Red Hat Enterprise Linux

Performance Tuning Guide

Red Hat Enterprise Linux: Performance Tuning Guide

1801 Varsity Drive
Raleigh NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park NC 27709 USA

This document is derived, in part, from papers by Henry Maine, Neil Horman, Rik Faith, Adrian Likins, Sherif Abdelgawad, Ulrich Drepper, Will Cohen, and Daniel Barrange.

A PDF version of this manual is available online at http://people.redhat.com/mbehm/.

Table of Contents

Preface

This guide describes how to determine if your system has performance problems and, if so, what to tune and what *not* to tune.

Caution

While this guide contains information that is field-tested and proven, your unique environment will require that you carefully consider the ramifications of implementing the recommendations made here. It is unlikely that anything catastrophic would occur, but it is always important to have your data and systems backed up, and that you have an implementation reversal plan should the need arise.

1. Who this Manual is For

This manual contains sections on system tuning and application tuning:

- The intended audience for the system tuning section is the same as for the Red Hat course RH401 *Red Hat Enterprise Deployment and Systems Management*; that is, primarily upper-level or senior system administrators. In addition to reading this manual, you should also take the Red Hat course RH442 *Red Hat Enterprise System Monitoring and Performance Tuning*.
  - Chapter 1 *Evidence-Based Performance Analysis and Tuning* gives you a strategy for performance tuning. Once you have determined where the problem is, then you can determine how to fix it—whether this requires reconfigured software, new hardware, or modifications to the application.
  - Chapter 2 *System Tuning Tools for Linux Servers* introduces many of the performance tuning tools that are available and describes how to use them to improve the performance of your system components.
  - Chapter 3 *Understanding the Virtual Memory Management System* describes the virtual memory management system.
  - Chapter 4 *Tuning Linux Servers* describes how to tune various components in your server.
  - Chapter 5 *Increasing Performance Through System Changes* looks at hardware upgrades that you could consider.
  - Appendix A *PerformanceAnalysisScript* is a performance-analysis script that you may find useful.

- The application-tuning chapters are for any application developer.
  - Chapter 6 *Detecting and Preventing Application Memory Problems* describes how to find memory-handling problems in applications. There is also a section on coding techniques that can help avoid memory problems in the future.
  - Chapter 7 *Application Tuning with OProfile* describes using OProfile to tune an application. Instructions are given for both the stand-alone OProfile application and the Eclipse OProfile plug-in.
  - Chapter 8 *Analyzing Java Application Performance Problems Using the IBM JVM* is a case study involving Java application performance problems.

- Appendix B *Additional Resources* lists links to more information.

2. Document Conventions

Certain words in this manual are represented in different fonts, styles, and weights. This highlighting indicates that the word is part of a specific category. The categories include the following:

`Courier font`

Courier font represents `commands`, `file names and paths`, and `prompts`.

When shown as below, it indicates computer output:

```
Desktop         about.html        logs       paulwesterberg.png
Mail            backupfiles       mail       reports
```

`bold Courier font`

Bold Courier font represents text that you are to type, such as: **`xload -scale 2`**

`italic Courier font`

Italic Courier font represents a variable, such as an installation directory: *`install_dir`*`/bin/`

bold font

Bold font represents **application programs**, a button on a graphical application interface (**OK**), or **text found on a graphical interface**.

Additionally, the manual uses different strategies to draw your attention to pieces of information. In order of how critical the information is to you, these items are marked as follows:

Note

Linux is case-sensitive: a rose is not a ROSE is not a rOsE.

Tip

The directory `/usr/share/doc/` contains additional documentation for installed packages.

Important

Modifications to the DHCP configuration file take effect when you restart the DHCP daemon.

Caution

Do not perform routine tasks as root—use a regular user account unless you need to use the root account for system administration tasks.


Warning

Be careful to remove only the listed partitions. Removing other partitions could result in data loss or a corrupted system environment.

3. We Need Feedback

If you have thought of a way to make this manual better, submit a bug report against the documentation component of the product Red Hat Application Server in Bugzilla at: http://bugzilla.redhat.com/bugzilla/

When submitting a bug report, be sure to mention the manual's identifier:

```
rhel-EN-3-PDF-RHI  (2005-05-01T01:08)
```

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

If you have a support question (for example, if you are not sure how to partition your hard drives), use the online support system by registering your product at: http://www.redhat.com/apps/activate/

redhat

Chapter 1.

Evidence-Based Performance Analysis and Tuning

You may see "performance problems" when deploying an application. This manual discusses how to approach the diagnosis of system performance issues, what areas to examine, and how to tune (and when not to tune) systems for various workloads.

Scheduling pressure may make you inclined to try to fix the problem before you diagnose the problem. You may have a list of suggestions based loosely on known operating system problems or the general kind of application being used. For example, if you are using an old RHEL AS 2.1 kernel, you may consider upgrading; if you are using Java, you might try settings that have been used for other Java applications; if you are using TCP/IP, you may try kernel-tuning parameters.

These initial, off-the-cuff suggestions never solve the problem.

Real-world performance problems are not solvable in 5 minutes with a short list of random suggestions. Instead, you must step back, understand the issues, and then make suggestions that have two key qualities:

- The suggestions are based on evidence.
- There is a metric that can be measured both before and after the change.

1.1. Is the Performance Problem 5% or 50%?

When looking at a performance problem, the first thing you must do is understand the level of performance increase required. You must also understand the level of performance increase typically provided by various technical solutions.

For example, binding IRQs to CPUs on SMP systems is standard operating procedure for high-throughput installations, and this is often the first suggestion for "performance problems" on SMP systems. However, a Samba netbench study shows that just enabling IRQ affinity can *decrease* performance. Would you be happy if the first thing someone suggested, without understanding the problem or taking any measurements, moved your performance by about -2%? IRQ affinity is useful for obtaining the last 5% performance improvement, but only after the rest of the system is tuned—it will not help a machine that is having trouble pushing more than 30Mbps through a 100baseT connection.

Rule: Do not try 5% solutions when you need a 50% increase in performance.

1.1.1. Avoid Jumping to Conclusions

It is common to call all application deployment issues "performance problems", and to assume that the "performance problem" is really a Linux kernel issue. This is seldom the case. Further, there might be two or more different issues that are independent, but that appear to be a single issue. Tackle the big issues first: a dozen "5% tweaks" will not fix the broken Ethernet switch to which the machine is attached.

How do you avoid jumping to conclusions? Gather evidence.

1.2. Evidence-Based Analysis

Evidence-based analysis requires:

1. *Developing* a model of the system in which you diagram all of the components.
2. *Planning* what type of evidence you need to gather.
3. *Gathering* evidence from the running application.
4. *Analyzing* the evidence.

1.2.1. Developing a Model

Construct a model of the system you want to tune—including all of the components. This may require separate diagrams for each subsystem.

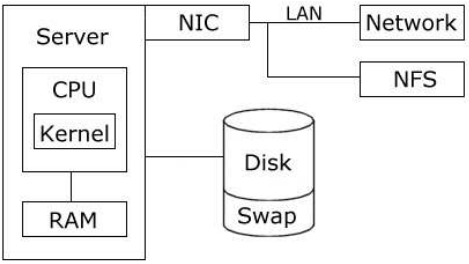

Figure 1-1. Basic System Diagram

System Components	
CPU MHz	RHEL Version
RAM	Bus
Disk Type	Swap Size
H/W RAID	S/W RAID
NFS	NFS Protocol
NIC	LAN

Once you have mapped the system, calculate the maximum theoretical data flows. This will be the baseline to which you will compare your observations.

1.2.2. Planning What Evidence to Gather

Your goal is to gather evidence that leads you to understand several basic facts:

- Why do you think there is a performance problem?

  The answer may be obvious (for example, an application can process 1000 transactions per second (tps) for long periods, but when the rate goes above 2000 tps, the machine hangs). However, in

many cases, the answer is less clear (for example, the application rate was 1000 tps under an older Solaris® machine and is 2000 tps on a new Linux machine, but does not "scale").

- What is the scope of the problem?

  Is the problem being experienced by a single user, or by all users? Does the problem appear only at certain times of the day, certain days of the week, or certain times of the year? If the onset of the problem was sudden, was it preceded by any change to the system (an upgrade to software, the replacement of hardware, or the addition of a new machine to the network)?

- What *metric* can be used to measure performance from one test run to another? Is this the *right* metric for the problem?

  If you are to successfully solve a performance problem, you must have a metric by which to judge your progress. Obvious metrics are things such as bytes per second, packets per second, or transactions per second.

- What are your goals? What is the measure of success?

  In the case of an application that has been ported from Solaris to Linux, you may be getting good performance under Linux and want only to rule out problems with the Linux kernel that may be limiting scalability. (In cases such as this, the application is usually hitting internal application-related performance bottlenecks that were never seen under Solaris because the transaction rate was never high enough to trigger the problem).

1.2.3. Gathering Evidence from the Running Application

After determining your metric and specific goals, the next step is to examine the running application. This is often difficult, especially when the application runs only in a production environment.

One of the main goals of your initial analysis will be to rule out common problems. With this goal in mind, a low-impact script (`PerformanceAnalysisScript.sh`) is provided with this document (see Appendix A *PerformanceAnalysisScript*) and online at http://people.redhat.com/mbehm/. This script can be run by hand before, during, and after the performance event (or you can modify it to be run every 30 minutes from `cron`). The script provides enough information to rule out obvious problems with swapping, process size, process count, Ethernet bandwidth availability, network errors, disk I/O bandwidth, disk errors, and interrupt counts. *Always* obtain at least two runs of any data-gathering scripts so that you can compare a "normal, loaded system" with the system when it is not performing as expected.

A complete list of all the things you should look for at this point would be large. Initially, see if there are common problems that may have been overlooked:

- Performance always degrades when the system starts swapping.
- Network performance can be poor due to bad hardware or a bad negotiation.
- Common applications errors that will result in poor Linux performance include such things as polling for I/O instead of using poll/select (or using these calls with a zero timeout!).

1.2.4. Analyzing the Evidence

After you have the run-time results in hand, you will understand the problem, the goal, the application, and the environment.

First, perform informal calculations (based on the performance logs—perhaps the output of a script or of `sar(1)` or `sal`) to ensure that the goals are realistic:

- Is there sufficient capacity: is there enough memory to prevent swapping? Are there enough CPU cycles to do the required computations in the required amount of time?

The `vmstat` utility will show how many processes are in the run queue. If more processes are waiting to run ("`r`" under "`procs`") than the machine has CPUs, then the system is probably CPU bound. Note that this does not necessarily mean it is running at maximum potential capacity—programming errors that waste CPU cycles on useless work will cause a system to appear CPU bound, but small changes in programming technique may be able to improve performance dramatically.

- Is there sufficient bandwidth: is there enough memory, PCI, network, and disk I/O bandwidth to service the number and size of transactions required to achieve the goals?

  If the `vmstat` utility shows many processes in the blocked statistic ("`b`" under "`procs`"), then the system may be I/O bound. This could be because of slow disks or slow Ethernet cards. It could also be because very fast cards have not been placed on a 64-bit bus. Check the negotiated Ethernet rate and compute the number of bits throughput. You can usually assume 5MBps for 10baseT, 80-90Bps for 100baseT, and 300-1000Mbps for 1000baseT, depending on cards, PCI bus width and speed, and drivers.
- Are you using hardware that enables fast performance, such as SCSI drives rather than IDE disk drives?

Second, determine what sort of improvement you need:

- Does the system appear to be highly tuned?

  If so, now is the time to make a list of "performance tweaks" that you predict will help eke out the remaining 5-10% of the performance requested. Make sure that you make only *one* change at a time and re-run the performance metrics to provide evidence that your tweak helped. Many tweaks have well documented performance "gains" in the -5% to 5% range—make certain you do not make things worse for your particular workload.
- Does the system or application crash, hang, or require a 50-100% improvement in performance? This is the case where applying small performance tweaks fails dramatically. However, Red Hat provides some options:
  - You can take the Red Hat course on system monitoring and performance tuning, RH442 (http://www.redhat.com/training/architect/courses/rh442.html).
  - You can engage Red Hat consulting services http://www.redhat.com/services/focus/hpc/).

- Confirm that you are running the latest update to your operating system. Red Hat updates fix bugs that can affect performance, so if the system is an old installation, run `up2date`.

Third, check the performance logs for obvious system-level problems:

- Is the system swapping? Most production systems are implemented to avoid swapping, so this is unusual.
- Are there networking or other issues that may have been overlooked? (for example, are there any errors reported in `ifconfig`?)
- Are some resource limits being exceeded? Look especially at the number of open file descriptors, the number of processes, and the number of processes linked with `libpthread` (these were limited in various ways in AS2.1, for example).

  Try installing and running `dklimits`, which is described in Section 2.2 *System-Monitoring Utilities*.
- Are some system components under their limits? See Section 5.1 *Can You Upgrade Your Hardware?*.

- Make sure nothing suspicious is in `/var/log/messages`. This may include hard disk or other hardware errors that have been overlooked.
- Is there any evidence of any known kernel-implementation issue? These are rare, but large enterprise installations may uncover them.
- Are there any binary-only modules loaded in the kernel, and are there any known issues with them? Just pointing to the first binary-only module you notice qualifies as a random answer—unless you have evidence that that particular module causes a problem.

Fourth, check the performance logs (from the `PerformanceAnalysisScript.sh` script (see Appendix A *PerformanceAnalysisScript*), from your custom script, or from `sar(1)` for obvious application-level problems:

- Is the application using 100% of the CPU? Why? That is, is the application really doing work, or did someone implement a call to `poll(2)` incorrectly?
- Is the memory footprint growing without bound? Is this by design or by accident?
- For ported applications, does the application take advantage of, or rely on, a feature (or implementation artifact) that is not available on Linux?
- Is the application making use of a feature that works well under Solaris that does not work well under Linux (for example, streams, certain uses of AI/O, threads in AS2.1)?

1.2.5. Summarize Findings and Propose an Experiment

After gathering and analyzing the evidence you have, prove your assertions by performing an experiment and measuring a change in the *metric* you have previously identified.

Experiments should change one thing at a time. This is often difficult, especially when testing on production systems. However, if you change more than one variable, you will not know which change provided an improvement (and you will miss the possibility that one change makes a positive impact that was canceled out by another change). *This is extremely important.*

1.3. Evidence-Based Tuning

System tuning should never be performed without experimental evidence that the tuned parameters have a positive impact on system performance. Sometime, this evidence can be gathered simply by getting a metric for a run, changing a parameter, and getting a metric for a second run. This is the *rare* case, however.

The more common case is that, without changing anything, the metric you are measuring will have 5-10% variation in it. Therefore, if you make a small tuning change that is known to generally produce a -5% to 5% performance change, doing a pair of runs is *worthless*. (In this case, a series of runs before and after the tuning change and appropriate statistical techniques (for example, ANOVA) can demonstrate a difference. Further, such small changes are likely to be dominated by other factors and are not worth the effort to make.)

If you are unable to demonstrate that a tuning change makes a measurable difference in performance, then *do not make the change*. Modern Linux kernels are designed to be self-tuning, and random changes you make could impact the effects of automatic tuning as the system runs for long periods of time.

Performance tuning is not without risks. If the machine is used for more than one task, you may be inadvertently degrading the performance of another application (for example, you may increase I/O throughput but severely degrade interactivity).

redhat

Chapter 2.

System Tuning Tools for Linux Servers

This chapter describes some useful tools and utilities.

2.1. Benchmarking Utilities

A good set of benchmarking utilities is very helpful in doing system-tuning work. It is impossible to duplicate "real-world" situations, but that is not really the goal of a good benchmark. A good benchmark typically tries to measure the performance of one particular thing very accurately. If you understand what the benchmarks are doing, they can be very useful tools.

Some useful benchmarking tools are listed below. Note that they have some overlap in functionality with the tools described later in this chapter.

`LMbench`

`LMbench` is a suite of simple, portable benchmarks that measure bandwidth (memory operations and TCP) as well as latency.

See http://www.bitmover.com/lmbench/ for details.

`Bonnie`

`Bonnie` is a utility for testing driver performance. Its only drawback is it sometimes requires the use of huge datasets on large-memory machines to get useful results.

`Bonnie` (http://www.textuality.com/bonnie/) has been around forever, and the numbers it produces are meaningful to many people. Thus, it is a good tool for producing information to share with others.

`Bonnie` produces the following measures:

- **Block Reads/Writes:** In these tests, `bonnie` tries to read a file it has already written in blocks of 8 KB at a time. It does not do anything useful with the data it reads, it just reads a block of data in, then reads the next block in when that is done. This gives some sort of indication how fast the operating system is at sequential disk reads. Keep in mind that this is a regular file read through the file system; no Async I/O and no Raw Disk I/O is being done here. As a result, do not try to compare these results to the disk-read results from something like IOMeter under Windows NT. They are testing two totally separate things (IOMeter uses Raw I/O and Async I/O under NT to get its speed, which totally bypasses all the file system layer entirely and tests only the speed of the disks and controller). By going through the file system, you get a closer-to-real-life metric as you see just how fast a normal application can expect the operating system to be able to feed it data. Most applications are not Async I/O and Raw I/O aware and will not be able to muster the numbers that IOMeter would suggest. For example, Microsoft Word under Windows NT does *not* use Raw I/O and Async I/O, so the numbers you get from `bonnie` would be a closer approximation than are the numbers from IOMeter as to how fast NT could supply a large file to Microsoft Word. The same is true for writes.
- **Character Reads/Writes:** In these tests, `bonnie` writes to, or reads from, the file one byte at a time using either `getc` or `putc`. This tests only the speed of your libc library (as how the libc library implements the buffering of these single-byte requests and how efficient it is outside of that determines the speed of this test). Under linux, the speed of putc/getc is also drastically affected by whether or not you use the thread safe version. When using glibc-2.0 or later, using the `putc_unlocked()` and `getc_unlocked()` routines greatly improve performance on these tests.

- **Rewrites:** In this test, `bonnie` reads an 8 KB chunk of data, modifies one byte, then writes that 8k chunk back to the original file. This tends to stress the file-system code quite heavily. The file system has to continually update data blocks on the disk or re-allocate blocks on the disk, etc. It also stresses how well the operating system caches data. This would most closely approximate what a smaller SQL database would be like during update phases. (`bonnie` does not even begin to tell you anything about databases, such as Oracle, that want to have access to plain, raw devices and to have no operating system intervention in their read/write patterns; it mimics only smaller databases that utilize plain files to store information in.)
- **Random Seeks:** This provides a reasonable test as to how well your operating system orders the requests it gets. This test randomly seeks into the file that was written and reads a small block of data. The more of these you can complete per second, the faster your operating system is going to be when performing real work. Since the biggest bottleneck to hard disks is not the actual disk throughput, but instead is the disk-seek times, better numbers here will mean better performance overall. This is especially true in multi-user environments.

There is also a somewhat newer version of `bonnie` that is called `bonnie++` (http://www.coker.com.au/bonnie++/) that fixes a few bugs, and includes extra tests.

`dkftpbench`

An ftp benchmarking utility that is designed to simulate real-world ftp usage (large number of clients, throttle connections to modem speeds, etc.). It also includes the useful `dklimits` utility.

`dkftpbench` is available from Dan Kegel's page (http://www.kegel.com/dkftpbench/).

2.1.1. Web Server Benchmarking Tools

You might want to apply benchmarking to high-traffic web sites or when you have an unexpected load spike (for example, a virus is flooding your servers with bogus requests).

`autobench`

`autobench` is a simple Perl script for automating the process of benchmarking a web server (or for conducting a comparative test of two different web servers). The script is a wrapper around httperf. `Autobench` runs httperf a number of times against each host, increasing the number of requested connections per second on each iteration, and extracts the significant data from the httperf output, delivering a CSV or TSV format file which can be imported directly into a spreadsheet for analysis/graphing.

For more information on `autobench`, see http://www.xenoclast.org/autobench/. To download, go to: http://www.xenoclast.org/autobench/downloads

`http_load`

`http_load` is a nice, simple http benchmarking application that does integrity checking, parallel requests, and simple statistics. `http_load` generates a load based off a test file of URLs to access, so it is flexible.

`http_load` is available from ACME Labs (http://www.acme.com/software/http_load/).

`httperf`

`httperf` is a popular web server benchmark tool. It provides a flexible facility for generating various HTTP workloads and for measuring server performance. The focus of `httperf` is not on implementing one particular benchmark but on providing a robust, high-performance tool that facilitates the construction of both micro- and macro-level benchmarks. The three distinguishing characteristics of `httperf` are its robustness, which includes the ability to generate and sustain server overload, support for the HTTP/1.1 protocol, and its extensibility to new workload generators and performance measurements.

For information, see http://www.hpl.hp.com/personal/David_Mosberger/httperf.html To download, go to ftp://ftp.hpl.hp.com/pub/httperf/

`WebStone`

`WebStone` is a highly-configurable client-server benchmark for HTTP servers. It makes HTTP 1.0 GET requests for specific pages on a web server and measures the throughput and latency of each HTTP transfer. By default, only statistical data are returned, but you can request data for each transaction. WebStone also reports transaction failures ("Connection Refused") errors.

For more information, see http://www.mindcraft.com/webstone/

2.2. System-Monitoring Utilities

Many system-monitoring utilities report the system load. A heavily loaded system will have a load average that is greater than two times the number of CPUs. Note that high loads may indicate a machine that is CPU-bound, but it can also indicate memory-bound or I/O-bound systems.

These system-monitoring utilities have some overlap in functionality with the tools described in Section 2.1 *Benchmarking Utilities*.

`xload`

`xload` provides a graphical view of the system load. Of course, this requires that you run the X Window system.

```
xload -scale 2
```

`uptime`

If you are not running the X Window system, you can see a snapshot of the load average for the previous 1, 5, and 15-minute periods by using `uptime`:

```
uptime
```

This provides information similar to the following:

```
9:10:01 up 120 days, 0 min,  9 users, load average: 9.23, 0.96, 0.00
```

The value for `users` is the number of interactive login sessions on the system. The load average is the number of processes that are waiting to run or have been blocked while waiting for I/O. Uniprocessor machines should give values around 1.00; multi-processor machines might run up to 10 times that rate.

`vmstat`

`vmstat`, which is part of the `procps` package, provides information about virtual memory.

Here is a sample `vmstat` output from a lightly used server:

```
procs                  memory    swap      io     system      cpu
 r  b   swpd  free   buff  cache  si  so  bi  bo in  cs us sy id  wa
 0  0 300492 16872 391760 436980   0   1   0   1  1   0  0  0  1   1
```

And here is some sample output from a heavily used server:

```
procs                  memory    swap      io     system           cpu
 r  b swpd   free   buff cache  si  so  bi  bo   in    cs  us  sy id wa
16  0 2360 264400  96672  9400   0   0   0   1   53    24   3   1 96  1
24  0 2360 257284  96672  9400   0   0   0   6 3063 17713  64  36  0  1
15  0 2360 250024  96672  9400   0   0   0   3 3039 16811  66  34  0  1
```

The most interesting number here is the first one, "`r`", in the `procs` (processes) section; this is the number of the process that are on the run queue. This value shows how many processes are ready to be executed, but cannot be ran at the moment because other processes need to finish.

For lightly loaded systems, this should be something less than two to four times the number of processors (so, less than 8 on a dual-processor machine).

The other main `procs` value is `b` (blocked processes). If a value regularly appears in this column, the system is waiting on disk access or network access.

In the `swap` area, `si` in the number of KB of memory swapped in and `so` in the number of KB of memory swapped out.

Other interesting values include the "system" numbers for `in` and `cs`. The `in` value is the number of interrupts per second a system is getting. A system doing a lot of network or disk I/O will have high values here, as interrupts are generated every time something is read or written to the disk or network.

The `cs` value is the number of context switches per second. A context switch occurs when the kernel has to remove the information stored in the processor's registers for the current process and replace it with the context information for a process that has just preempted the current process. It is actually *much* more complicated than that, but that is the basic idea. Excessive context switches are bad, as it takes a fairly large number of cycles to perform a context switch. Consequently, if you are doing lots of context switches, you are spending all your time changing jobs and not actually doing any work.

`vmstat -s` provides a list of statistics including "IO-wait cpu ticks", "pages paged in", and "pages paged out".

Chapter 1 *Evidence-Based Performance Analysis and Tuning* has an example of using `vmstat` (Section 1.2.4 *Analyzing the Evidence*).

`ps`

The `ps` displays a snapshot of the current processes. Consider using these options:

```
ps -eo pid,%cpu,vsz,args,wchan
```

This shows every process, their pid, % of cpu, memory size, name, and what syscall they are currently executing.

A basic form of the command is:

```
ps aux
```

To view a particular process over time:

`watch --interval=`*`interval`*` "ps auxw | grep `*`pid`*`"`

To view all processes in tree form:

`pstree`

Parameter	Show for every process:
args	Command name
%cpu	Percentage of CPU usage
comm	Name of the executable
cmajflt	Number of major page faults for the process and all its children
cminflt	Number of minor page faults for the process and all its children
cmdline	Command that is running
majflt	Number of major page faults
minflt	Number of minor page faults
pid	Process identifier
pstree	View all processes in tree form

Parameter	Show for every process:
rss	Resident set size in KB
vsz	Memory size (also written `vsize`)
wchan	The syscall currently executing

Table 2-1. Selected Parameters for ps

`top`

The `top` command displays the processes that are currently using the most processor time. In effect, this is `ps` run every 5 seconds. Use the `s` (secure) and `d x` (duration *`seconds`*) options to monitor operations.

```
top sd 10
```

You can also sort by the titles of the columns; for example, by the SIZE of the applications.

Note that `top` may execute often enough to influence the load average.

`vtad`

`vtad` is a rule-based performance monitoring system for Linux servers. It monitors the system's performance and makes recommendations for tuning based on *rulesets*. Rulesets are suggestions for performance configuration. For example:

- The configured maximum shared memory should be 25% of the physical memory.
- The configured maximum number of files should be one-quarter the number of inodes.
- The number of available local ports should be 10,000 (or up to 28,000 on servers that have many proxy server connections).

See: http://sourceforge.net/projects/vtad/

`dklimits`

A simple utility to check the actually number of file descriptors available, ephemeral ports available, and `poll()`-able sockets. Be warned that it can take a while to run if there are a large number of file descriptors available, as it will try to open that many files, and then unlink them.

This is part of the `dkftpbench` package (http://www.kegel.com/dkftpbench/).

`PerformanceAnalysisScript`

`PerformanceAnalysisScript.sh` is a low-impact script that is provided in Appendix A *PerformanceAnalysisScript* and online at http://people.redhat.com/mbehm/. You can run this script by hand before, during, and after the performance event, or you can modify it to run periodically from `cron`. The script provides enough information to rule out obvious problems with swapping, process size, process count, Ethernet bandwidth availability, network errors, disk I/O bandwidth, disk errors, and interrupt counts. *Always* obtain at least two runs of any data-gathering scripts so that you can compare a "normal, loaded system", with the system when it is not performing as expected.

2.3. Kernel Monitoring Utilities

`strace`

`strace` collects information about kernel system calls and signals of the command following it on the command line; this can produce a lot of output and more detail than desired.

The lower impact "-c" option counts the number of times that each kind of system call is invoked, the total time spent for each system call, and the average time per systemcall.

The default is for `strace` to print out the information.

To generate a usable amount of data from the process with pid 27692 and send that to a file called `strace_output.txt`, enter:

```
strace -c -p 27692 -o strace_output.txt
```

`readprofile`

`readprofile` reads and displays the time-based sample data collected by the profiler built into the kernel. The profiling is enabled as a kernel boot option, "`profile=2`" (the "2" indicates that each bin in the kernel histogram is 2^2, 4, bytes in size). `readprofile` provides information only on the kernel; it will not provide information on the modules or the user-space code.

`ipcs`

`ipcs` lists the interprocess communication mechanisms including shared-memory regions, message queues, and semaphores.

2.4. Disk Benchmarking and Monitoring Utilities

`dbench`

`dbench` is a good disk-I/O benchmarking utility. It is designed to simulate the disk I/O load of a system when running the NetBench benchmark suite; it seems to do an excellent job.

Dbench is available at the Samba ftp site and mirrors (ftp://ftp.samba.org/pub/tridge/dbench/).

`IOzone`

A file-system benchmark tool that generates and measures a variety of file operations. `IOzone` is useful for performing a broad file-system analysis. The benchmark tests file I/O performance for the following operations: read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read, pread, mmap, aio_read, and aio_write.

See the `IOzone` site (http://www.iozone.org/).

`tiobench`

A multithread disk I/O benchmarking utility that does a good job at stressing the disks. `tiobench` comes with some useful scripts for generating reports and graphs.

See the `tiobench` site (http://sourceforge.net/projects/tiobench).

`dt` (data test)

`dt` is a very good disk/scsi test program; it performs disk I/O, process creation, async I/O, etc. It is not really a benchmark program; it is more of a data-integrity testing program. However, it can also be used to benchmark things as one of the intended roles of this program is to help people tune their drivers once they have verified that the driver is reliable.

`dt` is available at: http://www.bit-net.com/~rmiller/dt.html

`iostat`

`iostat -d` shows the disk performance in transactions per second, and well as statistics on block reads and writes.

The example that follows gives the results from the command that monitors the disk at 2-second intervals for three samplings:

```
$ iostat -d 2 3

Linux 2.4.20-18.9smp (perftest.redhat.com)  09/28/2004

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
dev3-0            1.07         2.01        15.95    5904202   46798960

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
dev3-0            0.00         0.00         0.00          0          0

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
dev3-0            2.00         0.00        64.00          0        128
```

The initial values are the total number of blocks read and written since the last boot. The subsequent values are the reads and writes since that report; these very low values indicate a machine that is largely idle.

To get extended statistics, use the `-x [device]` argument:

```
iostat -x
```

For further information, see Section 4.4.4 *Disk I/O Elevators*.

Note

When disk device drivers actually controlled the movement of the read/write heads on the disk drive, it was easy for utilities, such as `iostat`, to calculate metrics for a device. For example, the access times for a disk drive could be calculated based on the amount of time it took to service a request. Modern disk controllers are much more advanced and perform many optimizations, so it becomes much more difficult to obtain granular performance metrics.

vmstat

`vmstat`, which is part of the `procps` package, provides useful information when diagnosing disk performance problems.

Here is a sample of `vmstat` output:

```
   procs                  memory    swap        io     system  cpu
r  b  w   swpd  free   buff  cache  si  so   bi   bo   in    cs  us  sy id
1  0  0      0  9200  31856  34612   0   1  992  861  140   122   2   1  1
```

The `bi` (blocks in) and `bo` (blocks out) values are the areas to watch for disk I/O.

Other fields are described in Section 2.2 *System-Monitoring Utilities*.

2.5. Memory Monitoring Utilities

These system-load monitoring utilities have some overlap in functionality with the tools described elsewhere in this chapter. See, for example, the description of `vmstat` in Section 2.1 *Benchmarking Utilities* and `top` in Section 2.2 *System-Monitoring Utilities*.

free

`/usr/bin/free` lists the amount of memory and swap space in the system, as well as the buffers and cache used by the kernel.

```
             total       used       free     shared    buffers     cached
Mem:        512688     501828      10860          0      91552     152664
-/+ buffers/cache:     257612     255076
Swap:      1638620     250692    1387928
```

If more than half the swap space is being used, add more swap space.

`pmap`

`pmap` takes pid numbers as input and produces a listing of the memory map for the process. This memory map includes information about whether the regions is read, write, or executable. Each entry lists the starting address and the size of the region.

`pmap` was introduced with RHEL 3.

`sar` (System Activity Report)

`sar` enables you to get real-time information about system resource usage. It displays activity records that are automatically stored in the system-activity data file (`/var/log/sa/sa`*`DD`* where *`DD`* is the date).

To gather data about memory usage, use the following options:

`-n` *`option`*

Displays network statistics. *`option`* can be `DEV` (for basic statistics), `EDEV` (for errors), `SOCK` (for socket statistics), or `FULL` (for all statistics).

`-q`

Queue length and load averages.

For example:

```
sar -q 5 2
Linux 2.4.9-e.16 (example.org)    04/04/2005
03:19:34 PM   runq-sz  plist-sz   ldavg-1     ldavg-5
03:19:39 PM         2        97      0.13        0.40
03:19:44 PM         2        98      0.15        0.40
```

`-r`

Memory-usage statistics.

`-R`

Rate of change of memory usage.

`-s` and `-e`

Start and end times.

For example:

```
sar -s 12:00:00 -e 12:15:00
12:00:00 PM       CPU     %user     %nice     %system     %idle
12:10:00 PM       all      2.29      0.13        0.40     97.18
12:20:00 PM       all      1.25      0.17        0.07     98.51
```

`-W`

Swap-page statistics.

For example, to display the rate of change of memory usage at 5 second intervals for three reports:

```
sar -R 5 3
```

`mprof`

`mprof` (Memory Profiler and Leak Detector) (http://www.software-facilities.com/devel-software/mprof.php) is another utility you can use to obtain memory profiling information.

2.6. Network Monitoring Utilities

These network monitoring utilities have some overlap in functionality with the tools described in Section 2.1 *Benchmarking Utilities*.

`netstat`

If you are primarily concerned with network servers, the `netstat` command can often be very useful. It can show status of all incoming and outgoing sockets, which can give very handy information about the status of a network server.

One of the more useful options is:

```
netstat -pa
```

The "`-p`" option tells it to try to determine what program has the socket open, which is often very useful information. For example, say someone NMAPs their system and wants to know what is using port 666. Running `netstat -pa` will show you its status and what is running on that TCP port.

Another useful option is:

```
netstat -i
```

The "`-i`" option displays network statistics about sent and received packets. Large numbers of errors or dropped packets can indicate congestion or hardware problems.

A complex, but useful, invocation is:

```
netstat -a -n|grep -E "^(tcp)"| cut -c 68-|sort|uniq -c|sort -n
```

The output shows a sorted list of how many sockets are in each connection state. For example:

```
 9  LISTEN
21  ESTABLISHED
```

`ttcp`

`TTCP` is a basic network speed tester. It really only pumps data from one side of a TCP/IP network socket to another as fast as possible. The numbers this gives are really just a base guideline as to how well your TCP stack can fill the particular pipe it is talking through, but it is more accurate than trying to estimate with ftp or other protocols.

On modern machines, the bottleneck usually is not the CPU, so those measurements become less useful. However, this is a useful means of testing reliability: `TTCP` generates a lot of TCP packet allocations/deallocations and network card interrupts, so it is useful in determining if your Ethernet card driver is reliable or if it might break under heavy loads.

See: http://www.pcausa.com/Utilities/pcattcp.htm

`netperf`

`NetPerf` is a benchmark that you can use to measure the performance of many different types of networking. It provides tests for both unidirectional throughput, and end-to-end latency. The environments currently measurable by `NetPerf` include: TCP and UDP via BSD Sockets, DLPI, Unix Domain Sockets, Fore ATM API, and HiPPI.

For more information, see http://www.netperf.org/netperf/NetperfPage.html. To download, go to ftp://ftp.sgi.com/sgi/src/netperf/

`NetPIPE`

`NetPIPE` is a tool that visually represents the network performance under a variety of conditions. It combines the best qualities of `TTCP` and `NetPerf`.

`pchar`

`pchar` is a tool to characterize the bandwidth, latency, and loss of links along an end-to-end path through the Internet.

For more information, see http://www.kitchenlab.org/www/bmah/Software/pchar/.

2.7. Tools for Improving Application Performance

Note

If your application uses NPTL, also see Section 2.7.2 *Tools for NPTL*.

`hoard` (`libhoard`)

`Hoard` is a fast, scalable and memory-efficient allocator for multiprocessors. It is a drop-in replacement for the C and C++ memory routines.

`Hoard` solves the heap contention problem caused when multiple threads call dynamic memory-allocation functions such as `malloc()` and `free()` (or `new` and `delete`). `Hoard` can dramatically improve the performance of multithreaded programs running on multiprocessors.

For more information, see: http://www.cs.umass.edu/~emery/hoard/

`oprofile`

`OProfile` collects data on all the running executables and the kernel. It is useful to identify hot spots where the processor spends significant amounts of time. OProfile can also use the performance monitoring hardware on processors to identify which processor resources are limiting performance; for example, cache and branch prediction hardware.

See Chapter 7 *Application Tuning with OProfile*.

`/usr/bin/gprof`

User code can be compiled and linked with "-pg" to enable profiling. This instruments the code to record function calls so that `gprof` can build a call graph that uses time-based sampling to indicate where the program spends time. The `gprof` support in RHEL is not thread safe, does not dump data until the program exits, and does not record information about shared libraries.

`/usr/bin/gcov`

`gcov` reads the information generated by programs compiled with GCC's "-ftest-coverage" and "-fprofile-arcs" options. When the compiled program is run, data is collected on which basic blocks in the program are executed and the number of times they are executed. `gcov` maps that information back to the source code and indicates the number of times that specific lines in the source code are executed.

`/usr/bin/sprof`

The loader included in `glibc` allows profiling of a shared library. For example, to profile the shared `libc` library for the `du` command and write the data out to a file `xyz`, type:

```
LD_PROFILE=libc.so.6  du
```

To analyze the data:

```
sprof libc.so.6 libc.so.6.profile|more
```

`/usr/bin/ltrace`

`ltrace` intercepts calls to shared library routines and prints out a trace of the shared library routines executed. This instrumentation can significantly slow the execution of the program.

2.7.1. Pointer-Use Errors

Many pointer-use errors relate to heap allocation. Writing past the end of a heap object, or accessing a pointer after a `free`, can sometimes be detected with nothing more than a library that replaces the standard library's heap functions (`malloc`, `free`, etc.). This section examines programs that can detect such errors.

ElectricFence

ElectricFence C memory debugging library (http://linux.maruhn.com/sec/electricfence.html) is a utility you can use to obtain more detailed statistics regarding memory usage by individual programs. It can also manage heap objects that abut inaccessible virtual memory pages. A buffer overrun there causes an instant segmentation fault. Some libraries provide a protected padding area around buffers. This padding is filled with code that can be periodically checked for changes, so program errors can be detected at a coarser granularity.

You link the ElectricFence library in at compile time and it warns you of possible problems, such as freeing memory that does not exist.

ElectricFence has a number of limitations:

- It uses `mmap()` to allocate memory. `mmap()` can allocate memory only in page units, which can quickly exhaust the available address space on 32-bit platforms.
- It can perform access checks only in one direction—either above or below the allocated buffer. You may want to run the program twice: once with `EF_PROTECT_BELOW` off and once with it on.
- If the address is only slightly wrong and the block is aligned at the top of the page, some accesses beyond the top of the object might go unnoticed if the object size and alignment require a few fill bytes.
- If the address is wrong by more than one page size, the access might be beyond the guard pages with `PROT_NONE` access created by the library; consequently, it might succeed.

Bounded-pointers GCC Extension

The bounded-pointers GCC extension (http://gcc.gnu.org/projects/bp/main.html) addresses pointer errors by replacing simple pointers with a three-word struct that also contains the legal bounds for that pointer. This changes the system ABI, making it necessary to recompile the entire application. The bounds are computed upon assignment from the address-of operator, and constructed for system calls within an instrumented version of the standard library. Each use of the pointer is quickly checked against its own bounds, and the application aborts upon a violation. Because there is no database of live objects, an instrumented program can offer no extra information to help debug the problem.

`gcc-checker Extension`

The gcc-checker extension (http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html) addresses pointer errors by mapping all pointer-manipulation operations and all variable lifetime scopes to

calls into a runtime library. In this scheme, the instrumentation is heavy-weight, but the pointer representation in memory remains ABI-compatible. It may be possible to detect the moment a pointer becomes invalid (say, through a bad assignment or increment), before it is ever used to access memory.

`StackGuard GCC Extension`

The StackGuard GCC extension (http://immunix.org/stackguard.html) addresses stack-smashing attacks via buffer overruns. It does this by instrumenting a function to rearrange its stack frame, so that arrays are placed away from vulnerable values such as return addresses. Further guard padding is added around arrays and is checked before a function returns. This is light-weight and reasonably effective, but provides no general protection for pointer errors or debugging assistance.

`Purify`

The Purify package (http://www.rational.com/products/purify_unix/) is a well-known proprietary package for detecting memory errors. Purify works by batch instrumentation of object files, reverse-engineering patterns in object code that represent compiled pointer operations, and replacing them with a mixture of inline code and calls into a runtime library.

`Mudflap`

`Mudflap` is a pointer-use-checking technology based on compile-time instrumentation. It transparently adds protective code to a variety of potentially unsafe C/C++ constructs that detect actual erroneous uses at run time. It can detect NULL pointer dereferencing, running off the ends of buffers and strings, and leaking memory. Mudflap has heuristics that allow some degree of checking even if only a subset of a program's object modules are instrumented.

`Mudflap` is described in detail in Chapter 6 *Detecting and Preventing Application Memory Problems*.

`valgrind`

`Valgrind` executes your program in a virtual machine, keeping track of the memory blocks that have been allocated, initialized, and freed. It simulates every instruction a program executes, which reveals errors not only in an application, but also in all supporting dynamically-linked (`.so-format`) libraries, including the GNU C library, the X client libraries, Qt (if you work with KDE), and so on.

`Valgrind` is described in detail in Chapter 6 *Detecting and Preventing Application Memory Problems*.

2.7.2. Tools for NPTL

NPTL Test and Trace Project

The NPTL Test and Trace Project aims to improve the NPTL library. Although the group is just underway, you may want to monitor their progress at http://nptl.bullopensource.org/home.php.

NPTL Trace Tool

NPTL Trace Tool provides a mechanism to trace the NPTL Library as unobtrusively as possible for the application dynamics. The post-mortem analysis allows measurements (of contension and other issues) to give you an understanding of hangs or bugs.

See http://sourceforge.net/projects/nptltracetool/.

Tip

See also Section 3.1.5 *Kernel Threads*.

Chapter 3. Understanding the Virtual Memory Management System

One of the most important aspects of an operating system is the Virtual Memory Management system. Virtual Memory (VM) allows an operating system to perform many of its advanced functions, such as process isolation, file caching, and swapping. As such, it is imperative that you understand the functions and tunable parameters of an operating system's virtual memory manager so that optimal performance for a given workload can be achieved.

This chapter provides a general overview of how a VM works, specifically the VM implemented in Red Hat Enterprise Linux 3. The VM may change between releases; however, with a well-grounded understanding of the general mechanics of a VM, it is fairly easy to convert your knowledge of VM tuning to another VM as the same general principles will apply. The documentation for a given kernel (including its specific tunable parameters), can be found in the corresponding kernel source tree in the file `/usr/src/linux_kernel/Documentation/sysctl/vm.txt`.

3.1. Virtual Memory Terminology

To properly understand how a Virtual Memory manager does its job, it helps to understand what components comprise a VM. While the low level details of a VM are overwhelming for most, a high level view is nonetheless helpful in understanding how a VM system works, and how it can be optimized for various workloads. A high level overview of the components that make up a Virtual Memory manager is presented in Figure 3-1

3.1.1. What Comprises a VM

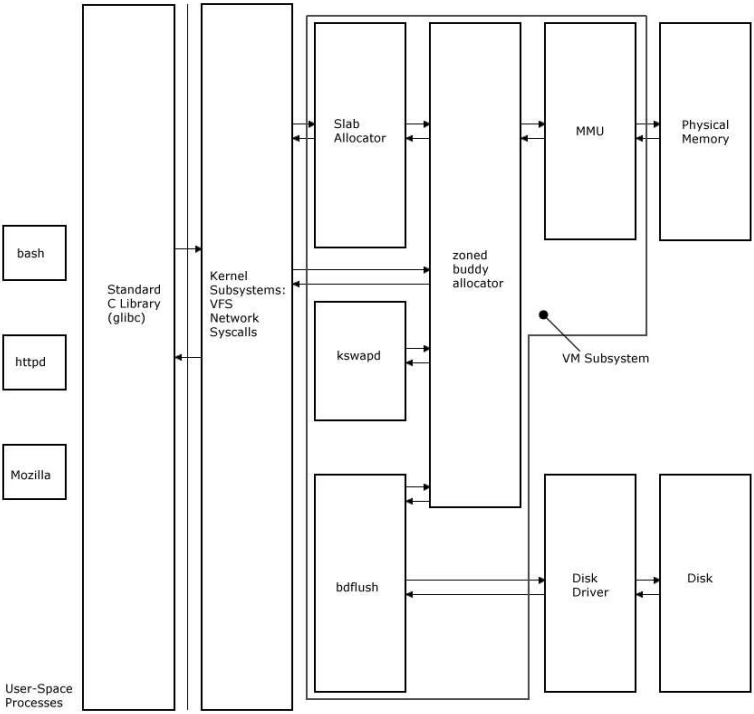

Figure 3-1. High Level Overview Of VM Subsystem

While a VM is actually far more complicated than illustrated in Figure 3-1, the high-level function of the system is accurate. The following sections describe each of the listed components in the VM.

3.1.2. Memory Management Unit (MMU)

The Memory Management Unit (MMU) is the hardware base that make a Virtual Memory system possible. The MMU allows software to reference physical memory by aliased addresses, quite often more than one. It accomplishes this through the use of *pages* and *page tables*. The MMU uses a section of memory to translate virtual addresses into physical addresses via a series of table lookups. Various processor architectures perform this function in slightly different ways, but in general, Figure 3-2 illustrates how a translation is performed from a virtual address to a physical address.

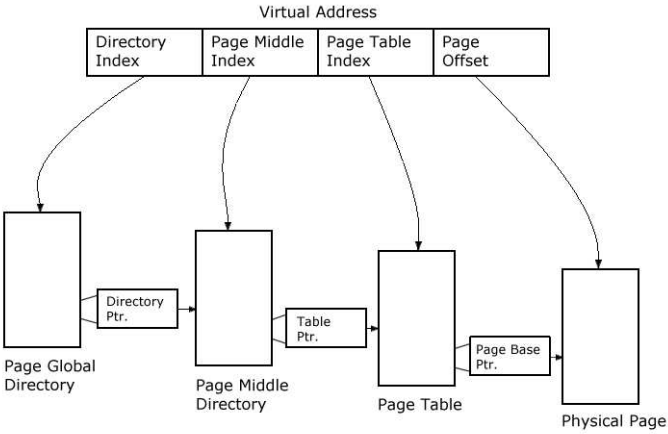

Figure 3-2. Illustration of a Virtual-to-Physical Memory Translation

Each table lookup provides a pointer to the base of the next table, as well as a set of extra bits that provide auxiliary data regarding that page or set of pages. This information typically includes the current page status, access privileges, and size. A separate portion of the virtual address being accessed provides an index into each table in the lookup process. The final table provides a pointer to the start of the physical page corresponding to the virtual address in RAM, while the last field in the virtual address selects the actual word in the page being accessed. Any one of the table lookups during this translation, may direct the lookup operation to terminate and drive the operating system to perform another action. Some of these actions are somewhat observable at a system level, and have common names or references.

Segmentation Violation

A user space process requests a virtual address, and during the translation the kernel is interrupted and informed that the requested translation has resulted in a page that it has not allocated, or that the process does not have permission to access. The kernel responds by signaling the process that it has attempted to access an invalid memory region, after which it is terminated.

Swapped out

During a translation of an address from a user space process, the kernel was interrupted and informed that the page table entry lists the page as accessible, but not present in RAM. The kernel interprets this to mean that the requested address is in a page that has been swapped to disk. The user process requesting the address is put to sleep and an I/O operation is started to retrieve the page.

3.1.3. Zoned Buddy Allocator

The Zoned Buddy Allocator is responsible for the management of page allocations to the entire system. This code manages lists of *physically contiguous* pages and maps them into the MMU page tables to provide other kernel subsystems with valid *physical* address ranges when the kernel requests them (Physical to Virtual Address mapping is handled by a higher layer of the VM and is collapsed into the kernel subsystems block of Figure 3-1.

The name Buddy Allocator is derived from the algorithm this subsystem uses to maintain it free page lists. All physical pages in RAM are cataloged by the buddy allocator and grouped into lists. Each list represents clusters of 2^n pages, where *n* is incremented in each list. There is a list of single pages, a list of 2 page clusters, a list of 4 page cluster, and so on. When a request comes in for an amount of memory, that value is rounded up to the nearest power of 2. An entry is then removed from the appropriate list, registered in the page tables of the MMU, and a corresponding physical address is returned to the caller, which is then mapped into a virtual address for kernel use. If no entries exist on the requested list, an entry from the next list up is broken into two separate clusters, and 1 is returned to the caller while the other is added to the next list down. When an allocation is returned to the buddy allocator, the reverse process happens. The allocation is returned to the requisite list, and the list is then examined to determine if a larger cluster can be made from the existing entries on the list that was just updated. This algorithm is advantageous in that it *automatically returns pages to the highest order free list possible*. That is to say, as allocations are returned to the free pool, they automatically form larger clusters, so that when a need arises for a large amount of physically contiguous memory (for example, for a DMA operation), it is more likely that the request can be satisfied. Note that the buddy allocator allocates memory in page multiples only. Other subsystems are responsible for finer grained control over allocation size. For more information regarding the finer details of a buddy allocator, refer to Bovet and Cesati's *Understanding the Linux Kernel*.

Note that the Buddy Allocator also manages memory *zones*, which define pools of memory that have different purposes. Currently there are three memory pools for which the buddy allocator manages access:

DMA

This zone consists of the first 16 MB of RAM, from which legacy devices allocate to perform direct memory operations.

NORMAL

This zone encompasses memory addresses from 16 MB to 1 GB[1] and is used by the kernel for internal data structures, as well as other system and user space allocations.

HIGHMEM

This zone includes all memory above 1 GB and is used exclusively for system allocations (file system buffers, user space allocations, etc).

3.1.4. Slab Allocator

The Slab Allocator provides a more usable front end to the Buddy Allocator for those sections of the kernel that require memory in sizes that are more flexible than the standard 4 KB page. The Slab Allocator allows other kernel components to create *caches* of memory objects of a given size. The Slab Allocator is responsible for placing as many of the caches' objects on a page as possible and monitoring which objects are free and which are allocated. When allocations are requested and no more are available, the Slab Allocator requests more pages from the Buddy Allocator to satisfy the request. This allows kernel components to use memory in a much simpler way. This way components that make use of many small portions of memory are not required to individually implement memory management code so that too many pages are not wasted.[2]

1. The hugemem kernel extends this zone to 3.9 GB of space.
2. The Slab Allocator may allocate only from the DMA and NORMAL zones.

3.1.5. Kernel Threads

The last component in the VM subsystem are the active tasks: `kscand`, `kswapd`, `kupdated`, and `bdflush`. These tasks are responsible for the recovery and management of in-use memory. All pages of memory have an associated state (for more info on the memory state machine, see Section 3.2 *The Life of A Page*).

Freshly allocated memory initially enters the *active* state. As the page sits in RAM, it is periodically visited by the `kscand` task, which marks the page as being *inactive*. If the page is subsequently accessed again by a task other than `kscand`, it is returned to the active state. If a page is modified during its allocation it is additionally marked as being dirty. The `kswapd` task scan pages periodically in much the same way, except that when it finds a page that is inactive, `kswapd` considers the page to be a candidate to be returned to the free pool of pages for later allocation. If `kswapd` selects a page for freeing, it examines its dirty bit. In the event the dirty bit is set, it locks the page, and initiates a disk I/O operation to move the page to swap space on the hard drive. When the I/O operation is complete, `kswapd` modifies the page table entry to indicate that the page has been migrated to disk, unlocks the page, and places it back on the free list, making it available for further allocations. In general the active tasks in the kernel relating to VM usage are responsible for attempting to move pages out of RAM. Periodically they examine RAM, trying to identify and free inactive memory so that it can be put to other uses in the system.

3.1.5.1. NPTL Threading Library

The NPTL Threading Library was introduced in Red Hat Enterprise Linux 3. If you have seen a reduction in performance after an upgrade to RHEL 3, it may be that your application does not use NPTL effectively.

3.1.5.1.1. Explaining LD_ASSUME_KERNEL

`LD_ASSUME_KERNEL` sets the kernel ABI that the application expects.

`LD_ASSUME_KERNEL` is handled by the dynamic linker, but the behavior of a specific value of `LD_ASSUME_KERNEL` is *not* hardcoded in the dynamic linker.

Every DSO (Dynamic Shared Object, also known as a shared library) can tell the dynamic linker in glibc which minimum operating system ABI version is needed. Note that dynamic linkers other than glibc's do not have this feature. The information about the minimum operating system ABI version is encoded in an ELF note section usually named `.note.ABI-tag`. This note section must be referenced by a `PT_NOTE` entry in the DSO's program header.

To examine the content of a DSO's note section, use the `readelf` program from `elfutils` (the version from `binutils` is inadequate). On a Red Hat system, the binary is called `eu-readelf`. Using it on a Fedora Core 2 system shows the following:

```
$ eu-readelf -n /lib/tls/libc-2.3.3.so

Note segment of 32 bytes at offset 0x174:
  Owner Data size  Type
  GNU 16  VERSION
OS: Linux, ABI: 2.4.20
```

This means the `/lib/tls/libc-2.3.3.so` DSO requires at least OS ABI version 2.4.20.

The specific ABI version requirements on a RHL9, and Fedora Core 1 and 2 system are as follows (note that this implies IA-32 is the architecture):

- DSOs in `/lib/tls` need ABI version 2.4.20.
- DSOs in `/lib/i686` need ABI version 2.4.1.
- DSOs in `/lib` need ABI version 2.2.5.

This means:

- `LD_ASSUME_KERNEL` supports only versions 2.2.5 and later.
- Versions from 2.2.5 to 2.4.0 will use the DSOs in `/lib`
- Versions from 2.4.1 to 2.4.19 will use the DSOs in `/lib/i686`
- Versions 2.4.20 and newer will use the DSOs in `/lib/tls`.

For the Red Hat releases, this layout was chosen to provide the maximum amount of backward compatibility for applications that violate the threading specifications or use implementation artifacts (correctly written applications will have no problems).

You must ensure that programmers use explicit thread stack sizes when creating new threads. For example, to cause all thread stacks to be 64k by default, use:

```
ulimit -s 64
```

This also sets the main thread's stack (the thread created by the kernel) to also have this size limitation. This is usually acceptable.

3.1.6. Components that Use the VM

It is worth mentioning the remaining components that sit on top of the VM subsystem. These components actively use the VM to acquire memory and presents it to users, providing the overall "feel" of the system:

Network Stack

> The Network Stack is responsible for the management of network buffers being received and sent out of the various network interfaces in a system.

Standard C Library

> Via various system calls, the standard C library manages pages of virtual memory and presents user applications with an API allowing fine-grained memory control.

Virtual File System

> The Virtual file system buffers data from disks for more rapid file access, and holds pages containing file data that has been memory-mapped by an application.

3.2. The Life of A Page

All of the memory managed by the VM is labeled by a *state*. These states help let the VM know what to do with a given page under various circumstances. Dependent on the current needs of the system, the VM may transfer pages from one state to the next, according to the state machine diagrammed in Figure 3-3:

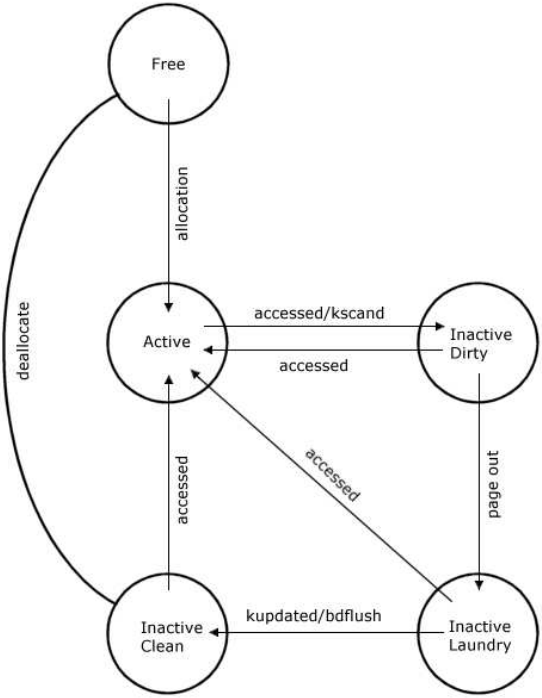

Figure 3-3. Basic Configuration

Using these states, the VM can determine what is being done with a page by the system at a given time and what actions the VM may take on the page. The states that have particular meanings are as follows:

FREE

All pages available for allocation begin in this state. This indicates to the VM that the page is not being used for any purpose and is available for allocation.

ACTIVE

Pages that have been allocated from the Buddy Allocator enter this state. It indicates to the VM that the page has been allocated and is actively in use by the kernel or a user process.

INACTIVE DIRTY

The state indicates that the page has fallen into disuse by the entity that allocated it, and as such is a candidate for removal from main memory. The kscand task periodically sweeps through all the pages in memory, taking note of the amount of time the page has been in memory since it was last accessed. If kscand finds that a page has been accessed since it last visited the page, it increments the pages age counter; otherwise, it decrements that counter. If `kscand` happens on a page that has its age counter at zero, then the page is moved to the inactive dirty state. Pages in the inactive dirty state are kept in a list of pages to be laundered.

INACTIVE LAUNDERED

This is an interim state in which those pages that have been selected for removal from main memory enter while their contents are being moved to disk. Only pages that were in the inactive dirty state enter this state. When the disk I/O operation is complete the page is moved to the inactive clean state, where it may be deallocated, or overwritten for another purpose. If, during the disk operation, the page is accessed, the page is moved back into the active state.

INACTIVE CLEAN

Pages in this state have been laundered. This means that the contents of the page are in sync with the backing data on disk. As such, they may be deallocated by the VM or overwritten for other purposes.

3.3. Tuning the VM

Now that the picture of the VM mechanism is sufficiently illustrated, how is it adjusted to fit certain workloads? There are two methods for changing tunable parameters in the Linux VM. The first is the `sysctl` interface. The sysctl interface is a programming oriented interface that enables software programs to directly modify various tunable parameters. The `sysctl` interface is exported to system administrators via the `sysctl` utility, which enables an administrator to specify a value for any of the tunable VM parameters on the command line, as in the following example:

```
sysctl -w vm.max_map_count=65535
```

The `sysctl` utility also supports the use of the `/etc/sysctl.conf` configuration file, in which all the desirable changes to a VM can be recorded for a system and restored after a restart of the operating system, making this access method suitable for long term changes to a system VM. The file is straightforward in its layout, using simple key-value pairs with comments for clarity, as in the following example:

```
#Adjust the min and max read-ahead for files
vm.max-readahead=64
vm.min-readahead=32

#turn on memory over-commit
vm.overcommit_memory=2

#bump up the percentage of memory in use to activate bdflush
vm.bdflush="40 500 0 0 500 3000 60 20 0"
```

The second method of modifying VM tunable parameters is via the `proc` file system. This method exports every group of VM tunables as a file, accessible via all the common Linux utilities used for modify file contents. The VM tunables are available in the directory `/proc/sys/vm`, and are most commonly read and modified using the Linux `cat` and `echo` utilities, as in the following example:

```
# cat kswapd
512     32      8

echo 511 31 7 > /proc/sys/vm/kswapd

# cat kswapd
511     31      7
```

The `proc` file system interface is a convenient method for making adjustments to the VM while attempting to isolate the peak performance of a system. For convenience, the following sections list the VM tunable parameters as the filenames they are exported as in `/proc/sys/vm`. Please note that unless otherwise stated, these tunables apply to the RHEL3 2.4.21-4 kernel.

3.3.1. bdflush

The `bdflush` file contains nine parameters, of which six are tunable. These parameters affect the rate at which pages in the buffer cache[3] are freed and returned to disk. By adjusting the various values in this file, a system can be tuned to achieve better performance in environments where large amounts of file I/O are performed. The following parameters are defined in the order they appear in the file.

Parameter	Description
nfract	The percentage of dirty pages in the buffer cache required to activate the bdflush task
ndirty	The maximum number of dirty pages in the buffer cache to write to disk in each bdflush execution
reserved1	Reserved for future use
reserved2	Reserved for future use
interval	The number of jiffies to delay between bdflush iterations
age_buffer	The time for a normal buffer to age before it is considered for flushing back to disk
nfract_sync	The percentage of dirty pages in the buffer cache required to cause the tasks that are dirtying pages of memory to start writing those pages to disk themselves, slowing the dirtying process
nfract_stop_bdflush	The percentage of dirty pages in buffer cache required to allow bdflush to return to idle state
reserved3	Reserved for future use

Generally, systems that require more free memory for application allocation will want to set these values higher (except for the age buffer, which would be moved lower), so that file data is sent to disk more frequently, and in greater volume, thus freeing up pages of RAM for application use. This of course comes at the expense of CPU cycles, as the system processor spends more time moving data to disk, and less time running applications. Conversely, systems that are required to perform large amounts of I/O would want to do the opposite to these values, allowing more RAM to be used to cache disk file, so that file access is faster.

3.3.2. dcache_priority

This file controls the bias of the priority for caching directory contents. When the system is under stress, it will selectively reduce the size of various file system caches in an effort to reclaim memory. By adjusting this value up, memory reclamation bias is shifted away from the dirent [4] cache. By reducing this amount, bias is shifted toward reclaiming dirent memory. This is not a particularly useful tuning parameter, but it can be helpful in maintaining the interactive response time on an otherwise heavily loaded system. If you experience intolerable delays in communicating with your system when it is busy performing other work, increasing this parameter may help you.

3. The subset of pagecache that stores files in memory
4. A dirent is a DIRectory ENTry. Directory entries contain information about the files in those directories and are cached in the kernel to speed disk access. They can either account for performance gains (if the same directory is accessed many times over) or a performance detriment (if many different directories are accessed, resulting in large memory allocations).

3.3.3. hugetlb_pool

This file is responsible for recording the number of megabytes used for 'huge' pages. Huge pages are just like regular pages in the VM, only they are an order of magnitude larger.[5] Hugepages are both beneficial and detrimental to a system. They are helpful in that each hugepage takes only one set of entries in the VM page tables, which allows for a higher degree of virtual address caching in the TLB[6] and a requisite performance improvement. On the downside, they are very large and can be wasteful of memory resources for those applications that do not need large amounts of memory. Some applications, however, do require large amounts of memory and if they are written to be aware of them, can make good use of hugepages. If a system runs applications that require large amounts of memory and is aware of this feature, then it is advantageous to increase this value to an amount satisfactory to that application or set of applications.

3.3.4. inactive_clean_percent

This control specifies the minimum percentage of pages in each page zone that must be in the clean or laundered state. If any zone drops below this threshold, and the system is under pressure for more memory, then that zone will begin having its inactive dirty pages laundered. Note that this control is only available on the 2.4.21-5EL kernels forward. Raising the value for the corresponding zone that is memory starved will cause pages to be paged out more quickly, eliminating memory starvation, at the expense of CPU clock cycles. Lowering this number will allow more data to remain in RAM, increasing the system performance, but at the risk of memory starvation.

3.3.5. kswapd

While this set of parameters previously defined how frequently and in what volume a system moved non-buffer cache pages to disk, in Red Hat Enterprise Linux 3, these controls are unused.

3.3.6. max_map_count

This file allows for the restriction of the number of VMAs (Virtual Memory Areas) that a particular process can own. A Virtual Memory Area is a contiguous area of virtual address space. These areas are created during the life of the process when the program attempts to memory map a file, link to a shared memory segment, or simply allocates heap space. Tuning this value limits the amount of these VMA's that a process can own. Limiting the amount of VMA's a process can own can lead to problematic application behavior, as the system will return out of memory errors when a process reaches its VMA limit, but can free up lowmem for other kernel uses. If your system is running low on memory in the ZONE_NORMAL zone, then lowering this value will help free up memory for kernel use.

3.3.7. max-readahead

This tunable affects how early the Linux VFS (Virtual File System) will fetch the next block of a file from memory. File readahead values are determined on a per file basis in the VFS and are adjusted based on the behavior of the application accessing the file. Anytime the current position being read in a file plus the current readahead value results in the file pointer pointing to the next block in the file, that block will be fetched from disk. By raising this value, the Linux kernel will allow the readahead value to grow larger, resulting in more blocks being prefetched from disks that predictably access files in uniform linear fashion. This can result in performance improvements, but can also result in

5. Note also that hugepages are not swappable
6. Translation Look-aside Buffer: A device that caches virtual address translations for faster lookups

excess (and often unnecessary) memory usage. Lowering this value has the opposite affect. By forcing readaheads to be less aggressive, memory may be conserved at a potential performance impact.

3.3.8. min-readahead

Like max-readahead, min-readahead places a floor on the readahead value. Raising this number forces a files readahead value to be unconditionally higher, which can bring about performance improvements, provided that all file access in the system is predictably linear from the start to the end of a file. This of course results in higher memory usage from the pagecache. Conversely, lowering this value, allows the kernel to conserve pagecache memory, at a potential performance cost.

3.3.9. overcommit_memory

The `overcommit_memory` sets the general kernel policy toward granting memory allocations. If the value in this file is 0, then the kernel will check to see if there is enough memory free to grant a memory request to a malloc call from an application. If there is enough memory, the request is granted. Otherwise it is denied and an error code is returned to the application. If the setting in this file is 1, the kernel will allow all memory allocations, regardless of the current memory allocation state. If the value is set to 2, then the kernel will grant allocations above the amount of physical RAM and swap in the system, as defined by the `overcommit_ratio` value (defined below). Enabling this feature can be somewhat helpful in environments that allocate large amounts of memory expecting worst-case scenarios, but do not use it all.

3.3.10. overcommit_ratio

This tunable defines the amount by which the kernel will overextend its memory resources, in the event that overcommit_memory is set to the value 2. The value in this file represents a percentage that will be added to the amount of actual RAM in a system when considering whether to grant a particular memory request. For instance, if this value was set to 50, then the kernel would treat a system with 1GB of RAM and 1GB of swap as a system with 2.5GB of allocatable memory when considering weather to grant a malloc request from an application. The general formula for this tunable is:

memory_{allocatable} = (sizeof_{swap} + (sizeof_{ram} * overcommit_{ratio}))

Use these previous two parameters with caution. Enabling memory overcommit can create significant performance gains at little cost, but only if your applications are suited to its use. If your applications use all of the memory they allocate, memory overcommit can lead to short performance gains followed by long latencies as your applications are swapped out to disk frequently when they must compete for oversubscribed RAM. Also ensure that you have at least enough swap space to cover the overallocation of RAM (meaning that your swap space should be *at least* big enough to handle the percentage of overcommit, in addition to the regular 50 percent of RAM that is normally recommended).

3.3.11. pagecache

The `pagecache` file adjusts the amount of RAM that can be used by the pagecache. The pagecache holds various pieces of data, such as open files from disk, memory mapped files and pages of executable programs. Modifying the values in this file dictates how much of memory is used for this purpose.

Parameter	Description
min	The minimum amount of memory to reserve for pagecache use

Parameter	Description
borrow	kswapd balances the reclaiming of pagecache pages and process memory to reach this percentage of pagecache pages
max	If more memory than this percentage is taken by the pagecache, kswapd will evict pages only from the pagecache. Once the amount of memory in pagecache is below this threshold, then kswapd will again allow itself to move process pages to swap.

Adjusting these values upward allows more programs and cached files to stay in memory longer, thereby allowing applications to execute more quickly. On memory starved systems however, this may lead to application delays, as processes must wait for memory to become available. Moving these values downward swaps processes and other disk-backed data out more quickly, allowing for other processes to obtain memory more easily, increasing execution speed. For most workloads the automatic tuning works fine. However, if your workload suffers from excessive swapping and a large cache, you may want to reduce the values until the swapping problem goes away.

3.3.12. page-cluster

The kernel will attempt to read multiple pages from disk on a page fault, in order to avoid excessive seeks on the hard drive. This parameter defines the number of pages the kernel will try to read from memory during each page fault. The value is interpreted as $2^{\text{page-cluster}}$ pages for each page fault. A page fault is encountered every time a virtual memory address is accessed for which there is not yet a corresponding physical page assigned, or for which the corresponding physical page has been swapped to disk. If the memory address has been requested in a valid way (that is, the application contains the address in its virtual memory map). then the kernel will associate a page of RAM with the address, or retrieve the page from disk and place it back in RAM, and restart the application from where it left off. By increasing the page-cluster value, pages subsequent to the requested page will also be retrieved, meaning that if the workload of a particular system accesses data in RAM in a linear fashion, increasing this parameter can provide significant performance gains (much like the file read-ahead parameters described earlier). Of course if your workload accesses data discreetly in many separate areas of memory, then this can just as easily cause performance degradation.

To view the page fault activity for a process, use:

```
ps –o minflt,majflt pid
```

Where `minflt` (minor page fault) indicates that the virtual memory system allocated a new page frame for the process and `majflt` (major page fault) indicates that the kernel blocked processes while it read the contents of the page from disk. (Major page faults are more 'expensive' than minor page faults.)

Chapter 4.
Tuning Linux Servers

Warnings

- This chapter suggests parameter tuning changes you can make to improve the performance of particular subsystems. Some of these parameter changes *might* improve that *particular* kind of workload, but it might also *degrade* performance in other areas. The only way you can find out if a change improves or degrades the performance of the subsystem and of the total system is by taking careful measurements before and after the change. If the effect of the change cannot be measured, it should not be made permanent.

  As these changes are non-standard and can affect other kinds of workloads, they should be noted and included in any Support-related interactions with Red Hat.

- If the changes you make to entries under `/proc` or `/sys` render your system unbootable or unusable, you should be able to remedy the problem by booting into either single-user mode or emergency mode. See the *Red Hat Enterprise Linux Step By Step Guide* for details about booting into single-user mode.

If your system is encountering performance, memory, or scalability issues, there are several suggested techniques to analyze and investigate the problem.

Isolate the problem to the database or application services. The `top` command is useful for this purpose. If there is high CPU usage on the application services relative to the database services, or vice versa, then you have an area to focus on.

- If the problem is with the database server, turn on developer support and identify redundant or slow queries. Once these queries are identified, caching, denormalization, query optimization, and/or creating indexes should improve performance.

  Database servers typically benefit from any tuning you can do (or upgrades you can perform) on the disk I/O. See Section 4.2 *Disk Tuning*.

- If the problem is with the application server, system profiling is recommended. To isolate application bottlenecks or memory usage, use a profiler such as OptimizeIt (http://www.borland.com/optimizeit/) or JProbe (http://java.quest.com/jprobe/).

If there appears to be memory pressure, turning on verbose logging for garbage collection can be helpful. Each JVM has a different way to turn on verbose garbage collection. With the IBM JVM, start your server with:

```
java –verbose:gc
```

This logs all garbage collection activity to your server log file.

Important

Once the issue in question has been resolved, be sure to turn off options such as developer support and verbose logging. Excess logging can be the cause of very noticeable performance problems.

4.1. Reducing System Load

You can determine which processes are loading the system by using `ps` and `top` (which are described in Section 2.2 *System-Monitoring Utilities*). You may be able to use this information to reduce the system load:

- If there are programs that can be run during off-peak times, use `cron` to reschedule them.
- If the programs can be run on a different server, move them.
- If the programs are not critical, use `renice` to raise their *nice* value and thus reduce their priority.

  Nice values range from 19 (the lowest priority) to -20 (the highest priority); the default is 0. To reduce the priority to 10 of a running program with a pid of 8435, use:
  `renice 10 8435`

  If this proves to be beneficial, in the future start the program with `nice`:
  `nice -n 10 program_name`

4.1.1. Avoiding Swapping

The graphic below shows the difference in speed between RAM and disk access; note that the scale is logarithmic.

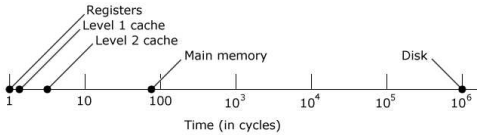

Figure 4-1. Data access time in cycles

If you have run the tools in Section 2.5 *Memory Monitoring Utilities* and see that your system is using swap space heavily, you should see if you can add RAM. Section 5.1 *Can You Upgrade Your Hardware?* describes the limits of each version of Red Hat Enterprise Linux.

If you must use swap space, there are ways to improve swap performance:

- Place swap areas on the lowest-numbered partitions on fast disks spread across two or more controllers. (It may be wasteful to dedicate a whole disk to swap, so you can use the remainder of the disk for infrequently used storage.)
- Assign higher priority to the fastest disks. If you have multiple disks, set the swap partitions on each disk to the same priority by using the `pri` option for the `mount` command. This stripes swap space across the disks like a RAID array, thus improving performance.
- If you have an immediate need for extra swap space but have no available partitions, you can set up a temporary swap file on a lightly-used disk. However, use the `pri` command to prioritize the true swap partitions over the slower swap file:

```
/dev/sda1      swap  swap  pri= 3  0  0
/dev/sdb1      swap  swap  pri= 3  0  0
/dev/sdc1      swap  swap  pri= 3  0  0
/dev/swapfile  swap  swap  pri= 1  0  0
```


Tip

If a swap space must be on a busy disk, locate it as close to the busy partition as possible. This reduces the seek time for the drive head and reduces swap latency.

4.1.1.1. Tuning Parameters that Affect Swapping

A *page fault* occurs whenever the kernel needs to access a page of data that is not currently available in the current process' resident memory. *Page fault latency* is the time required to move data from the hard drive into RAM. There are several parameters you can tune to optimize paging performance:

- `/proc/sys/vm/page-cluster` sets the number of pages the kernel reads per page fault. Increasing this value reduces the cost of head-seek time.
- `/proc/sys/vm/pagetable-cache` sets the minimum and maximum number of pages of memory that are used per processor for pagetable caches. On SMP systems, this enables each processor to do pagetable caches without having a kernel memory lock.
- For Linux versions *prior to*= Red Hat Enterprise Linux 3, `/proc/sys/vm/kswapd` contains the values that the `kswapd` kernel thread uses for swapping:
  - `tries_base` is the maximum number of pages to be freed in a cycle. The value should be divisible by 4 or 8. Increasing this number effectively increases the swap bandwidth. This should be large enough to make the writes efficient without overloading the disk queue. Experiment with this value while the system is paging to find the optimum value.
  - `tries_min` is the minimum number of times the kernel swap thread tries to free some pages. This attempts to ensure that some pages are freed even when the thread is called with a minimum priority.
  - `swap_cluster` is the number of pages the kernel-swap thread writes in a cycle. This should be large enough to minimize head seeks without flooding the request queue.

4.1.2. ptys and ttys

In older versions of Red Hat Linux, the number of ptys and ttys could sometimes be a limiting factor for login servers and database servers. This is less likely to be a problem in Red Hat Enterprise Linux. However, if you want to test whether your system is being slowed by pty/tty limits, be aware that modifying these values requires a kernel recompile, which invalidates your support agreement.

4.2. Disk Tuning


Tip

Before you begin tuning, wherever possible spread the load for I/O as evenly as possible across the available disk drives; if one drive is busier than the other drives in a system, that drive will most likely be the bottleneck for applications.

Placing different files on different disks can also help performance. For example:

- Place the mail spool directory on a separate device, if a system's primary role is that of a mailserver.
- For a system with a large number of interactive users, place the home directories for different users on different disk drives.
- Use separate disks for the operating system, the data, and the log files.

System performance often depends heavily on disk input/output performance, so it is critical that you ensure the disk I/O subsystem performs as efficiently as possible. In particular, it is important to satisfy read requests quickly as these block other processes and directly affect the user's perception of the system's performance. Some tools that you can use to monitor disk performance are described in Section 2.4 *Disk Benchmarking and Monitoring Utilities*.

Systems that have an I/O bottleneck at the disk typically have high disk and disk-controller utilization, low network utilization, and possibly low CPU utilization.

Of course, faster drives improve performance, so you should choose SCSI over IDE/EIDE disks. Within a given drive type, there is a roughly linear improvement in performance based on disk speed. Comparing a 15,000 rpm drive to a 7,200 rpm one shows that the faster drive has less than half the latency and seek time, and provides almost twice the total throughput.

Note

Red Hat Enterprise Linux does not currently support iSCSI.

Fibre Channel, a fast, expensive alternative to SCSI, is not discussed in this document.

Whatever type of disks you have, you can tune them for better performance. For example, most modern disk controllers have built-in cache that can be used to speed up both read and write accesses. Because write caching can lead to data loss or corruption in the event of a power failure, some manufacturers may ship their disk drives with write caching disabled. Turning on write caching can significantly boost disk performance, albeit at some risk.

4.2.1. SCSI Tuning

SCSI tuning is highly dependent on the particular SCSI cards and drives in question. The most effective variable when it comes to SCSI card performance is tagged-command queueing.

Tagged-command queuing (TCQ) can improve disk performance by allowing the the disk controller to re-order I/O requests in such a way that head-seek movement is minimized. The requests are tagged with an identifier by the disk controller so that the blocks requested for a particular I/O operation can be returned in the proper sequence for the order in which they were received. This parameter is typically passed as an argument to the kernel at system boot time. For certain applications, you may wish to limit the queue depth (number of commands that can be queued up).

For the Adaptec aic7xxx series cards (2940's, 7890's, *160's, etc.), this can be enabled with a module option such as:

```
aic7xxx=tag_info:{{0,0,0,0,0,0,0,0}}
```

This enables the default tagged-command queuing on the first device, on the first four SCSI IDs.

To set the TCQ depth to 64:

- To set the depth temporarily, use:
  `insmod aic7xxxaic7xxx=tag_info:{{0,0,64,0,0,0,0,0}}`

- To make this the default on RHEL 3, add the following line to `/etc/modules.conf`:
  `options aic7xxxaic7xxx=tag_info:{{0,0,64,0,0,0,0}}`
- To make this the default on RHEL 4, add the following line to `/etc/modprobe.conf`:
  `options aic7xxxaic7xxx=tag_info:{{0,0,64,0,0,0,0}}`

Alternatively, add the following line to the "kernel" line in `/boot/grub/grub.conf`:

```
aic7xxxaic7xxx=tag_info:{{0,0,64,0,0,0,0}}
```

Check the driver documentation for your particular SCSI modules for more information.

Tip

To understand the output provided by the `modinfo` command, install the documentation for your device drivers:

- For RHEL 3, and earlier, this information is provided, along with the kernel source code, in the `kernel-source` package.

  Another source of information is the files in `/usr/src/linux-2.4/Documentation`.
- For RHEL 4, this information is provided in the `kernel-doc` package.

4.2.2. Read-ahead

Read-ahead is the practice of reading multiple blocks of data, anticipating that having read block A, the next most likely request will be blocks B, C, and so on. Caching as yet unrequested pages in memory means that the kernel can satisfy requests for that data more quickly while reducing the load on the disk controller.

The default number of pages to read-ahead is controlled by `/proc/sys/vm/min-readahead`. When the read-ahead algorithm succeeds (that is, the cached data is actually searched), the `min-readahead` value is doubled until it reaches the value stored in `/proc/sys/vm/max-readahead`.

Of course, these performance increases do not occur when requests are for random data or for the same data continuously. In these situations, the read-ahead algorithm turns itself off; therefore, you can increase performance if you are able to change random access patterns into sequential access patterns.

Note

The 2.6 kernels used in RHEL 4 and Fedora use the "`blockdev`" command to set the number of pages to read-ahead. The `/proc` entries are no longer there.

4.2.3. File System Fragmentation

Disk fragmentation can be a serious problem for some filesystems, but but normally is not an issue for the ext2/ext3 filesystems.

When extending a file, the ext2/ext3 filesystem will try to get a new block for the file near the last block that was allocated for the file. If a free block cannot be found near the last-allocated block for the file, an attempt will be made to get a free block from the same block group. Because the blocks

in the block group are located close to each other, any head movement resulting from fragmentation should be small.

Fragmentation becomes much more likely as a disk partition becomes close to full and the filesystem device drivers have to resort to placing data blocks for files in different block groups. Keep a reserved block count of a few percent to prevent the possibility of serious fragmentation.

4.2.4. Tuning the ext2 File System

The ext2 file system defaults to a block size of 1024. If you are serving mainly large files, you can change this size to 2048 or 4096 to reduce the number of seeks for data. However, if you have smaller files in the future, the new setting will result in wasted space.

Note that changing the block size requires that you rebuild the partition (or drive) in question; therefore you must back up your data before you rebuild the partition. The rebuild command is:

```
/sbin/mke2fs /dev/partition /dev/hda7 -b newsize
```

4.2.5. Tuning the ext3 File System

The ext3 file system is essentially an enhanced version of the ext2 file system that provides stronger data integrity in the event that an unclean system shutdown occurs. Despite writing some data more than once, in most cases ext3 has a higher throughput than ext2 because ext3's journaling optimizes hard drive head motion. You can choose from three journaling modes to optimize speed, but doing so means trade-offs with regards to data integrity.

It is easy to change from ext2 to ext3 to gain the benefits of a robust journaling file system without reformatting.

For applications that require the best possible performance from ext3, you should use an external journal partition. The partition should be located on a device with similar or better performance characteristics than the device that contains the filesystem. Also, the journal partition must be created with the same block size as that used by the filesystem it is journaling. (Note that the external journal will utilize the entire partition.)

The following command creates a journal device to be used for a filesystem with a 4 KiB block size:

```
mke2fs -b 4096 -O journal_dev /dev/sdc1
```

The following command associates this journal with a filesystem:

```
mke2fs -b 4096 -J device=/dev/sdc1 /dev/sda1
```

You can specify other journal options using the `-J` option for `mke2fs` or `tune2fs`.

Tip

Every time a file is read or written, an access timestamp is updated for that file, which entails a read and write for the metadata for that file. On a busy filesystem, this can cause a large number of writes. This effect is magnified on ext3 and other filesystems that journal metadata, as the access timestamps must be written to the on-disk journal as well. To improve performance, you can disable the writing of that metadata. However, this will break `tmpwatch`, which uses access times to determine if files in temporary directories need to be removed. To learn more, refer to Section 4.9.4 *Reduce Disk Writes with noatime*.

4.2.6. Creating a Software RAID Device

Note

This section includes a summary of software RAID configuration. You can find more detailed information elsewhere, such as in the Software-RAID-HOWTO paper for 2.4 kernels (http://en.tldp.org/HOWTO/Software-RAID-HOWTO.html), as well as in the documentation and `man` pages for the `raidtools` package.

A faster, more expensive alternative to a software RAID is a hardware RAID. See Section 5.2 *Adding a Hardware RAID* for details.

Depending on the array, the disks used, and the controller, you may want to try a software RAID (Redundant Arrays of Inexpensive Disks). You will see excellent software RAID performance on a modern CPU that has a fast disk controller.

The easiest way to configure software RAID device is to do it during the Red Hat Linux installation. If you use the GUI installer, there are options in the disk-partition screen to create an "md" (meta-device), which is Linux terminology for a software RAID partition. You need to make partitions on each of the drives of type "Linux RAID", and then after creating all these partitions, create a new partition, (for example, `/test`), and select `md` as its type. Then you can select all the partitions that should be part of it, as well as the RAID type. For pure performance, RAID 0 is the best choice; however, there is no redundancy in this level.

Tip

If you are setting up a RAID 5 array and have the option to specify the parity algorithm, "left-symmetric", the default under RHEL, is the best choice for large reads.

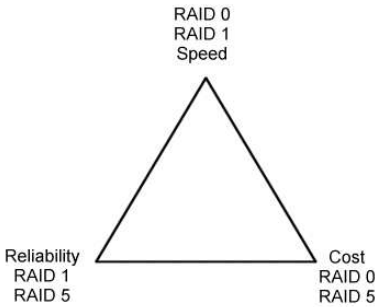

Figure 4-2. RAID Trade-offs

Note

Note that by default, you are limited to 27 drives in an MD device, so your application may be limited by that. However, if the drives are fast enough, you should be able to achieve >100 MB/s consistently.

4.2.6.1. Software RAID Tuning

To get the best performance, have a dedicated controller for each disk in the array, and make sure that the controller bus has sufficient bandwidth.

When configuring your RAID device, you can set the `chunk-size`. This is the minimum amount of data written to a disk in a single operation. For a RAID 5 device, the chunk size should be (the size of a typical I/O operation) divided by (the number of disks - 1). Typical values for the chunk size are between 32KB and 128KB, and are divisible by 4KB (the largest likely file system block size).

Tip

File system block size effects performance: large blocks are faster, but store small files inefficiently; small blocks use disk space efficiently, but are slower.

If you are using an ext2 or ext3 file system on your RAID device, set the `stride` option to the number of file-system blocks in a chunk. For example, if the chunk is 32KB and the file-system block is 4KB, the stride value should be 8:

```
/sbin/mke2fs -b 4096 stride=8 /dev/md0
```

4.2.6.2. How Partition Position Affects Drive Performance

The position of a partition on a hard drive has performance implications. Partitions stored at the very outer edge of a drive tend to be significantly faster than those on the inside. A good benchmarking trick is to use RAID across several drives, but only use a very small partition on the outside of the disk. This give both consistent performance, and the best performance. On most modern drives, or least drives using ZCAV (Zoned Constant Angular Velocity), this tends to be sectors with the lowest address (that is, the first partitions). To see the differences illustrated, see the ZCAV page (http://www.coker.com.au/bonnie++/zcav/).

4.2.7. Use a Red Hat Global File System (GFS)

If you have a large group of servers, consider using a Red Hat Global File System (GFS). Red Hat GFS helps Red Hat Enterprise Linux servers achieve high I/O throughput for demanding applications in database, file, and compute serving. You can scale performance incrementally for hundreds of Red Hat Enterprise Linux servers using Red Hat GFS and storage area networks (SANs) constructed with iSCSI or Fibre Channel.

GFS performs best in an environment with few, large files and little contention. The larger the reads/writes and the more independence each node in the cluster has, the better the performance.

For more information, see http://www.redhat.com/software/rha/gfs/.

4.3. Virtual Memory System Tuning

This section describes how to apply the information from Chapter 3 *Understanding the Virtual Memory Management System* to some example workloads and the various tuning parameters that may improve system performance.

4.3.1. File (IMAP, Web, etc.) Servers

This workload is geared towards performing a large amount of I/O to and from the local disk, thus benefiting from an adjustment allowing more files to be maintained in RAM. This would speed up I/O by caching more files in RAM and eliminating the need to wait for disk I/O to complete. A simple change to `sysctl.conf` as follows usually benefits this workload:

```
#increase the amount of RAM pagecache is allowed to use
#before we start moving it back to disk
vm.pagecache="10 40 100"
```

4.3.2. General Compute Server With Many Active Users

This workload is a very general kind of configuration. It involves many active users who will likely run many processes, all of which may or may not be CPU intensive or I/O intensive or a combination thereof. As the default VM configuration attempts to find a balance between I/O and process memory usage, it may be best to leave most configuration settings alone in this case. However, this environment will also likely contain many small processes, which regardless of workload, consume memory resources, particularly lowmem. It may help, therefore, to tune the VM to conserve low memory resources when possible:

```
#lower pagecache max to avoid using up all memory with cache
vm.pagecache="10 25 50"

#lower max_readahead to reduce the amount of unneeded IO
vm.max-readahead=16
```

See Section 3.3.11 *pagecache* for details.

4.3.3. Non-interactive (Batch) Computing Server

A batch computing server is usually the exact opposite of a file server. Applications run without human interaction, and they commonly perform very little I/O. The number of processes running on the system can be very closely controlled. Consequently this system should be tuned exclusively to allow for maximum throughput:

```
#Reduce the amount of pagecache normally allowed
vm.pagecache="1 10 100"

#do not worry about conserving lowmem; not that many processes
vm.max_map_count=128000

#increase overcommit; non-interactive processes can sleep
vm.overcommit=2

vm.overcommit_ratio=75
```

4.4. Kernel Tuning

Note

You can learn about the features in the Red Hat Linux 2.4 kernel at: http://www.redhat.com/software/rhel/kernel26/

The SYSV interprocess communication mechanisms allow programs to exchange information with each other via shared data structures in memory. You can tune SYSV mechanisms by using entries in `/proc/sys/kernel`:

- `/proc/sys/kernel/sem` controls:
  - The maximum number of semaphores per semaphores array
  - The maximum number of semaphores in the entire system
  - The maximum number of allowed operations per semaphore system call
  - The maximum number of semaphore arrays.

- `/proc/sys/kernel/msgmnb` is the maximum number of bytes in a single message queue.
- `/proc/sys/kernel/msgmni` is the maximum number of message queue identifiers.
- `/proc/sys/kernel/msgmax` is the maximum size of a message that can be passed between processes. Note that this memory cannot be swapped.
- `/proc/sys/kernel/shmmni` is the maximum number of shared-memory segments in the system.
- `/proc/sys/kernel/shmall` is the total amount of shared memory, in pages, that the system can use at one time. This should be at least `SHMMAX/PAGE_SIZE`.
- `/proc/sys/kernel/shmmax` is the maximum size of a shared memory segment. The kernel supports shared memory segments up to 4 GB-1.

4.4.1. Kernel Tuning Tool

There are a variety of ways to view and change kernel parameters—both with the graphical **Kernel Tuning** tool and from the command line. The **Kernel Tuning** tool has the advantage of making the tunable system values obvious. If you prefer to use the command line, you can find the tunable parameters by looking in the `/proc` and `/proc/sys/` files.

- To make changes using the **Kernel Tuning** graphical interface, run as root:
  `/usr/bin/redhat-config-proc`

  The **Kernel Tuning** tool appears:

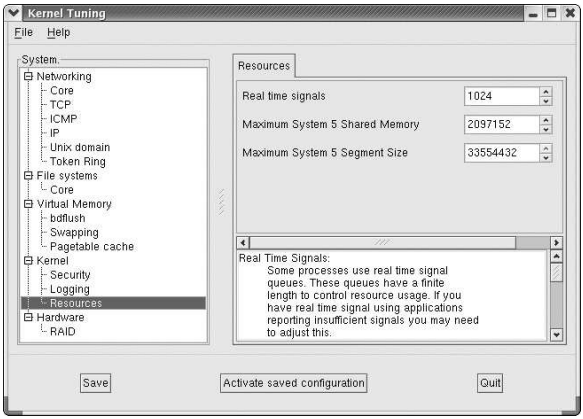

Figure 4-3. Kernel Tuning Tool

The text box at the bottom of the **Kernel Tuning** tool offers suggestions for improving performance of some types of application, such as Oracle.

- To view kernel parameter settings from the command line, use `cat` or `/sbin/sysctl`:
  `cat /proc/sys/kernel/shmmax`

  or
  `/sbin/sysctl kernel.shmmax`
- To change kernel parameter settings for the current session from the command line, use `echo` or `/sbin/sysctl`:
  `echo 33554430 > /proc/sys/kernel/shmmax`

  or
  `/sbin/sysctl -w kernel.shmmax=33554430`

  Of these options, `/sbin/sysctl` is preferred because it checks the validity of the command before making the change. This lessens the chance of misconfiguration caused by an input error.
- To make a change that will be used after the next reboot, edit the `/etc/sysctl.conf` file:
  `kernel.shmmax = 33554430`

  To make this change take effect before the next reboot, use the following command:
  `/sbin/sysctl -p`

Note

You use the `sysctl` command to tune entries under `/proc/sys` in a running kernel without rebooting. However, changes made in this way do not persist across reboots. To make a setting take effect at boot, configure the settings in the file `/etc/sysctl.conf`, which is read by `sysctl` early in the boot process.

To view the current values of all `sysctl` parameters, use:

```
sysctl -a
```

4.4.2. Flushing Old Pages with pdflush (RHEL 4) and kupdated (RHEL 3)

As memory is a finite resource, the kernel must flush back to disk the pages that have been dirtied but are no longer being used by a process. The mechanisms for doing this on RHEL 4 is `bdflush`.

Old pages are flushed by a different method. The mechanisms for doing this on RHEL 4 is `pdflush`; on RHEL 3 it is `kupdated`.

4.4.3. Setting bdflush for Server Operation (RHEL 3)

Some of the default kernel parameters for system performance are geared more towards workstation performance than to file server and large disk-I/O types of operations. The most important of these is the `bdflush` value in `/proc/sys/vm/bdflush`.

The `bdflush` thread is woken up when the kernel needs additional buffers but cannot find enough free memory to allocate the buffers, or when the appropriate sysrequest key is pressed.

Both `bdflush` and `kupdated` can be tuned with the `vm.bdflush` sysctl. The file contains nine settings:

nfract

Percentage of the buffer cache that must be dirty to activate `bdflush`.

ndirty

Maximum number of dirty blocks to write out per wake-cycle.

Setting 3

Unused.

Setting 4

Unused.

interval

Delay, in jiffies, between `kupdated` flushes.

age_buffer

Time for normal buffer to age before being flushed.

nfract_sync

Percentage of the buffer cache that can be dirty before `bdflush` begins synchronous behavior.

nfract_stop

Percentage of the dirty buffer cache at which `bdflush` stops.

Setting 9

Unused.

The `/usr/src/linux-2.4/Documentation/sysctl/vm.txt` file documents these in detail.

Parameter	Minimum	Default	Maximum
nfract	0	50	100
ndirty	1	500	50000
unused	0	0	20000
unused	0	0	20000

Parameter	Minimum	Default	Maximum
interval	0	5*HZ	10000*HZ
age_buffer	1*HZ	30*HZ	10000*HZ
nfract_sync	0	80	100
nfract_stop	0	50	100
unused	0	0	0

Table 4-1. `vm.bdflush` Settings

A good set of values for a large file server is:

```
echo 100 5000 640 2560 150 30000 5000 1884 2 > /proc/sys/vm/bdflush
```

4.4.4. Disk I/O Elevators

On systems that are consistently doing a large amount of disk I/O, tuning the disk I/O elevators may be useful. This is a 2.4 kernel feature that allows some control over latency vs. throughput by changing the way disk I/O elevators operate.

This works by changing how long the I/O scheduler will let a request sit in the queue before it has to be handled. Because the I/O scheduler can collapse some requests together, having a lot of items in the queue means more can be coalesced, which can increase throughput.

Changing the maximum latency on items in the queue allows you to trade disk I/O latency for throughput, and vice versa.

The tool `/sbin/elvtune` (part of util-linux) enables you to change these maximum latency values. Lower values means less latency, but also less throughput. The values can be set for the read and write queues separately.

To determine what the current settings are, issue:

```
/sbin/elvtune /dev/hda1
```

substituting the appropriate device of course. Default values are 4096 for read, 8192 for writes on Ext3 systems.

To set new values of 2000 for read and 4000 for write, use:

```
/sbin/elvtune -r 2000 -w 4000 /dev/hda1
```

Note that these values are examples only, and are *not* recommended tuning values—those values depend on your situation. However, read values should be half of write values.

If you make changes that you want to have after the next reboot, add the `/sbin/elvtune` command to the `/etc/rc.d/rc.local` file.

The units of these values are basically "sectors of writes before reads are allowed." The kernel attempts to do all reads, then all writes, etc. in an attempt to prevent disk I/O mode switching, which can be slow; this enables you to alter how long it waits before switching.

Monitor the output of `iostat -d -x` *`device`* to get an idea of the effectiveness of these changes. The `avgrq-sz` and `avgqu-sz` values (average size of request and average queue length) should be affected by these elevator changes. (See the man page for `iostat` for more information.) Lowering the latency should cause the `avgrq-sz` to go down, for example.

See the `elvtune` man page for more information. Some information from when this feature was introduced is also at Lwn.net (http://lwn.net/2000/1123/kernel.php3).

Tip

If you are running RHEL 3 on a desktop (*not* on a server), the following command can speed up desktop responsiveness:

```
elvtune -r 8 -w 8 /dev/hda
```

4.4.5. File Descriptor Limits

Open TCP sockets and some applications are prone to opening a large number of file descriptors. The default number of available file descriptors is 1024, but this may need to be increased. To view the currently allowed open files for a user, enter:

```
ulimit -a
```

The system returns output similar to the following:

```
core file size        (blocks, -c) 0
data seg size         (kbytes, -d) unlimited
file size             (blocks, -f) unlimited
max locked memory     (kbytes, -l) unlimited
max memory size       (kbytes, -m) unlimited
open files                    (-n) 1024
pipe size          (512 bytes, -p) 8
stack size            (kbytes, -s) 8192
cpu time             (seconds, -t) unlimited
max user processes            (-u) 4091
virtual memory        (kbytes, -v) unlimited
```

The theoretical limit is roughly a million file descriptors, however the practical limit is much less. When increasing file limit descriptors, you may want to simply double the value. If you need to increase the default value of 1024, increase the value to 2048 first. If you need to increase it again, try 4096, etc.

Note that each resource has a hard and a soft limit. The hard limit represents the maximum value a soft limit may have, and the soft limit represents the limit being actively enforced on the system at that time. Hard limits can be lowered by all users, but not raised, and soft limits cannot be set higher than hard limits. Only root may raise hard limits.

For example:

```
echo 128000 > /proc/sys/fs/inode-max
echo 2048 > /proc/sys/fs/file-max
```

and as root:

```
ulimit -n 2048
```

Note

On 2.4 kernels, the `inode-max` entry is no longer needed.

You probably want to add these to `/etc/rc.d/rc.local` so they get set on each boot.

There are other ways to make these changes persist after reboots. For example, you can use `/etc/sysctl.conf` and `/etc/security/limits.conf` to set and save these values.

```
Tom  hard rss 13000
@faculty hard rss 25000
```

Example 4-1. Setting Limits for User Tom and Group faculty

If you get errors of the variety "Unable to open file descriptor," you definitely need to increase these values.

You can examine the contents of `/proc/sys/fs/file-nr` to determine the number of allocated file handles, the number of file handles currently being used, and the max number of file handles.

4.5. Network Interface Card Tuning

Warning

In almost all circumstances, Network Interface Cards auto-negotiate the correct speed and mode. If you force the card to run at a certain speed or mode, you must be absolutely certain the device at the other end of the connection has the same parameters; otherwise the connection will not work or will work very slowly.

Most benchmarks benefit greatly when the network interface cards (NICs) in use are well supported, with a well-written driver. Examples of such cards include eepro100, tulip, relatively new 3Com cards, and AceNIC and Syskonnect gigabit cards.

Making sure the cards are running in full-duplex mode is also very often critical to benchmark performance. Depending on the networking hardware used, some of the cards may not autosense properly and may not run full duplex by default.

Many cards include module options that can be used to force the cards into full-duplex mode. Some examples for common cards include:

```
alias eth0 eepro100
options eepro100 full_duplex=1
alias eth1 tulip
options tulip full_duplex=1
```

4.5.1. Network Interface Configuration

You must set the frame size to match all systems on the LAN or VLAN. This is the default unless you are using jumbo frames (9,000-byte frames).

Other characteristics like mode and frame size just need to match the hub, switch, or router at the other end of the link.

4.5.2. Ethernet Channel Bonding

If your machine has two network interface cards, you can use the Ethernet bonding driver to bond them into a single virtual interface. The driver will then alternate between cards when sending and receiving packets on the virtual interface.

- On RHEL 3, you configure the virtual interface in `/etc/modules.conf`:
  ```
  alias bond0 bonding
  options bond0 mode=5 miimon=100 use_carrier=0
  ```
- On RHEL 4, you configure the virtual interface in `/etc/modprobe.conf`:
  ```
  alias bond0 bonding
  options bond0 mode=5 miimon=100 use_carrier=0
  ```

where `mode=5` is the load-balancing mode, `miimon` is the number of milliseconds between checks of each interface for their link status, `use_carrier` specifies how the link is checked: use `1` to specify the `net_if_carrier_ok()` function, or `0` to specify that the driver supports only `mii-tool` or `ethtool`.

Note

Neither `/sbin/mii-tool` nor `/sbin/ethtool` has support for all Linux drivers. Try using `ethtool` first; if that does not work, try `mii-tool`.

Each interface also requires a configuration file in `/etc/sysconfig/network-scripts/`:

```
DEVICE=bond0
IPADDR=192.168.0.1
NETMASK=255.255.255.0
GATEWAY=192.168.0.254
ONBOOT=yes
BOOTPROTO=none
```

Example 4-2. ifcfg-bond0

```
DEVICE=eth0
MASTER=bond0
SLAVE=yes
ONBOOT=yes
BOOTPROTO=none
```

Example 4-3. ifcfg-eth0

```
DEVICE=eth1
MASTER=bond0
SLAVE=yes
ONBOOT=yes
BOOTPROTO=none
```

Example 4-4. ifcfg-eth1

4.6. TCP Tuning

For servers that are serving up huge numbers of concurrent sessions, there are some TCP options that should probably be enabled. With a large number of clients making server requests, its not uncommon for the server to have 20000 or more open sockets.

In order to optimize TCP performance for this situation, try tuning the following parameters:

```
echo 1024 65000 > /proc/sys/net/ipv4/ip_local_port_range
```

This enables more local ports to be available. Generally this is not an issue, but in a benchmarking scenario you often need more ports available. A common example is clients running **ab**, **http_load**, or similar software.

In the case of firewalls or servers doing NAT or masquerading, you may not be able to use the full port range this way, because of the need for high ports for use in NAT.

Increasing the amount of memory associated with socket buffers can often improve performance. In particular, NFS servers and Apache systems that have large buffers configured can benefit from this.

```
echo 262143 > /proc/sys/net/core/rmem_max
echo 262143 > /proc/sys/net/core/rmem_default
```

This will increase the amount of memory available for socket input queues. The corresponding `wmem_*` values do the same for output queues.

Note

With 2.4.x kernels, these values "autotune" fairly well, so some people suggest instead just changing the values in:

```
/proc/sys/net/ipv4/tcp_rmem
/proc/sys/net/ipv4/tcp_wmem
```

There are three values here, "min, default, and max." These reduce the amount of work the TCP stack has to do, so it is often helpful in this situation.

```
echo 0 > /proc/sys/net/ipv4/tcp_sack
echo 0 > /proc/sys/net/ipv4/tcp_timestamps
```

Tip

Be sure to also see `LMbench`, the TCP benchmarking utility described in Section 2.1 *Benchmarking Utilities*, and `TTCP` and `NetPerf`, which are described in Section 2.6 *Network Monitoring Utilities*.

4.6.1. When a CPU is Overloaded by TCP

It is possible to optimize the network beyond the capability of the CPU to handle the network traffic. In this situation, there are steps you can take:

- Reduce or offload the TCP/IP overhead.

One way to reduce TCP/IP overhead is to replace TCP with UDP (User Datagram Protocol). Note that while UDP is potentially faster because of reduced overheads, it is less reliable: messages may be lost, duplicated, or received out of order.

Another option is to reduce the number of TCP packets by employing network cards that support jumbo frames (9,000-byte frames). This can provide a great increase in the transmission rate.

- Reduce context switching by using interrupt coalescence or by offloading packet checksums.

  The CPU learns about the arrival of data packets from interrupts. If interrupts arrive more quickly than the CPU can process them, the CPU enters a "livelock" state. Interrupt coalescence is the grouping of multiple packets into a single interrupt. While this technique is effective, it can make TCP measurements inaccurate.

  The checksum is a number that is calculated from the data packet and attached to the end of the packet. By comparing the checksum to the data, data corruption can be detected. Typically this check is performed on the CPU; however, some network cards can perform this operation, which frees the CPU to perform other tasks.

4.7. NFS Tuning

Before performing NFS tuning, ensure that TCP performance is satisfactory (see Section 4.6 *TCP Tuning*).

The basic NFS-tuning steps include:

- Try using NFSv3 if you are currently using NFSv2. There can be very significant performance increases with this change.

  

  Note

  TCP is recommended for NFSv3 and required for NFSv4. NFSv4 is in 2.6 kernels only (RHEL 4 and Fedora).

- Increasing the read write block size. This is done with the `rsize` and `wsize` mount options. They need to the mount options used by the NFS clients. Values of 4096 and 8192 reportedly increase performance substantially. However, see the notes in the NFS Tuning HOWTO (http://nfs.sourceforge.net/nfs-howto/performance.html) for information about experimenting and measuring the performance implications. The limits on these are 8192 for NFSv2 and 32768 for NFSv3
- Another approach is to increase the number of nfsd threads running. You can increase the number of nfs daemons by setting the value of `RPCNFSDCOUNT` in `/etc/sysconfig/nfs`. The best way to determine if you need this is to experiment. The HOWTO mentions a way to determine thread usage, but that does not seem supported in all kernels.
- Another good tool for getting some handle on NFS server performance is `nfsstat`. This utility reads the information in `/proc/net/rpc/nfs[d]` and displays it in a somewhat readable format. Some information intended for tuning Solaris, but useful for its description of the nfsstat format, is here: http://www.princeton.edu/~unix/Solaris/troubleshoot/nfsstat.html.

4.7.1. Tuning NFS Buffer Sizes

Busy NFS servers may get improved performance from increasing the buffer sizes:

```
sysctl -w net.core.rmem_default=262144
```

```
sysctl -w net.core.rmem_max=262144
sysctl -w net.ipv4.ipfrag_high_thresh=524288
sysctl -w net.ipv4.ipfrag_low_thresh=262144
```

4.7.2. NFS over UDP

Some users of NFS over UDP report that setting larger socket buffer sizes in `/etc/sysctl` can improve performance:

```
net.core.rmem_max=262143
net.core.wmem_max=262143
net.core.rmem_default=262143
net.core.wmem_default=262143
```

Tip

By default, NFS uses UDP. However, TCP may be preferable over WAN connections or in other high-loss networks.

If you do use TCP in an environment that has high packet loss, you could adjust the `net.ipv4.tcp_syn_retries` parameter. The `net.ipv4.tcp_syn_retries` parameter specifies the maximum number of SYN packets to send to try to establish a TCP connection. The default is 5; the maximum is 255. The default value corresponds to a connection time of approximately 180 seconds.

4.8. Java Tuning

If there appears to be memory pressure, turning on verbose logging for garbage collection can be helpful. Each JVM has a different way to turn on verbose garbage collection. With the IBM JVM, start your server with:

```
java -verbose:gc
```

This logs all garbage collection activity to your server log file.

There are many other diagnostic tools you can use, including the following:

JProbe Suite

A complete performance toolkit for Java code tuning. JProbe helps you diagnose and resolve performance bottlenecks, memory leaks, excessive garbage collection, threading issues, and coverage deficiencies in your J2EE and J2SE applications.

See JProbe (http://java.quest.com/jprobe/)

Optimizeit

Borland® Optimizeit Enterprise Suite 6 enables you to isolate and resolve performance problems in J2EE applications. It contains memory and CPU profiling, as well as thread and code coverage analysis. You can track performance bottlenecks from the JDBC, JMS, JNDI, JSP, and EJB level to the exact line of source code.

See Optimizeit (http://www.borland.com/optimizeit/).

For an example of Java tuning, see Chapter 8 *Analyzing Java Application Performance Problems Using the IBM JVM*.

4.9. Apache Configuration

Before you begin tuning Apache, you need to establish benchmarks for your system. You may use the tools described in Section 2.1.1 *Web Server Benchmarking Tools*.

You need to measure the throughput (the amount of work a system can perform in a period of time) and latency (the amount of time the system takes to respond to a request) for static pages, dynamic content, and if appropriate, secure content.

In a well-tuned system, static pages stress the network, dynamic content makes use of the system's RAM, and secure content uses the CPU. For all types of content, memory is the first or second most important system characteristic. Whenever possible, have enough RAM to minimize the chance of the kernel having to access the swap partition when Apache is running.

4.9.1. Configuring Apache for Benchmark Scores

Start many initial daemons in order to get good benchmark scores:

```
#######
MinSpareServers 20
MaxSpareServers 80
StartServers 32

# this can be higher if Apache is recompiled
MaxClients 256

MaxRequestsPerChild 10000
```

Note

Starting a massive number of httpd processes is really a benchmark hack. In most real-world cases, setting a high number for max servers and a sane spare server setting will be more than adequate. It is just the instant on load that benchmarks typically generate that the StartServers helps with.

The `MaxRequestPerChild` should be increased if you are sure that your httpd processes do not leak memory. Setting this value to 0 will cause the processes to never reach a limit.

One of the best resources on tuning these values, especially for application servers, is the mod_perl performance tuning documentation http://perl.apache.org/guide/performance.html).

4.9.2. Compiling Apache 2.0 for Your Environment

You can compile Apache to disable features that you do not use in your environment. See the *Modules enabled by default* section of the `configure` man page (http://httpd.apache.org/docs-2.0/programs/configure.html). If there are features that you do not use, see http://httpd.apache.org/docs-2.0/install.html to learn how to compile Apache.

4.9.3. Increasing the Number of Available File Handles

To increase the number of pages that Apache can serve concurrently, increase the number of file handles. To determine the current setting, enter:

```
/sbin/sysctl fs.file-max
```

To increase the number of file handles to 256,000, enter:

```
/sbin/sysctl -w fs.file-max=256000
```

4.9.4. Reduce Disk Writes with noatime

For pure file server applications such as web servers and Samba servers, disable the `atime` option on the local file system. This disables updating the `atime` value for the file, which indicates the last time a file was accessed. Because this information is not very useful in this situation, and causes extra disk writes, it is typically disabled. To do this, just edit `/etc/fstab` and add `noatime` as a mount option for the local file system.

For example:

```
/dev/sdb3 /test ext3 noatime 1 2
```

Notes

Turning off `atime` updates breaks `tmpwatch`, which uses access times to determine whether or not to remove temporary directories.

This technique cannot be used on NFS-mounted file systems.

Incremental backups require information normally provided by `atime`; by setting `noatime`, incremental backups become full backups.

4.9.5. ListenBacklog

One of the most frustrating thing for a user of a website, is to get "connection refused" error messages. With Apache, the common cause of this is for the number of concurrent connections to exceed the number of available httpd processes that are available to handle connections.

The Apache `ListenBacklog` parameter lets you specify what backlog parameter is set to `listen()`. By default on Linux, this can be as high as 128.

Increasing this allows a limited number of HTTPDs to handle a burst of attempted connections.

4.9.6. Using Static Content Servers

If the servers are serving lots of static files (images, videos, PDFs, etc.), a common approach is to serve these files off a dedicated server. This could be a very light Apache setup, or any many cases, something like thttpd, boa, khttpd, or TUX. In some cases it is possible to run the static server on the same server, addressed via a different hostname.

For purely static content, some of the smaller, more lightweight web servers can offer very good performance. They are not nearly as powerful or as flexible as Apache, but for very specific performance-crucial tasks, they can be very useful.

- Boa: http://www.boa.org/
- thttpd: http://www.acme.com/software/thttpd/
- mathopd: http://mathop.diva.nl

If you need even more extreme web server performance, you probably want to take a look at TUX.

4.9.7. Proxy Usage

For servers that are serving dynamic content, or ssl content, a better approach is to employ a reverse-proxy. Typically, this would done with either Apache's `mod_proxy`, or Squid. There can be several advantages from this type of configuration, including content caching, load balancing, and the prospect of moving slow connections to lighter-weight servers.

The easiest approach is probably to use `mod_proxy` and the "ProxyPass" directive to pass content to another server. mod_proxy supports a degree of caching that can offer a significant performance boost. But another advantage is that since the proxy server and the web server are likely to have a very fast interconnect, the web server can quickly serve up large content, freeing up an Apache process, while the proxy slowly feeds out the content to clients. This can be further enhanced by increasing the amount of socket-buffer memory that is available for the kernel. See Section 4.6 *TCP Tuning*.

4.9.8. Samba Tuning

Samba is an Open Source/Free Software suite that provides seamless file and print services to SMB/CIFS clients.

Samba performance should be roughly 75% of ftp performance. If you are see less performance, there are a number of tweaks you can try to improve Samba's performance over the default. The default is best for general-purpose file sharing, but for extreme uses, there are a couple of options.

The first option is to rebuild it with mmap support. In cases where you are serving up a large number of small files, this seems to be particularly useful. You just need to add `--with-mmap` to the configure line.

You also want to make sure the following options are enabled in the `/etc/samba/smb.conf` file:

```
read raw = no
write raw = no
read prediction = true
level2 oplocks = true
```

Tip

While the above settings are reasonable rules-of-thumb, test the performance of the `read raw = no` and `write raw = no` settings in your environment.

In some circumstances you may want to change the `socket options` setting. By default, TCP buffers characters so that larger packets are transmitted, rather than single characters. To instead send packets immediately, set:

```
socket options = TCP_NODELAY
```

On a LAN, you can set:

```
socket options = IPTOS_LOWDELAY TCP_NODELAY
```

On a WAN, you can set:

```
socket options = IPTOS_THROUGHPUT TCP_NODELAY
```

If your system's disk access and network access speeds are similar, you can set the `read size` parameter to enable the server to begin reading the data before the entire packet has been received:

```
read size = 32768
```

Notes

You can reduce time-consuming disk writes by disabling updates to file's `atime` value (which indicates the last time the file was accessed). See Section 4.9.4 *Reduce Disk Writes with noatime*.

Also, ensure that logging is set no higher than 2 to avoid excessive logging.

4.9.9. OpenLDAP Tuning

Lightweight Directory Access Protocol (LDAP) is a set of open protocols used to access centrally stored information over a network. It is based on the X.500 standard for directory sharing, but is less complex and resource intensive.

The most important tuning aspect for OpenLDAP is deciding which attributes you want to build indexes on.

Some sample values:

```
cachesize 10000
dbcachesize 100000
sizelimit 10000
loglevel 0
dbcacheNoWsync

index cn,uid
index uidnumber
index gid
index gidnumber
index mail
```

If you add those parameters to `/etc/openldap/slapd.conf` before entering the information into the database, they will all get indexed and performance will increase.

Chapter 5.

Increasing Performance Through System Changes

The greatest change to the Red Hat Enterprise Linux operating system (in terms of performance) is the NPTL threading library (which appeared in Red Hat Enterprise Linux 3). Other significant changes include hardware support—such as the CPU type and speed, and the amount of memory and other system resources that are supported.

This chapter looks at changes you can make to your system beyond the configuration changes discussed previously.

- Upgrading your system hardware.
- Adding a hardware RAID device.

5.1. Can You Upgrade Your Hardware?

You may be able to increase performance by:

- Upgrading the CPU. If your application is single-threaded, you may get better performance from a very fast CPU than from a multiple-CPU system.
- Having more than one CPU. Note, however, that adding more processors may not help if the memory bandwidth is the limitation. This may be an issue for some of Intel machines that have all the processors on the same memory bus.
- Increasing the amount of RAM in your system. If you have run the utilities described in Section 2.5 *Memory Monitoring Utilities* and discovered that your system is swapping, you should add RAM.

The table that follows shows, where known, the limits for the number of CPUs and the amount of RAM. For example, if your application would benefit from running on a server with more than two CPUs, you must run RHEL AS. To learn about limits for various processors and subsystems, see Table 5-2.

Red Hat Enterprise Linux Product:	AS	ES	WS	Desktop
Maximum memory supported	-[a]	32-bit systems: 8GB; 64-bit systems: 16GB	-	4GB
Maximum CPUs supported	-	2	2	1
Notes: a. Entries marked '-' have upper-limits defined by hardware, software, or architectural capabilities.				

Table 5-1. Potential RAM Upgrades and Additional CPUs

Notes

A CPU chip containing hyper-threaded processing elements is counted as a single CPU. Similarly, multi-core processors are also counted as a single CPU.

If you add memory, make sure that the memory is of a type that is compatible with the kernel.

Some motherboards support memory interleaving; this is similar in concept to disk striping. Memory interleaving increases bandwidth by enabling simultaneous access to multiple chunks of memory. With memory interleaving, having eight 128MB memory modules will be faster than having two 512MB memory modules

For updated information, see http://www.redhat.com/software/rhel/comparison/.

5.1.1. Red Hat Enterprise Linux 3 System Limits

The following table lists some Red Hat Enterprise Linux 3 supported system and software limits. As further testing is completed, this table will be updated here: http://www.redhat.com/software/rhel/configuration/.

These minimum and maximum system configuration limits identify the technical capabilities of the Red Hat Enterprise Linux technology. Please note that specific products, such as Red Hat Enterprise Linux WS and ES, support limited configurations as noted here: http://www.redhat.com/software/rhel/comparison/.

Characteristic	Minimum	Maximum	Comments
X86 Memory:	256MB	64GB	Maximum varies with chosen kernel; Red Hat Enterprise Linux ES supports up to 8GB
X86 CPUs:	1 (300MHz, i686)	16	16 physical CPUs or 8 Hyperthreaded CPUs; AMD K6 (i586) is not supported, Red Hat Enterprise Linux WS and ES support up to 2 physical CPUs (4 Hyperthreaded) CPUs per system.
Itanium2* Memory:	512MB	128GB	128GB applies to HP Integrity systems. Maximum memory for Intel Tiger-based systems is 32GB.
Itanium2 CPUs:	1	8	Red Hat Enterprise Linux WS for Itanium supports up to 2 CPUs per system.
AMD64 Memory:	512MB	64GB	
AMD64 CPUs:	1	4	Red Hat Enterprise Linux WS for AMD64 supports up to 2 CPUs per system.
Network Interface Cards	0	30	
IDE Disks	0	20	
SCSI Disks	0	256	
SATA Disks	0	2	
Devices in a RAID/MD device		27	
Block device size		1TB	

Characteristic	Minimum	Maximum	Comments
File system size	800MB	1TB	Quoted minimum is for a custom installation. Sparse files can be up to 4TB
NFS mounts	0	800	
Graphics heads	0	2	
Cluster Suite nodes	2	8	
Processes	1	32000	
Threaded Processes	0	32000	
Thread Limit	0	32000	Total threads per system; this is a kernel limit.

Table 5-2. System Limits by RHEL Version

* Itanium2 only; the original Intel Itanium processor is not supported.

Note

Any updated information for *System Limits by RHEL Version* can be found at: http://www.redhat.com/software/rhel/configuration/

5.1.2. Applications with Hardware Prerequisites

Developer's Suite has specific hardware prerequisites.

RHEL Product	Minimum	Recommended
Developer's Suite (Eclipse)	0.5 GB RAM	1.0 GB RAM
	Pentium III	Pentium III

Table 5-3. Developer's Suite Hardware Prerequisites

5.2. Adding a Hardware RAID

You may be able to boost I/O performance by adding a hardware RAID (Redundant Array of Inexpensive Disks) device. Of course, a hardware RAID device can improve performance only if the SCSI bus to which it is attached has the capacity to transfer data—several fast SCSI devices can overwhelm a bus.

Another alternative—a software RAID—is discussed in Section 4.2.6 *Creating a Software RAID Device*. In general, on a lightly loaded system, software RAID will usually be faster. On a heavily loaded system, hardware RAID may be faster (the custom ASICs on the cards are usually slower than modern general purpose CPUs). Hardware RAIDs have the additional benefit of higher reliability.

RAIDS can be of different "levels" (types). Some of the more common types are:

RAID level 0

A virtual device in which data written to the RAID is "striped" across the physical disks in the array.

A *stripe* is a fixed-sized band of storage that spans each device in the array. The width of the stripe is critical: if the stripe is too large, one physical device may handle a disproportionate number of requests. Therefore you should make the stripe width size match the size of the average data request:

```
/sbin/mke2fs -R stride=stripe_size
```

RAID 0 arrays can boost both sequential transfers and random accesses. Sequential transfers benefit because the transfer rate for a RAID is greater than for a single disk. Random transfers benefit because there are fewer seek requests to any particular drive. However, performance does not scale linearly.

Because data is not stored redundantly, this configuration is vulnerable to disk failure.

RAID level 1

A virtual device composed of two or three physical devices. All writes to the virtual device are mirrored to each physical disk. If a third drive is used, it can be taken offline to perform backups.

The read performance of RAID 1 is the same as for a single disk; the write performance depends on the write policy. A *parallel write policy* at best will be as fast as for a single disk. A *serial write policy* at best will be half as fast as for a single disk. Not all RAID hardware controllers support a parallel write policy.

RAID level 5

A virtual device composed of at least three physical devices. RAID 5 is similar to RAID 0 except that one of the chunks in the stripe stores parity information for the remainder of the stripes in the device. The parity chunk is calculated by XORing the information in the data chunks. If a disk containing a member chunk of the stripe fails, the data can be calculated by performing the XOR operation using the parity chunk in place of the data chunk. To avoid bottlenecks, the parity chunk for each data chunk alternates between disks.

RAID 5 has the best reliability-to-cost ratio. It also has performance similar to a RAID 0 device that has one less disk (that is, a 4-disk RAID 5 performs as well as a 3-disk RAID 0). However, write performance is slower because of the need to update the parity stripe.

As with RAID 0, you should make the stripe width size match the size of the average data request.

RAID level 10

A mirrored, striped array whose components are RAID 1 arrays.

RAID level 0+1

Combines the advantages of RAID 0 and RAID 1.

Note

Further definitions with cost and performance analyses are available at the RAID tutorial at http://www.acnc.com/raid.html.

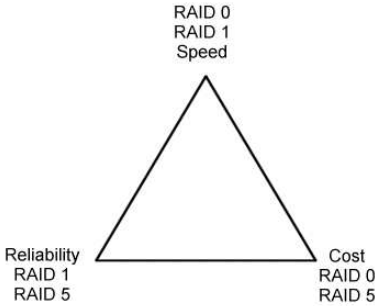

Figure 5-1. RAID Trade-offs

The factors that affect RAID performance are:

- The RAID level. Which RAID level is best depends on the application: RAID 10 provides excellent read/write performance, but for environments that mainly have reads, RAID 5 is a more cost-effective solution. For operations requiring fast, redundant read/write operations, such as database redo logs, RAID 1 is recommended.
- The number of disks
- The speed of the disks
- The speed of the disk controllers
- The size of the stripe (it should be a multiple of the data block size). Note that with Samba, the cache is specified in bytes, so if the RAID stripe size is 32KB, you would set:
  ```
  write cache size = 32768
  ```
- The RAID controller's cache size (particularly with RAID 5 systems). This cache can enable several small write requests to be grouped into a single large request.
- The database block size, when used with database servers.

redhat

Chapter 6.

Detecting and Preventing Application Memory Problems

C, and to a lesser extent C++, are sometimes referred to as a "portable assembly language." This means that they are portable across platforms, but are low-level enough to comfortably deal with hardware and raw bits in memory. This makes them particularly suited for writing systems software such as operating systems, databases, network servers, and data/language processors. However, the runtime model of C/C++ does not include any checking of pointer use, so errors can easily creep in.

Several kinds of pointer-use errors are widely known by every C/C++ programmer; accessing freed objects, going past buffer boundaries, dereferencing NULL or other bad pointers, can each result in a spectrum of effects, including random glitches, outright crashes, and security breaches (stack-smashing buffer overrun errors, where malevolent input causes pointers go beyond their bounds to corrupt memory). Such bugs can induce hard-to-debug delayed failures.

6.1. Using valgrind

`valgrind` executes your program in a virtual machine, keeping track of the memory blocks that have been allocated, initialized, and freed. It simulates every instruction a program executes, which reveals errors not only in an application, but also in all supporting dynamically-linked (`.so-format`) libraries, including the GNU C library, the X client libraries, Qt (if you work with KDE), and so on.

`valgrind` can detect invalid write accesses, attempted reads of uninitialized memory, and the use of undefined values:

Using uninitialized memory.

Uninitialized data can come from uninitialized local variables or from `malloc`'ed blocks before your program has written data to that area.

Reading/writing memory after it has been freed.
Reading/writing after the end of `malloc`'ed blocks.

Errors that occur from attempting to access data at the wrong time or the wrong place.

Reading/writing inappropriate areas on the stack.

This can happen in two ways:

- Illegal read/write errors occur when you try to access an address that is not in the address range of your program.
- Invalid free errors occur when you try to free areas that have not been allocated.

Memory leaks.

Here, pointers to `malloc`'ed blocks are lost, so the memory is never freed.

Mismatched allocate/free functions.

In C++ you can choose from more than one function allocate memory, as long as you follow these rules to free that memory:

- Memory allocated with `malloc`, `calloc`, `realloc`, `valloc`, or `memalign` must be deallocated with `free`.

- Memory allocated with `new[]` must be deallocated with `delete[]`.
- Memory allocated with `new` must be deallocated with `delete`.

Memory errors can be difficult to detect as the symptoms of the error may occur far from the cause. However, `valgrind` can detect all of the above errors, as well as:

- Errors that occur because of invalid system-call parameters (overlapping src and dst pointers in `memcpy()` and related functions)
- Some errors resulting from non-conformance to the POSIX pthreads API.

6.1.1. valgrind Overview

`valgrind` consists of a core, which simulates the operation of an x86 CPU in software, and a series of tools for debugging and profiling. The tools include:

Memcheck

Memcheck detects memory-management problems in programs by tracking all `malloc`/`new` calls and the coresponding `free`/`delete` calls.

Note that program run up to 50 times slower when you use `memcheck`.

Addrcheck

Addrcheck is identical to `memcheck` except that it does not do any uninitialised-value checks. Addrcheck does not catch the uninitialised-value errors that `memcheck` can find, but programs run twice as fast as on `memcheck`, and much less memory is used.

Cachegrind

`Cachegrind`, the cache profiler, simulates the I1, D1 and L2 caches in the CPU so that it can pinpoint the sources of cache misses in the code. It can show the number of cache misses, memory references, and instructions accruing to each line of source code, with per-function, per-module, and whole-program summaries. It can also show counts for each individual x86 instruction.

Cachegrind is complemented by the KCacheGrind visualization tool (http://kcachegrind.sourceforge.net), a KDE application that graphs these profiling results.

Helgrind

`Helgrind` finds data races in multithreaded programs. Helgrind looks for memory locations that are accessed by more than one (POSIX p-)thread, but for which no consistently used (pthread_mutex_)lock can be found. Such locations indicate missing synchronization between threads, which can cause timing-dependent problems.

Note

Some minor tools (`corecheck`, `lackey`, and `Nulgrind`) are supplied mainly to illustrate how to create simple tools.

6.1.2. valgrind Limitations and Dependencies

- Not all invalid memory access are detected. For example, writes that are too large for their allocated area will succeed.
- `valgrind` has a library implementation for multi-threaded applications, which is both slow and not fully POSIX-compliant. Applications that work under `libpthread` may fail under `valgrind`.
- False-positives are known; false negatives are a possibility.
- `valgrind` is currently of production quality only for IA-32.
- `valgrind` is up to 50 times slower than native execution.
- `valgrind` works only on the Linux platform (kernels 2.2.X or 2.4.X) on x86s. Glibc 2.1.X or 2.2.X is also required for `valgrind`; glibc-2.3.2+, with the NPTL (Native Posix Threads Library) package, will not work.

6.1.3. Before Running valgrind

You should recompile your application and libraries with debugging info enabled (the `-g` flag) to enable `valgrind` to know to which function a particular piece of code belongs.

You should also set the `-fno-inline` option, which makes it easier to see the function-call chain and to navigate in large C++ applications. You do not have to use this option, but doing so helps `valgrind` produce more accurate and usable error reports.

Note

Sometimes optimization levels at `-O2` and above generate code that leads Memcheck to wrongly report uninitialised value errors. The best solution is to turn off optimization altogether, but as this often makes things unmanagably slow, a compromise is to use `-O`. This gets you the majority of the benefits of higher optimization levels while keeping relatively small the chances of false errors from Memcheck.

6.1.4. Running valgrind

To run `valgrind`, use

```
# valgrind [ --tool=toolname ] commandname
```

Memcheck is the default `--tool`.

For example:

```
valgrind --tool=memcheck ls -l
```

Errors are reported before the associated operation actually happens. If you are using a tool (Memcheck, Addrcheck) which does address checking and your program attempts to read from address zero, the tool will emit a message to this effect, then the program will die with a segmentation fault.

6.1.4.1. valgrind Processing Options

`valgrind` has many processing options that you may find useful:

`--log-file=`*`filename`*

Writes the commentary to *`filename`*`.pid`*`pidnumber`* This is helpful when running `valgrind` on a tree of processes at once, as each process writes to its own logfile.

`-v`

Reports how many times each error occurred. When execution finishes, all the reports are printed out, sorted by their occurrence counts. This makes it easy to see which errors have occurred most frequently.

`--leak-check=yes`

Searches for memory leaks when the program being tested exits.

`--error-limit=no flag`

Disables the cutoff for error reports (300 different errors or 30000 errors in total—after suppressed errors are removed).

Note

These cutoff limits are set in `vg_include.h` and can be modified.

6.1.5. How to Read the valgrind Error Report

Here is a sample of output for a test program:

```
01 ==1353== Invalid read of size 4
02 ==1353==  at 0x80484F6: print (valg_eg.c:7)
03 ==1353==  by 0x8048561: main (valg_eg.c:16)
04 ==1353==  by 0x4026D177: __libc_start_main (../libc-start.c   :129)
05 ==1353==  by 0x80483F1: free@@GLIBC_2.0 (in /valg/a.out)
06 ==1353==  Address 0x40C9104C is 0 bytes after a block of size 40 malloc'ed
07 ==1353==  at 0x40046824: malloc (vg_clientfuncs.c:100)
08 ==1353==  by 0x8048524: main (valg_eg.c:12)
09 ==1353==  by 0x4026D177: __libc_start_main (../libc-start.c   :129)
10 ==1353==  by 0x80483F1: free@@GLIBC_2.0 (in /valg/a.out)
```

`1353` is the process ID. The remaining text is described below:

- 02: A read error is at line 7, in the function `print`.
- 03: The function `print` is in the function `main`.
- 04: The function `main` is called by the function `__libc_start_main` at line number 129, in `../libc-start.c`.
- The function `__libc_start_main` is called by `free@@GLIBC_2.0` in the file `/valg/a.out`.

The remaining text describes a `malloc` error.

6.2. Detecting Memory-Handling Problems with mudflap

`mudflap` is a `gcc4` compiler option that detects memory-handling problems. It works looking for unsafe source-level pointer operations. Constructs are replaced with expressions that normally evaluate to the same value, but include parts that refer to `libmudflap`, the `mudflap` runtime. To evaluate memory accesses, the runtime maintains a database of valid memory objects.

Consider this code:

```
int a[10];
int b[10];

int main(void) {
return a[11];
}
```

`valgrind` detects no error; the compiler allocates memory for `a` and `b` consecutively, and an access of `a[11]` will likely read `b[1]`.

However, if you compile the code with:

```
gcc4 -o bug bug.c -g -fmudflap -lmudflap
```

When executed, the program will fail and `mudflap` prints a warning.

`mudflap` can also print a list of memory leaks; just specify the `-print-leaks` option (as described in Section 6.2.1.1 *Runtime options*). However, `mudflap` does not detect reads of uninitialized memory.

`Mudflap` instrumentation and runtime costs extra time and memory. At build time, the compiler needs to process the instrumentation code. When running, it takes time to perform the checks, and memory to represent the object database. The behavior of the application has a strong impact on the run-time slowdown, affecting the lookup cache hit rate, the overall number of checks, and the number of objects tracked in the database, and their rates of change.

The source code of GCC with `mudflap` extensions, and of `libmudflap`, are available by anonymous CVS. See http://gcc.gnu.org/ for instructions.

6.2.1. Using Mudflap

Using `mudflap` is easy. If you have a single-threaded program, you build a `mudflap`-protected program by adding an extra compiler option (`-fmudflap`) to objects to be instrumented, and link with the same option, plus perhaps `-static`. You may run such a program by just starting it as usual.

If you have a multie-threaded program, use instead the `-fmudflapth` compiler option and the `-lmudflapth` library.

If you are interested in maximum performance instrumentation that protects only against erroneous *writes*, add the compiler option `-fmudflapir`. The `ir`, which stands for "ignore reads", causes the compiler to omit checking of pointer reads. This automatically turns on the `-ignore-reads` `libmudflap` option for its suite of intercepted library calls.

In the default configuration, a `mudflap`-protected program will print detailed violation messages to `stderr`. They are tricky to decode at first. Section 6.2.2 *How to Read the libmudflap Error Report* contains a sample message and its explanation.

6.2.1.1. Runtime options

`libmudflap` observes an environment variable `MUDFLAP_OPTIONS` at program startup, and extracts a list of options. Include the string `-help` in that variable to see all the options and their default values (as shown in Section 6.2.2 *How to Read the libmudflap Error Report*).

6.2.1.1.1. Violation Handling

The `-viol-` series of options control what `libmudflap` should do when it determines a violation has occurred. The `-modeseries` controls whether `libmudflap` should be active.

`-viol-abort`

Call abort(), requesting a core dump and exit.

`-viol-gdb`

Create a GNU debugger session on this suspended program. The debugger process may examine program data, but it needs to quit in order for the program to resume.

`-viol-nop`

Do nothing. The program may continue with the erroneous access. This may corrupt its own state, or `libmudflap`'s. (The default is "active".)

`-viol-segv`

Generate a `SIGSEGV`, which a program may opt to catch.

`-mode-check`

Normal checking mode in which `mudflap` checks for memory violations. (The default is "active".)

`-mode-nop`

Disable all main `libmudflap` functions. Since these calls are still tabulated if using `-collect-stats`, but the lookup cache is disabled, this mode is useful to count total number of checked pointer accesses.

`-mode-populate`

Act like every `libmudflap` check succeeds. This mode merely populates the lookup cache but does not actually track any objects. Performance measured with this mode would be a rough upper bound of an instrumented program running an ideal `libmudflap` implementation.

`-mode-violate`

Trigger a violation for every main `libmudflap` call. This is a dual of `-mode-populate`, and is perhaps useful as a debugging aid.

6.2.1.1.2. Extra checking and tracing with mudflap

A variety of options add extra checking and tracing:

`-abbreviate`

Abbreviate repeated detailed printing of the same tracked memory object. (The default is "active".)

`-backtrace=`*`N`*

Save or print *`N`* levels of stack backtrace information for each allocation, deallocation, and violation. Turning this option off (`-backtrace=0`) results in a significant speed increase. (The default is "4".)

`-check-initialization`

Check that memory objects on the heap have been written to before they are read. Section 6.2.2 *How to Read the libmudflap Error Report* explains a violation message due to this check.

`-collect-stats`

Print a collection of statistics at program shutdown. These statistics include the number of calls to the various main `libmudflap` functions, and an assessment of lookup-cache utilization.

`-crumple-zone=`*`N`*

Create extra inaccessible regions of *`N`* bytes before and after each allocated heap region. This is good for finding buggy assumptions of contiguous memory allocation. (The default is "32".)

`-free-queue-length=`*`N`*

Defer an intercepted free for *`N`* rounds, to make sure that immediately following malloc calls will return new memory. This is good for finding bugs in routines manipulating list or tree-like structures. (The default is "4".)

`-internal-checking`

Periodically traverse `libmudflap` internal structures to assert the absence of corruption.

`-persistent-count=`*`N`*

Keep the descriptions of *`N`* recently valid (but now deallocated) objects around, in case a later violation may occur near them. This is useful to help debug use of buffers after they are freed. (The default is "100".)

`-print-leaks`

At program shutdown, print a list of memory objects on the heap that have not been deallocated.

`-sigusr1-report`

Handle signal `SIGUSR1` by printing the same sort of `libmudflap` report that will be printed at shutdown. This is useful for monitoring the `libmudflap` interactions of a long-running program.

`-trace-calls`

Print a line of text to stderr for each `libmudflap` function.

`-verbose-trace`

Add even more tracing of internal `libmudflap` events.

`-verbose-violations`

Print details of each violation, including nearby recently valid objects. (The default is "active".)

`-wipe-heap`

Do the same for heap objects being deallocated.

`-wipe-stack`

Clear each tracked stack object when it goes out of scope. This can be useful as a security or debugging measure.

6.2.1.1.3. mudflap Heuristics

`libmudflap` contains several heuristics that it may use when it suspects a memory access violation. These heuristics are only useful when running a hybrid program that has some uninstrumented parts. Memory regions suspected valid by heuristics are given the special guess storage type in the object database, so they do not interfere with concrete object registrations in the same area.

`-heur-proc-map`

On Linux systems, the special file `/proc/self/map` contains a tabular description of all the virtual memory areas mapped into the running process. This heuristic looks for a matching row that may contain the current access. If this heuristic is enabled, (roughly speaking) `libmudflap` will permit all accesses that the raw operating system kernel would allow (that is, not earn a `SIGSEGV`).

`-heur-start-end`

Permit accesses to the statically linked text/data/bss areas of the program.

`-heur-stack-bound`

Permit accesses within the current stack area. This is useful if uninstrumented functions pass local variable addresses to instrumented functions they call.

`-heur-stdlib`

Register standard library data (argv, errno, stdin, ...). (The default is "active".)

6.2.1.1.4. Tuning mudflap

There are some other parameters available to tune performance-sensitive behaviors of `libmudflap`. Picking better parameters than default is a trial-and-error process and should be undertaken only if `-collect-stats` suggests unreasonably many cache misses, or the application's working set changes much faster or slower than the defaults accommodate.

`-ignore-reads`

Ignore read accesses (that is, assume that they are good).

`-lc-shift=N`

Set the lookup-cache shift value to *`N`*. *`N`* should be just a little smaller than the power-of-2 alignment of the memory objects in the working set.

`-timestamps`

Track object lifetime timestamps (the default is "active". Turning this option off (`-timestamps=0`) results in a some speed increase. (The default is "active".)

6.2.1.1.5. mudflap Introspection

`libmudflap` provides some additional services to applications or developers trying to debug them. Functions listed in the `mf-runtime.h` header may be called from an application, or interactively from within a debugging session.

`__mf_watch`

Given a pointer and a size, `libmudflap` will specially mark all objects overlapping this range. When accessed in the future, a special violation is signaled. This is similar to a GDB watchpoint.

`__mf_unwatch`

Undo the above marking.

`__mf_report`

Print a report just like the one possibly shown at program shutdown or upon receipt of `SIGUSR1`.

`__mf_set_options`

Parse a given string as if it were supplied at startup in the `MUDFLAP_OPTIONS` environment variable, to update `libmudflap` runtime options.

6.2.2. How to Read the libmudflap Error Report

This section looks at a sample `libmudflap` violation message.

```
mudflap violation 3 (check/read): time=1049824033.102085 ptr=080c0cc8 size=1
```

This is the third violation taken by this program. It was attempting to read a single-byte object with base pointer 0x080c0cc8. The timestamp can be decoded as 102 ms after Tue Apr 8 13:47:13 2003 via ctime.

```
pc=08063299 location='nbench1.c:3077 (SetCompBit)'
    nbench [0x8063299]
    nbench [0x8062c59]
    nbench(DoHuffman+0x4aa) [0x806124a]
```

The pointer access occurred at the given PC value in the instrumented program, which is associated with the file `nbench1.c` at line 3077, within function `SetCompBit`. (This does not require debugging data.) The following lines provide a few levels of stack backtrace information, including PC values in square brackets, and sometimes module/function names.

```
Nearby object 1: checked region begins 8B into and ends 8B into
```

There was an object near the accessed region, and in fact the access is entirely within the region, referring to its byte #8.

```
mudflap object 080958b0: name='malloc region'
bounds=[080c0cc0,080c2057] size=5016 area=heap check=1r/0w liveness=1
```

This object was created by the `malloc` wrapper on the heap, and has the given bounds, and size. The check part indicates that it has been read once (this current access), but never written. The liveness part relates to an assessment of how frequently this object has been accessed recently.

```
alloc time=1049824033.100726 pc=4004e482
    libmudflap.so.0(__real_malloc+0x142) [0x4004e482]
    nbench(AllocateMemory+0x33) [0x806a153]
    nbench(DoHuffman+0xd5) [0x8060e75]
```

The allocation moment of this object is described here, by time and stack backtrace. If this object was also deallocated, there would be a similar `dealloc` clause. Its absence means that this object is still alive, or generally legal to access.

```
Nearby object 2: checked region begins 8B into and ends 8B into
mudflap object 080c2080: name='malloc region'
bounds=[080c0cc0,080c2057] size=5016 area=heap check=306146r/1w liveness=4562
```

```
alloc time=1049824022.059740 pc=4004e482
    libmudflap.so.0(__real_malloc+0x142) [0x4004e482]
    nbench(AllocateMemory+0x33) [0x806a153]
    nbench(DoHuffman+0xd5) [0x8060e75]
```

Another nearby object was located by `libmudflap`. This one too was a `malloc` region, and happened to be placed at the exact same address. It was frequently accessed.

```
dealloc time=1049824027.761129 pc=4004e568
    libmudflap.so.0(__real_free+0x88) [0x4004e568]
    nbench(FreeMemory+0xdd) [0x806a41d]
    nbench(DoHuffman+0x654) [0x80613f4]
    nbench [0x8051496]
```

This object was deallocated at the given time, so this object may not be legally accessed anymore.

```
number of nearby objects: 2

This is a GCC "mudflap" memory-checked binary.
Mudflap is Copyright (C) 2002-2003 Free Software Foundation, Inc.
```

No more nearby objects have been found. The conclusion? Some code on line 3077 of `nbench1.c` is reading a heap-allocated block that has not yet been initialized by being written into. This is a situation detected by the `-check-initialization libmudflap` option.

6.3. Best Programming Practices

This appendix provides some suggestions about options you have when you create C/C++ programs. Generally these suggestions enable you to reduce the risk of problems with memory allocation. For more information, see Chapter 6 *Detecting and Preventing Application Memory Problems*.

6.3.1. General Suggestions

6.3.1.1. Limit Resource Usage

Applications that have "runaway processes" are much like a denial-of-service attack. Thus, it is a good practice to limit the amount of resources that applications need while allowing them all the resources they need to operate perfectly. By using the `setrlimit()` interface, you can restrict memory usage, stack size, and the number of file descriptors, file locks, POSIX message queues, and pending signals.

It is difficult to determine usable limits; while you can track calls such as `open()` and `socket()` to determine the largest descriptor, in general you have to monitor the application and its resource usage.

6.3.1.2. Compile with _FORTIFY_SOURCE Defined (RHEL4)

Starting with Red Hat Enterprise Linux 4, you can compile programs with `-D_FORTIFY_SOURCE=2` defined (when optimization is enabled). This macro performs two types of tests:

- Functions that operate on memory blocks are checked, if the size of the memory blocks is known.
- Ignored return values for functions are flagged when this should not happen.

These security checks have little or no cost at runtime.

6.3.1.3. Compile with Warnings Enabled

It is generally a good practice to compile with warnings enabled. To enable warnings in `gcc`, add these compiler options:

```
-Wall -Wextra
```

Optionally, you can choose to have compilation fail when a warning is emitted:

```
-Werror
```

However, some code, such as that from generated sources, cannot be made to compile without errors. In such cases you should edit the `makefile` to disable `-Werror` for the appropriate files.

6.3.1.4. Use dmalloc

The `dmalloc` library contains alternative implementations of the `malloc()` functions. Programs that link to this library perform additional tests at runtime to detect memory leaks.

To use `dmalloc`, include `<dmalloc.h>` as the last header, then link the resulting binary with `libdmalloc` (or, for applications that use threads, `libdmallocth`). At runtime, set the `DMALLOC_OPTIONS` environment variable. To determine the value to use for the environment variable, use the `dmalloc` program. For example, to write the debug output to a file named `LOG`, perform internal checks every 100th call, and perform all possible tests, use:

```
dmalloc -l LOG -i 100 high
```

The program produces a log file that shows the amount of memory used, the number of call made, and the memory leaks. If memory corruption is detected, the program stops immediately.

6.3.2. Notes About C Functions

Use `malloc` carefully

In the following example:

```
struct example *ptr = (struct example *) malloc(sizeof(ptr));
```

the parameter for `malloc` should be `sizeof(*ptr)` instead.

You can use a macro in this situation:

```
#define alloc(type) (type) malloc(sizeof(*(type)));
```

Avoid `char *gets(char *s);`
Avoid `char *getwd(char *buf);`

Never use these functions. `gets()` stores input from standard input into the buffer that `s` points to. Such input could cause a buffer overflow.

`getwd()` is similar; it stores the path of the current directory in the buffer that `buf` points to. Again the input can be arbitrarily large and could cause a buffer overflow.

Usable alternatives that have almost the same interface—but which also pass the length of the buffer—are: `fgets()` and `getcwd()`.

Avoid `realpath()`

This function operate much like `getcwd()`. Instead, use:

```
car *canonicalize_file_name(const char *)
```

The effect is the same as that for `realpath()`, but the result is always returned in a newly allocated buffer.

Avoid `sprintf()` when the maximum buffer size is unknown

If the maximum buffer size is unknown, you would have to call `snprintf()` in a loop where the function is called with ever larger buffers.

Instead, use the `int asprintf(char ** strp, const char *fmt, ...);` or the `int vasprintf(char ** strp, const char *fmt, va_list ap);` function. Note that you must free the buffer after it is no longer needed.

Be careful using `scanf()`

The `%s` and `%[` format specifiers allow strings from uncontrolled sources, which could be arbitrarily long.

You can specify that `scanf()` allocate the memory required by the parsed string (the `a` modifier in this example):

```
int num
char *str
scanf("%d %as", &num, &str);
```

Never use `rand_r()`

The whole state the pseudo-random number generator can have is in the `unsigned int` the parameter points to; these are only 32 bits on Linux platforms.

A reasonable alternative is `random()`.

Note

Other possibilities include `/dev/random` and `/dev/urandom` when you require only small amounts of randomness. `/dev/random` provides high-quality data, but reading numbers from this device will block if there are not enough random numbers left and if there are no events that can be used to generate randomness. Calls to `/dev/urandom` will never block.

Chapter 7.

Application Tuning with OProfile

Performance tuning is one of the most difficult elements of a product development cycle. It requires a broad knowledge of not only the product, but also the product's hardware and software environment. Products having to meet strict performance criteria often interact directly with the hardware (specifically hardware drivers) and the kernel. Unfortunately, as device driver developers have long known, conventional profiling tools do not adequately monitor the environmental aspects of a product's performance.

To address the growing need for environmental performance tuning, Red Hat Enterprise Linux includes OProfile, a powerful, low-overhead system profiler that utilizes processor-hardware performance counters to enable system-wide profiling of a wide variety of CPU events, such as the number of clock cycles that the processor is halted, the number of instruction fetch misses, the number of L2 data loads or stores, and much more. Capable of profiling any application, shared library, kernel, or kernel module without recompilation, it runs on a variety of hardware platforms, including Intel's Pentium and AMD's Athlon processor families.

OProfile includes a kernel module, a daemon that controls the runtime behavior of the kernel module, and command-line tools to configure/run the daemon and to perform data analysis. For more information on using these command-line-only tools, refer to the documentation available with Red Hat Enterprise Linux (in `/usr/share/oprofile-*`) or the OProfile project on sourceforge (http://oprofile.sourceforge.net).

This chapter describes how to determine whether OProfile is installed on the system and how to use OProfile to search for common performance problems in code. In Section 7.8 *Performance Tuning with Eclipse* you will learn how to use the OProfile plug-in that comes with Eclipse (which is part of the Red Hat Developer Suite).

Note

Documentation for OProfile exists in the `oprofile` package; To obtain a listing of the available documentation after installing Red Hat Enterprise Linux, issue the command:

```
rpm -qd oprofile
```

The kernel support for OProfile in Red Hat Enterprise Linux 3 is based on the backported code from the 2.6 kernel. Therefore, if you refer to the OProfile documentation, keep in mind that features listed as being 2.6-specific actually apply to the Red Hat Enterprise Linux kernel, even though the kernel version is 2.4. Likewise, this means that features listed as being specific to the 2.4 kernel do not apply to the Red Hat Enterprise Linux kernel.

7.1. OProfile Installation

OProfile is closely tied to the Linux kernel and the processor architecture. Currently OProfile on Red Hat supports the processors listed in Table 7-1. The `TIMER_INT` is a fallback mechanism for processors without supported performance monitoring hardware. Use of the `TIMER_INT` mechanism may also be seen on machines that have problems with the interrupt handling hardware; for example, laptop computers.

For the Red Hat-supplied kernels only, the Symmetric Multi-Processor (SMP) kernels have OProfile support enabled. The uniprocessor kernels do not have OProfile support enabled because some laptops have problems with OProfile. However, if the SMP kernels runs on the laptop, OProfile can be used.

First, check to see that appropriate Linux kernel and OProfile packages are installed:

```
#checking oprofile installed
rpm -qa |grep oprofile


#checking kernel
rpm -qa |grep kernel
```

To use OProfile, the SMP kernel must be running. Verify an smp kernel is running with:

```
#check which kernel is running
uname -a
```

If an SMP kernel is running and OProfile is installed, you should be able to load the OProfile module and set up the oprofile file system by running the following command as root:

```
opcontrol --init
```

You should be able to verify that the oprofile module is loaded with the `/sbin/lsmod` command.

The oprofile file system should be available as `/dev/oprofile`. The contents of `/dev/oprofile` should look similar to the following:

```
ls /dev/oprofile
0  2  4  6  buffer       buffer_watershed  cpu_type  enable       stats
1  3  5  7  buffer_size  cpu_buffer_size   dump      kernel_only
```

The file `/dev/oprofile/cpu_type` contains a string to indicate the processor type oprofile is going to use. You can run `cat /dev/oprofile/cpu_type` to get the string. Table 7-1 lists the possible strings and the number of events that can be monitored concurrently and the number of numerical directories you should see in `/dev/oprofile`.

Processors	cpu_type	#counters
Pentium Pro	i386/ppro	2
Pentium II	i386/pii	2
Pentium III	i386/piii	2
Pentium 4 (non-HT)	i386/p4	8
Pentium 4 (HT)	i386/p4-ht	4
Athlon	i386/athlon	4
AMD64	x86-64/hammer	4
Itanium	ia64/itanium	4
Itanium 2	ia64/itanium2	4
TIMER_INT	timer	1 (not user settable)
IBM iseries	timer	1 (not user settable)
IBM pseries	timer	1 (not user settable)
IBM s390	timer	1 (not user settable)

Table 7-1. OProfile Processors

After the verifying that the OProfile module is installed and the OProfile file system is mounted, the next step is to configure OProfile to collect data.

7.2. OProfile Configuration

The configuration of oprofile consists of indicating the kernel that is running and what events to monitor. For `TIMER_INT` (/dev/cpu_type "timer"), there is no choice on the events to monitor or the rate. The following will set up the profiling for the `TIMER_INT` event:

```
opcontrol --setup --vmlinux=/boot/vmlinux-`uname -r`
```

The Pentium, Athlon, AMD64, and ia64 processors, which have support for the performance monitoring hardware, can be set up in the same way as the `TIMER_INT` event above. In the default setup, a time-based event set is selected to produce about 2,000 sample per second per processor. Events and sample rates can be set explicitly. The following is an example for the Intel Pentium 4 to get time-based measurements:

```
opcontrol --setup --vmlinux=/boot/vmlinux-`uname -r`
 --ctr0-event=GLOBAL_POWER_EVENTS --ctr0-count=240000 --ctr0-unit-mask=1
```

This command selects the currently running kernel, and the Pentium 4 time-based sampling with a sample generated every 240,000 clock cycles. Smaller values for `--ctr0-count` produce more frequent sampling; larger values produce less frequent sampling. For rarer events, a lower count may be needed to get a sufficient number of samples. The `--ctr0-unit-mask=1` option tells oprofile to count clock cycles only when the processor is unhalted.

Table 7-2 lists processors and time-based metrics for other cpu_types. The `op_help` command lists the events that are available on machine. The `--setup` option causes oprofile to store the configuration information in `/home/root/.oprofile/daemonrc`, but it does not start the daemon.

Processor	-ctr0-event=	--ctr0-unit-mask=
Pentium Pro/PII/PIII	CPU_CLK_UNHALTED	
Pentium 4 (HT and non-HT)	GLOBAL_POWER_EVENTS	1
Athlon/AMD64	CPU_CLK_UNHALTED	
Itanium 2	CPU_CYCLES	
TIMER_INT	(nothing to set)	

Table 7-2. Time-based events

In some cases it is useful to associate the samples for shared libraries with the executable that called the shared library. Adding the `--separate=library` option to the opcontrol setup line will cause the oprofile daemon to prepend the application executable to the name and allow the analysis tools to determine which application executable those samples are associated with.

Once oprofile is configured, the daemon is started with:

```
opcontrol --start
```

The end of the output from the `opcontrol --start` command states the following:

```
Using log file /var/lib/oprofile/oprofiled.log
Profiler running.
```

Check that oprofile daemon is running:

```
ps -aux|grep oprofiled
```

The daemon can be shut down with:

```
opcontrol --shutdown
```

You can also use the GUI program `oprof_start` to control and to configure oprofile.

7.3. Collecting And Analyzing Data

Once the daemon is running, OProfile will collect data on all executables run on the machine. You can run the code that you want to measure. OProfile writes the profiling data to `/var/lib/oprofile/samples` at its convenience. To ensure that all the current profiling data is flushed to the sample files before each analysis of profiling data, you should type in (as root):

```
opcontrol --dump
```

You can check for sample files in `/var/lib/oprofile/samples`. Each executable that has samples should be in that directory. Each sample filename is coded as the path to the executable with "}" replacing "/" and with a "#" followed by the counter number used for that sample file.

For this OProfile tutorial, a FORTRAN Linpack benchmark was downloaded from http://netlib.org/benchmark/1000d. The program was saved as `linpack1000d.f`. The program was compiled with the following command:

```
g77 -g -O3 linpack1000d.f -o linpack1000d
```

To start with a clean slate we can run to save the previously collected samples into `/var/lib/oprofile/samples/junk`:

```
opcontrol --save=junk
```

OProfile automatically collects data on any program that runs while OProfile is operating. Thus, the application is run normally:

```
./linpack1000d
```

7.4. Viewing the Overall System Profile

Once the profiling data is dumped to the sample files, you can use `op_time` to see which executables are being run and analyze the data.

```
op_time -r --counter 0 | more
```

Produces:

```
7310      76.3845  0.0000 /root/linpack1000d
882        9.2163  0.0000 /boot/vmlinux-2.4.21-1.1931.2.317.entsmp
691        7.2205  0.0000 /usr/lib/gcc-lib/i386-redhat-linux/3.2.3/f771
327        3.4169  0.0000 /bin/bash
85         0.8882  0.0000 /usr/bin/gdmgreeter
62         0.6479  0.0000 /usr/bin/ld
24         0.2508  0.0000 /usr/bin/as
...
```

Notice that there are 7310 samples for `/root/linpack1000d`, the application executable. Because `GLOBAL_POWER_EVENT` is being measured by counter 0, shows the relative amount of time the system spent in the various executables. On the Pentium 4 (and other processors), there are restrictions on which counters can measure specific events. Thus, you may need to use a counter other than counter 0 to collect and analyze data. The command `op_help` lists the available events and the suitable registers for them.

7.5. Examining A Single Executable's Profile

In most cases a more detailed analysis of an application executable is desired. This requires debugging information to map the addresses in the program back to the source code. Thus, GCC's `-g` option should be used when producing executables. Note that the GCC option `-g` does not change the executable code, it just adds the mapping information needed by the debugger to map addresses back to line numbers. Most RPMs are distributed without debugging information to save space, so you will not be able to get detailed profiling information from just the RPM. Recently, the Red Hat build system started building debuginfo RPMs for each normal RPM. The OProfile supplied by Red Hat can read the information installed by the debuginfo RPMs to allow better characterization of the executables distributed in RPMs.

For the example `linpack100d` program, you can find where it spends its time:

```
oprofpp --counter 0 -l -r /root/linpack1000d | more
Cpu type: P4 / Xeon
Cpu speed was (MHz estimation) : 2386.61
Counter 0 counted GLOBAL_POWER_EVENTS events
 (time during which processor is not stopped)
 with a unit mask of 0x01
 (count cycles when processor is active)
 count 1193500
vma      samples  %           symbol name
08048fa0 6591     90.1888     daxpy_
08048b68 470      6.43131     dgefa_
08049b64 152      2.07991     ran_
08049ce0 63       0.862069    matgen_
08049ddc 16       0.218938    idamax_
08049330 15       0.205255    dmxpy_
08049254 1        0.0136836   dscal_
...
```

You can see that most of the time is spent in the `daxpy_` subroutine. To get the detailed view of the addresses in the particular function in the executable:

```
oprofpp --counter 0 -s daxpy_  /root/linpack1000d | more
Cpu type: P4 / Xeon
Cpu speed was (MHz estimation) : 2386.61
Counter 0 counted GLOBAL_POWER_EVENTS events
 (time during which processor is not stopped)
 with a unit mask of 0x01
 (count cycles when processor is active)
 count 1193500
vma      samples  %           symbol name
08048fa0 6591     100         daxpy_
 08048fa0 1        0.0151722
 08048fa4 1        0.0151722
 08048fb5 2        0.0303444
 08048fc0 1        0.0151722
 08048fc2 2        0.0303444
 08048fc4 2        0.0303444
 08048fca 1        0.0151722
```

```
 08048fd3 1         0.0151722
 08048fd5 1         0.0151722
 08048fd8 2         0.0303444
 0804902e 1         0.0151722
...
 080490b9 68        1.03171
 080490bb 71        1.07723
 080490be 25        0.379305
 080490c1 1371      20.8011
 080490c4 132       2.00273
 080490c8 72        1.0924
 080490ca 1460      22.1514
 080490cc 143       2.16963
 080490d1 92        1.39584
 080490d4 1408      21.3625
 080490d7 74        1.12274
 080490d9 63        0.955849
 080490dc 15        0.227583
 080490df 1356      20.5735
 080490e2 133       2.0179
 080490e8 64        0.971021
 080490ed 6         0.0910332
```

These samples can be related back to the original source code with the `op_to_source` command:

```
op_to_source --source-dir /root /root/linpack1000d
```

For each file used to build the executable, an equivalent file is annotated with the sample counts for each line. The created file also has information at the end of the file describing the processor and the events that the counters measured. Below is the `daxpy_ subroutine` from the created file. Notice that most of the samples are for the inner loop at the bottom of the function.

```
----daxpy----
      2 0.027% :      subroutine daxpy(n,da,dx,incx,dy,incy)
      /* daxpy_ total:    6591 90.18% */
               :c
               :c     constant times a vector plus a vector.
               :c     uses unrolled loops for increments equal to one.
               :c     jack dongarra, linpack, 3/11/78.
               :c
               :      double precision dx(1),dy(1),da
               :      integer i,incx,incy,ix,iy,m,mp1,n
               :c
               :      if(n.le.0)return
      8 0.109% :      if (da .eq. 0.0d0) return
      4 0.054% :      if(incx.eq.1.and.incy.eq.1)go to 20
               :c
               :c        code for unequal increments or equal
               :c          increments not equal to 1
               :c
               :      ix = 1
               :      iy = 1
               :      if(incx.lt.0)ix = (-n+1)*incx + 1
               :      if(incy.lt.0)iy = (-n+1)*incy + 1
               :      do 10 i = 1,n
               :        dy(iy) = dy(iy) + da*dx(ix)
               :        ix = ix + incx
               :        iy = iy + incy
      3 0.041% :   10 continue
               :      return
               :c
```

```
                :c        code for both increments equal to 1
                :c
                :c
                :c        clean-up loop
                :c
                :   20 m = mod(n,4)
                :      if( m .eq. 0 ) go to 40
      2 0.027% :      do 30 i = 1,m
     13 0.177% :        dy(i) = dy(i) + da*dx(i)
      3 0.041% :   30 continue
                :      if( n .lt. 4 ) return
                :   40 mp1 = m + 1
      3 0.041% :      do 50 i = mp1,n,4
   1535 21.00% :        dy(i) = dy(i) + da*dx(i)
   1664 22.76% :        dy(i + 1) = dy(i + 1) + da*dx(i + 1)
   1643 22.48% :        dy(i + 2) = dy(i + 2) + da*dx(i + 2)
   1508 20.63% :        dy(i + 3) = dy(i + 3) + da*dx(i + 3)
    203 2.777% :   50 continue
                :      return
----daxpy----
```

7.6. Optimizing Code

The general procedure for optimization is to find out where the hotspots in the code are and revise the code to avoid those performance problems. Optimizing rarely-executed code does little to improve the overall performance. For the Intel Pentium Pro, Pentium II, and Pentium III, `CPU_CLOCK_UNHALTED` can be used to get time-based measurements. For the Pentium 4, the `GLOBAL_POWER_EVENTS` provide time-based measurements. The `TIMER_INT` provides time-based sampling using a periodic interrupt on processors without support performance monitoring hardware. You can refer to Table 7-2 for the events to monitor for time-based sampling on various processors. The hotspots can be checked for common performance problems. Both Intel and AMD have produced documents that describe techniques on how to optimize code for their processors in much greater detail.

Here is a short (and incomplete) list of things to look for:

- Section 7.6.1 *Memory References And Data Cache Misses*
- Section 7.6.2 *Unaligned/Partial Memory Accesses*
- Section 7.6.3 *Branch Misprediction*
- Section 7.6.4 *Instruction Cache Misses*

7.6.1. Memory References And Data Cache Misses

Processor speed has increased at a much greater rate than memory-access speed. As a result, each data cache miss is relatively much more expensive on current processors than on older processors. A cache miss that requires access to main memory on a 3GHz Pentium 4 costs on the order of a hundred clock cycles. To get good processor performance, the cache-miss rate needs to be low. Table 7-3 and Table 7-4 show the processor events related to memory references and cache misses respectively.

Processor	event=

Processor	event=
Pentium Pro/PII/PIII	DATA_MEM_REFS
Pentium 4 (HT and non-HT)	--
Athlon/AMD64	DATA_CACHE_ACCESSES
Itanium 2	DATA_REFERENCES_SET0 DATA_REFERENCES_SET1
TIMER_INT	(nothing available)

Table 7-3. Memory reference events

Processor	event=
Pentium Pro/PII/PIII	DCU_MISS_OUTSTANDING
Pentium 4 (HT and non-HT)	BSQ_CACHE_REFERENCE unit-mask=0x10f
Athlon/AMD64	DATA_CACHE_MISSES
Itanium 2	L1D_READ_MISSES
TIMER_INT	(nothing available)

Table 7-4. Data Cache Miss Events

Similarly, you can look for sections in the code where there is significant bus traffic. The goal is to reduce the amount of bus traffic in these problem code sections. Table 7-5 lists events for sampling bus activity.

Processor	event=
Pentium Pro/PII/PIII	BUS_DRDY_CLOCKS unit-mask=0
Pentium 4 (HT and non-HT)	FSB_DATA_ACTIVITY unit-mask=11
Athlon/AMD64	DATA_CACHE_REFILLS_FROM_SYSTEM
Itanium 2	BUS_DATA_CYCLE
TIMER_INT	(nothing available)

Table 7-5. System Bus Traffic

7.6.2. Unaligned/Partial Memory Accesses

Some architectures allow memory access to be unaligned. This can lead to data that spans two cache lines and is more expensive to access. If possible, organize data structures with the elements with the strictest alignment constraints at the beginning of the data structure.

The Pentium 4 has forwarding logic to speed up load/store and store/load sequences. However, for situations where a series of small writes are grouped together to form larger value, for example, writing two 32-bit words to form a 64-bit double, the logic will fail and the stores will have to complete before the load can be started. In a similar manner, a small load extracting data out of a larger store can stall

waiting for the larger store to complete. Thus, using C unions to access parts of larger items can have performance penalties.

7.6.3. Branch Misprediction

Much of the speed of current processors is due to the deep pipelines running at high clock rates. However, when a conditional branch is encountered the processor may be required to wait until a value is available and the next instruction to fetch and process can be determined. Most processors now have branch prediction circuitry allowing the processor to start executing instructions on the expected path before the outcome of the branch instruction is known. If the prediction is correct, a significant improvement in performance is obtained because the pipeline sits idle for less time. However, if the branch prediction is incorrect, the processor must nullify the work done for the incorrect branch. On the Intel Pentium III a branch misprediction costs about 10 clock cycles. On newer processors with deeper pipelines, the cost is even higher.

Table 7-6 shows the performance monitoring events for sampling branches and Table 7-7 shows the performance monitoring events for sampling branch mispredictions. Branches do not occur every clock cycle; it is recommend that you set the branch-count interval to be about 1/5 of what is used time-based sampling, for example, (processor clock rate)/10,000. If the code has good branch prediction behavior (less than 10% of the branches are mispredicted), set the counter interval for branch misprediction to same rate used for branches.

With the exception of the Hyperthreaded Pentium 4, OProfile can monitor both branches and branch mispredicts concurrently. Configure OProfile to sample on both branches and branch mispredictions, then analyze the sample files with `oprofpp` and `op_to_source`. Look for places where there are significantly more branch mispredictions compared to the number of branches in the samples.

Processor	event=
Pentium Pro/PII/PIII	BR_INST_RETIRED
Pentium 4 (HT and non-HT)	RETIRED_BRANCH_TYPE unit-mask=15
Athlon/AMD64	RETIRED_BRANCHES
Itanium 2	BR_MISPRED_DETAIL unit-mask=0
TIMER_INT	(nothing available)

Table 7-6. Branches

Processor	event=
Pentium Pro/PII/PIII	BR_MISS_PRED_RETIRED
Pentium 4 (HT and non-HT)	RETIRED_MISPRED_BRANCH_TYPE unit-mask=15
Athlon/AMD64	RETIRED_BRANCHES_MISPREDICTED
Itanium 2	BR_MISPRED_DETAIL unit-mask=2 (path)
Itanium 2	BR_MISPRED_DETAIL unit-mask=3 (target)
TIMER_INT	(nothing available)

Table 7-7. Branch Mispredictions

When compiling, GCC attempts to order the code to help the static prediction of branches; forward

branches not taken (such as error handling code) and backward branches taken (such as loops). However, GCC's static estimates of branches' relative frequency may be incorrect. GCC can use information collected with the `-fprofile-arcs` option. The `-fprofile-arcs` option instruments the generated code to collect data on how often various branches are taken through the code. The instrumented program is run with typical input. When the program completes execution, it writes the data out to `.da` files in the build directory. The program is compiled a second time with the same options, except replacing the `-fprofile-arcs` with `-fbranchprobabilities`. GCC will use the `.da` files to reorder the basic blocks in functions to help the static prediction. Infrequently used blocks of code are moved to the end function to improve cache performance.

GCC can also eliminate branches (calls) by inlining functions. This occurs when the `-O3` or `-finline-functions` options are used to compile code. Inlining functions can also reduce the overhead of function prologue and epilogue.

7.6.4. Instruction Cache Misses

The instruction-cache miss rate can also influence performance. Excessive cache-miss rates could be found by comparing "Instruction Fetch" (event in Table 7-8) to "Instruction Fetch Miss" (event in Table 7-9). For microbenchmarks this is unlikely to be a problem. However, for large programs this could be a concern.

The Pentium 4 does not have a traditional instruction cache; it has a trace cache that holds decoded micro ops. There is an event that can be used to determine the time that the trace cache is in build mode (decoding a single instructions at a time and placing the micro ops in the trace cache) and the time the trace cache is in deliver mode (providing up to three micro ops per cycle to the execution unit). A unit mask must be specified for the `TC_DELIVER_MODE` to indicate what to measure for the Pentium 4.

Processor	event=
Pentium Pro/PII/PIII	IFU_IFETCH
Pentium 4 (non-HT)	TC_DELIVER_MODE unit-mask=7
Athlon/AMD64	ICACHE_FETCHES
Itanium 2	INST_DISPERSED
TIMER_INT	(nothing available)

Table 7-8. Instruction Fetch

Processor	event=
Pentium Pro/PII/PIII	IFU_IFETCH_MISS
Pentium 4 (non-HT)	TC_DELIVER_MODE unit-mask=0x38
Athlon/AMD64	ICACHE_MISSES
Itanium 2	L1I_FILLS
TIMER_INT	(nothing available)

Table 7-9. Instruction Fetch Miss

7.7. Cautions About OProfile

OProfile is a useful tool, but there are some limitations that you should be aware of when using OProfile:

- Use of shared libraries: Samples for code in shared library are not attributed to the particular application unless the `--separate=library` option is used.
- Performance monitoring samples are inexact: When a performance-monitoring register triggers a sample, the interrupt is not precise like a divide by zero exception. Due to the out-of-order execution of instruction by the processor, the sample is recorded on a nearby instruction.
- `Oprofpp` misattributes inlined functions' samples: `oprofpp` uses a simple address range mechanism to determine which function an address is in. Inline functions samples are not attributed to the inlined function, but rather to the function the inline function was inserted into.
- OProfile accumulates data from runs: OProfile is a system-wide profile and expects processes to start up and shut down multiple times. Thus, samples from multiple runs will accumulate. You can use the command `opcontrol --reset` to clear out the samples.
- Non-CPU-limited performance problems: OProfile is oriented to finding problems with CPU-limited processes. OProfile does not identify processes that are asleep because they are waiting on locks or for some other event to occur, such as waiting for an I/O device to finish an operation.

7.8. Performance Tuning with Eclipse

For developers who want to use an integrated development environment, Red Hat Developer Suite includes the OProfile plug-in, which provides graphical OProfile tools that integrate control, configuration, and analysis into the Eclipse workbench.

The OProfile plug-in consist of two main functional areas: the configuration and launch of the OProfile daemon and the analysis of the collected data. These tasks are accomplished within the Eclipse workbench via common Eclipse workflows. Experienced Eclipse users will have little trouble using the OProfile plug-ins.

7.8.1. Configuring the OProfile Daemon with the Launch Manager

Note

In order to use the OProfile plug-ins, development machines *must* have OProfile support enabled for the kernel. Without rebuilding a kernel, it suffices to run the SMP kernel (even on UP machines), which already contains support for OProfile.

The Eclipse launch manager controls the "launching" of the OProfile daemon. In command-line terms, the launch manager stores the command-line arguments that you want to use to start OProfile; you can store different launch profiles under different names.

The launcher consists of several panes that enable you to specify generic daemon-configuration options, counter configuration, and an optional Eclipse CDT launch profile to run when the daemon is started. To access the Eclipse launch manager, click **Run** > **Run...** The Eclipse launch manager appears:

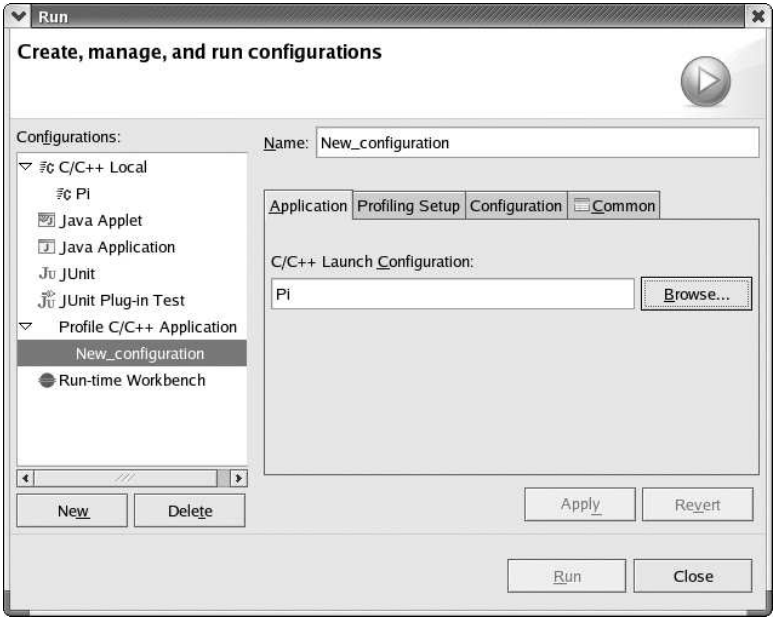

Figure 7-1. The Launch Manager

To facilitate fine-grained application tuning, the Application Configuration pane enables you to specify an Eclipse C/C++ launcher to run when the daemon starts collecting data.

You specify generic OProfile daemon runtime options via the **Profiling Setup** pane. You can enable verbose logging, per-application shared library profiling (which associates shared library samples with the image running the library), and per-application kernel sample files (which associates kernel samples with the image running code in the kernel).

The launcher's **Configuration** pane lists the available counters in the system. When you select a counter, the panel lists all events that can be collected, a description of the event, and a table from which you can choose event-specific options (unit masks). The counter can also be configured to profile the kernel, user binaries, or both. The events available on any counter are dictated by the type of processor in the system.

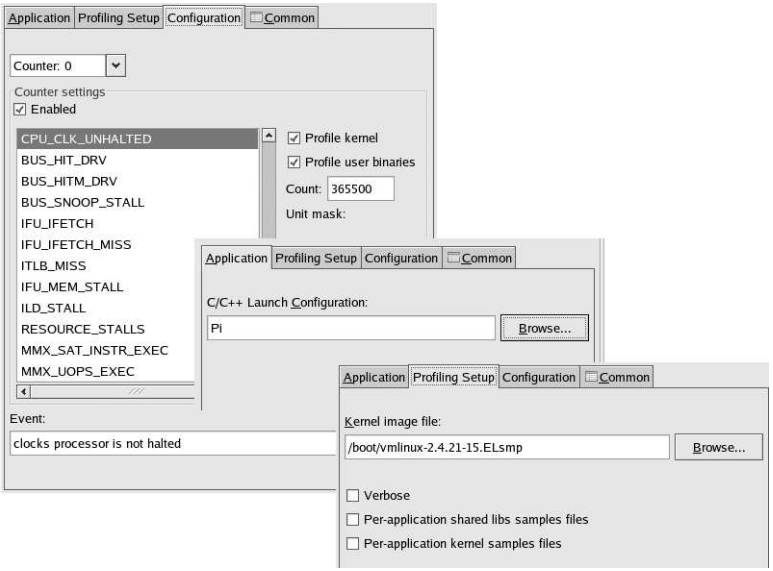

Figure 7-2. OProfile Launch Manager Configuration Panes

For more information, see the Event Type Reference available in the documentation area on the OProfile homepage (http://oprofile.sourceforge.net/docs).

7.8.2. Data Analysis with the OProfile Perspective

When data collection is complete (or even while it is still running), data analysis can begin. You use the **Profiling** perspective and its views to navigate and view the collected data. There are three views in the perspective:

The **System Profile** view is the primary user interface for the **Profiling** perspective. It lists all of the profiling sessions (collections of samples) for any given collected event. These sessions list the collected sample information in the form of a hierarchical tree, which lists the containing images (executables, shared libraries, kernel, and kernel modules) and symbol information (derived from the executable's minimal symbol tables, not from its debugging information).

Clicking on an element in the **System Profile** view causes specific sample data to be displayed in the **Sample** view. The data displayed varies depending on which element was selected in the **System Profile** view.

For example, selecting a session in the view will display collection statistics for the entire session in the **Sample** view: a list of images, including shared libraries and kernel modules (if the daemon was appropriately configured), sorted by the number of total samples collected in the image.

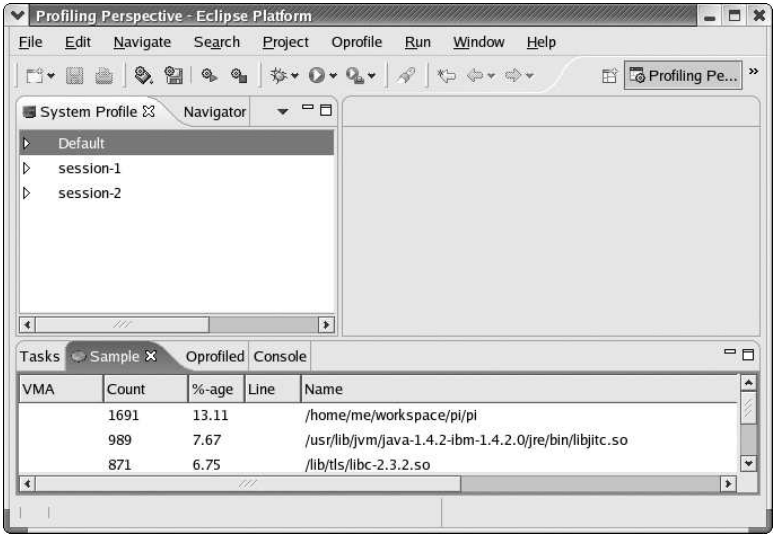

Figure 7-3. Viewing a session

Clicking on an image such as `/home/me/workspace/pi/pi` in the **System Profile** view causes the **Sample** view to display a summary of samples collected in the image: how often and in which symbols the event collection occurred (also sorted by total sample count).

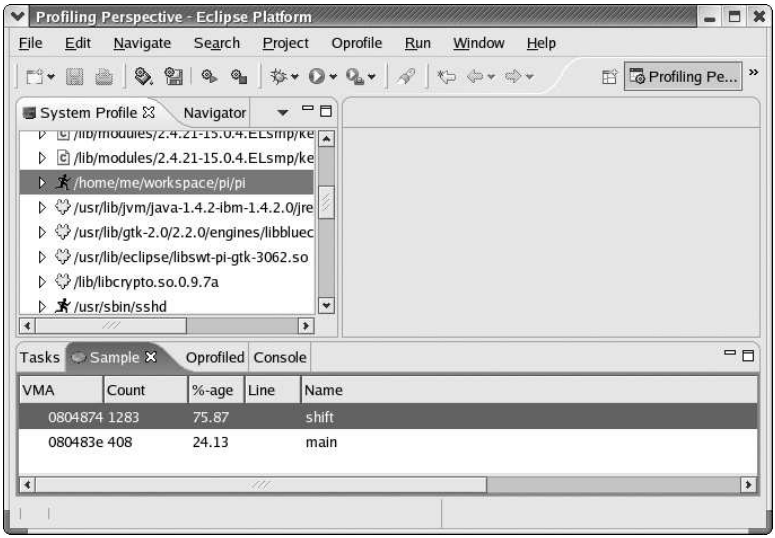

Figure 7-4. Viewing an image file

If the image was compiled with debugging information, the **Sample** view will also list the symbol's source file and line number, which can be opened in an editor with the click of a mouse button.

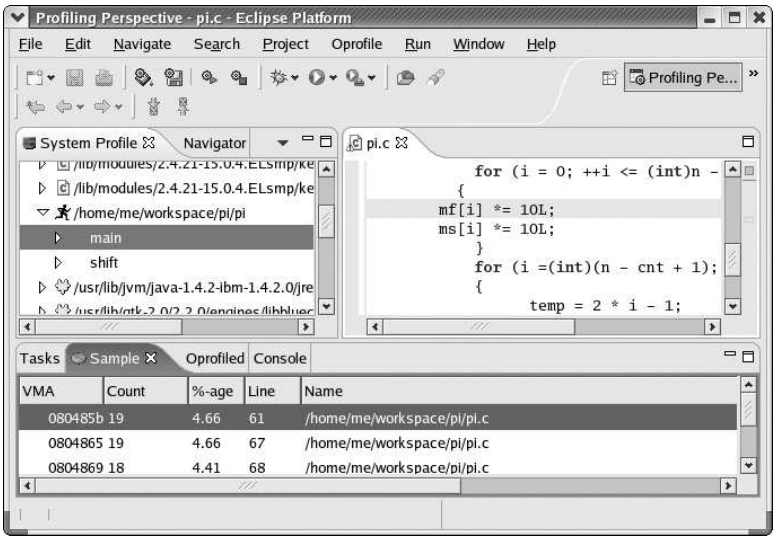

Figure 7-5. Viewing a symbol and its source

Finally, the **OProfile** view displays the daemon's log file so that you can monitor the daemon for runtime status and errors without disrupting your workflow by opening a shell to view it.

redhat

Chapter 8.

Analyzing Java Application Performance Problems Using the IBM JVM

Debugging problems on a production server requires basic debugging capabilities traditionally provided by GDB, but sadly lacking in the Java world. However, you can use the IBM JVM diagnostics to diagnose the problem.

8.1. Problem Scenario

In this scenario the following problems have been observed:

- A Web application running in a production environment is having periodic OutOfMemory and extreme CPU load problems.
- The server is not responding to HTTP requests.
- Running `ps -ef` shows a couple of Java processes (out of several hundred processes in total) consuming 99% of CPU time.
- The problem reoccurs with a frequency of between 6 and 24 hours.
- There is, as yet, no known procedure to reproduce the problem on demand.

These are some questions to answer:

- Which Java threads correspond to each UNIX process?
- Can you get a stack trace for every Java thread?
- Which locks are held? Which are being waited on?
- What objects are allocated?

8.2. Diagnostic Tools

The IBM JVM comes with a number of custom, built-in tools for collecting the information required to answer these questions. Unlike the standard tools in the Java Platform Debugging Architecture, these IBM-specific facilities impose near zero performance overhead on the JVM, so they can be left permanently enabled on production systems. What is more, these tools are free, quick, and easy to use.

There is extensive documentation available from the IBM JVM diagnosis documentation site (http://www-106.ibm.com/developerworks/java/jdk/diagnosis/index.html). What follows is a brief overview of some of the details.

8.2.1. Javadump

The `javadump` tool (which is also known as `Javacore`), produces text files that contain diagnostic information related to the JVM and the application running within. This facility is enabled by default. A `javadump` can be manually triggered by sending `SIGQUIT` to the root process of the JVM. This does not stop the JVM, but it creates a dump file in the first of the following locations that is writable:

1. The directory specified by the IBM_JAVACOREDIR environment variable.

2. The JVM's current working directory.
3. The directory specified by the TMPDIR environment variable.
4. The `/tmp` directory.

A dump file will be generated automatically if a native exception occurs in the JVM. If the IBM_JAVADUMP_OUTOFMEMORY environment variable is set, then a dump will also be generated when the heapspace is exhausted. The file name will be of the form `javacore_[pid].[time].txt`

Some of the information included in the dump file includes:

- The OS memory information (for example, from `/proc/meminfo`)
- User resource limits
- Signal handlers
- Environment variables
- OS memory maps (for example, from `/proc/[pid]/maps`)
- A list of all object monitors being waited on
- A list of all system locks (and what thread, if any, is holding them)
- A list of all threads, their Java names and their UNIX process IDs.
- A Java stack trace for each thread
- A native OS stack trace for each thread
- A list of all classloaders
- A list of all loaded classes.

This is all output in a well-formatted text files, which makes for fairly easy human reading and trivial Perl scripting.

8.2.2. Heapdump

The `heapdump` tool produces text files that contain a dump of all objects currently allocated on the heap and a list of references between objects. This facility is *disabled* by default, but can be enabled by setting the environment variable IBM_HEAP_DUMP=true prior to starting the JVM. When enabled, a `heapdump` is generated at the same time as the `javadump`, for example when you send a `SIG_QUIT` signal.

The filename will be of the form `heapdump_[pid].[time].txt`. While the file is in plain text, the format of this file is not directly interesting to humans; however, when it is processed with the tool `heaproots`, interesting information appears:

- Objects table: a list of objects in the heap sorted by size, address, name, subtree size, and so on.
- Types table: a list of classes with objects in the heap sorted by total size, instance count, or name.
- Gaps table: a list of free spaces in the heap.
- Object info: a list of references to and from an object.
- Paths/routes: finds a route between two objects (if one exists).

8.2.3. JFormat

The `JFormat` tool is a TUI/GUI for extracting and displaying information from JVM core dumps. To enable core dumps, there are two things to do before starting the JVM:

1. Remove any resource limits by running:
   `ulimit -c unlimited`

2. Change to a writable directory.

Now to generate a core dump, simply send the root process a `SIGABRT` signal.

Because the core dump is operating-system specific, you must now process it with the `jextract` tool to create a platform-independent core dump.

```
$JAVA_HOME/jre/bin/jextract path-to-native-core-dump
```

Launch the `JFormat` GUI:

```
$JAVA_HOME/bin/jformat -g
```

The `JFormat` GUI provides menus to extract many kinds of useful information. You will notice that there is a lot of overlap with the `javadump` and `heapdump` metrics.

8.3. Diagnosing the Problem

Having set the IBM_HEAPDUMP=true environment variable and restarted the servers, wait for the performance slowdown to reoccur (which indicates that the garbage collector is starting to thrash). When it does, send a `SIGQUIT` to the root process, which generates a `heapdump` file about 850 MB in size. Then load this into the **HeapRoots** analysis program:

```
java -jar /tmp/HR205.jar heapdump3632.1083842645.txt
```

After parsing the file, `heaproots` shows a few basic statistics about the dump:

```
# Objects            : 7,727,411
# Refs               : 12,807,015
# Unresolved refs    : 31
Heap usage           : 315,413,344
```

The first thing to do is to invoke the `process` command. This resolves all references between the objects, detects roots, and calculates total size metrics per class and sub-tree.

```
Pure roots           : 174,088
... which reach      : 7,706,600
Artificial roots     : 1,531
Softlinks followed   : 16,014
```

Now, you are ready to answer some interesting queries. Look to see what the biggest tree roots are. This was done by running the `dt` command (Dump from roots, sort by total size). The command prompts for some values such as the maximum depth to print out and the minimum size threshold, then it produces something similar to this:

```
<0> [270,167,808] 0x100a2ef8 [72] java/lang/Thread
<1>   [270,167,680] 0x10053c70 [168]
        com/caucho/java/CompilingClassLoader
<2>     [270,164,496] 0x10053d18 [168]
          com/caucho/util/DirectoryClassLoader
          - 14 children of 0x10053d18 too deep.
<2>     {270,167,680} 0x10053c70 [168]
          com/caucho/java/CompilingClassLoader
<2>     <270,163,200 parent:0x10053d18> 0x10053dc0 [168]
          com/caucho/java/CompilingClassLoader
<2>     <270,163,200 parent:0x10053d18> 0x10053dc0 [168]
          com/caucho/java/CompilingClassLoader
```

```
        - 13 children of 0x10053c70 filtered.
      - 4 children of 0x100a2ef8 filtered.
<0> [5,234,712] 0x16a85b90 [447,296] array of java/lang/Object
      - 111820 children of 0x16a85b90 filtered.
```

Unsurprisingly, the root with highest memory consumption is the top level thread of the JVM. This implies the memory leak is in the application code, so look for the biggest object in a package whose name began with the name of the application provider (`com/arsdigita/` in this example). To do this, use the "Objects dump, sort by total size" command (`ot com/arsdigita/`). Again this prompts for some more optional filters, but accepting the defaults produces a list of large objects:

```
  Addr       Size  Root-owner Parent     Total-size  Name
-------------------------------------------------------------------------
N 0x284ffb48 256   0x100a2ef8 0x284ffc88 224,107,816
    class com/arsdigita/templating/Templating
N 0x220d3458 96    0x100a2ef8 0x22bc0ee8 4,585,520
    com/arsdigita/london/theme/util/ThemeDevelopmentFileManager
N 0x1cbc24b0 32    0x100a2ef8 0x1e18a5a8 2,460,464
    com/arsdigita/templating/XSLTemplate
N 0x23180b28 80    0x100a2ef8 0x22bc0ee8 2,164,688
    com/arsdigita/london/search/RemoteSearcher
N 0x1bf215e0 56    0x100a2ef8 0x1bf2a850 1,844,152
    com/arsdigita/persistence/Session
N 0x1bf2ab98 40    0x100a2ef8 0x1bf215e0 1,770,096
    com/arsdigita/persistence/TransactionContext
N 0x1affda20 16    0x100a2ef8 0x1aa4d388 1,681,056
    com/arsdigita/persistence/DataQueryImpl$UnaliasMapper
N 0x1affc508 96    0x100a2ef8 0x1affda20 1,681,040
    com/arsdigita/persistence/DataAssociationCursorImpl
N 0x1a985ff0 56    0x100a2ef8 0x1a98e8e8 1,138,696
    com/arsdigita/bebop/PageState
```

As you can see, the class object for `com.arsdigita.templating.Templating` is taking up 224 MB of the heap. A quick look at the code shows it has a static HashMap storing XSLTemplate objects with no size bounding. To confirm this is the problem, got back to the "Dump from roots, sort by total size" command, this time starting the display from the object at address 0x284ffb48:

```
<0> [224,107,816] 0x284ffb48 [256] class
      com/arsdigita/templating/Templating
<1>   [224,092,408] 0x28500840 [56] java/util/HashMap
<2>     [224,092,352] 0x1cdf9160 [1,552] array of
          java/util/HashMap$Entry
<3>       [4,288,800] 0x2dbfa500 [32] java/util/HashMap$Entry
<4>         [3,216,568] 0x1bf5cc00 [32] java/util/HashMap$Entry
              - 3 children of 0x1bf5cc00 too deep.
<4>         [1,069,960] 0x2c824ca8 [32]
              com/arsdigita/templating/XSLTemplate
              - 4 children of 0x2c824ca8 too deep.
            - 1 child of 0x2dbfa500 filtered.
<3>       [3,216,584] 0x263c8ff8 [32] java/util/HashMap$Entry
<4>         [2,144,416] 0x3d00ade8 [32] java/util/HashMap$Entry
              - 3 children of 0x3d00ade8 too deep.
<4>         [1,069,912] 0x237796b0 [32]
              com/arsdigita/templating/XSLTemplate
              - 4 children of 0x237796b0 too deep.
            - 1 child of 0x263c8ff8 filtered.
<3>       [3,216,568] 0x19319d58 [32] java/util/HashMap$Entry
<4>         [2,144,400] 0x33ddd918 [32] java/util/HashMap$Entry
              - 3 children of 0x33ddd918 too deep.
<4>         [1,069,912] 0x1864f950 [32]
              com/arsdigita/templating/XSLTemplate
```

```
              - 4 children of 0x1864f950 too deep.

...

<1>   {224,107,816} 0x284ffb48 [256]
      class com/arsdigita/templating/Templating
      - 15 children of 0x284ffb48 filtered.
There were 362 objects expanded.
```

From this it is clear that the problem is a huge HashMap in the `Templating` class containing around one hundred XSLTemplate objects, each around 1 MB in size.

8.4. Conclusions

If you implement a static cache in a server, make sure that you:

1. Benchmark the average size of objects in the cache.
2. Decide how much memory can be set aside to it.
3. Use the results from the above steps to calculate a good maximum entry count for the cache before invoking a purge algorithm.

redhat

Appendix A.

PerformanceAnalysisScript

To learn how to use this script, see Section 2.1 *Benchmarking Utilities*. This script is available online at http://people.redhat.com/mbehm/.

```
#!/bin/sh
# perf1.sh --
#
# Copyright 2004 Red Hat Inc., Durham, North Carolina. All Rights Reserved.
#
# Permission is hereby granted, free of charge, to any person obtaining
# a copy of this software and associated documentation files (the
# "Software"), to deal in the Software without restriction, including
# without limitation on the rights to use, copy, modify, merge,
# publish, distribute, sublicense, and/or sell copies of the Software,
# and to permit persons to whom the Software is furnished to do so,
# subject to the following conditions:
#
# The above copyright notice and this permission notice (including the
# next paragraph) shall be included in all copies or substantial
# portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
# EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NON-INFRINGEMENT.  IN NO EVENT SHALL RED HAT AND/OR THEIR SUPPLIERS
# BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
# ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
# CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
#

if test x"$1" == x""; then
  echo "Usage: ./perf1.sh <logfile-name>"
  exit 1
fi

FILE=$1
ITER=10

if test -e $FILE; then
  echo "WARNING: $FILE exists -- will NOT overwrite"
  exit 1
fi

function log () {
  d=`date`
  echo "$d: $1"
  echo "================================================" >> $FILE
  echo "$d: $1"                                           >> $FILE
  if test x"$2" != x""; then
    shift
  fi
  $* >> $FILE 2>&1
}

function logi () {
  d=`date`
```

```
  printf "$d: %-20.20s (PASS $1 of $ITER)\n" $2
  echo "==================================================" >> $FILE
  printf "$d: %-20.20s (PASS $1 of $ITER)\n" $2             >> $FILE
  shift
  if test x"$2" != x""; then
    shift
  fi
  $* >> $FILE 2>&1
}

function ccc () {
  log $1 cat $1
}

function ccci () {
  logi $1 $2 cat $2
}

function note () {
  echo "`date`: (NOTE: $*)"
}

function banner () {
  d=`date`
  echo "================================================="
  echo "===== $d: $* ====="
  echo "================================================" >> $FILE
  echo "===== $d: $* ====="                               >> $FILE
}

banner "Start of Testing ($FILE)"

banner General System Information
log uname uname -a
log free
log df    df -h
log mount
log lsmod
log lspci lspci -v
log dmidecode
log route route -n
log ifconfig
log "ip rule ls" ip rule ls
log "ip route ls" ip route ls
log iptables "iptables -L -n -v"
log sysctl sysctl -a
ccc /proc/cpuinfo
ccc /proc/meminfo
ccc /proc/net/dev
ccc /proc/interrupts
ccc /proc/devices
ccc /proc/cmdline
ccc /proc/scsi/scsi
ccc /etc/modules.conf
ccc /var/log/dmesg

banner Performance Snapshot
log ps ps auxwww
log sar sar -A

let t="10*$ITER"
note "The following takes about $t seconds"
```

```
log "vmstat" vmstat $ITER 10

note "The following takes about $t seconds"
log "iostat" iostat -k $ITER 10

note "Each pass takes 10 seconds"
for i in `seq 1 $ITER`; do
  note "**** PASS $i of $ITER"
  logi $i uptime
  logi $i free
  ccci $i /proc/interrupts
  ccci $i /proc/stat
  logi $i ifconfig ifconfig -a
  sleep 10
done

banner "End of Testing ($FILE)"
```

redhat

Appendix B.

Additional Resources

Not all performance tuning information that you can find on the Internet is effective or even safe. However, you may find the following resources to be useful.

B.1. General Linux Tuning

See http://www.redbooks.ibm.com/abstracts/redp3861.html?Open for *Tuning Red Hat Enterprise Linux on xSeries Servers*.

See http://perso.wanadoo.fr/sebastien.godard/ for information on the `sysstat` collection of performance monitoring utilities.

See http://www.oreilly.com/catalog/linuxkernel2/index.html for *Understanding the Linux Kernel*.

See http://www.kegel.com for the "c10k problem" page in particular, but the entire site has useful tuning information.

See http://linuxperf.nl.linux.org/ for a very good Linux-specific system tuning page (in Dutch).

B.2. Shared Memory/Virtual Memory

See http://ps-ax.com/shared-mem.html for *Tuning/tuneable kernel parameters*.

See http://surriel.com/lectures/ols2003/ for *Towards an O(1) VM*.

B.3. NFS Tuning

See http://nfs.sourceforge.net/nfs-howto/performance.html for information on tuning Linux kernel (NFS in particular), and Linux, network, and disk I/O in general.

B.4. TCP Tuning

See http://www.psc.edu/networking/perf_tune.html#Linux for a summary of tcp tuning information.

See http://www.web100.org/ for the Web100 project, which will provide the software and tools necessary for end-hosts to automatically and transparently achieve high-bandwidth data rates (100 Mbps) over high-performance research networks.

B.5. mod_perl Tuning

See http://perl.apache.org/guide/performance.html for the mod_perl performance-tuning documentation.

B.6. Application Tuning

See http://people.redhat.com/drepper/optimtut1.ps.gz for *Optimizing with gcc and glibc*, a tutorial on optimizing applications by using gcc and glibc features. It is meant for somewhat experienced programmers.

See http://people.redhat.com/drepper/optimtut2.ps.gz for *Application Optimization on Linux*, a hands-on tutorial that is similar to *Optimizing with gcc and glibc*.

B.7. Samba Tuning Resources

See http://k12linux.mesd.k12.or.us/using_samba/appb_02.html for one of the better resources for tuning Samba. It is from the *Using Samba* book from O'Reilly.

B.8. mod_proxy Tuning

See http://www.webtechniques.com/archives/1998/05/engelschall for an article on using `mod_proxy`.

See http://httpd.apache.org/docs/mod/mod_proxy.html for the `mod_proxy` home page.

See http://www.zope.org/Members/rbeer/caching for a description of using mod_proxy with Zope.

See http://www.squid-cache.org/ for the Squid home page.

B.9. OProfile Resources

- *AMD Athlon Processor X86 Code Optimization Guide*, February 2002, AMD Publication 22007. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf
- GCC homepage, July 2003. http://gcc.gnu.org/
- *IA-32 Intel® Architecture Optimization Reference Manual*, 2003, Intel Order number 248966-09. ftp://download.intel.com/design/Pentium4/manuals/24896611.pdf For more information, see: http://www.intel.com/design/Pentium4/documentation.htm
- *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, June 2002, Intel Document 251110-002. http://www.intel.com/design/itanium2/manuals/251110.htm
- OProfile, http://oprofile.sourceforge.net/

Index

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z