin to figure out where the `bonobo_window_add_popup` call is made. We can search all the source files in the `nautilus` directory using the commands in [Listing 11.9](#).

Listing 11.9.

```
[nautilus ]$ find -type f | xargs grep bonobo_window_add_popup

./src/file-manager/fm-directory-view.c:
bonobo_window_add_popup\
(get_bonobo_window (view), menu, popup_path);
```

Fortunately, it appears as if `bonobo_window_add_popup` is only called from a single function, `create_popup_menu`, as shown in [Listing 11.10](#).

Listing 11.10.

```
static GtkWidget *create_popup_menu (FMDirectoryView *view,
                                     const char *popup_path)
{
    GtkWidget *menu;

    menu = GTK_MENU (gtk_menu_new ());
    gtk_menu_set_screen (menu, gtk_widget_get_screen
(GTK_WIDGET (view)));

    gtk_widget_show (GTK_WIDGET (menu));

    bonobo_window_add_popup (get_bonobo_window (view), menu,
popup_path);

    g_signal_connect_object (menu, "hide",
                             G_CALLBACK (popup_menu_hidden),
                             G_OBJECT (view),
                             G_CONNECT_SWAPPED);

    return menu;
}
```

 PREV

< Day Day Up >

NEXT 

11.8. Using gdb to Generate Call Traces

The two different tools for retrieving information about which functions our application was calling gave us different information about which functions were the hot functions. We theorized that the high-level functions that `ltrace` reported were calling the low-level function that `oprofile` was reporting. It would be nice to have a performance tool that could show us exactly which functions were calling `g_type_check_instance_is_a` to verify this theory.

Although no Linux performance tool shows us exactly which functions are calling a particular function, `gprof` should be able to present this callback information, but this requires recompiling the application and all the libraries that it relies on with the `-pg` flag to be effective. For `nautilus`, which relies on 72 shared libraries, this can be a daunting and infeasible task, so we have to look for another solution. Newer versions of `oprofile` can also provide this type of information, but because `oprofile` only samples periodically, it will still not be able to account for every call to any given function.

Fortunately, we can creatively use `gdb` to extract that information. Using `gdb` to trace the application greatly slows down the run; however, we do not really care whether the trace takes a long time. We are interested in finding the number of times that a particular function is called rather than the amount of time it is called, so it is acceptable for the run to take a long time. Luckily, the creation of the pop-up menu is in the millisecond range; even if it is 1,000 times slower with `gdb`, it still only takes about 15 minutes to extract the full trace. The value of the information outweighs our wait to retrieve it.

In particular, to find which functions are calling `g_type_check_instance_is_a`, we are going to use a few different features of `gdb`. First, we use `gdb`'s ability to set a breakpoint at that function. Then we use `gdb`'s ability to generate a backtrace with `bt` at that breakpoint. These two features are really all that we need to figure out which functions are calling this `g_type_check_instance_is_a`, but manually recording the information and continuing would be tedious. We would need to type `bt ; cont` after each time `gdb` breaks in the function.

To solve this, use another one of `gdb`'s features. `gdb` can execute a given set of commands when it hits a breakpoint. By using the `command` command, we can tell `gdb` to execute `bt ; cont` every time it hits the breakpoint in our function. So now the backtrace displays automatically, and the application continues running every time it hits `g_type_check_instance_is_a`.

Now we have to isolate when the trace actually runs. We could just set up the breakpoint in `g_type_check_instance_is_a` at the start of the `nautilus` execution, and `gdb` would show tracing information when it is called by any function. Because we only care about those functions that are called when we are creating a pop-up menu, we want to limit that tracing to only when pop-ups are being created. To do this, we set another breakpoint at the beginning and end of the `fm_directory_view_pop_up_background_context_menu` function. When we reach the first breakpoint, we turn on the backtracing in `g_type_check_instance_is_a`; when we reach the second breakpoint, we exit the debugger. This limits the backtrace information to that which is generated when we are creating a pop-up menu. Finally, we want to be able to save this backtrace information for post-processing. We can use `gdb`'s ability to log its output to a file to save the information for later. The commands passed into `gdb` to extract this information are shown in [Listing 11.12](#).

Listing 11.12.

```
# Prevent gdb from stopping after a screenful of output
set height 0
# Turn on output logging to a file (default: gdb.txt)
set logging on
# Turn off output to the screen
set logging redirect on
# Stop when a popup menu is about to be created
break fm-directory-view.c:5730
```

 PREV

< Day Day Up >

NEXT 

11.9. Finding the Time Differences

Now that we have narrowed down which functions do the work of creating the menu, we want to figure out which pieces are taking up all the time and which pieces are relatively lightweight. A great way to do that, without using any performance tools at all, is to just disable pieces of code and see how it changes performance. Even though this causes nautilus to function incorrectly, it will at least indicate which of the functions are taking all the time.

We first have to start by taking a baseline, because the binaries we are testing have been compiled with different flags than those provided by Red Hat. We time the scripts as we did before. In this case, a run of 100 iterations takes 30.5 seconds on the version that we have compiled ourselves. Next, we comment out the `eel_pop_up_context_menu` call. This shows us how much time it took nautilus to detect the mouse click and decide that a context menu needed to be created. Even if we completely optimize away all the commands in these functions, we will not be able to run any faster than this. In this case, it takes 7.6 seconds to run all 100 iterations. Next, we comment out `bonobo_window_add_popup` to see how much time it costs us to actually call the function that `ltrace` says is taking the most amount of time. If we comment out `bonobo_window_add_popup`, the 100 iterations take 21.9 seconds to complete. This says that if we optimize away the `bonobo_window_add_popup`, it can shave ~8 seconds off the total run, which is nearly a 25 percent improvement.

11.10. Trying a Possible Solution

So, as we have seen, `bonobo_window_add_popup` is an expensive function that must be called every time we want to create a pop-up menu. If we are repeatedly calling it with the same parameters, it may be possible to cache the value it returns from the initial call and use that every time after that instead of repeatedly calling that expensive function. [Listing 11.19](#) shows an example of a rewritten function to do just that.

Listing 11.19.

```
void
fm_directory_view_pop_up_background_context_menu
(FMDirectoryView *view,
                                     GdkEventButton *event)
{
    /* Primitive Cache */
    static FMDirectoryView *old_view = NULL;
    static GtkMenu *old_menu = NULL;

    g_assert (FM_IS_DIRECTORY_VIEW (view));

    /* Make the context menu items not flash as they update to
       proper disabled,
       * etc. states by forcing menus to update now.
       */
    if ((old_view != view) ||
view->details->menu_states_untrustworthy)
    {
        update_menus_if_pending (view);
        old_view = view;
        old_menu = create_popup_menu(view,
FM_DIRECTORY_VIEW_POPUP_PATH_BACKGROUND);
    }

    eel_pop_up_context_menu (old_menu,
        EEL_DEFAULT_POPUP_MENU_DISPLACEMENT,
        EEL_DEFAULT_POPUP_MENU_DISPLACEMENT,
        event);
}
```

In this case, we remember the menu that was generated last time. If we are past it in the same view, and we do not believe that the menu for that view has changed, we just use the same menu that we used last time instead of creating a new one. This is not a sophisticated technique, and it will break down if the user does not open a pop-up menu in the same directory repeatedly. For example, if the user opens a pop-up in directory 1, and then opens one in directory 2, if the user then opens a pop-up in directory 1, nautilus will still create a new menu. It is possible to create a simple cache that stores menus as they are created. When opening a menu, the first check is to see whether these views already have menus in the cache. If they do, the cached menus could be viewed; otherwise, new ones could be created. This cache would be especially useful for some special directories, such as the desktop, computer, or home directory where the user will most likely open a pop-up menu more than once. After applying this proposed solution and timing it with the 100 iterations, the time has dropped to 24.0 seconds. This is a ~20 percent performance improvement, and close to the theoretical improvement that we would get if we did not create the menu at all (21.9 seconds). Creating pop-up menus in various directories worked

 PREV

< Day Day Up >

NEXT 

11.11. Chapter Summary

In this chapter, we determined why a particular component of an application had high latency (pop-up menus in nautilus). We figured out how to automate the creation of the pop-up menus (xautomation) and extend the amount of time that nautilus spent creating pop-up menus (100 iterations). We used `oprofile` to figure out in which function nautilus was spending all of its time. We then used `ltrace` and `gdb` to determine which shared library calls were responsible for making all the calls. After we figured out which library calls were high cost, we tried to reduce or limit the number of times they were called. In this case, we stored a pointer to a new menu when it was allocated and used it later to avoid unneeded reallocations. We created a proposed patch and then ran our performance test against it to see whether performance improved. Performance improved, and functionality did not appear to be affected. The next step is to submit the patch to the nautilus developers for comment. Whereas this chapter focused on optimizing a single application's latency, the next chapter presents a performance hunt that concentrates on solving a system-level performance problem. This type of hunt can often involve investigating many different areas of the system, including hardware (disk, network, and memory) and software (applications, shared libraries, and the Linux kernel).

Chapter 12. Performance Hunt 3: The System-Wide Slowdown (prelink)

This chapter contains an example of how to use the Linux performance tools to find and fix a performance problem that affects the entire system rather than a specific application.

After reading this chapter, you should be able to

- Track down which individual process is causing the system to slow down.
- Use `strace` to investigate the performance behavior of a process that is not CPU-bound.
- Use `strace` to investigate how an application is interacting with the Linux kernel.
- Submit bug reports that describe a performance problem so that an author or maintainer has enough information to fix the problem.

12.1. Investigating a System-Wide Slowdown

In this chapter, we investigate a system-wide slowdown. Initially, we will notice that the system is behaving slowly, and we will use the Linux performance tools to pinpoint the exact cause. This sort of problem happens quite often. As a user or system administrator, you may sometimes notice the Linux machine becoming sluggish or taking a long time to complete a task. It is valuable to be able to figure out why the machine is slowing down.

12.2. Identify a Problem

Once again, our first step is to identify the exact problem that we will investigate. In this case, we are going to investigate a periodic slowdown that occurs when I use my Fedora Core 2 desk_{top}. Typically, the desk_{top} performance is reasonable, but occasionally, the disk starts grinding and, as a result, menus and applications take forever to open. After a while, the disk grinding subsides, and then the desk_{top} behavior goes back to normal. In this chapter, we figure out exactly what is causing this problem, and why.

This type of problem is different from the problems in the two previous chapters, because we initially have absolutely no idea what part of the system is causing the problem. When investigating the GIMP's and nautilus's performance, we knew which application was responsible for the problem. In this case, we just have a misbehaving system, and the performance problem could theoretically be in any part of the system. This type of situation is common. When confronted with it, it is important to use the performance tools to actually track down the cause of the problem rather than just guess the cause and try a solution.

12.3. Find a Baseline/Set a Goal

Once again, the first step is to determine the current state of the problem.

However, in this case, it is not so easy to do. We do not know when the problem will begin or how long it will last, so we cannot really set a baseline without more investigation. As far as a goal, ideally we would like the problem to disappear completely, but the problem might be caused by essential OS functions, so eliminating it entirely might not be possible.

First, we need to do a little more investigation into why this problem is happening to figure out a reasonable baseline. The initial step is to run `top` as the slowdown is happening. This gives us a list of processes that may be causing the problem, or it may even point at the kernel itself.

In this case, as shown in [Listing 12.1](#), we run `top` and ask it to show only nonidle processes (by pressing `<I>` as `top` runs).

Listing 12.1.

```
top - 12:03:40 up 12 min,  7 users,  load average: 1.35, 0.98,
0.53
Tasks:  86 total,   2 running,  84 sleeping,   0 stopped,
0 zombie
Cpu(s):  2.3% us,  5.0% sy,  1.7% ni,  0.0% id, 91.0% wa,
0.0% hi,  0.0% si
Mem:   320468k total,   317024k used,    3444k free,
24640k buffers
Swap:   655192k total,    0k used,   655192k free,
183620k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+
COMMAND
 5458 root      34   19  4920 1944 2828  R   1.7   0.6   0:01.13
prelink
 5389 ezolt    17    0  3088  904 1620  R   0.7   0.3   0:00.70
top
```

The `top` output in [Listing 12.1](#) has several interesting properties. First, we notice that no process is hogging the CPU; both nonidle tasks are using less than 2 percent of the total CPU time. Second, the system is spending 91 percent waiting for I/O to happen. Third, the system is not using any of the swap space, so the grinding disk is NOT caused by swapping. Finally, an unknown process, `prelink`, is running when the problem happens. It is unclear what this `prelink` command is, so we will remember that application name and investigate it later.

Our next step is to run `vmstat` to see what the system is doing. [Listing 12.2](#) shows the result of `vmstat` and confirms what we saw with `top`. That is, ~90 percent of the time the system is waiting for I/O. It also tells us that the disk subsystem is reading in about 1,000 blocks a second of data. This is a significant amount of disk I/O.

Listing 12.2.

```
[ezolt@localhost ezolt]$ vmstat 1 10
procs -----memory----- ---swap-- ----io----
--system-- ----cpu----
 r b  swpd  free  buff  cache   si   so    bi    bo    in
cs us sy id wa
```

 PREV

< Day Day Up >

NEXT 

12.4. Configure the Application for the Performance Hunt

The next step in the investigation is to set up the application for the performance hunt. `prelink` is a small and self-contained application. In fact, it does not even use any shared libraries. (It is statically linked.) However, it is a good idea to recompile it with all the symbols so that we can examine it in the debugger (`gdb`) if we need to. Again, this tool uses the `configure` command to generate the makefiles. We must download the source to `prelink` and recompile it with symbols. We can once again download the source `rpms` for `prelink` from Red Hat. The source will be installed in `/usr/src/redhat/SOURCES`. Once we unpack `prelink`'s source code, we compile it as shown in [Listing 12.6](#).

Listing 12.6.

```
env CFLAGS=-g3 ./configure
make
```

After `prelink` is configured and compiled, we can use the binary we compiled to investigate the performance problems.

12.5. Install and Configure Performance Tools

The next step in the hunt is to install the performance tools. In this case, neither `ltrace` nor `oprofile` will be of help. `oprofile` is used to profile applications that use a significant amount of CPU time, and because `prelink` uses only about 3 percent of the CPU when running, `oprofile` will not help us. Because the `prelink` binary is statically linked and does not use any shared libraries, `ltrace` will also not help us. However, `strace`, the system call tracer, may help, so we need to install that.

12.6. Run Application and Performance Tools

Now we can finally begin to analyze the performance characteristics of the different modes of `prelink`. As you just saw, `prelink` does not spend much time using the CPU; instead, it spends all of its time on disk I/O. Because `prelink` must call the kernel for disk I/O, we should be able to trace its execution using the `strace` performance tool. Because the quick mode of `prelink` does not appear to be that much faster than the standard `full-run` mode, we compare both runs using `strace` to see whether any suspicious behavior shows up.

At first, we ask `strace` to trace the slower full run of `prelink`. This is the run that creates the initial cache that is used when `prelink` is running in quick mode. Initially, we ask `strace` to show us the summary of the system calls that `prelink` made and see how long each took to complete. The command to do this is shown in [Listing 12.7](#).

Listing 12.7.

```
[root@localhost prelink]# strace -c -o af_sum
/usr/sbin/prelink -af
....
/usr/sbin/prelink:
/usr/libexec/autopackage/luau-downloader.bin: Could
not parse '/usr/libexec/autopackage/luau-downloader.bin: error
while
loading shared libraries: libuau.so.2: cannot open shared
object file: No
such file or directory'
...
/usr/sbin/prelink: /usr/lib/mozilla-1.6/regchrome: Could not
parse
'/usr/lib/mozilla-1.6/regchrome: error while loading shared
libraries:
libxpcor.so: cannot open shared object file: No such file or
directory'
...
```

[Listing 12.7](#) is also a sample of `prelink`'s output. `prelink` is struggling when trying to `prelink` some of the system executables and libraries. This information becomes valuable later, so remember it.

[Listing 12.8](#) shows the summary output file that the `strace` command in [Listing 12.7](#) generated.

Listing 12.8.

```
[root@localhost prelink] # cat af_sum
execve("/usr/sbin/prelink", ["/usr/sbin/prelink", "-af"], [/*
31 vars
*/]) = 0
% time      seconds  usecs/call   calls     errors syscall
-----  -
77.87    151.249181      65    2315836         read
11.93     23.163231      55     421593         pread
 3.59      6.976880      63     110585         pwrite
 1.70      3.294913      17     196518         mmap
 1.02      1.977743      32      61774         lstat64
 0.97      1.890977      40      47820         open
 0.72      1.406801     249      5639         vfork
```

 PREV

< Day Day Up >

NEXT 

12.7. Simulating a Solution

The information revealed by `strace` shows that `prelink` is spending a lot of time trying to open and analyze binaries that it cannot possibly prelink. The best way to test whether caching of nonprelinkable binaries could improve `prelink`'s performance is to modify `prelink` so that it adds all these unprelinkable binaries to its initial cache. Unfortunately, adding code to cache these "unprelinkable" binaries could be a complicated process that involves a good amount of knowledge about the internals of the `prelink` application. An easier method is to simulate the cache by replacing all the unprelinkable binaries with a known prelinkable binary. This causes all the formerly unprelinkable binaries to be ignored when quick mode is run. This is exactly what would happen if we had a working cache, so we can use it to estimate the performance increase we would see if `prelink` were able to cache and ignore unprelinkable binaries.

To start the experiment, we copy all the files in `/usr/bin/` to the `sandbox` directory and run `prelink` on this directory. This directory includes normal binaries, and shell scripts, and other libraries that cannot be prelinked. We then run `prelink` on the `sandbox` directory and tell it to create a new cache rather than rely on the system cache. This is shown in [Listing 12.15](#).

Listing 12.15.

```
/usr/sbin/prelink -C new_cache -f sandbox/
```

Next, in [Listing 12.16](#), we time how long it takes the quick mode of `prelink` to run. We had to run this multiple times until it gave a consistent result. (The first run warmed the cache for each of the succeeding runs.) The baseline time in [Listing 12.16](#) is .983 seconds. We have to beat this time for our optimization (improving the cache) to be worth investigating.

Listing 12.16.

```
time /usr/sbin/prelink -C new_cache -q sandbox/
real    0m0.983s
user    0m0.597s
sys     0m0.386s
```

Next, in [Listing 12.17](#), we run `strace` on this `prelink` command. This is to record which files `prelink` opens in the `sandbox` directory.

Listing 12.17.

```
strace -o strace_prelink_sandbox /usr/sbin/prelink -C
new_cache -q
sandbox/
```

Next we create a new directory, `sandbox2`, into which we once again copy all the binaries in the `/usr/bin` directory. However, we overwrite all the files that `prelink` "opened" in the preceding `strace` output with a known good binary, `less`, which can be prelinked. We copy the `less` on to all the problem binaries rather than just deleting them, so that both sandboxes contain the same number of files. After we set up the second sandbox, we run the full version of `prelink` on this new directory using the command in [Listing 12.18](#).

Listing 12.18.

```
[root@localhost prelink]#/usr/sbin/prelink -C new_cache2 -f
sandbox2/
```

 PREV

< Day Day Up >

NEXT 

12.8. Reporting the Problem

Because we have found a problem and potential solution in a pretty low-level piece of system software, it is a good idea to work with the author to resolve the problem. We must at least submit a bug report so that the author knows that a problem exists. Submitting the tests we used to discover the problem helps him to reproduce the problem and hopefully fix it. In this case, we will add a bug report to Red Hat's bugzilla (bugzilla.redhat.com) tracking system. (Most other distributions have similar bug tracking systems.) Our bug report describes the problem that we encountered and the possible solution that we discovered.

When arriving at bugzilla, we first search for bug reports in `prelink` to see whether anyone else has reported this problem. In this case, no one has, so we enter the bug report in [Listing 12.22](#) and wait for the author or maintainer to respond and possibly fix the bug.

Listing 12.22.

From Bugzilla Helper:

```
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.6)
Gecko/20040510
```

Description of problem:

```
When running in quick mode, prelink does not cache the fact
that some
binaries can not be prelinked. As a result it rescans them
every time ,
even if prelink is running in quick mode. This causes the disk
to grind
and dramatically slows down the whole system.
```

```
There are 3 types of executables that it retries during quick
mode:
```

- 1) Static Binaries
- 2) Shell Scripts
- 3) Binaries that rely on unprelinkable binaries. (Such as OpenGL)

```
For 1&2, it would be nice if prelink cached that fact that
these
executables can not be prelinked, and then in quick mode check
their
ctime & mtime, and don't even try to read them if it already
knows that
they can't be prelinked.
```

```
For 3, it would be nice if prelink recorded which libraries
are causing
the prelink to fail (Take the OpenGL case for example), and
record that
```

 PREV

< Day Day Up >

NEXT 

12.9. Testing the Solution

Because we have not solved the problem in the `prelink` code, but instead reported a bug, we cannot test the fixed `prelink` time against our original baseline immediately. However, if the author or maintainer is able to implement the proposed changes, or even find a better way to optimize things, we will be able to check out the performance of the updated version when it arrives.

12.10. Chapter Summary

In this chapter, we started with a misbehaving system and used performance tools to pinpoint which subsystem was used excessively (the disk subsystem as shown by `vmstat`) and which component caused the problem (`prelink`). We then investigated the `prelink` application to determine why it used so much disk I/O (using `strace`). We discovered in `prelink`'s documentation a cached mode that should dramatically reduce disk I/O. We investigated the performance of the cached mode and found that it did not eliminate disk I/O as much as it should, because it was trying to `prelink` files that could not be prelinked. We then simulated a cache that avoided trying to `prelink` files that could not be prelinked and verified that it significantly reduced the amount of disk I/O and runtime of `prelink` in quick mode. Finally, we submitted a bug report to the author of `prelink` in the hopes that the author will recognize the problem and fix it. This chapter was the last chapter of Linux performance hunts.

In the next chapter, the final chapter, we look at the higher-level picture of Linux performance and performance tools. We review methodologies and tools covered in this book and look at some of the areas of Linux performance tools that are ripe for improvement.

Chapter 13. Performance Tools: What's Next?

This chapter contains musings about the current state of Linux performance tools, what still needs to be improved, and why Linux is currently a great platform to do performance investigation.

After reading this chapter, you should be able to

- Understand the holes in the Linux performance toolbox, and understand some of the ideal solutions
- Understand the benefits of Linux as a platform for performance investigation

13.1. The State of Linux Tools

This book tours the current Linux performance tools, how to use them individually, and how to use them together to solve performance problems. As with every aspect of Linux, these performance tools are constantly evolving, so when investigating a problem, it is always a good idea to review the man page or documentation of a performance tool to determine whether its usage has changed. The fundamental functions of the performance tools rarely change, but new features are commonly added, so it is helpful to review the latest release notes and documentation for a given tool.

13.2. What Tools Does Linux Still Need?

As we toured some of the Linux performance tools, we saw some holes in the overall performance-investigation functionality. Some of these holes are the result of kernel limitations, and some exist just because no one has written a tool to solve the problem. However, filling some of these holes would make it dramatically easier to track down and fix Linux performance problems.

13.2.1. Hole 1: Performance Statistics Are Scattered

One glaring hole is that Linux has no single tool that provides all relevant performance statistics for a particular process. `ps` was meant to fill this hole in the original UNIX, and on Linux, it is pretty good but it does not cover all the statistics that other commercial UNIX implementations provide. Some statistics are invaluable in tracking down performance problems—for example, `inblk` (I/O blocks read in) and `oublk` (I/O blocks written out), which indicate the amount of disk I/O a process is using; `vcsw` (voluntary context switches) and `invsw` (involuntary context switches), which often indicate a process was context-switched off the CPU; `msgrcv` (messages received on pipes and sockets) and `msgsnd` (messages sent on pipes and sockets), which show the amount of network and pipe I/O an application is using. An ideal tool would add all these statistics and combine the functionality of many performance tools presented so far (including `oprofile`, `top`, `ps`, `strace`, `ltrace`, and the `/proc` file system) into a single application. A user should be able point this single application at a process and extract all the important performance statistics. Each statistic would be updated in real time, enabling a user to debug an application as it runs. It would group statistics for a single area of investigation in the same location.

For example, if I were investigating memory usage, it would show exactly how memory was being used in the heap, in the stack, by libraries, shared memory, and in `mmap`. If a particular memory area was much higher than I expected, I could drill down, and this performance tool would show me exactly which functions allocated the memory. If I were investigating CPU usage, I would start with overall statistics, such as how much time is spent in system time versus user time, and how many system calls a particular process is making, but then I would be able to drill down into either the system or user time and see exactly which functions are spending all the time and how often they are being called. A smart shell script that used the appropriate preexisting tools to gather and combine this information would go a long way to achieving some of this functionality, but fundamental changes in the behavior of some of the tools would be necessary to completely realize this vision.

13.2.2. Hole 2: No Reliable and Complete Call Tree

The next performance tool hole is the fact that there is currently no way to provide a complete call tree of a program's execution. Linux has several incomplete implementations. `oprofile` provides call-tree generation, but it is based on sampling, so it will not catch every call that is made. `gprof` supports call trees, but it will not be able to profile the full application unless every library that a particular process calls is also compiled with profile support. This most promising tool, `valgrind`, has a skin called `calltree`, described in the section, "5.2.5 `kcachegrind`," in Chapter 5, "Performance Tools: Process-Specific Memory," which has a goal of providing a completely accurate call tree. However, it is still in development and does not work on all binaries.

This call-tree tool would be useful even if it dramatically slowed down application performance as it runs. A common way of using this would be to run `oprofile` to figure out which functions in an application are "hot," and then run the call-tree program to figure out why the application called them. The `oprofile` step would provide an accurate view of the application's bottlenecks when it runs at full speed, and the call tree, even if it runs slowly, would show how and why the application called those functions. The only problem would be if the program's behavior was timing sensitive and it would change if it was run slowly (for example, something that relied on network or disk I/O). However, many problems exist that are not timing sensitive, and an accurate call-tree mechanism would go a long way to fixing these.

13.2.3. Hole 3: I/O Attribution

 PREV

< Day Day Up >

NEXT 

13.3. Performance Tuning on Linux

Even with the holes just mentioned, Linux is still an ideal place to find and fix performance problems. It was written for developers by developers and, as a result, it is very friendly to the performance investigator. Linux has a few characteristics that make it a great platform to track down performance problems.

13.3.1. Available Source

First, a developer has access to most (if not all) source code for the entire system. This is invaluable when tracking down a problem that appears to exist outside of your code. On a commercial UNIX or other operating systems where source is not available, you might have to wait for a vendor to investigate the problem, and you have no guarantee that he will fix it if it is his problem. However, on Linux, you can investigate the problem yourself and figure out exactly why the performance problem is happening. If the problem is outside your application, you can fix it and submit a patch, or just run with a fixed version. If, by reading the source of the Linux code, you realize that the problem is in your code, you can then fix the problem. In either case, you can fix it immediately and are not gated by waiting for someone else.

13.3.2. Easy Access to Developers

The second advantage of Linux is that it is relatively easy to find and contact the developers of a particular application or library. In contrast to most other proprietary operating systems, where it is difficult to figure out which engineer is responsible for a given piece of code, Linux is much more open. Usually, the names or contact information of the developers for a particular piece of software are with the software package. Access to the developers allows you to ask questions about how a particular piece of code behaves, what slow-running code intends to do, and whether a given optimization is safe to perform. The developers are usually more than happy to help with this.

13.3.3. Linux Is Still Young

The final reason that Linux is a great platform on which to optimize performance is because it is still young. Features are still being developed, and Linux has many opportunities to find and fix straightforward performance bugs. Because most developers focus on adding functionality, performance issues can be left unresolved. An ambitious performance investigator can find and fix many of the small performance problems in the ever-developing Linux. These small fixes go beyond a single individual and benefit the entire Linux community.

13.4. Chapter Summary

In this chapter, we investigated a few of the areas where the set of Linux performance tools has shortcomings and proposed some ideal solutions. We also discussed why Linux is a good platform on which to try performance investigation and optimization.

It is up to you, the reader, to change Linux performance for the better. The opportunities for improvement of Linux performance and Linux performance tools abound. If you find a performance problem that annoys you, fix it or report it to the developers and work with them to fix it. Either way, no one else will be hit by the problem, and the entire Linux community benefits.

Appendix A. Performance Tool Locations

The performance tools described in this book originated from many different locations on the Internet. Fortunately, most major distributions have pulled them together and included them in the current versions of their distributions. [Table A-1](#) describes all the tools, provides pointers to their original source locations, and indicates whether they are included in the following distributions: Fedora Core 2 (FC2), Red Hat Enterprise Linux (EL3), and SUSE 9.1 (S9.1).

Table A-1. Locations of Performance Tools

Tool	Distro	Source Location
<code>bash</code>	FC2, EL3, S9.1	http://cnswww.cns.cwru.edu/~chet/bash/bashtop.html
<code>etherape</code>	None	http://etherape.sourceforge.net/
<code>ethtool</code>	FC2, EL3, S9.1	http://sourceforge.net/projects/gkernel/
<code>free</code>	FC2, EL3, S9.1	Part of the <code>procps</code> package: http://procps.sourceforge.net/
<code>gcc</code>	FC2, EL3, S9.1	http://gcc.gnu.org/
<code>gdb</code>	FC2, EL3, S9.1	http://sources.redhat.com/gdb/
<code>gkrellm</code>	FC2, S9.1	http://web.wt.net/~billw/gkrellm/gkrellm.html
<code>gnome-system-monitor</code>	FC2, EL3, S9.1	Part of the GNOME project, and available from: ftp://ftp.gnome.org/pub/gnome/sources/gnome-system-monitor/
<code>gnumeric</code>	FC2, EL3, S9.1	http://www.gnome.org/projects/gnumeric/
<code>gprof</code>	FC2, EL3, S9.1	Part of the <code>binutils</code> package: http://sources.redhat.com/binutils
<code>ifconfig</code>	FC2, EL3, S9.1	Part of the <code>net-tools</code> : http://www.tazenda.demon.co.uk/phil/net-tools/
<code>iostat</code>	FC2, S9.1	Part of the <code>sysstat</code> package: http://perso.wanadoo.fr/sebastien.godard/
<code>ip</code>	FC2	Part of the <code>iproute</code> package: ftp://ftp.inr.com/ip_routing

 PREV

< Day Day Up >

NEXT 

Appendix B. Installing `oprofile`

Although the system profiler `oprofile` is a powerful performance tool, its installation/use can be tricky. This appendix describes some of the issues when installing `oprofile` on Fedora Core 2 (FC2), Red Hat Enterprise Linux (EL3), and SUSE 9.1 (S9.1).

B.1 Fedora Core 2 (FC2)

For FC2, Red Hat provides packages for `oprofile` that should be used rather than those downloaded from the `oprofile` Web site. The uniprocessor kernel does not provide the necessary `oprofile` drivers. Red Hat packages the necessary `oprofile` kernel modules with the `smp` version of the kernel. If you want to run `oprofile`, you must use the `smp` kernel, even if you are running it on a single-processor machine.

B.2 Enterprise Linux 3 (EL3)

For EL3, once again, Red Hat provides packages for `oprofile` that should be used rather than those downloaded from the `oprofile` Web site. The uniprocessor kernel does not provide the necessary `oprofile` drivers. Red Hat packages the necessary `oprofile` kernel modules with the `smp` or `hugemem` versions of the kernel. If you want to run `oprofile`, you must use the `smp` or `hugemem` kernel, even if you are running it on a single-processor machine.

More details on using `oprofile` in EL3 are provided at <http://www.redhat.com/docs/manuals/enterprise/RHEL-3-Manual/sysadmin-guide/ch-oprofile.html>.

B.3 SUSE 9.1

For SUSE 9.1, SUSE provides packages for `oprofile` that should be used rather than those downloaded from the `oprofile` Web site. All versions of the SUSE kernels (`default`, `smp`, and `bigmp`) provide support for `oprofile`, so any of the supplied kernels will work.

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

- % time option
 - ltrace tool
 - strace tool
- %MEM option, top (v. 2.x and 3.x) tool
- %memused option
 - sar (II) tool
- %swpused option
 - sar (II) tool
- +D directory option
 - lsof (List Open Files) tool
 - disk I/O subsystem usage
- +d directory option
 - lsof (List Open Files) tool
 - disk I/O subsystem usage
- annotated-source option
 - gprof command
- assembly option
 - opreport tool
- brief option
 - gprof command
- delay option
 - slabtop tool
- details option
 - opreport tool
- flat-profile option
 - gprof command
- follow-exec option
 - memprof tool
 - application use of memory
- follow-fork option
 - memprof tool
 - application use of memory
- graph option
 - gprof command
- help option
 - kcachegrind tool
 - application use of memory
 - ltrace tool
 - strace tool
- interfaces=name option
 - netstat tool
 - network I/O
- long-filenames option
 - opreport tool
- raw|-w option
 - netstat tool
 - network I/O
- sort option
 - slabtop tool
- source -- option
 - opreport tool
- statistics|-s option
 - netstat tool
 - network I/O
- symbols option
 - opreport tool
- tcp|-t option
 - netstat tool
 - network I/O
- trace-jump=yes|no option
 - kcachegrind tool
 - application use of memory
- udp|-u option
 - netstat tool
 - network I/O
- /+ buffers/cache option
 - free tool
- A option
 - gprof command
- a option
 - opreport tool 2nd
- A option
 - ns command

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

- a option
 - slabtop tool
- active
 - vs. inactive memory 2nd 3rd
- Active option
 - /proc/meminfo file
- active option
 - vmstat II tool
- active option, top (v. 2.x and 3.x) tool
- application optimization
 - CPU usage 2nd
 - disk I/O usage
 - loaders
 - memory
 - network I/O usage
 - startup time
- application performance investigation
 - analyzing tool results 2nd 3rd 4th 5th 6th 7th
 - configuring applications 2nd
 - identifying problems 2nd
 - installing/configuring performance tools
 - latency problems 2nd
 - analyzing time use 2nd
 - analyzing tool results 2nd 3rd 4th
 - configuring applications 2nd
 - identifying problems 2nd
 - installing/configuring tools
 - running applications and tools 2nd 3rd 4th 5th 6th 7th 8th 9th 10th
 - setting baseline/goals 2nd 3rd 4th
 - solutions 2nd 3rd
 - tracing function calls 2nd 3rd 4th 5th 6th
 - running applications and performance tools 2nd 3rd
 - setting baseline/goals 2nd 3rd
 - solutions
 - accessing image tiles 2nd
 - accessing image tiles, with local arrays 2nd 3rd 4th
 - increasing image cache 2nd
 - searching Web for functions 2nd
 - verifying
 - system-wide problems 2nd 3rd 4th 5th 6th
 - configuring application 2nd
 - configuring/installing performance tools
 - running applications/tools 2nd 3rd 4th 5th 6th 7th 8th 9th
 - simulating solution 2nd 3rd 4th 5th 6th
 - submitting bug report 2nd 3rd
 - testing solution
- application tests
 - automating 2nd
- applications
 - CPU cache
 - kernel mode
 - subdividing time use
- time use
 - gprof command 2nd 3rd 4th 5th 6th 7th 8th 9th
 - oprofile (II) tool 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th
- time use versus library time use 2nd
 - ltrace tool 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th
- use of CPU cache
 - cachegrind tool
 - oprofile tool
- use of memory
 - /proc//PID tool 2nd 3rd 4th 5th 6th 7th 8th 9th 10th
 - kcachegrind tool 2nd 3rd 4th 5th 6th 7th 8th 9th 10th
 - memprof tool 2nd 3rd 4th 5th 6th 7th
 - oprofile (III) tool 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th
 - ps tool 2nd 3rd 4th 5th 6th
 - tools supported for Java, Mono, Python, and Perl 2nd
 - valgrind tool 2nd 3rd 4th 5th 6th 7th 8th
- use of shared memory
 - ipcs tool 2nd 3rd 4th 5th 6th 7th
- user mode
- automation of tasks
 - application tests 2nd

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

- b option
 - slabtop tool
 - vmstat tool
- bar() function
 - memprof tool
 - application use of memory 2nd 3rd 4th 5th 6th
- baseline of system performance
- bash shell
 - automating/executing long commands
 - example 2nd 3rd
 - options 2nd 3rd
 - time command 2nd
- bash tool
 - source location
- bi statistic
 - vmstat tool
 - disk I/O subsystem usage
- Blk_read statistic
 - iostat tool
 - disk I/O subsystem usage
- Blk_read/s statistic
 - iostat tool
 - disk I/O subsystem usage
- Blk_wrtn statistic
 - iostat tool
 - disk I/O subsystem usage
- Blk_wrtn/s statistic
 - iostat tool
 - disk I/O subsystem usage
- blocked processes
 - queue statistics 2nd
- bo statistic
 - vmstat tool
 - disk I/O subsystem usage
- bt option
 - GNU debugger
- buff option
 - vmstat II tool
- buffers
 - memory 2nd 3rd 4th
- Buffers option
 - /proc/meminfo file
 - free tool
 - procinfo II tool
- buffers option, top (v. 2.x and 3.x) tool
- bufpg/s option
 - sar (II) tool

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

- c option
 - slabtop tool
- C/C++
 - static versus dynamic languages 2nd
- cache option
 - vmstat II tool
- cache subsystem, CPUs
 - application use of memory use
 - oprofile
 - applications use
 - cachegrind
 - Levels 1 and 2 caches
- Cached option
 - /proc/meminfo file
 - free tool
- cachegrind tool
 - application use of CPU cache
 - oprofile
- caches
 - hot functions
- caches, memory 2nd 3rd 4th
- call trees
 - process time use
 - unreliable/incomplete 2nd
- calls option
 - ltrace tool
 - strace tool
- carrier statistic
 - ip tool
 - network I/O
 - ipconfig tool
 - network I/O
- CODE option, top (v. 2.x and 3.x) tool
- coll/sstatistic
 - sar tool
 - network I/O
- collsns statistic
 - ip tool
 - network I/O
- command option
 - ps command
 - ps tool
 - application use of memory
- COMMAND statistic
 - lsdf (List Open Files) tool
 - disk I/O subsystem usage
- command-line mode, top (v. 2.0.x) tool
- command-line mode, top (v. 3.x.x) tool 2nd
- command-line options
 - memory performance
 - free tool 2nd
 - sar (II) tool
 - slabtop tool 2nd
 - mpstat
 - procinfo tool 2nd
 - sar tool 2nd
 - top (v. 2.0.x) tool
 - vmstat II tool
 - memory performance
 - vmstat tool 2nd
- Committed_AS option
 - /proc/meminfo file
- context switches 2nd 3rd 4th
- count option
 - iostat tool
 - disk I/O subsystem usage
 - sar tool
 - network I/O
 - vmstat tool
 - disk I/O subsystem usage
- CPU cache
 - application use of memory
 - applications use

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

- DATA option, top (v. 2.x and 3.x) tool
- Debian (testing) distribution
- developers
 - source for performance investigation 2nd
- Device field, maps file
 - /proc//PID tool
 - processes, maps file
- device option
 - iostat tool
 - disk I/O subsystem usage
- DEVICE statistic
 - lsdf (List Open Files) tool
 - disk I/O subsystem usage
- Dirty option
 - /proc/meminfo file
- disk I/O performance
 - application use
 - files accessed 2nd
 - single drives 2nd
- disk I/O subsystem performance tools
 - inadequacies 2nd
 - iostat
 - example 2nd 3rd 4th
 - options 2nd 3rd
 - statistics 2nd 3rd
 - lsdf (List Open Files)
 - example 2nd
 - options 2nd 3rd
 - statistics 2nd
 - prelink application
 - running application/tools 2nd 3rd 4th 5th 6th 7th 8th 9th
 - simulating solution 2nd 3rd 4th 5th 6th
 - submitting bug report 2nd 3rd
 - testing solution
 - sar
 - example 2nd
 - options 2nd
 - statistics 2nd 3rd
 - vmstat (ii)
 - example 2nd 3rd 4th
- disk I/O SUBSYSTEM performance tools
 - vmstat (ii)
 - options
- disk I/O subsystem performance tools
 - vmstat (ii)
 - options
 - statistics 2nd 3rd 4th 5th 6th 7th
- disk I/O subsystem usage
 - inadequate performance tools 2nd
 - system-wide performance 2nd
- disk I/O usage
 - application problems
- disks statistic
 - vmstat tool
 - disk I/O subsystem usage
- do option
 - bash shell
- documentation
 - performance investigation 2nd 3rd 4th 5th 6th 7th 8th
- done option
 - bash shell
- dropped statistic
 - ip tool
 - network I/O
 - ipconfig tool
 - network I/O
- dsiz option
 - ps tool
 - application use of memory
- dynamic languages
 - versus static languages 2nd
- dynamic loader
 - ld so tool 2nd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

EDIT NOTE

- Delete entries beginning with performance tools for Ch09
disk or hard disk?

- Earlier refs for vmstat II, average and sample modes were for CPU performance so make that addition
For Ch10, performance investigation
applications, INSERT the type of application problem for GIMP

- Elapsed time option
time command

- environmental variables
ld.so tool 2nd

- errors option
strace tool

- errors statistic
ip tool
network I/O
ipconfig tool
network I/O

- etherape tool
network I/O
example 2nd
options 2nd 3rd
source location

- Ethernet network I/O
ethtool performance tool
options
ip performance tool
example 2nd 3rd 4th
options 2nd 3rd
statistics 2nd
ipconfig performance tool
example 2nd
layers
link layer 2nd
physical layer 2nd
mii-tool performance tool
example 2nd
options 2nd
netstat performance tool
example 2nd 3rd 4th 5th
sar performance tool
example 2nd 3rd

- ethtool tool
source location

- ethtool tool tool
network I/O
options

- etime option
ps command

- Exit status option
time command

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

- fault/s option
 - sar (II) tool
- FD statistic
 - lsdf (List Open Files) tool
 - disk I/O subsystem usage
- Fedora Core 2 (FC2) distribution
 - installing
 - oprofile tool
 - performance tools included 2nd
- file option
 - script command
- File Transport Protocol (FTP)
- foo() function
 - memprof tool
 - application use of memory 2nd 3rd 4th 5th 6th
- forks option
 - vmstat tool
- frame statistic
 - ipconfig tool
 - network I/O
- frames
 - network statistics 2nd
- Free option
 - free tool
 - procinfo II tool
- free option
 - vmstat II tool
- free swap option
 - vmstat II tool
- free tool
 - memory performance
 - example 2nd 3rd 4th
 - options 2nd 3rd 4th
 - statistics 2nd
 - source location
- frmpg/s option
 - sar (II) tool
- FTP (File Transport Protocol)
- function option
 - ltrace tool
- functions
 - memory subsystem use
 - function library size 2nd
 - function text size 2nd
 - heap sizes 2nd
 - process time usage
 - call trees
 - hot functions 2nd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

- gcc (GNU compiler collection)
 - example 2nd 3rd 4th
 - options 2nd 3rd
- gcc tool
 - source location
- gdb (GNU debugger)
 - example 2nd 3rd 4th
 - options 2nd 3rd
- gdb tool
 - source location
 - tracing function calls 2nd 3rd 4th 5th 6th 7th
- get_pixel function 2nd 3rd
- getpixel function
- GIMP application
 - analyzing tool results 2nd 3rd 4th 5th 6th 7th
 - performance investigation
 - configuring applications 2nd
 - identifying problems 2nd
 - installing/configuring performance tools
 - setting baseline/goals 2nd 3rd
 - versus nautilus file manager 2nd
 - running applications and performance tools 2nd 3rd
 - solutions
 - accessing image tiles 2nd
 - accessing image tiles, with local arrays 2nd 3rd 4th
 - increasing image cache 2nd
 - searching Web for functions 2nd
 - verifying
- gimp\#208pixel_rgb_set_uchar function
 - Web search 2nd
- gimp\#208pixel_rgn_get_pixel function
 - Web search 2nd
- gimp_bilinear_rgb function
- gimp_pixel_rgn_get_pixel function
- gimp_rgba_set_uchar function
- gimp_tile_cache_ntiles function
- gkrellm tool
 - network I/O
 - example 2nd
 - options 2nd
 - statistics 2nd
 - source location
- gnome-system-monitor
 - CPU-related options 2nd
 - example 2nd 3rd
- gnome-system-monitor (II) tool
 - memory performance
 - example 2nd
 - options 2nd
- GNU compiler collection (gcc)
 - example 2nd 3rd 4th
 - options 2nd 3rd
- GNU compiler collection. [See [gcc tool](#)]
- GNU debugger (gdb)
 - example 2nd 3rd 4th
 - options 2nd 3rd
- GNU debugger. [See [gdb tool](#)]
- gnumeric spreadsheet
 - example 2nd 3rd 4th
 - options 2nd 3rd
- gnumeric tool
 - source location
- gprof command
 - example 2nd 3rd 4th 5th
 - options 2nd 3rd
- gprof tool 2nd
 - source location

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

hardware

- interrupts 2nd
- performance investigation

hardware and software layers 2nd

- link layer
 - network I/O
- physical layer
 - network I/O 2nd

heap memory subsystem use 2nd

High option

- free tool

HighFree option

- /proc/meminfo file

HighTotal option

- /proc/meminfo file

hot functions

- process time use
 - cache misses

HTTP (Hypertext Transfer Protocol)

HugePages 2nd

Hypertext Transfer Protocol (HTTP)

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

ICMP (Internet Control Message Protocol)

id option

vmstat tool 2nd

idle option

mpstat tool

ifconfig tool

source location

in option

vmstat tool

inactive

vs. active memory 2nd 3rd

Inactive option

/proc/meminfo file

inactive option

vmstat II tool

inactive option, top (v. 2.x and 3.x) tool

inblk (I/O blocks read in) tool

Inode field, maps file

/proc//PID tool

processes, maps file

inprogress IO statistic

vmstat tool

disk I/O subsystem usage

Internet Control Message Protocol (ICMP)

Internet Protocol (IP)

interrupts, hardware 2nd

interval option

iostat tool

disk I/O subsystem usage

sar tool

network I/O

vmstat tool

disk I/O subsystem usage

invcs (involuntary context switches) tool

Involuntary context switches: option

time command

IO: cur statistic

vmstat tool

disk I/O subsystem usage

IO: s statistic

vmstat tool

disk I/O subsystem usage

iostat tool

disk I/O subsystem usage

example 2nd 3rd 4th

options 2nd 3rd

statistics 2nd 3rd

IP (Internet Protocol)

ip tool

network I/O

example 2nd 3rd 4th

options 2nd 3rd

statistics 2nd

source location

ip-fragstatistic

sar tool

network I/O

ipconfig tool

network I/O

example 2nd

options

statistics 2nd

ipcs tool

application use of memory

supported for Java, Mono, Python, and Perl 2nd

application use of shared memory

example 2nd 3rd 4th

options 2nd 3rd

iptraf tool

network I/O

example 2nd 3rd

options 2nd 3rd

source location

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

Java

- memory performance tools
 - application use 2nd
 - static versus dynamic languages 2nd

java command

- Xrunhprof command-line option

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

- kbbuffers option
 - sar (II) tool
- kbcached option
 - sar (II) tool
- kbmemfree option
 - sar (II) tool
- kbmemused option
 - sar (II) tool
- kbswpcad option
 - sar (II) tool
- kbswpfree option
 - sar (II) tool
- kbswpused option
 - sar (II) tool
- kcachegrind tool
 - application use of memory
 - example 2nd 3rd 4th 5th 6th 7th
 - options 2nd 3rd
 - source location
- kernel mode
 - applications
- kernel scheduling
 - context switches
- kernel space
 - CPU usage
- kernel usage
 - system-wide performance

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

I option

- slabtop tool

latency performance investigation

- analyzing time use 2nd
- analyzing tool results 2nd 3rd 4th
- configuring applications 2nd
- identifying problems 2nd
- installing/configuring tools
- running applications and tools 2nd 3rd 4th 5th 6th 7th 8th 9th 10th
- setting baseline/goals 2nd 3rd 4th
- solutions 2nd 3rd
- tracing function calls 2nd 3rd 4th 5th 6th

latency performance problems

- investigating 2nd

ld (The linux loader) tool

- source location

ld.so tool 2nd

- environmental variables 2nd
- example
- options
- statistics 2nd

ldd command

- example 2nd
- options 2nd

ldd tool

- source location

Level 1 and 2 CPU caches

libraries

- memory subsystem use
 - function library size 2nd
- time use versus application time use 2nd
 - ltrace tool 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th
- utility performance helpers
 - example 2nd
 - options 2nd

link layer

link layer, network I/O

Linux kernel

- memory usage (slabs) 2nd 3rd 4th 5th
- time use versus user time use

load average

- queue statistics 2nd

loaders

- application problems

Low option

- free tool

LowFree option

- /proc/meminfo file

LowTotal option

- /proc/meminfo file

lsdf (List Open Files) tool

- disk I/O subsystem usage
 - example 2nd
 - options 2nd 3rd
 - statistics 2nd

lsdf tool

- source location

ltrace command

ltrace tool 2nd

- analyzing results 2nd
 - latency-sensitive applications 2nd 3rd 4th
- example 2nd 3rd 4th 5th
- installing/configuring 2nd
- options 2nd
- running 2nd
 - latency-sensitive applications 2nd 3rd 4th
- source location
- statistics 2nd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

- m option, top (v. 2.x and 3.x) tool
- M option, top (v. 2.x and 3.x) tool
- majflt option
 - ps tool
 - application use of memory
- majflt/s option
 - sar (II) tool
- Major page faults option
 - time command
- Mapped option
 - /proc/meminfo file
- maximum transfer unit (MTUs) 2nd
- mcast statistic
 - ip tool
 - network I/O
- MemFree option
 - /proc/meminfo file
- memory
 - swap memory use
- memory performance
 - application use
 - application use of memory
- memory performance tools
 - /proc/meminfo file
 - example 2nd
 - options 2nd
 - statistics 2nd 3rd
 - application use of memory
 - /proc//PID
 - kcachegrind 2nd 3rd 4th 5th 6th 7th 8th 9th 10th
 - memprof 2nd 3rd 4th 5th 6th 7th
 - oprofile (III) 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th
 - ps 2nd 3rd 4th 5th 6th
 - tools supported for Java, Mono, Python, and Perl 2nd
 - valgrind 2nd 3rd 4th 5th 6th 7th 8th
 - application use of shared memory
 - ipcs 2nd 3rd 4th 5th 6th 7th
- free
 - example 2nd 3rd 4th
 - options 2nd 3rd 4th
 - statistics 2nd
- gnome-system-monitor (II)
 - example 2nd
 - options 2nd
- processes, maps file
 - /proc//PID 2nd 3rd
- processes, status
 - /proc//PID
- processes, status file
 - /proc//PID 2nd 3rd 4th 5th
- procinfo II
 - CPU statistics 2nd
 - example 2nd
 - options
- sar (II)
 - example 2nd 3rd
 - options 2nd 3rd
 - statistics 2nd
- slabtop
 - example 2nd
 - options 2nd 3rd
- top (v. 2.x and 3.x)
 - example 2nd 3rd
 - runtime mode 2nd
 - statistics 2nd
- vmstat II 2nd
 - average mode
 - command-line options
 - example 2nd 3rd 4th 5th
 - output statistics 2nd
- memory subsystem
 - active vs. inactive memory 2nd 3rd
 - kernal usage (slahs)

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

n option

- slabtop tool

nautilus file manager

- latency performance investigation 2nd

 - analyzing time use 2nd

 - analyzing tool results 2nd 3rd 4th

 - configuring applications 2nd

 - identifying problems 2nd

 - installing/configuring tools

 - running applications and tools 2nd 3rd 4th 5th 6th 7th 8th 9th 10th

 - setting baseline/goals 2nd 3rd 4th

 - solutions 2nd 3rd

 - tracing function calls 2nd 3rd 4th 5th 6th

 - versus GIMP application 2nd

nDRT option, top (v. 2.x and 3.x) tool

netstat tool

- network I/O

 - example 2nd 3rd 4th 5th

 - options 2nd 3rd 4th

- source location

network configuration tools

- MTU settings 2nd

network I/O

- layers 2nd

 - link layer

 - physical layer 2nd

- protocol-level network I/O

network I/O performance

- error-prone devices

- limits

- traffic

 - application sockets

 - process time use

 - remote processes 2nd

network I/O performance tools

- etherape

 - example 2nd

 - options 2nd 3rd

- ethtool

 - options

- gkrellm

 - example 2nd

 - options 2nd

 - statistics 2nd

- ip

 - example 2nd 3rd 4th

 - options 2nd 3rd

 - statistics 2nd

- ipconfig

 - example 2nd

 - options

 - statistics 2nd

- iptraf

 - example 2nd 3rd

 - options 2nd 3rd

- mii-tool

 - example 2nd

 - options 2nd

- netstat

 - example 2nd 3rd 4th 5th

 - options 2nd 3rd 4th

- sar

 - example 2nd 3rd

 - options 2nd 3rd

 - statistics 2nd

network I/O usage

- application problems

- system-wide performance

network layer

network performance tools

- inadequate tools 2nd

NODE statistic

- lsuf (List Open Files) tool

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

- o option
 - slabtop tool
- objdump command
 - example 2nd
 - options 2nd
- objdump tool
 - source location
- Offset field, maps file
 - /proc//PID tool
 - processes, maps file
- opannotate tool
 - analyzing results 2nd
 - example
 - options 2nd 3rd
- opcontrol program 2nd
 - event handling 2nd
 - options 2nd
- opreport program 2nd 3rd 4th 5th 6th 7th 8th
- opreport tool
 - analyzing results
 - example 2nd
 - options 2nd 3rd 4th
 - running
 - latency-sensitive applications
- oprofile (II) tool
 - example 2nd 3rd 4th 5th 6th
 - opannotate options 2nd 3rd
 - opreport options 2nd 3rd 4th
 - options
- oprofile (III) tool
 - application use of memory
 - example 2nd 3rd 4th 5th 6th 7th 8th 9th 10th
 - options 2nd
- oprofile tool 2nd 3rd 4th
 - analyzing results 2nd 3rd 4th
 - latency-sensitive applications 2nd 3rd 4th
 - application use of CPU cache
 - oprofile
 - CPU-related options 2nd 3rd
 - example 2nd 3rd 4th 5th 6th
 - installing
 - on Fedora Core 2 FC2
 - on Red Hat Enterprise Linux (EL3) 2nd
 - on SUSE 9.1 (S9.1)
 - installing/configuring 2nd
- opcontrol program
 - event handling 2nd
 - options 2nd
- opreport program 2nd 3rd
 - running
 - latency-sensitive applications 2nd 3rd 4th 5th 6th
 - source location
- option
 - ps tool
 - application use of memory
- oublk (I/O blocks written out) tool
- overruns statistic
 - ip tool
 - network I/O
 - ipconfig tool
 - network I/O

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

- P option
 - slabtop tool
- packets statistic
 - ip tool
 - network I/O
- Page oinoption
 - procinfo II tool
- Page out option
 - procinfo II tool
- Page size option
 - time command
- pages paged in option
 - vmstat II tool
- pages paged out option
 - vmstat II tool
- pages swapped in option
 - vmstat II tool
- pages swapped in/out option
 - vmstat II tool
- partitions statistic
 - vmstat tool
 - disk I/O subsystem usage
- Pathname field, maps file
 - /proc//PID tool
 - processes, maps file
- pcpu option
 - ps command
- peek function
- Percent of CPU this job got option
 - time command
- performance investigation
 - applications
 - analyzing tool results 2nd 3rd 4th 5th 6th 7th
 - configuring applications 2nd
 - identifying problems 2nd
 - installing/configuring performance tools
 - running applications and performance tools 2nd 3rd
 - searching Web for functions 2nd
 - setting baseline/goals 2nd 3rd
 - solutions, accessing image tiles 2nd
 - solutions, accessing image tiles, with local arrays 2nd 3rd 4th
 - solutions, increasing image cache 2nd
 - solutions, verifying
 - automating tasks 2nd 3rd
 - documentation 2nd 3rd 4th
 - guidelines 2nd
 - hardware/software configuration
 - performance results
 - research information/URLs
 - establishing baseline
 - establishing metric
 - establishing target 2nd
 - general guidelines 2nd
 - initial use of performance tools
 - latency-sensitive applications 2nd
 - analyzing time use 2nd
 - analyzing tool results 2nd 3rd 4th
 - configuring applications 2nd
 - identifying problems 2nd
 - installing/configuring tools
 - running 2nd 3rd 4th 5th 6th 7th 8th 9th 10th
 - setting baseline/goals 2nd 3rd 4th
 - solutions 2nd 3rd
 - tracing function calls 2nd 3rd 4th 5th 6th
 - low-overhead tools 2nd
 - multiple tool use 2nd
 - solutions earlier by others 2nd 3rd
 - system-wide slowdown
 - configuring application 2nd
 - configuring/installing performance tools
 - identifying problems 2nd
 - running applications/tools 2nd 3rd 4th 5th 6th 7th 8th 9th
 - setting baseline/goals 2nd 3rd 4th 5th 6th 7th

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

- r/s statistic
 - iostat tool
 - disk I/O subsystem usage
- RAM
- rawsckstatistic
 - sar tool
 - network I/O
- rd_sec/s statistic
 - sar tool
 - disk I/O subsystem usage
- read sectors statistic
 - vmstat tool
 - disk I/O subsystem usage 2nd
- reads statistic
 - vmstat tool
 - disk I/O subsystem usage
- reads: merged statistic
 - vmstat tool
 - disk I/O subsystem usage
- reads: ms statistic
 - vmstat tool
 - disk I/O subsystem usage
- reads: sectors statistic
 - vmstat tool
 - disk I/O subsystem usage
- reads: total statistic
 - vmstat tool
 - disk I/O subsystem usage
- Red Hat Enterprise Linux (EL3)
 - installing
 - oprofile tool 2nd
 - performance tools included 2nd
- requested writes statistic
 - vmstat tool
 - disk I/O subsystem usage
- RES option, top (v. 2.x and 3.x) tool
- rkB/s statistic
 - iostat tool
 - disk I/O subsystem usage
- rrqm/s statistic
 - iostat tool
 - disk I/O subsystem usage
- rsec/s statistic
 - iostat tool
 - disk I/O subsystem usage
- rss option
 - ps tool
 - application use of memory
- runnable processes
 - queue statistics 2nd
- runtime mode, top (v. 2.0.x) tool 2nd 3rd
- runtime mode, top (v. 2.x and 3.x) tool 2nd
- runtime mode, top (v. 3.x.x) tool 2nd 3rd 4th
- RX packets statistic
 - ipconfig tool
 - network I/O
- rxbyt/sstatistic
 - sar tool
 - network I/O
- rxcmp/sstatistic
 - sar tool
 - network I/O
- rxdrop/sstatistic
 - sar tool
 - network I/O
- rxerr/sstatistic
 - sar tool
 - network I/O
- rxfifo/sstatistic
 - sar tool
 - network I/O
- rxfram/sstatistic
 - sar tool

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

- S option
 - ltrace tool
- s option
 - slabtop tool
- sample mode, vmstat II tool
 - memory performance
- sample mode, vmstat tool 2nd 3rd
- sar (II) tool
 - memory performance
 - example 2nd 3rd
 - options 2nd 3rd
 - statistics 2nd
- sar tool
 - CPU-related options 2nd 3rd 4th
 - CPU-related statistics 2nd
 - disk I/O subsystem usage
 - example 2nd
 - options 2nd
 - statistics 2nd 3rd
 - example 2nd 3rd 4th 5th
 - network I/O
 - example 2nd 3rd
 - options 2nd 3rd
 - statistics 2nd
- script command
 - example 2nd 3rd
 - options 2nd 3rd
- script tool
 - source location
- seconds option
 - ltrace tool
 - strace tool
- Secure Shell (SSH) service
- Serial Line Internet Protocol (SLIP)
- Shared option
 - free tool
 - procinfo II tool
- SHR option, top (v. 2.x and 3.x) tool
- si option
 - vmstat II tool
- SIZE statistic
 - lsdf (List Open Files) tool
 - disk I/O subsystem usage
- Slab option
 - /proc/meminfo file
- slabs, memory 2nd 3rd 4th 5th
- slabtop tool
 - memory performance
 - example 2nd
 - options 2nd 3rd
 - source location
- SLIP (Serial Line Internet Protocol)
- so option
 - vmstat II tool
- software
 - performance investigation
- SSH (Secure Shell) service
- startup time of applications
- static languages
 - versus dynamic languages 2nd
- strace tool 2nd
 - disk I/O subsystem usage
 - example 2nd
 - options 2nd
 - source location
 - statistics 2nd
 - system-wide slowdown
 - configuring/installing tool
 - running applications/tools 2nd 3rd 4th 5th
 - simulating solution 2nd 3rd 4th 5th
- SUSE 9.1 (S9.1) distribution
 - installing
 - onprofile tool

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

T option

- top (v. 2.0.x) tool

target for system performance 2nd

TCP (Transport Control Protocol)

- network I/O

tcpsckstatistic

- sar tool

- network I/O

tee command

- example 2nd

- options 2nd

tee tool

- source location

thrashing

time command

- example 2nd 3rd 4th

- options 2nd 3rd

- statistics 2nd

time option

- ps command

time tool

- application use of time

- source location

time use

- applications

- gprof command 2nd 3rd 4th 5th 6th 7th 8th 9th

- oprofile (II) tool 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th

- libraries versus applications 2nd

- ltrace tool 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th

- Linux kernel versus users

- subdividing application use

time use performance tools

- time command

- example 2nd 3rd 4th

- options 2nd 3rd

- statistics 2nd

top (v. 2.0.x) tool 2nd

- command-line mode

- command-line options

- CPU-related options 2nd

- example 2nd 3rd 4th

- runtime mode 2nd 3rd

- sorting/display options

- system-wide statistics 2nd

top (v. 2.x and 3.x)

- memory performance

top (v. 2.x and 3.x) tool

- memory performance

- example 2nd 3rd

- statistics 2nd

- runtime mode 2nd

top (v. 3.x.x) tool

- command-line mode

- command-line options

- CPU-related options

- example 2nd 3rd

- runtime mode

- runtime options 2nd 3rd

- system-wide options

top tool

- source location

- system-wide slowdown 2nd

Total option

- free tool

- procinfo II tool

total reads statistic

- vmstat tool

- disk I/O subsystem usage

total swap option

- vmstat II tool

total, used, free option, top (v. 2.x and 3.x) tool

Totals option

- free tool

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

- UDP (User Datagram Protocol)
- udpsckstatistic
 - sar tool
 - network I/O
- us option
 - vmstat tool 2nd
- usecs/call option
 - ltrace tool
- Used option
 - free tool
- used swap option
 - vmstat II tool
- Usedoption
 - procinfo II tool
- User Datagram Protocol (UDP)
 - network I/O
- user mode
 - applications
- user space
 - CPU usage
- USER statistic
 - lsuf (List Open Files) tool
 - disk I/O subsystem usage
- User time option
 - time command
- users/call option
 - strace tool
- utility performance helpers
 - bash shell
 - example 2nd 3rd
 - options 2nd 3rd
 - GNU compiler collection
 - example 2nd 3rd 4th
 - options 2nd 3rd
 - GNU debugger
 - example 2nd 3rd 4th
 - options 2nd 3rd
 - gnumeric
 - example 2nd 3rd 4th
 - gnumeric spreadsheet
 - options 2nd 3rd
 - ldd command
 - example 2nd
 - options 2nd
 - ltrace command
 - objdump command
 - example 2nd
 - options 2nd
 - script command
 - example 2nd 3rd
 - options 2nd 3rd
 - tee command
 - example 2nd
 - options 2nd
 - watch command 2nd
 - options 2nd 3rd

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

- v option
 - slabtop tool
- valgrind tool
 - application use of memory
 - example 2nd 3rd 4th 5th 6th
 - options 2nd 3rd
 - source location
- vcswh (voluntary context switches) tool
- VIRT option, top (v. 2.x and 3.x) tool
- virtual memory
- Virtual Memory Statistics. [See [vmstat tool](#)]
- VmData statistic
 - /proc//PID tool
 - processes: status file
- VmExe statistic
 - /proc//PID tool
 - processes: status file
- VmLck statistic
 - /proc//PID tool
 - processes: status file
- VmRSS statistic
 - /proc//PID tool
 - processes: status file
- vmsta tool
 - source location
- vmstat II tool
 - average mode
 - memory performance
 - memory performance 2nd
 - command-line options
 - example 2nd 3rd 4th 5th
 - output statistics 2nd
 - sample mode
 - memory performance
- vmstat tool 2nd
 - average mode 2nd 3rd
 - command-line options 2nd
 - CPU-specific statistics 2nd 3rd 4th 5th 6th 7th
 - disk I/O subsystem usage
 - example 2nd 3rd 4th
 - options 2nd
 - statistics 2nd 3rd 4th 5th 6th 7th
 - sample mode 2nd 3rd
 - system-wide slowdown 2nd 3rd
- VmStk statistic
 - /proc//PID tool
 - processes: status file
- Voluntary context switches option
 - time command
- vsz option
 - ps tool
 - application use of memory

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X]

- w/s statistic
 - iostat tool
 - disk I/O subsystem usage
- wa option
 - vmstat tool 2nd
- wa statistic
 - vmstat tool
 - disk I/O subsystem usage
- watch command
 - automating/executing long commands
 - options 2nd 3rd 4th
- Web searches
 - source for performance investigation
- while condition option
 - bash shell
- wkB/s statistic
 - iostat tool
 - disk I/O subsystem usage
- wr_sec/s statistic
 - sar tool
 - disk I/O subsystem usage
- Writeback option
 - /proc/meminfo file
- writes statistic
 - vmstat tool
 - disk I/O subsystem usage 2nd
- writes: merged statistic
 - vmstat tool
 - disk I/O subsystem usage
- writes: ms statistic
 - vmstat tool
 - disk I/O subsystem usage
- writes: sectors statistic
 - vmstat tool
 - disk I/O subsystem usage
- writes: total statistic
 - vmstat tool
 - disk I/O subsystem usage
- written sectors statistic
 - vmstat tool
 - disk I/O subsystem usage
- wrqm/s statistic
 - iostat tool
 - disk I/O subsystem usage
- wsec/s statistic
 - iostat tool
 - disk I/O subsystem usage

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

xautomation package

xeyes command

ltrace tool 2nd 3rd 4th 5th