Red Hat Enterprise Linux 6 Performance Tuning Guide

Optimizing subsystem throughput in Red Hat Enterprise Linux 6

Red Hat Subject Matter Experts

Red Hat Enterprise Linux 6 Performance Tuning Guide
Optimizing subsystem throughput in Red Hat Enterprise Linux 6
Edition 1.0

Author	Red Hat Subject Matter Experts
Editor	Don Domingo
Editor	Laura Bailey

The *Performance Tuning Guide* describes how to optimize the performance of a system running Red Hat Enterprise Linux 6. It also documents performance-related upgrades in Red Hat Enterprise Linux 6.

While this guide contains procedures that are field-tested and proven, Red Hat recommends that you properly test all planned configurations in a testing environment before applying it to a production environment. You should also back up all your data and pre-tuning configurations.

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file `my_next_bestselling_novel` in your current working directory, enter the `cat my_next_bestselling_novel` command at the shell prompt and press `Enter` to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press `Enter` to execute the command.

> Press `Ctrl`+`Alt`+`F2` to switch to the first virtual terminal. Press `Ctrl`+`Alt`+`F1` to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in `mono-spaced bold`. For example:

> File-related classes include `filesystem` for file systems, `file` for files, and `dir` for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click

---

[1] https://fedorahosted.org/liberation-fonts/

> **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or *Proportional Bold Italic*

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **ssh** *username@domain.name* at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.
>
> The **mount -o remount** *file-system* command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.
>
> To see the version of a currently installed package, use the **rpm -q** *package* command. It will return a result as follows: *package-version-release*.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books          Desktop    documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads        images  notes  scripts  svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;
```

```
public class ExClient
{
   public static void main(String args[])
      throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object        ref    = iniCtx.lookup("EchoBean");
      EchoHome      home   = (EchoHome) ref;
      Echo          echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at *http://access.redhat.com*. Through the customer portal, you can:

• search or browse through a knowledgebase of technical support articles about Red Hat products.

• submit a support case to Red Hat Global Support Services (GSS).

- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at *https://www.redhat.com/mailman/listinfo*. Click on the name of any mailing list to subscribe to that list or to access the list archives.

2.2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: *http://bugzilla.redhat.com/* against the product **Red Hat Enterprise Linux 6.**

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

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.

Overview

The *Performance Tuning Guide* is a comprehensive reference on the configuration and optimization of Red Hat Enterprise Linux. While this release also contains information on Red Hat Enterprise Linux 5 performance capabilities, all instructions supplied herein are specific to Red Hat Enterprise Linux 6.

This book is divided into chapters discussing specific subsystems in Red Hat Enterprise Linux. The *Performance Tuning Guide* focuses on three major themes per subsystem:

Features

> Each subsystem chapter describes performance features unique to (or implemented differently) in Red Hat Enterprise Linux 6. These chapters also discuss Red Hat Enterprise Linux 6 updates that significantly improved the performance of specific subsystems over Red Hat Enterprise Linux 5.

Analysis

> The book also enumerates performance indicators for each specific subsystem. Typical values for these indicators are described in the context of specific services, helping you understand their significance in real-world, production systems.

> In addition, the *Performance Tuning Guide* also shows different ways of retrieving performance data (i.e. profiling) for a subsystem. Note that some of the profiling tools showcased here are documented elsewhere with more detail.

Configuration

> Perhaps the most important information in this book are instructions on how to adjust the performance of a specific subsystem in Red Hat Enterprise Linux 6. The *Performance Tuning Guide* explains how to fine-tune a Red Hat Enterprise Linux 6 subsystem for specific services.

Keep in mind that tweaking a specific subsystem's performance may affect the performance of another, sometimes adversely. The default configuration of Red Hat Enterprise Linux 6 is optimal for *most* services running under *moderate* loads.

The procedures enumerated in the *Performance Tuning Guide* were tested extensively by Red Hat engineers in both lab and field. However, Red Hat recommends that you properly test all planned configurations in a secure testing environment before applying it to your production servers. You should also back up all data and configuration information before you start tuning your system.

1.1. Audience

This book is suitable for two types of readers:

System/Business Analyst

> This book enumerates and explains Red Hat Enterprise Linux 6 performance features at a high level, providing enough information on how subsystems perform for specific workloads (both by default and when optimized). The level of detail used in describing Red Hat Enterprise Linux 6 performance features helps potential customers and sales engineers understand the suitability of this platform in providing resource-intensive services at an acceptable level.

> The *Performance Tuning Guide* also provides links to more detailed documentation on each feature whenever possible. At that detail level, readers can understand these performance features enough to form a high-level strategy in deploying and optimizing Red Hat Enterprise Linux 6. This allows readers to both develop *and* evaluate infrastructure proposals.

> This feature-focused level of documentation is suitable for readers with a high-level understanding of Linux subsystems and enterprise-level networks.

System Administrator

> The procedures enumerated in this book are suitable for system administrators with RHCE [1] skill level (or its equivalent, that is, 3-5 years experience in deploying and managing Linux). The *Performance Tuning Guide* aims to provide as much detail as possible about the effects of each configuration; this means describing any performance trade-offs that may occur.

> The underlying skill in performance tuning lies not in knowing how to analyze and tune a subsystem. Rather, a system administrator adept at performance tuning knows how to balance and optimize a Red Hat Enterprise Linux 6 system *for a specific purpose*. This means *also* knowing which trade-offs and performance penalties are acceptable when attempting to implement a configuration designed to boost a specific subsystem's performance.

1.2. Horizontal Scalability

Red Hat's efforts in improving the performance of Red Hat Enterprise Linux 6 focus on *scalability*. Performance-boosting features are evaluated primarily based on how they affect the platform's performance in different areas of the workload spectrum — that is, from the lonely web server to the server farm mainframe.

Focusing on scalability allows Red Hat Enterprise Linux to maintain its versatility for different types of workloads and purposes. At the same time, this means that as your business grows and your workload scales up, re-configuring your server environment is less prohibitive (in terms of cost and man-hours) and more intuitive.

Red Hat makes improvements to Red Hat Enterprise Linux for both *horizontal scalability* and *vertical scalability*; however, horizontal scalability is the more generally applicable use case. The idea behind horizontal scalability is to use multiple *standard computers* to distribute heavy workloads in order to improve performance and reliability.

In a typical server farm, these standard computers come in the form of 1U rack-mounted servers and blade servers. Each standard computer may be as small as a simple two-socket system, although some server farms use large systems with more sockets. Some enterprise-grade networks mix large and small systems; in such cases, the large systems are high performance servers (for example, database servers) and the small ones are dedicated application servers (for example, web or mail servers).

This type of scalability simplifies the growth of your IT infrastructure: a medium-sized business with an appropriate load might only need two pizza box servers to suit all their needs. As the business hires more people, expands its operations, increases its sales volumes and so forth, its IT requirements increase in both volume and complexity. Horizontal scalability allows IT to simply deploy additional machines with (mostly) identical configurations as their predecessors.

To summarize, horizontal scalability adds a layer of abstraction that simplifies system hardware administration. By developing the Red Hat Enterprise Linux platform to scale horizontally, increasing the capacity and performance of IT services can be as simple as adding new, easily configured machines.

1.2.1. Parallel Computing

Users benefit from Red Hat Enterprise Linux's horizontal scalability not just because it simplifies system hardware administration; but also because horizontal scalability is a suitable development philosophy given the current trends in hardware advancement.

---

[1] Red Hat Certified Engineer. For more information, refer to *http://www.redhat.com/certification/rhce/*.

Consider this: most complex enterprise applications have thousands of tasks that must be performed simultaneously, with different coordination methods between tasks. While early computers had a single-core processor to juggle all these tasks, virtually all processors available today have multiple cores. Effectively, modern computers put multiple cores in a single socket, making even single-socket desktops or laptops multi-processor systems.

As of 2010, standard Intel and AMD processors were available with two to sixteen cores. Such processors are prevalent in pizza box or blade servers, which can now contain as many as 40 cores. These low-cost, high-performance systems bring large system capabilities and characteristics into the mainstream.

To achieve the best performance and utilization of a system, each core must be kept busy. This means that 32 separate tasks must be running to take advantage of a 32-core blade server. If a blade chassis contains ten of these 32-core blades, then the entire setup can process a minimum of 320 tasks simultaneously. If these tasks are part of a single job, they must be coordinated.

Red Hat Enterprise Linux was developed to adapt well to hardware development trends and ensure that businesses can fully benefit from them. *Section 1.3, "Distributed Systems"* explores the technologies that enable Red Hat Enterprise Linux's horizontal scalability in greater detail.

1.3. Distributed Systems

To fully realize horizontal scalability, Red Hat Enterprise Linux uses many components of *distributed computing*. The technologies that make up distributed computing are divided into three layers:

Communication
> Horizontal scalability requires many tasks to be performed simultaneously (in parallel). As such, these tasks must have *interprocess communication* to coordinate their work. Further, a platform with horizontal scalability should be able to share tasks across multiple systems.

Storage
> Storage via local disks is not sufficient in addressing the requirements of horizontal scalability. Some form of distributed or shared storage is needed, one with a layer of abstraction that allows a single storage volume's capacity to grow seamlessly with the addition of new storage hardware.

Management
> The most important duty in distributed computing is the *management* layer. This management layer coordinates all software and hardware components, efficiently managing communication, storage, and the usage of shared resources.

The following sections describe the technologies within each layer in more detail.

1.3.1. Communication

The communication layer ensures the transport of data, and is composed of two parts:

• Hardware

• Software

The simplest (and fastest) way for multiple systems to communicate is through *shared memory*. This entails the usage of familiar memory read/write operations; shared memory has the high bandwidth, low latency, and low overhead of ordinary memory read/write operations.

Ethernet

The most common way of communicating between computers is over Ethernet. Today, *Gigabit Ethernet* (GbE) is provided by default on systems, and most servers include 2-4 ports of Gigabit Ethernet. GbE provides good bandwidth and latency. This is the foundation of most distributed systems in use today. Even when systems include faster network hardware, it is still common to use GbE for a dedicated management interface.

10GbE

Ten Gigabit Ethernet (10GbE) is rapidly growing in acceptance for high end and even mid-range servers. 10GbE provides ten times the bandwidth of GbE. One of its major advantages is with modern multi-core processors, where it restores the balance between communication and computing. You can compare a single core system using GbE to an eight core system using 10GbE. Used in this way, 10GbE is especially valuable for maintaining overall system performance and avoiding communication bottlenecks.

Unfortunately, 10GbE is expensive. While the cost of 10GbE NICs has come down, the price of interconnect (especially fibre optics) remains high, and 10GbE network switches are extremely expensive. We can expect these prices to decline over time, but 10GbE today is most heavily used in server room backbones and performance-critical applications.

Infiniband

Infiniband offers even higher performance than 10GbE. In addition to TCP/IP and UDP network connections used with Ethernet, Infiniband also supports shared memory communication. This allows Infiniband to work between systems via *remote direct memory access* (RDMA).

The use of RDMA allows Infiniband to move data directly between systems without the overhead of TCP/IP or socket connections. In turn, this reduces latency, which is critical to some applications.

Infiniband is most commonly used in *High Performance Technical Computing* (HPTC) applications which require high bandwidth, low latency and low overhead. Many supercomputing applications benefit from this, to the point that the best way to improve performance is by investing in Infiniband rather than faster processors or more memory.

RoCCE

RDMA over Ethernet (RoCCE) implements Infiniband-style communications (including RDMA) over a 10GbE infrastructure. Given the cost improvements associated with the growing volume of 10GbE products, it is reasonable to expect wider usage of RDMA and RoCCE in a wide range of systems and applications.

Each of these communication methods is fully-supported by Red Hat for use with Red Hat Enterprise Linux 6.

1.3.2. Storage

An environment that uses distributed computing uses multiple instances of shared storage. This can mean one of two things:

- Multiple systems storing data in a single location

- A storage unit (e.g. a volume) composed of multiple storage appliances

The most familiar example of storage is the local disk drive mounted on a system. This is appropriate for IT operations where all applications are hosted on one host, or even a small number of hosts.

However, as the infrastructure scales to dozens or even hundreds of systems, managing as many local storage disks becomes difficult and complicated.

Distributed storage adds a layer to ease and automate storage hardware administration as the business scales. Having multiple systems share a handful of storage instances reduces the number of devices the administrator needs to manage.

Consolidating the storage capabilities of multiple storage appliances into one volume helps both users and administrators. This type of distributed storage provides a layer of abstraction to storage pools: users see a single unit of storage, which an administrator can easily grow by adding more hardware. Some technologies that enable distributed storage also provide added benefits, such as failover and multipathing.

NFS

Network File System (NFS) allows multiple servers or users to mount and use the same instance of remote storage via TCP or UDP. NFS is commonly used to hold data shared by multiple applications. It is also convenient for bulk storage of large amounts of data.

SAN

Storage Area Networks (SANs) use either Fibre Channel or iSCSI protocol to provide remote access to storage. Fibre Channel infrastructure (such as Fibre Channel host bus adapters, switches, and storage arrays) combines high performance, high bandwidth, and massive storage. SANs separate storage from processing, providing considerable flexibility in system design.

The other major advantage of SANs is that they provide a management environment for performing major storage hardware administrative tasks. These tasks include:

- Controlling access to storage

- Managing large amounts of data

- Provisioning systems

- Backing up and replicating data

- Taking snapshots

- Supporting system failover

- Ensuring data integrity

- Migrating data

GFS2

The Red Hat *Global File System 2* (GFS2) file system provides several specialized capabilities. The basic function of GFS2 is to provide a single file system, including concurrent read/write access, shared across multiple members of a cluster. This means that each member of the cluster sees exactly the same data "on disk" in the GFS2 filesystem.

GFS2 allows all systems to have concurrent access to the "disk". To maintain data integrity, GFS2 uses a *Distributed Lock Manager* (DLM), which only allows one system to write to a specific location at a time.

GFS2 is especially well-suited for failover applications that require high availability in storage.

For more information about GFS2, refer to the *Global File System 2 Guide*[4].

For more information about storage in general, refer to the *Storage Administration Guide*[5].

1.3.3. Converged Networks

Communication over the network is normally done through Ethernet, with storage traffic using a dedicated Fibre Channel SAN environment. It is common to have a dedicated network or serial link for system management, and perhaps even *heartbeat*[6]. As a result, a single server is typically on multiple networks.

Providing multiple connections on each server is expensive, bulky, and complex to manage. This gave rise to the need for a way to consolidate all connections into one. *Fibre Channel over Ethernet* (FCoE) and *Internet SCSI* (iSCSI) address this need.

FCoE

With FCoE, standard fibre channel commands and data packets are transported over a 10GbE physical infrastructure via a single *converged network card* (CNA). Standard TCP/IP ethernet traffic and fibre channel storage operations can be transported via the same link. FCoE uses one physical network interface card (and one cable) for multiple logical network/storage connections.

FCoE offers the following advantages:

Reduced number of connections
>   FCoE reduces the number of network connections to a server by half. You can still choose to have multiple connections for performance or availability; however, a single connection provides both storage and network connectivity. This is especially helpful for pizza box servers and blade servers, since they both have very limited space for components.

Lower cost
>   Reduced number of connections immediately means reduced number of cables, switches, and other networking equipment. Ethernet's history also features great economies of scale; the cost of networks drops dramatically as the number of devices in the market goes from millions to billions, as was seen in the decline in the price of 100Mb Ethernet and gigabit Ethernet devices.
>
>   Similarly, 10GbE will also become cheaper as more businesses adapt to its use. Also, as CNA hardware is integrated into a single chip, widespread use will also increase its volume in the market, which will result in a significant price drop over time.

iSCSI

Internet SCSI (iSCSI) is another type of converged network protocol; it is an alternative to FCoE. Like fibre channel, iSCSI provides block-level storage over a network. However, iSCSI does not provide a complete management environment. The main advantage of iSCSI over FCoE is that iSCSI provides much of the capability and flexibility of fibre channel, but at a lower cost.

---

[4] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html-single/Global_File_System_2/index.html

[5] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html-single/Storage_Administration_Guide/index.html

[6] *Heartbeat* is the exchange of messages between systems to ensure that each system is still functioning. If a system "loses heartbeat" it is assumed to have failed and is shut down, with another system taking over for it.

Red Hat Enterprise Linux 6 Performance Features

2.1. 64-Bit Support

Red Hat Enterprise Linux 6 supports 64-bit processors; these processors can theoretically use up to 18 *exabytes* of memory. As of general availability (GA), Red Hat Enterprise Linux 6 is tested and certified to support up to 8TB of physical memory.

The size of memory supported by Red Hat Enterprise Linux 6 is expected to grow over several minor updates, as Red Hat continues to introduce and improve more features that enable the use of larger memory blocks. Examples of such improvements (as of Red Hat Enterprise Linux 6 GA) are:

- Huge pages and transparent huge pages

- Non-Uniform Memory Access improvements

These improvements are outlined in greater detail in the sections that follow.

Huge pages and transparent huge pages

The implementation of *huge pages* in Red Hat Enterprise Linux 6 allows the system to manage memory use efficiently across different memory workloads. Huge pages dynamically utilize 2 MB pages compared to the standard 4 KB page size, allowing applications to scale well from processing gigabytes and even terabytes of memory.

Huge pages are difficult to manually create, manage, and use. To address this, Red Hat Enterprise 6 also features the use of *transparent huge pages* (THP). THP automatically manages many of the complexities involved in the use of huge pages.

For more information on huge pages and THP, refer to *Section 5.2, "Huge Pages and Transparent Huge Pages"*.

NUMA improvements

Many new systems now support *Non-Uniform Memory Access* (NUMA). NUMA simplifies the design and creation of hardware for large systems; however, it also adds a layer of complexity to application development. For example, NUMA implements both local and remote memory, where remote memory can take several times longer to access than local memory. This feature (among others) has many performance implications that impact operating systems, applications, and system configurations should be deployed.

Red Hat Enterprise Linux 6 is better optimized for NUMA use, thanks to several additional features that help manage users and applications on NUMA systems. These features include CPU affinity, CPU pinning (cpusets), numactl and control groups, which allow a process (affinity) or application (pinning) to "bind" to a specific CPU or set of CPUs.

For more information about NUMA support in Red Hat Enterprise Linux 6, refer to *Section 4.1, "CPU and NUMA Topology"*.

2.2. Ticket Spinlocks

A key part of any system design is ensuring that one process does not alter memory used by another process. Uncontrolled data change in memory can result in data corruption and system crashes. To

prevent this, the operating system allows a process to lock a piece of memory, perform an operation, then unlock or "free" the memory.

One common implementation of memory locking is through *spin locks*, which allow a process to keep checking to see if a lock is available and take the lock as soon as it becomes available. If there are multiple processes competing for the same lock, the first one to request the lock after it has been freed gets it. When all processes have the same access to memory, this approach is "fair" and works quite well.

Unfortunately, on a NUMA system, not all processes have equal access to the locks. Processes on the same NUMA node as the lock have an unfair advantage in obtaining the lock. Processes on remote NUMA nodes experience lock starvation and degraded performance.

To address this, Red Hat Enterprise Linux implemented *ticket spinlocks*. This feature adds a reservation queue mechanism to the lock, allowing *all* processes to take a lock in the order that they requested it. This eliminates timing problems and unfair advantages in lock requests.

While a ticket spinlock has slightly more overhead than an ordinary spinlock, it scales better and provides better performance on NUMA systems.

2.3. Dynamic List Structure

The operating system requires a set of information on each processor in the system. In Red Hat Enterprise Linux 5, this set of information was allocated to a fixed-size array in memory. Information on each individual processor was obtained by indexing into this array. This method was fast, easy, and straightforward for systems that contained relatively few processors.

However, as the number of processors for a system grows, this method produces significant overhead. Because the fixed-size array in memory is a single, shared resource, it can become a bottleneck as more processors attempt to access it at the same time.

To address this, Red Hat Enterprise Linux 6 uses a *dynamic list structure* for processor information. This allows the array used for processor information to be allocated dynamically: if there are only eight processors in the system, then only eight entries are created in the list. If there are 2048 processors, then 2048 entries are created as well.

A dynamic list structure allows more fine-grained locking. For example, if information needs to be updated at the same time for processors 6, 72, 183, 657, 931 and 1546, this can be done with greater parallelism. Situations like this obviously occur much more frequently on large, high-performance systems than on small systems.

2.4. Tickless Kernel

In previous versions of Red Hat Enterprise Linux, the kernel used a timer-based mechanism that continuously produced a system interrupt. During each interrupt, the system *polled*; that is, it checked to see if there was work to be done.

Depending on the setting, this system interrupt or *timer tick* could occur several hundred or several thousand times per second. This happened every second, regardless of the system's workload. On a lightly loaded system, this impacts *power consumption* by preventing the processor from effectively using sleep states. The system uses the least power when it is in a sleep state.

The most power-efficient way for a system to operate is to do work as quickly as possible, go into the deepest sleep state possible, and sleep as long as possible. To implement this, Red Hat Enterprise Linux 6 uses a *tickless kernel*. With this, the interrupt timer has been removed from the idle loop, transforming Red Hat Enterprise Linux 6 into a completely interrupt-driven environment.

The tickless kernel allows the system to go into deep sleep states during idle times, and respond quickly when there is work to be done.

Refer to the *Power Management Guide*[2] for more information.

2.5. Control Groups

Red Hat Enterprise Linux provides many useful options for performance tuning. Large systems, scaling to hundreds of processors, can be tuned to deliver superb performance. But tuning these systems requires considerable expertise and a well-defined workload. When large systems were expensive and few in number, it was acceptable to give them special treatment. Now that these systems are mainstream, more effective tools are needed.

To further complicate things, more powerful systems are being used now for service consolidation. Workloads that may have been running on four to eight older servers are now placed into a single server. And as discussed earlier in *Section 1.2.1, "Parallel Computing"*, many mid-range systems nowadays contain more cores than yesterday's high-performance machines.

Many modern applications are designed for parallel processing, using multiple threads or processes to improve performance. However, few applications can make effective use of more than eight threads. Thus, multiple applications typically need to be installed on a 32-CPU system to maximize capacity.

Consider the situation: small, inexpensive mainstream systems are now at parity with the performance of yesterday's expensive, high-performance machines. Cheaper high-performance machines gave system architects the ability to consolidate more services to fewer machines.

However, some resources (such as I/O and network communications) are shared, and do not grow as fast as CPU count. As such, a system housing multiple applications can experience degraded overall performance when one application hogs too much of a single resource.

To address this, Red Hat Enterprise Linux 6 now supports *control groups* (cgroups). Cgroups allow administrators to allocate resources to specific tasks as needed. This means, for example, being able to allocate 80% of four CPUs, 60GB of memory, and 40% of disk I/O to a database application. A web application running on the same system would be given two CPUs, 2GB of memory, and 50% of available network bandwidth.

As a result, both database and web applications deliver good performance, as the system prevents both from excessively consuming system resources. In addition, many aspects of cgroups are *self-tuning*, allowing the system to respond accordingly to changes in workload.

A cgroup has two major components:

• A list of tasks assigned to the cgroup

• Resources allocated to those tasks

Tasks assigned to the cgroup run *within* the cgroup. Any child tasks they spawn also run within the cgroup. This allows an administrator to manage an entire application as a single unit. An administrator can also configure allocations for the following resources:

• CPUsets

• Memory

---

[2] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html-single/Power_Management_Guide/index.html

- I/O

- Network (bandwidth)

Within CPUsets, cgroups allow administrators to configure the number of CPUs, affinity for specific CPUs or nodes [3], and the amount of CPU time used by a set of tasks. Using cgroups to configure CPUsets is vital for ensuring good overall performance, preventing an application from consuming excessive resources at the cost of other tasks while simultaneously ensuring that the application is not starved for CPU time.

I/O bandwidth and network bandwidth are managed by other resource controllers. Again, the resource controllers allow you to determine how much bandwidth the tasks in a cgroup can consume, and ensure that the tasks in a cgroup neither consume excessive resources nor are starved of resources.

Cgroups allow the administrator to define and allocate, at a high level, the system resources that various applications need (and will) consume. The system then automatically manages and balances the various applications, delivering good predictable performance and optimizing the performance of the overall system.

For more information on how to use control groups, refer to the Red Hat Enterprise Linux 6 *Resource Management Guide*[4].

2.6. Storage and File System Improvements

Red Hat Enterprise Linux 6 also features several improvements to storage and file system management. Two of the most notable advances in this version are ext4 and XFS support. For more comprehensive coverage of performance improvements relating to storage and file systems, refer to *Chapter 7, Storage*.

Ext4

Ext4 is the default file system for Red Hat Enterprise Linux 6. It is the fourth generation version of the EXT file system family, supporting a theoretical maximum file system size of 1 exabyte, and single file maximum size of 16TB. Red Hat Enterprise Linux 6 supports a maximum file system size of 16TB, and a single file maximum size of 16TB. Other than a much larger storage capacity, ext4 also includes several new features, such as:

- Extent-based metadata

- Delayed allocation

- Journal check-summing

For more information about the ext4 file system, refer to *Section 7.3.1, "The Ext4 File System"*.

XFS

XFS is a robust and mature 64-bit journaling file system that supports very large files and file systems on a single host. This file system was originally developed by SGI, and has a long history of running on extremely large servers and storage arrays. XFS features include:

- Delayed allocation

---

[3] A node is generally defined as a set of CPUs or cores within a socket.
[4] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/index.html

- Dynamically-allocated inodes

- B-tree indexing for scalability of free space management

- Online defragmentation and file system growing

- Sophisticated metadata read-ahead algorithms

While XFS scales to exabytes, the maximum XFS file system size supported by Red Hat is 100TB. For more information about XFS, refer to *Section 7.4, "The XFS File System"*.

Large Boot Drives

Traditional BIOS supports a maximum disk size of 2.2TB. Red Hat Enterprise Linux 6 systems using BIOS can support disks larger than 2.2TB by using a new disk structure called *Global Partition Table* (GPT). GPT can only be used for data disks; it cannot be used for boot drives with BIOS; therefore, boot drives can only be a maximum of 2.2TB in size. The BIOS was originally created for the IBM PC; while BIOS has evolved considerably to adapt to modern hardware, *Unified Extensible Firmware Interface* (UEFI) is designed to support new and emerging hardware.

Red Hat Enterprise Linux 6 also supports UEFI, which can be used to replace BIOS (still supported). Systems with UEFI running Red Hat Enterprise Linux 6 allow the use of GPT and 2.2TB (and larger) partitions for both boot partition and data partition.

> **Important — UEFI for 32-bit x86 systems**
>
> Red Hat Enterprise Linux 6 does not support UEFI for 32-bit x86 systems.

> **Important — UEFI for AMD64 and Intel 64**
>
> Note that the boot configurations of UEFI and BIOS differ significantly from each other. Therefore, the installed system must boot using the same firmware that was used during installation. You cannot install the operating system on a system that uses BIOS and then boot this installation on a system that uses UEFI.
>
> Red Hat Enterprise Linux 6 supports version 2.2 of the UEFI specification. Hardware that supports version 2.3 of the UEFI specification or later should boot and operate with Red Hat Enterprise Linux 6, but the additional functionality defined by these later specifications will not be available. The UEFI specifications are available from *http://www.uefi.org/specs/agreement/*.

Monitoring and Analyzing System Performance

This chapter briefly introduces tools that can be used to monitor and analyze system and application performance, and points out the situations in which each tool is most useful. The data collected by each tool can reveal bottlenecks or other system problems that contribute to less-than-optimal performance.

3.1. The proc File System

The **proc** "file system" is a directory that contains a hierarchy of files that represent the current state of the Linux kernel. It allows applications and users to see the kernel's view of the system.

The **proc** directory also contains information about the hardware of the system, and any currently running processes. Most of these files are read-only, but some files (primarily those in **/proc/sys**) can be manipulated by users and applications to communicate configuration changes to the kernel.

For further information about viewing and editing files in the **proc** directory, refer to the *Deployment Guide*[1].

3.2. GNOME and KDE System Monitors

The GNOME and KDE desktop environments both have graphical tools to assist you in monitoring and modifying the behavior of your system.

GNOME System Monitor

The **GNOME System Monitor** displays basic system information and allows you to monitor system processes, and resource or file system usage. Open it with the **gnome-system-monitor** command in the **Terminal**, or click on the **Applications** menu, and select **System Tools** > **System Monitor**.

GNOME System Monitor has four tabs:

System

Displays basic information about the computer's hardware and software.

Processes

Shows active processes, and the relationships between those processes, as well as detailed information about each process. It also lets you filter the processes displayed, and perform certain actions on those processes (start, stop, kill, change priority, etc.).

Resources

Displays the current CPU time usage, memory and swap space usage, and network usage.

File Systems

Lists all mounted file systems alongside some basic information about each, such as the file system type, mount point, and memory usage.

---

[1] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html-single/Deployment_Guide/index.html

For further information about the **GNOME System Monitor**, refer to the **Help** menu in the application, or to the *Deployment Guide*[2].

KDE System Guard

The **KDE System Guard** allows you to monitor current system load and processes that are running. It also lets you perform actions on processes. Open it with the `ksysguard` command in the **Terminal**, or click on the **Kickoff Application Launcher** and select **Applications** > **System** > **System Monitor**.

There are two tabs to **KDE System Guard**:

Process Table

Displays a list of all running processes, alphabetically by default. You can also sort processes by a number of other properties, including total CPU usage, physical or shared memory usage, owner, and priority. You can also filter the visible results, search for specific processes, or perform certain actions on a process.

System Load

Displays historical graphs of CPU usage, memory and swap space usage, and network usage. Hover over the graphs for detailed analysis and graph keys.

For further information about the **KDE System Guard**, refer to the **Help** menu in the application.

3.3. Built-in Command-line Monitoring Tools

In addition to graphical monitoring tools, Red Hat Enterprise Linux provides several tools that can be used to monitor a system from the command line. The advantage of these tools is that they can be used outside run level 5. This section discusses each tool briefly, and suggests the purposes to which each tool is best suited.

top

The **top** tool provides a dynamic, real-time view of the processes in a running system. It can display a variety of information, including a system summary and the tasks currently being managed by the Linux kernel. It also has a limited ability to manipulate processes. Both its operation and the information it displays are highly configurable, and any configuration details can be made to persist across restarts.

By default, the processes shown are ordered by the percentage of CPU usage, giving an easy view into the processes that are consuming the most resources.

For detailed information about using **top**, refer to its man page: `man top`.

ps

The **ps** tool takes a snapshot of a select group of active processes. By default this group is limited to processes owned by the current user and associated with the same terminal.

It can provide more detailed information about processes than **top**, but is not dynamic.

For detailed information about using **ps**, refer to its man page: `man ps`.

---

[2] http://documentation-stage.bne.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html-single/Deployment_Guide/index.html

vmstat

vmstat (Virtual Memory Statistics) outputs instantaneous reports about your system's processes, memory, paging, block I/O, interrupts and CPU activity.

Although it is not dynamic like **top**, you can specify a sampling interval, which lets you observe system activity in near-real time.

For detailed information about using **vmstat**, refer to its man page: **man vmstat**.

sar

sar (System Activity Reporter) collects and reports information about today's system activity so far. The default output covers today's CPU utilization at ten minute intervals from the beginning of the day:

```
12:00:01 AM     CPU     %user     %nice    %system    %iowait    %steal      %idle
12:10:01 AM     all      0.10      0.00       0.15       2.96      0.00       96.79
12:20:01 AM     all      0.09      0.00       0.13       3.16      0.00       96.61
12:30:01 AM     all      0.09      0.00       0.14       2.11      0.00       97.66
...
```

This tool is a useful alternative to attempting to create periodic reports on system activity through **top** or similar tools.

For detailed information about using **sar**, refer to its man page: **man sar**.

3.4. Tuned and ktune

Tuned is a daemon that monitors and collects data on the usage of various system components, and uses that information to dynamically tune system settings as required. It can react to changes in CPU and network use, and adjust settings to improve performance in active devices or reduce power consumption in inactive devices.

The accompanying **ktune** partners with the **tuned-adm** tool to provide a number of tuning profiles that are pre-configured to enhance performance and reduce power consumption in a number of specific use cases. Edit these profiles or create new profiles to create performance solutions tailored to your environment.

Further information about **tuned** and **ktune** is available in the Red Hat Enterprise Linux 6 *Power Management Guide*, available from *http://docs.redhat.com/*.

3.5. Application Profilers

Profiling is the process of gathering information about a program's behavior as it executes. You profile an application to determine which areas of a program can be optimized to increase the program's overall speed, reduce its memory usage, etc. Application profiling tools help to simplify this process.

There are three supported profiling tools for use with Red Hat Enterprise Linux 6: **SystemTap**, **OProfile** and **Valgrind**. Documenting these profiling tools is outside the scope of this guide; however, this section does provide links to further information and a brief overview of the tasks for which each profiler is suitable.

3.5.1. SystemTap

SystemTap is a tracing and probing tool that lets users monitor and analyze operating system activities (particularly kernel activities) in fine detail. It provides information similar to the output of tools like

netstat, **top**, **ps** and **iostat**, but includes additional filtering and analysis options for the information that is collected.

SystemTap provides a deeper, more precise analysis of system activities and application behavior to allow you to pinpoint system and application bottlenecks.

The Function Callgraph plug-in for Eclipse uses SystemTap as a back-end, allowing it to thoroughly monitor the status of a program, including function calls, returns, times, and user-space variables, and display the information visually for easy optimization.

For further information about Systemtap, refer to the *SystemTap Beginners Guide*[3].

3.5.2. OProfile

OProfile (**oprofile**) is a system-wide performance monitoring tool. It uses the processor's dedicated performance monitoring hardware to retrieve information about the kernel and system executables, such as when memory is referenced, the number of L2 cache requests, and the number of hardware interrupts received. It can also be used to determine processor usage, and which applications and services are used most.

OProfile can also be used with Eclipse via the Eclipse OProfile plug-in. This plug-in allows users to easily determine the most time-consuming areas of their code, and perform all command-line functions of OProfile with rich visualization of the results.

However, users should be aware of several OProfile limitations:

- Performance monitoring samples may not be precise - because the processor may execute instructions out of order, a sample may be recorded from a nearby instruction, instead of the instruction that triggered the interrupt.

- Because OProfile is system-wide and expects processes to start and stop multiple times, samples from multiple runs are allowed to accumulate. This means you may need to clear sample data from previous runs.

- It focuses on identifying problems with CPU-limited processes, and therefore does not identify processes that are sleeping while they wait on locks for other events.

For further information about using OProfile, refer to the *Deployment Guide*[4], or to the **oprofile** documentation on your system, under **/usr/share/doc/oprofile-<version>**.

3.5.3. Valgrind

Valgrind provides a number of detection and profiling tools to help improve the performance and correctness of your applications. These tools can detect memory and thread-related errors as well as heap, stack and array overruns, allowing you to easily locate and correct errors in your application code. They can also profile the cache, the heap, and branch-prediction to identify factors that may increase application speed and minimize application memory use.

Valgrind analyzes your application by running it on a synthetic CPU and instrumenting the existing application code as it is executed. It then prints "commentary" clearly identifying each process involved in application execution to a user-specified file descriptor, file, or network socket. The level of

---

[3] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html-single/SystemTap_Beginners_Guide/
[4] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html-single/Deployment_Guide/

instrumentation varies depending on the Valgrind tool in use, and its settings, but it is important to note that executing the instrumented code can take 4-50 times longer than normal execution.

Valgrind can be used on your application as-is, without recompiling. However, because Valgrind uses debugging information to pinpoint issues in your code, if your application and support libraries were not compiled with debugging information enabled, recompiling to include this information is highly recommended.

More information about Valgrind is available from the *Developer Guide*[5], or by using the **man valgrind** command once the *valgrind* package is installed. Accompanying documentation can also be found in:

- **/usr/share/doc/valgrind-<version>/valgrind_manual.pdf**

- **/usr/share/doc/valgrind-<version>/html/index.html**

3.5.4. Perf

The **perf** tool provides a number of useful performance counters that let the user assess the impact of other commands on their system:

perf stat
    This command provides overall statistics for common performance events, including instructions executed and clock cycles consumed. You can use the option flags to gather statistics on events other than the default measurement events. For further information, read the man page: **man perf-stat**.

perf record
    This command records performance data into a file which can be later analyzed using **perf report**. For further details, read the man page: **man perf-record**.

perf report
    This command reads the performance data from a file and analyzes the recorded data. For further details, read the man page: **man perf-report**.

perf list
    This command lists the events available on a particular machine. These events will vary based on the performance monitoring hardware and the software configuration of the system. For further information, read the man page: **man perf-list**.

perf top
    This command performs a similar function to the **top** tool. It generates and displays a performance counter profile in realtime. For further information, read the man page: **man perf-top**.

More information about **perf** is available in the Red Hat Enterprise Linux *Developer Guide* at *http:// docs.redhat.com*.

3.6. Red Hat Enterprise MRG

Red Hat Enterprise MRG's Realtime component includes **Tuna**, a tool that allows users to both adjust the tunable values their system and view the results of those changes. While it was developed for use with the Realtime component, it can also be used to tune standard Red Hat Enterprise Linux systems.

---

[5] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Developer_Guide/valgrind.html

With Tuna, you can adjust or disable unnecessary system activity, including:

- BIOS parameters related to power management, error detection, and system management interrupts;

- network settings, such as interrupt coalescing, and the use of TCP;

- journaling activity in journaling file systems;

- system logging;

- whether interrupts and user processes are handled by a specific CPU or range of CPUs;

- whether swap space is used; and

- how to deal with out-of-memory exceptions.

For more detailed conceptual information about tuning Red Hat Enterprise MRG with the Tuna interface, refer to the "General System Tuning" chapter of the *Realtime Tuning Guide*. For detailed instructions about using the Tuna interface, refer to the *Tuna User Guide*. Both guides appear under **Red Hat Enterprise MRG** at *http://docs.redhat.com*.

CPU

The term CPU, which stands for *central processing unit*, is a misnomer for most systems, since *central* implies *single*, whereas most modern systems have more than one processing unit, or core. Physically, CPUs are contained in a package attached to a motherboard in a *socket*. Each socket on the motherboard has various connections: to other CPU sockets, memory controllers, interrupt controllers, and other peripheral devices. A socket to the operating system is a logical grouping of CPUs and associated resources. This concept is central to most of our discussions on CPU tuning.

Red Hat Enterprise Linux keeps a wealth of statistics about system CPU events; these statistics are useful in planning out a tuning strategy to improve CPU performance.

SMP and NUMA

Older computers had relatively few CPUs per system, which allowed an architecture known as *Symmetric Multi-Processor* (SMP). This meant that each CPU in the system had similar (or symmetric) access to available memory. In recent years, CPU count-per-socket has grown to the point that trying to give symmetric access to all RAM in the system has become very expensive. Most high CPU count systems these days have an architecture known as *Non-Uniform Memory Access* (NUMA) instead of SMP.

AMD processors have had this type of architecture for some time with their *Hyper Transport* (HT) interconnects, while Intel has begun implementing NUMA in their *Quick Path Interconnect* (QPI) designs. NUMA and SMP are tuned differently, since you need to account for the *topology* of the system when allocating resources for an application.

Threads

Inside the Linux operating system, the unit of execution is known as a *thread*. Threads have a register context, a stack, and a segment of executable code which they run on a CPU. It is the operating system's job to schedule these threads on the available CPUs.

The OS maximizes CPU utilization by load-balancing the threads across available cores. Since the OS is primarily concerned with keeping CPUs busy, it may not make optimal decisions with respect to application performance. Moving an application thread to a CPU on another socket may worsen performance more than simply waiting for the current CPU to become available, since memory access operations may slow drastically across sockets. For high-performance applications, it is usually better for the designer to determine where threads should be placed. *Section 4.4, "CPU Scheduler"* discusses how to best allocate CPUs and memory to best execute application threads.

Interrupts

One of the less obvious (but nonetheless important) system events that can impact application performance is the *interrupt* (also known as IRQs in Linux). These events are handled by the operating system, and are used by peripherals to signal the arrival of data or the completion of an operation, such as a network write or a timer event.

The manner in which the OS or CPU that is executing application code handles an interrupt does not affect the application's functionality. However, it may impact the performance of the application. This chapter also discusses tips on preventing interrupts from adversely impacting application performance.

4.1. CPU and NUMA Topology

The first computer processors were *uniprocessors*, meaning that the system had a single CPU. The illusion of executing processes in parallel was done by the operating system rapidly switching the

single CPU from one thread of execution (process) to another. In the quest for increasing system performance, designers noted that increasing the clock rate to execute instructions faster only worked up to a point (usually the limitations on creating a stable clock waveform with the current technology). In an effort to get more overall system performance, designers added another CPU to the system, allowing two parallel streams of execution. This trend of adding processors has continued over time.

Most early multiprocessor systems were designed so that each CPU had the same logical path to each memory location (usually a parallel bus). This let each CPU access any memory location in the same amount of time as any other CPU in the system. This type of architecture is known as a Symmetric Multi-Processor (SMP) system. SMP is fine for a small number of CPUs, but once the CPU count gets above a certain point (8 or 16), the number of parallel traces required to allow equal access to memory uses too much of the available board real estate, leaving less room for peripherals.

Two new concepts combined to allow for a higher number of CPUs in a system:

1. Serial buses

2. NUMA topologies

A serial bus is a single-wire communication path with a very high clock rate, which transfers data as packetized bursts. Hardware designers began to use serial buses as high-speed interconnects between CPUs, and between CPUs and memory controllers and other peripherals. This means that instead of requiring between 32 and 64 traces on the board from *each* CPU to the memory subsystem, there was now *one* trace, substantially reducing the amount of space required on the board.

At the same time, hardware designers were packing more transistors into the same space by reducing die sizes. Instead of putting individual CPUs directly onto the main board, they started packing them into a processor package as multi-core processors. Then, instead of trying to provide equal access to memory from each processor package, designers resorted to a Non-Uniform Memory Access (NUMA) strategy, where each package/socket combination has one or more dedicated memory area for high speed access. Each socket also has an interconnect to other sockets for slower access to the other sockets' memory.

As a simple NUMA example, suppose we have a two-socket motherboard, where each socket has been populated with a quad-core package. This means the total number of CPUs in the system is eight; four in each socket. Each socket also has an attached memory bank with four gigabytes of RAM, for a total system memory of eight gigabytes. For the purposes of this example, CPUs 0-3 are in socket 0, and CPUs 4-7 are in socket 1. Each socket in this example also corresponds to a NUMA node.

It might take three clock cycles for CPU 0 to access memory from bank 0: a cycle to present the address to the memory controller, a cycle to set up access to the memory location, and a cycle to read or write to the location. However, it might take six clock cycles for CPU 4 to access memory from the same location; because it is on a separate socket, it must go through two memory controllers: the local memory controller on socket 1, and then the remote memory controller on socket 0. If memory is contested on that location (that is, if more than one CPU is attempting to access the same location simultaneously), memory controllers need to arbitrate and serialize access to the memory, so memory access will take longer. Adding cache consistency (ensuring that local CPU caches contain the same data for the same memory location) complicates the process further.

The latest high-end processors from both Intel (Xeon) and AMD (Opteron) have NUMA topologies. The AMD processors use an interconnect known as HyperTransport™ or HT, while Intel uses one named QuickPath Interconnect™ or QPI. The interconnects differ in how they physically connect to other interconnects, memory, or peripheral devices, but in effect they are a switch that allows transparent access to one connected device from another connected device. In this case, transparent refers to the fact that there is no special programming API required to use the interconnect, not a "no cost" option.

Because system architectures are so diverse, it is impractical to specifically characterize the performance penalty imposed by accessing non-local memory. We can say that each *hop* across an interconnect imposes at least some relatively constant performance penalty per hop, so referencing a memory location that is two interconnects from the current CPU imposes at least *2N + memory cycle time* units to access time, where N is the penalty per hop.

Given this performance penalty, performance-sensitive applications should avoid regularly accessing remote memory in a NUMA topology system. The application should be set up so that it stays on a particular node and allocates memory from that node.

To do this, there are a few things that applications will need to know:

1.   What is the *topology* of the system?

2.   Where is the application currently executing?

3.   Where is the closest memory bank?

The topology of a system is how a system's components are connected: CPUs, memory, and peripheral devices. The **/sys** file system contains a wealth of information about how CPUs and devices are connected via NUMA interconnects.

The **/sys/devices/system/cpu** directory contains information about how CPUs are connected to one another. The **/sys/devices/system/node** directory contains information about the NUMA nodes in the system; in particular, the relative distances between nodes.

4.1.1. Using numactl and libnuma

Red Hat Enterprise Linux provides two methods for modifying how an application runs on a NUMA system. The first is a low-level library named **libnuma**, which contains functions for determining the NUMA topology of the host system, setting the CPUs upon which a thread can execute, and setting which NUMA nodes a thread should use when allocating memory.

The second method is **numactl**, a utility program that uses libnuma to set up NUMA execution parameters for an arbitrary application. numactl takes that application (and the arguments of that application) as an argument of its own. The application specified will be started, and the NUMA options passed to numactl; for example:

```
numactl --cpubind=0 --membind=0,1  myprogram arg1 arg2
```

This example tells the operating system to execute the program *myprogram* with arguments arg1 and arg2, to bind the myprogram process to NUMA node 0, and to only allocate memory for myprogram from NUMA nodes 0 and 1.

Two useful options to numactl are the **--show** and **--hardware** options. These are information display options, and are not used when executing programs on specific nodes. The **--show** option shows the current NUMA policy defaults, while the **--hardware** option shows the available NUMA nodes, as well as the CPUs and memory banks available on each node.

Avoid making direct libnuma calls from within an application; doing so ties the application to a particular NUMA platform, or requires further application code to support various NUMA topologies. A more practical solution is to start your application with numactl and specify parameters that bind your threads to particular nodes/CPUs, and allocate memory from local memory banks. This lets you experiment with various NUMA settings to find those that yield the best performance for your application on a given platform. When the next generation of hardware is deployed, you can retest to verify that the settings are still valid, and, if not, adjust them without any changes to application code.

For more information on **numactl**, refer to the *numactl* man page: `man numactl(8)`. For information about the functions available from within the **libnuma** shared library, read the *numa* man page: `man numa(3)`.

4.2. NUMA and Multi-Core Support

Originally, NUMA was a mechanism that connected single processors to multiple memory banks. As CPU manufacturers refined their processes and die sizes shrank, multiple CPU cores could be included in one package. These CPU cores were *clustered* so that each had equal access time to a local memory bank, and cache could be shared between the cores.

When multi-core packages are in use, applications should generally be bound to a single socket or node rather than being bound to a CPU core, particularly if the application is multi-threaded. Binding multiple threads to a single CPU core is not recommended. First-level caches are generally quite small (around 32 KB), so when multiple threads execute and access data on a single CPU, each thread may potentially access data that will evict previously-accessed data from the cache. This means that when the operating system tries to multitask between these threads, a large percentage of execution time is used on cache line replacement operations. This is known as *cache thrashing*. Binding an application to a node (and therefore all CPUs belonging to that node) allows threads to share cache lines on multiple levels (first-, second- and third-level cache), minimizing the need for cache fill operations.

The performance of a NUMA system can be improved primarily by ensuring that information travels efficiently. To do so, you must be aware of your system's topology - the CPUs, the memory banks, and the paths between them.

The system in *Figure 4.1, "An example of NUMA topology"* contains two NUMA nodes. Each node has four CPUs, a memory bank, and a memory controller. Any CPU on a node has direct access to the memory bank on that node. Following the arrows on Node 1, the steps are as follows:

1. A CPU (any of 0-3) presents the memory address to the local memory controller.

2. The memory controller sets up access to the memory address.

3. The CPU performs read or write operations on that memory address.
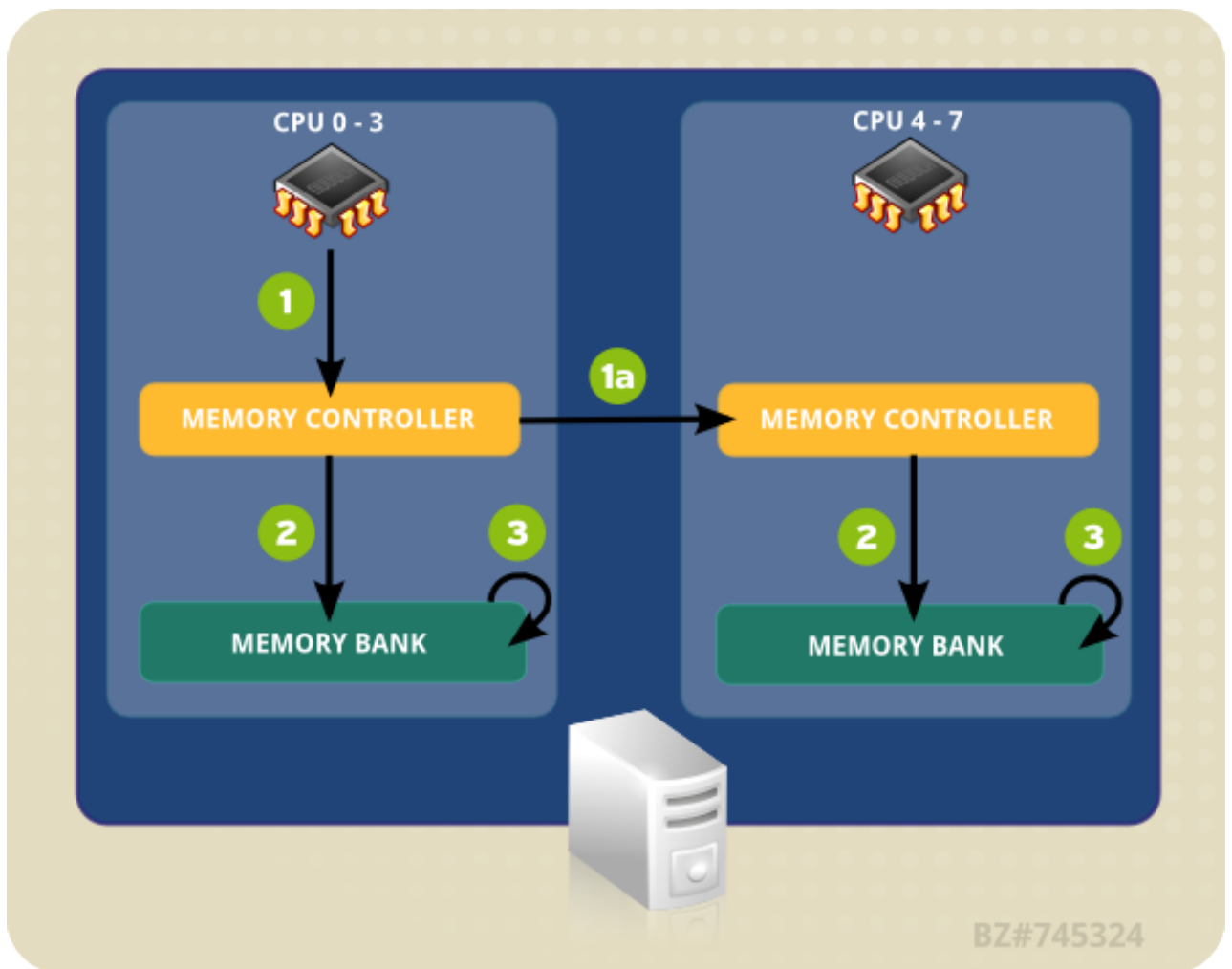
Figure 4.1. An example of NUMA topology

However, if a CPU on one node needs to access code that resides on the memory bank of a different NUMA node, the path it has to take is less direct:

1. A CPU (any of 0-3) presents the remote memory address to the local memory controller.

    a. The CPU's request for that remote memory address is passed to a remote memory controller, local to the node containing that memory address.

2. The remote memory controller sets up access to the remote memory address.

3. The CPU performs read or write operations on that remote memory address.

Since in the second case every action needs to pass through multiple memory controllers, access can take more than twice as long when attempting to access remote memory addresses. It is therefore important to ensure that information travels the shortest possible path.

The best way to ensure this is to use the **numactl** utility or **numa(7)** library calls to locate the nearest memory bank to the CPU(s) where your application is running, and bind the application to that local memory bank.

The **numactl --hardware** command lists the available hardware:

```
$ numactl --hardware
```

```
available: 4 nodes (0-3)

node 0 cpus: 0 1 2 3
node 0 size: 8189 MB
node 0 free: 7616 MB

node 1 cpus: 4 5 6 7
node 1 size: 8192 MB
node 1 free: 7756 MB

node 2 cpus: 8 9 10 11
node 2 size: 8192 MB
node 2 free: 7594 MB

node 3 cpus: 12 13 14 15
node 3 size: 8192 MB
node 3 free: 7756 MB

node distances:
node   0   1   2   3
0:    10  20  20  20
1:    20  10  20  20
2:    20  20  10  20
3:    20  20  20  10
```

This example system is a four-node NUMA system with 16 CPUs (four per node) and 32 GB of memory (8 GB per node).

The node distance matrix output by the **numactl --hardware** command shows the relative "cost" of accessing one node from another. The distance matrix in the example output shows that accessing something on node 0 from node 0 has a cost of 10, while accessing any other node from node 0 has a cost of 20. This means that all other nodes are directly connected to node 0. Since the cost to access any node from any node other than itself is 20, all four nodes in the example output are directly connected to each other.

> ### Check the distance matrix on new systems
>
> Not every system has a connection between every node and every other node, so it is prudent to check the distance matrix on new systems to determine the relative costs of inter-node access.

```
$ numactl --membind 1 --cpunodebind 1 --localalloc  myapplication
```

This command runs a program called **myapplication**, binds it to memory node 1, and constrains its threads to only run on the CPUs on node 1 (CPUs 4, 5, 6, and 7). You could also do this using **numa(7)** calls from the application, but doing so reduces the administrator's ability to change the location of the application when deploying on different hardware.

4.3. NUMA enhancements in Red Hat Enterprise Linux 6

Red Hat Enterprise Linux 6 includes a number of enhancements to capitalize on the full potential of today's highly scalable hardware. This section gives a high-level overview of the most important NUMA-related performance enhancements provided by Red Hat Enterprise Linux 6.

4.3.1. Bare-metal and scalability optimizations

4.3.1.1. Enhancements in topology-awareness

The following enhancements allow Red Hat Enterprise Linux to detect low-level hardware and architecture details, improving its ability to automatically optimize processing on your system.

enhanced topology detection

> This allows the operating system to detect low-level hardware details (such as logical CPUs, hyper threads, cores, sockets, NUMA nodes and access times between nodes) at boot time, and optimize processing on your system.

completely fair scheduler

> This new scheduling mode ensures that runtime is shared evenly between eligible processes. Combining this with topology detection allows processes to be scheduled onto CPUs within the same socket to avoid the need for expensive remote memory access, and ensure that cache content is preserved wherever possible.

malloc

> malloc is now optimized to ensure that the regions of memory that are allocated to a process are as physically close as possible to the core on which the process is executing. This increases memory access speeds.

skbuff I/O buffer allocation

> Similarly to malloc, this is now optimized to use memory that is physically close to the CPU handling I/O operations such as device interrupts.

device interrupt affinity

> Information recorded by device drivers about which CPU handles which interrupts can be used to restrict interrupt handling to CPUs within the same physical socket, preserving cache affinity and limiting high-volume cross-socket communication.

4.3.1.2. Enhancements in multi-processor synchronization

Coordinating tasks between multiple processors requires frequent, time-consuming operations to ensure that processes executing in parallel do not compromise data integrity. Red Hat Enterprise Linux includes the following enhancements to improve performance in this area:

Read-Copy-Update (RCU) locks

> Typically, 90% of locks are acquired for read-only purposes. RCU locking removes the need to obtain an exclusive-access lock when the data being accessed is not being modified. This locking mode is now used in page cache memory allocation: locking is now used only for allocation or deallocation operations.

per-CPU and per-socket algorithms

> Many algorithms have been updated to perform lock coordination among cooperating CPUs on the same socket to allow for more fine-grained locking. Numerous global spinlocks have been replaced with per-socket locking methods, and updated memory allocator zones and related memory page lists allow memory allocation logic to traverse a more efficient subset of the memory mapping data structures when performing allocation or deallocation operations.

4.3.2. Virtualization optimizations

Because KVM utilizes kernel functionality, KVM-based virtualized guests immediately benefit from all bare-metal optimizations. Red Hat Enterprise Linux also includes a number of enhancements to allow

virtualized guests to approach the performance level of a bare-metal system. These enhancements focus on the I/O path in storage and network access, allowing even intensive workloads such as database and file-serving to make use of virtualized deployment. NUMA-specific enhancements that improve the performance of virtualized systems include:

CPU pinning
> Virtual guests can be bound to run on a specific socket in order to optimize local cache use and remove the need for expensive inter-socket communications and remote memory access.

transparent hugepages (THP)
> With THP enabled, the system automatically performs NUMA-aware memory allocation requests for large contiguous amounts of memory, reducing both lock contention and the number of translation lookaside buffer (TLB) memory management operations required and yielding a performance increase of up to 20% in virtual guests.

kernel-based I/O implementation
> The virtual guest I/O subsystem is now implemented in the kernel, greatly reducing the expense of inter-node communication and memory access by avoiding a significant amount of context switching, and synchronization and communication overhead.

4.4. CPU Scheduler

The *scheduler* is responsible for keeping the CPUs in the system busy. The Linux scheduler implements a number of *scheduling policies*, which determine when and for how long a thread runs on a particular CPU core.

Scheduling policies are divided into two major categories:

1. Realtime policies

   - SCHED_FIFO

   - SCHED_RR

2. Normal policies

   - SCHED_OTHER

   - SCHED_BATCH

   - SCHED_IDLE

4.4.1. Realtime scheduling policies

Realtime threads are scheduled first, and normal threads are scheduled after all realtime threads have been scheduled.

The *realtime* policies are used for time-critical tasks that must complete without interruptions.

SCHED_FIFO
> This policy is also referred to as *static priority scheduling*, because it defines a fixed priority (between 1 and 99) for each thread. The scheduler scans a list of SCHED_FIFO threads in priority order and schedules the highest priority thread that is ready to run. This thread runs until it blocks, exits, or is preempted by a higher priority thread that is ready to run.

> Even the lowest priority realtime thread will be scheduled ahead of any thread with a non-realtime policy; if only one realtime thread exists, the **SCHED_FIFO** priority value does not matter.

SCHED_RR

> A round-robin variant of the **SCHED_FIFO** policy. **SCHED_RR** threads are also given a fixed priority between 1 and 99. However, threads with the same priority are scheduled round-robin style within a certain quantum, or time slice. The `sched_rr_get_interval(2)` system call returns the value of the time slice, but the duration of the time slice cannot be set by a user. This policy is useful if you need multiple thread to run at the same priority.

For more detailed information about the defined semantics of the realtime scheduling policies, consult the IEEE 1003.1 POSIX standard, System Interfaces volume, Realtime section: *http:// pubs.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_08.html*.

Best practice in defining thread priority is to start low and increase priority only when a legitimate latency is identified. Realtime threads are not time-sliced like normal threads; **SCHED_FIFO** threads run until they block, exit, or are pre-empted by a thread with a higher priority. Setting a priority of 99 is therefore not recommended, as this places your process at the same priority level as migration and watchdog threads. If these threads are blocked because your thread goes into a computational loop, they will not be able to run. Uniprocessor systems will eventually lock up in this situation.

In the Linux kernel, the **SCHED_FIFO** policy includes a bandwidth cap mechanism. This protects realtime application programmers from realtime tasks that might monopolize the CPU. This mechanism can be adjusted through the following **/proc** file system parameters:

/proc/sys/kernel/sched_rt_period_us

> Defines the time period to be considered one hundred percent of CPU bandwidth, in microseconds ('us' being the closest equivalent to 'μs' in plain text). The default value is 1000000μs, or 1 second.

/proc/sys/kernel/sched_rt_runtime_us

> Defines the time period to be devoted to running realtime threads, in microseconds ('us' being the closest equivalent to 'μs' in plain text). The default value is 950000μs, or 0.95 seconds.

4.4.2. Normal scheduling policies

There are three normal scheduling policies: **SCHED_OTHER**, **SCHED_BATCH** and **SCHED_IDLE**. However, the **SCHED_BATCH** and **SCHED_IDLE** policies are intended for very low priority jobs, and as such are of limited interest in a performance tuning guide.

SCHED_OTHER, or **SCHED_NORMAL**

> The default scheduling policy. This policy uses the Completely Fair Scheduler (CFS) to provide fair access periods for all threads using this policy. CFS establishes a dynamic priority list partly based on the *niceness* value of each process thread. (Refer to the *Deployment Guide* for more details about this parameter and the **/proc** file system.) This gives users some indirect level of control over process priority, but the dynamic priority list can only be directly changed by the CFS.

4.4.3. Policy Selection

Selecting the correct scheduler policy for an application's threads is not always a straightforward task. In general, realtime policies should be used for time critical or important tasks that need to be scheduled quickly and do not run for extended periods of time. Normal policies will generally yield better data throughput results than realtime policies because they let the scheduler run threads more efficiently (that is, they do not need to reschedule for pre-emption as often).

If you are managing large numbers of threads and are concerned mainly with data throughput (network packets per second, writes to disk, etc.) then use **SCHED_OTHER** and let the system manage CPU utilization for you.

If you are concerned with event response time (latency) then use **SCHED_FIFO**. If you have a small number of threads, consider isolating a CPU socket and moving your threads onto that socket's cores so that there are no other threads competing for time on the cores.

4.5. Tuned IRQs

An interrupt request (IRQ) is a request for service, sent at the hardware level. Interrupts can be sent by either a dedicated hardware line, or across a hardware bus as an information packet (a Message Signaled Interrupt, or MSI).

When interrupts are enabled, receipt of an IRQ prompts a switch to interrupt context. Kernel interrupt dispatch code retrieves the IRQ number and its associated list of registered Interrupt Service Routines (ISRs), and calls each ISR in turn. The ISR acknowledges the interrupt and ignores redundant interrupts from the same IRQ, then queues a deferred handler to finish processing the interrupt and stop the ISR from ignoring future interrupts.

The **/proc/interrupts** file lists the number of interrupts per CPU per I/O device. It displays the IRQ number, the number of that interrupt handled by each CPU core, the interrupt type, and a comma-delimited list of drivers that are registered to receive that interrupt. (Refer to the proc(5) man page for further details: **man 5 proc**)

IRQs have an associated "affinity" property, *smp_affinity*, which defines the CPU cores that are allowed to execute the ISR for that IRQ. This property can be used to improve application performance by assigning both interrupt affinity and the application's thread affinity to one or more specific CPU cores. This allows cache line sharing between the specified interrupt and application threads.

The interrupt affinity value for a particular IRQ number is stored in the associated **/proc/ irq/*IRQ_NUMBER*/smp_affinity** file, which can be viewed and modified by the root user. The value stored in this file is a hexadecimal bit-mask representing all CPU cores in the system.

As an example, to set the interrupt affinity for the Ethernet driver on a server with four CPU cores, first determine the IRQ number associated with the Ethernet driver:

```
# grep eth0 /proc/interrupts
32:   0     140     45        850264      PCI-MSI-edge      eth0
```

Use the IRQ number to locate the appropriate *smp_affinity* file:

```
# cat /proc/irq/32/smp_affinity
f
```

The default value for smp_affinity is **f**, meaning that the IRQ can be serviced on any of the CPUs in the system. Setting this value to **1**, as follows, means that only CPU 0 can service this interrupt:

```
# echo 1 >/proc/irq/32/smp_affinity
# cat /proc/irq/32/smp_affinity
1
```

Commas can be used to delimit *smp_affinity* values for discrete 32-bit groups. This is required on systems with more than 32 cores. For example, the following example shows that IRQ 40 is serviced on all cores of a 64-core system:

```
# cat /proc/irq/40/smp_affinity
ffffffff,ffffffff
```

To service IRQ 40 on only the upper 32-cores of a 64-core system, you would do the following:

```
# echo 0xffffffff,00000000 > /proc/irq/40/smp_affinity
# cat /proc/irq/40/smp_affinity
ffffffff,00000000
```

> **Note**
>
> On systems that support *interrupt steering*, modifying the $smp\_affinity$ of an IRQ sets up the hardware so that the decision to service an interrupt with a particular CPU is made at the hardware level, with no intervention from the kernel.

Memory

Read this chapter for an overview of the memory management features available in Red Hat Enterprise Linux, and how to use these management features to optimize memory utilization in your system.

5.1. Huge Translation Lookaside Buffer (HugeTLB)

Physical memory addresses are translated to virtual memory addresses as part of memory management. The mapped relationship of physical to virtual addresses is stored in a data structure known as the page table. Since reading the page table for every address mapping would be time consuming and resource-expensive, there is a cache for recently-used addresses. This cache is called the Translation Lookaside Buffer (TLB).

However, the TLB can only cache so many address mappings. If a requested address mapping is not in the TLB, the page table must still be read to determine the physical to virtual address mapping. This is known as a "TLB miss". Applications with large memory requirements are more likely to be affected by TLB misses than applications with minimal memory requirements because of the relationship between their memory requirements and the size of the pages used to cache address mappings in the TLB. Since each miss involves reading the page table, it is important to avoid these misses wherever possible.

The Huge Translation Lookaside Buffer (HugeTLB) allows memory to be managed in very large segments so that more address mappings can be cached at one time. This reduces the probability of TLB misses, which in turn improves performance in applications with large memory requirements.

Information about configuring the HugeTLB can be found in the kernel documentation: **`/usr/share/doc/kernel-doc-`*`version`*`/Documentation/vm/hugetlbpage.txt`**

5.2. Huge Pages and Transparent Huge Pages

Memory is managed in blocks known as *pages*. A page is 4096 bytes. 1MB of memory is equal to 256 pages; 1GB of memory is equal to 256,000 pages, etc. CPUs have a built-in *memory management unit* that contains a list of these pages, with each page referenced through a *page table entry*.

There are two ways to enable the system to manage large amounts of memory:

*   Increase the number of page table entries in the hardware memory management unit

*   Increase the page size

The first method is expensive, since the hardware memory management unit in a modern processor only supports hundreds or thousands of page table entries. Additionally, hardware and memory management algorithms that work well with thousands of pages (megabytes of memory) may have difficulty performing well with millions (or even billions) of pages. This results in performance issues: when an application needs to use more memory pages than the memory management unit supports, the system falls back to slower, software-based memory management, which causes the entire system to run more slowly.

Red Hat Enterprise Linux 6 implements the second method via the use of *huge pages*.

Simply put, huge pages are blocks of memory that come in 2MB and 1GB sizes. The page tables used by the 2MB pages are suitable for managing multiple gigabytes of memory, whereas the page tables of 1GB pages are best for scaling to terabytes of memory.

Huge pages must be assigned at boot time. They are also difficult to manage manually, and often require significant changes to code in order to be used effectively. As such, Red Hat Enterprise Linux 6 also implemented the use of *transparent huge pages* (THP). THP is an abstraction layer that automates most aspects of creating, managing, and using huge pages.

THP hides much of the complexity in using huge pages from system administrators and developers. As the goal of THP is improving performance, its developers (both from the community and Red Hat) have tested and optimized THP across a wide range of systems, configurations, applications, and workloads. This allows the default settings of THP to improve the performance of most system configurations.

Note that THP can currently only map anonymous memory regions such as heap and stack space.

5.3. Capacity Tuning

Read this section for an outline of memory, kernel and file system capacity, the parameters related to each, and the trade-offs involved in adjusting these parameters.

To set these values temporarily during tuning, echo the desired value to the appropriate file in the proc file system. For example, to set *overcommit_memory* temporarily to **1**, run:

```
# echo 1 > /proc/sys/vm/overcommit_memory
```

Note that the path to the parameter in the proc file system varies depending on the system affected by the change.

To set these values persistently, you will need to use the **sysctl** command. Refer to *The proc File System*[1] appendix of the Red Hat Enterprise Linux 6 *Deployment Guide* for further information.

Capacity-related Memory Tunables

Each of the following parameters is located under **/proc/sys/vm/** in the proc file system.

overcommit_memory
    Defines the conditions that determine whether a large memory request is accepted or denied. There are three possible values for this parameter:

    - **0** — The default setting. The kernel performs heuristic memory overcommit handling by estimating the amount of memory available and failing requests that are blatantly invalid. Unfortunately, since memory is allocated using a heuristic rather than a precise algorithm, this setting can sometimes allow available memory on the system to be overloaded.

    - **1** — The kernel performs no memory overcommit handling. Under this setting, the potential for memory overload is increased, but so is performance for memory-intensive tasks.

    - **2** — The kernel denies requests for memory equal to or larger than the sum of total available swap and the percentage of physical RAM specified in *overcommit_ratio*. This setting is best if you want a lesser risk of memory overcommitment.

---

[1] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html-single/Deployment_Guide/index.html#ch-proc

> **Note**
>
> This setting is only recommended for systems with swap areas larger than their physical memory.

overcommit_ratio

Specifies the percentage of physical RAM considered when *overcommit_memory* is set to **2**. The default value is **50**.

max_map_count

Defines the maximum number of memory map areas that a process may use. In most cases, the default value of **65530** is appropriate. Increase this value if your application needs to map more than this number of files.

nr_hugepages

Defines the number of hugepages configured in the kernel. It is only possible to allocate (or deallocate) hugepages if there are sufficient physically contiguous free pages in the system. Pages reserved by this parameter cannot be used for other purposes. Further information is available from the installed documentation: **/usr/share/doc/kernel-doc-*kernel_version*/Documentation/vm/hugetlbpage.txt**

Capacity-related Kernel Tunables

Each of the following parameters is located under **/proc/sys/kernel/** in the proc file system.

msgmax

Defines the maximum allowable size in bytes of any single message in a message queue. This value must not exceed the size of the queue (*msgmnb*). The default value is **65536**.

msgmnb

Defines the maximum size in bytes of a single message queue. The default value is **65536** bytes.

msgmni

Defines the maximum number of message queue identifiers (and therefore the maximum number of queues). The default is **7454**.

shmall

Defines the total amount of shared memory in bytes that can be used on the system at one time. The default value for machines with 64-bit architecture is **4294967296**; for 32-bit architecture the default value is **268435456**.

shmmax

Defines the maximum shared memory segment allowed by the kernel, in bytes. The default value is **68719476736**. Note, however, that the kernel supports values much larger than this.

shmmni

Defines the system-wide maximum number of shared memory segments. The default value is **4096**.

threads-max

> Defines the system-wide maximum number of threads (tasks) to be used by the kernel at one time. The default value on machines with 64-bit architecture is **59344**; for 32-bit architecture, the default value is **127600**.

Capacity-related File System Tunables

Each of the following parameters is located under **/proc/sys/fs/** in the proc file system.

aio-max-nr

> Defines the maximum allowed number of events in all active asynchronous I/O contexts. The default value is **65536**. Note that changing this value does not pre-allocate or resize any kernel data structures.

file-max

> Lists the maximum number of file handles that the kernel allocates. The default value on machines with 64-bit architecture is **374263**; for 32-bit architecture, the default value is **811491**. Raising this value can resolve errors caused by a lack of available file handles.

Out-of-Memory Kill Tunables

Out of Memory (OOM) refers to a computing state where all available memory, including swap space, has been allocated. By default, this situation causes the system to panic and stop functioning as expected. However, setting the */proc/sys/vm/panic_on_oom* parameter to **0** instructs the kernel to call the oom_killer function when OOM occurs. Usually, oom_killer can kill rogue processes and the system survives.

The following parameter can be set on a per-process basis, giving you increased control over which processes are killed by the oom_killer function. It is located under **/proc/pid/** in the proc file system, where *pid* is the process ID number.

oom_adj

> Defines a value from **-16** to **15** that helps determine the *oom_score* of a process. The higher the *oom_score* value, the more likely the process will be killed by the oom_killer. Setting a *oom_adj* value of **-17** disables the oom_killer for that process.

> ### Important
>
> Any processes spawned by an adjusted process will inherit that process's *oom_score*. For example, if an **sshd** process is protected from the oom_killer function, all processes initiated by that SSH session will also be protected. This can affect the oom_killer function's ability to salvage the system if OOM occurs.

5.4. Tuning Virtual Memory

Virtual memory is typically consumed by processes, file system caches, and the kernel. Virtual memory utilization depends on a number of factors, which can be affected by the following parameters:

swappiness

> A value from 0 to 100 which controls the degree to which the system swaps. A high value prioritizes system performance, aggressively swapping processes out of physical memory when they are not active. A low value prioritizes interactivity and avoids swapping processes out of

physical memory for as long as possible, which decreases response latency. The default value is **60**.

`min_free_kbytes`

The minimum number of kilobytes to keep free across the system. This value is used to compute a watermark value for each low memory zone, which are then assigned a number of reserved free pages proportional to their size.

> ⚠️ **Extreme values can break your system**
>
> Be cautious when setting this parameter, as both too-low and too-high values can be damaging.
>
> Setting `min_free_kbytes` too low prevents the system from reclaiming memory. This can result in system hangs and OOM-killing multiple processes.
>
> However, setting this parameter to a value that is too high (5-10% of total system memory) will cause your system to become out-of-memory immediately. Linux is designed to use all available RAM to cache file system data. Setting a high `min_free_kbytes` value results in the system spending too much time reclaiming memory.

`dirty_ratio`

Defines a percentage value. Writeout of dirty data begins (via **pdflush**) when dirty data comprises this percentage of total system memory. The default value is **20**.

`dirty_background_ratio`

Defines a percentage value. Writeout of dirty data begins in the background (via **pdflush**) when dirty data comprises this percentage of total memory. The default value is **10**.

`drop_caches`

Setting this value to **1**, **2**, or **3** causes the kernel to drop various combinations of page cache and slab cache.

1

The system invalidates and frees all page cache memory.

2

The system frees all unused slab cache memory.

3

The system frees all page cache and slab cache memory.

This is a non-destructive operation. Since dirty objects cannot be freed, running **sync** before setting this parameter's value is recommended.

> ⭐ **Important**
>
> Using the `drop_caches` to free memory is not recommended in a production environment.

To set these values temporarily during tuning, echo the desired value to the appropriate file in the proc file system. For example, to set *swappiness* temporarily to **50**, run:

```
# echo 50 > /proc/sys/vm/swappiness
```

To set this value persistently, you will need to use the **sysctl** command. Refer to *The proc File System*[2] appendix of the Red Hat Enterprise Linux 6 *Deployment Guide* for further information.

---

[2] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html-single/Deployment_Guide/index.html#ch-proc

Input/Output

6.1. Features

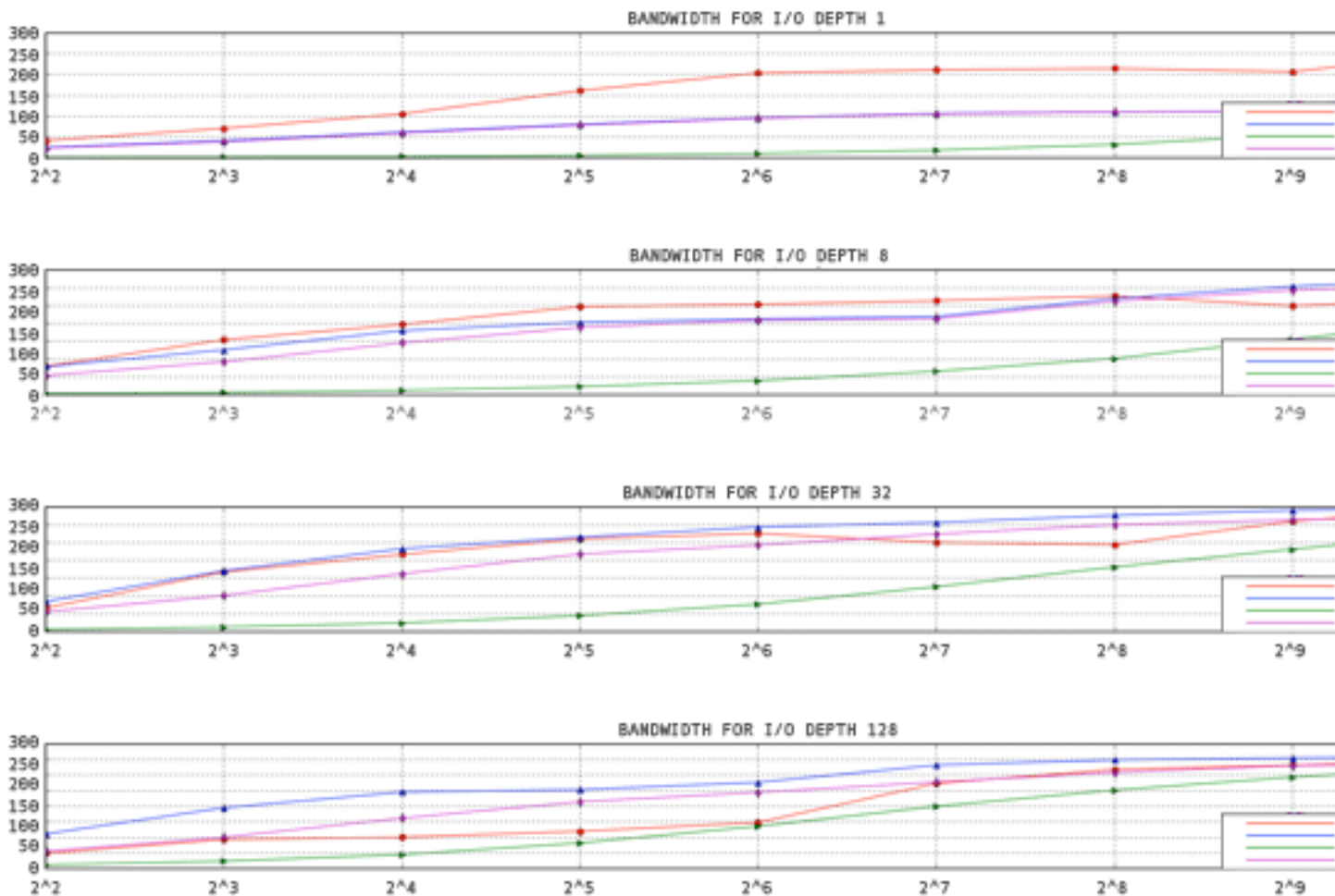Red Hat Enterprise Linux 6 introduces a number of performance enhancements in the I/O stack:

* Solid state disks (SSDs) are now recognized automatically, and the performance of the I/O scheduler is tuned to take advantage of the high I/Os per second (IOPS) that these devices can perform.

* Discard support has been added to the kernel to report unused block ranges to the underlying storage. This helps SSDs with their wear-leveling algorithms. It also helps storage that supports logical block provisioning (a sort of virtual address space for storage) by keeping closer tabs on the actual amount of storage in-use.

* The file system barrier implementation was overhauled in Red Hat Enterprise Linux 6.1 to make it more performant.

* `pdflush` has been replaced by per-backing-device flusher threads, which greatly improves system scalability on configurations with large LUN counts.

6.2. Analysis

Successfully tuning storage stack performance requires an understanding of how data flows through the system, as well as intimate knowledge of the underlying storage and how it performs under varying workloads. It also requires an understanding of the actual workload being tuned.

Whenever you deploy a new system, it is a good idea to profile the storage from the bottom up. Start with the raw LUNs or disks, and evaluate their performance using direct I/O (I/O which bypasses the kernel's page cache). This is the most basic test you can perform, and will be the standard by which you measure I/O performance in the stack. Start with a basic workload generator (such as **aio-stress**) that produces sequential and random reads and writes across a variety of I/O sizes and queue depths.

Following is a graph from a series of **aio-stress** runs, each of which performs four stages: sequential write, sequential read, random write and random read. In this example, the tool is configured to run across a range of record sizes (the x axis) and queue depths (one per graph). The queue depth represents the total number of I/O operations in progress at a given time.

BANDWIDTH FOR I/O DEPTH 1

BANDWIDTH FOR I/O DEPTH 8

BANDWIDTH FOR I/O DEPTH 32

BANDWIDTH FOR I/O DEPTH 128

The y-axis shows the bandwidth in megabytes per second. The x-axis shows the I/O Size in kilobytes.

Figure 6.1. aio-stress output for 1 thread, 1 file

Notice how the throughput line trends from the lower left corner to the upper right. Also note that, for a given record size, you can get more throughput from the storage by increasing the number of I/Os in progress.

By running these simple workloads against your storage, you will gain an understanding of how your storage performs under load. Retain the data generated by these tests for comparison when analyzing more complex workloads.

If you will be using device mapper or md, add that layer in next and repeat your tests. If there is a large loss in performance, ensure that it is expected, or can be explained. For example, a performance drop may be expected if a checksumming raid layer has been added to the stack. Unexpected performance drops can be caused by misaligned I/O operations. By default, Red Hat Enterprise Linux aligns partitions and device mapper metadata optimally. However, not all types of storage report their optimal alignment, and so may require manual tuning.

After adding the device mapper or md layer, add a file system on top of the block device and test against that, still using direct I/O. Again, compare results to the prior tests and ensure that you understand any discrepancies. Direct-write I/O typically performs better on pre-allocated files, so ensure that you pre-allocate files before testing for performance.

Synthetic workload generators that you may find useful include:

- **aio-stress**

- **iozone**

- **fio**

6.3. Tools

There are a number of tools available to help diagnose performance problems in the I/O subsystem. **vmstat** provides a coarse overview of system performance. The following columns are most relevant to I/O: si (swap in), so (swap out), bi (block in), bo (block out), and wa (I/O wait time). si and so are useful when your swap space is on the same device as your data partition, and as an indicator of overall memory pressure. si and bi are read operations, while so and bo are write operations. Each of these categories is reported in kilobytes. wa is idle time; it indicates what portion of the run queue is blocked waiting for I/O complete.

Analyzing your system with **vmstat** will give you an idea of whether or not the I/O subsystem may be responsible for any performance issues. The $free$, $buff$, and $cache$ columns are also worth noting. The $cache$ value increasing alongside the bo value, followed by a drop in $cache$ and an increase in $free$ indicates that the system is performing write-back and invalidation of the page cache.

Note that the I/O numbers reported by **vmstat** are aggregations of all I/O to all devices. Once you have determined that there may be a performance gap in the I/O subsystem, you can examine the problem more closely with **iostat**, which will break down the I/O reporting by device. You can also retrieve more detailed information, such as the average request size, the number of reads and writes per second, and the amount of I/O merging going on.

Using the average request size and the average queue size (**avgqu-sz**), you can make some estimations about how the storage should perform using the graphs you generated when characterizing the performance of your storage. Some generalizations apply: for example, if the average request size is 4KB and the average queue size is 1, throughput is unlikely to be extremely performant.

If the performance numbers do not map to the performance you expect, you can perform more fine-grained analysis with **blktrace**. The **blktrace** suite of utilities gives fine-grained information on how much time is spent in the I/O subsystem. The output from **blktrace** is a set of binary trace files that can be post-processed by other utilities such as **blkparse**.

blkparse is the companion utility to **blktrace**. It reads the raw output from the trace and produces a short-hand textual version.

The following is an example of **blktrace** output:

```
8,64   3        1     0.000000000  4162  Q  RM 73992 + 8 [fs_mark]
8,64   3        0     0.000012707     0  m   N cfq4162S / alloced
8,64   3        2     0.000013433  4162  G  RM 73992 + 8 [fs_mark]
8,64   3        3     0.000015813  4162  P   N [fs_mark]
8,64   3        4     0.000017347  4162  I   R 73992 + 8 [fs_mark]
8,64   3        0     0.000018632     0  m   N cfq4162S / insert_request
8,64   3        0     0.000019655     0  m   N cfq4162S / add_to_rr
8,64   3        0     0.000021945     0  m   N cfq4162S / idle=0
8,64   3        5     0.000023460  4162  U   N [fs_mark] 1
8,64   3        0     0.000025761     0  m   N cfq workload slice:300
8,64   3        0     0.000027137     0  m   N cfq4162S / set_active wl_prio:0 wl_type:2
8,64   3        0     0.000028588     0  m   N cfq4162S / fifo=(null)
8,64   3        0     0.000029468     0  m   N cfq4162S / dispatch_insert
8,64   3        0     0.000031359     0  m   N cfq4162S / dispatched a request
```

```
8,64   3        0     0.000032306      0  m   N cfq4162S / activate rq, drv=1
8,64   3        6     0.000032735   4162  D   R 73992 + 8 [fs_mark]
8,64   1        1     0.004276637      0  C   R 73992 + 8 [0]
```

As you can see, the output is dense and difficult to read. You can tell which processes are responsible for issuing I/O to your device, which is useful, but **blkparse** can give you additional information in an easy-to-digest format in its summary. **blkparse** summary information is printed at the very end of its output:

```
Total (sde):
Reads Queued:           19,       76KiB  Writes Queued:     142,183,   568,732KiB
Read Dispatches:        19,       76KiB  Write Dispatches:   25,440,   568,732KiB
Reads Requeued:          0               Writes Requeued:       125
Reads Completed:        19,       76KiB  Writes Completed:   25,315,   568,732KiB
Read Merges:             0,        0KiB  Write Merges:      116,868,   467,472KiB
IO unplugs:         20,087               Timer unplugs:           0
```

The summary shows average I/O rates, merging activity, and compares the read workload with the write workload. For the most part, however, **blkparse** output is too voluminous to be useful on its own. Fortunately, there are several tools to assist in visualizing the data.

btt provides an analysis of the amount of time the I/O spent in the different areas of the I/O stack. These areas are:

- Q — A block I/O is Queued

- G — Get Request

  A newly queued block I/O was not a candidate for merging with any existing request, so a new block layer request is allocated.

- M — A block I/O is Merged with an existing request.

- I — A request is Inserted into the device's queue.

- D — A request is issued to the Device.

- C — A request is Completed by the driver.

- P — The block device queue is Plugged, to allow the aggregation of requests.

- U — The device queue is Unplugged, allowing the aggregated requests to be issued to the device.

btt breaks down the time spent in each of these areas, as well as the time spent transitioning between them, like so:

- Q2Q — time between requests sent to the block layer

- Q2G — how long it takes from the time a block I/O is queued to the time it gets a request allocated for it

- G2I — how long it takes from the time a request is allocated to the time it is Inserted into the device's queue

- Q2M — how long it takes from the time a block I/O is queued to the time it gets merged with an existing request

- I2D — how long it takes from the time a request is inserted into the device's queue to the time it is actually issued to the device

- M2D — how long it takes from the time a block I/O is merged with an exiting request until the request is issued to the device

- D2C — service time of the request by the device

- Q2C — total time spent in the block layer for a request

You can deduce a lot about a workload from the above table. For example, if Q2Q is much larger than Q2C, that means the application is not issuing I/O in rapid succession. Thus, any performance problems you have may not be at all related to the I/O subsystem. If D2C is very high, then the device is taking a long time to service requests. This can indicate that the device is simply overloaded (which may be due to the fact that it is a shared resource), or it could be because the workload sent down to the device is sub-optimal. If Q2G is very high, it means that there are a lot of requests queued concurrently. This could indicate that the storage is unable to keep up with the I/O load.

Finally, **seekwatcher** consumes **blktrace** binary data and generates a set of plots, including Logical Block Address (LBA), throughput, seeks per second, and I/Os Per Second (IOPS).
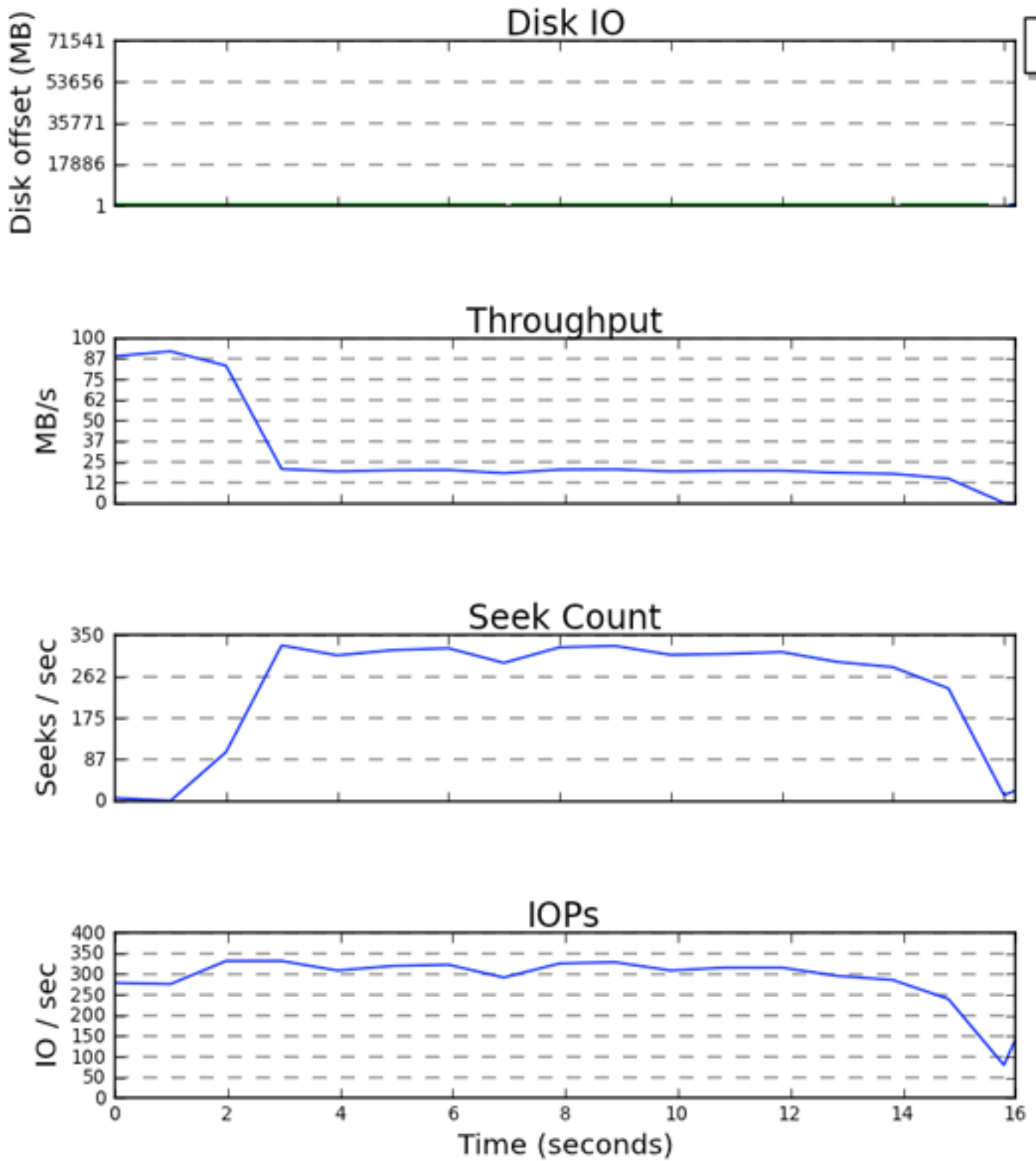
Figure 6.2. Example seekwatcher output

All plots use time as the X axis. The LBA plot shows reads and writes in different colors. It is interesting to note the relationship between the throughput and seeks/sec graphs. For storage that is seek-sensitive, there is an inverse relation between the two plots. The IOPS graph is useful if, for example, you are not getting the throughput you expect from a device, but you are hitting its IOPS limitations.

6.4. Configuration

One of the first decisions you will need to make is which I/O scheduler to use. This section provides an overview of each of the main schedulers to help you decide which is best for your workload.

6.4.1. Completely Fair Queuing (CFQ)

CFQ attempts to provide some fairness in I/O scheduling decisions based on the process which initiated the I/O. Three different scheduling classes are provided: real-time (RT), best-effort (BE), and idle. A scheduling class can be manually assigned to a process with the **ionice** command, or programmatically assigned via the `ioprio_set` system call. By default, processes are placed in the best-effort scheduling class. The real-time and best-effort scheduling classes are further subdivided into eight I/O priorities within each class, priority 0 being the highest and 7 the lowest. Processes in the real-time scheduling class are scheduled much more aggressively than processes in either best-effort or idle, so any scheduled real-time I/O is always performed before best-effort or idle I/O. This means that real-time priority I/O can starve out both the best-effort and idle classes. Best effort scheduling is the default scheduling class, and 4 is the default priority within this class. Processes in the idle scheduling class are only serviced when there is no other I/O pending in the system. Thus, it is very important to only set the I/O scheduling class of a process to idle if I/O from the process is not at all required for making forward progress.

CFQ provides fairness by assigning a time slice to each of the processes performing I/O. During its timeslice, a process may have (by default) up to 8 requests in flight at a time. The scheduler tries to anticipate whether an application will issue more I/O in the near future based on historical data. If it is expected that a process will issue more I/O, then CFQ will idle, waiting for that I/O, even if there is I/O from other processes waiting to be issued.

Because of the idling performed by CFQ, it is often not a good fit for hardware that does not suffer from a large seek penalty, such as fast external storage arrays or solid state disks. If using CFQ on such storage is a requirement (for example, if you would also like to use the cgroup proportional weight I/O scheduler), you will need to tune some settings to improve CFQ performance. Set the following parameters in the files of the same name located in **/sys/block/*device*/queue/iosched/**:

```
slice_idle = 0
quantum = 64
group_idle = 1
```

When `group_idle` is set to 1, there is still the potential for I/O stalls (whereby the back-end storage is not busy due to idling). However, these stalls will be less frequent than idling on every queue in the system.

CFQ is a non-work-conserving I/O scheduler, which means it can be idle even when there are requests pending (as we discussed above). The stacking of non-work-conserving schedulers can introduce large latencies in the I/O path. An example of such stacking is using CFQ on top of a host-based hardware RAID controller. The RAID controller may implement its own non-work-conserving scheduler, thus causing delays at two levels in the stack. Non-work-conserving schedulers operate best when they have as much data as possible to base their decisions on. In the case of stacking such

scheduling algorithms, the bottom-most scheduler will only see what the upper scheduler sends down. Thus, the lower layer will see an I/O pattern that is not at all representative of the actual workload.

Tunables

back_seek_max

Backward seeks are typically bad for performance, as they can incur greater delays in repositioning the heads than forward seeks do. However, CFQ will still perform them, if they are small enough. This tunable controls the maximum distance in KB the I/O scheduler will allow backward seeks. The default is **16** KB.

back_seek_penalty

Because of the inefficiency of backward seeks, a penalty is associated with each one. The penalty is a multiplier; for example, consider a disk head position at 1024KB. Assume there are two requests in the queue, one at 1008KB and another at 1040KB. The two requests are equidistant from the current head position. However, after applying the back seek penalty (default: 2), the request at the later position on disk is now twice as close as the earlier request. Thus, the head will move forward.

fifo_expire_async

This tunable controls how long an async (buffered write) request can go unserviced. After the expiration time (in milliseconds), a single starved async request will be moved to the dispatch list. The default is **250** ms.

fifo_expire_sync

This is the same as the fifo_expire_async tunable, for for synchronous (read and O_DIRECT write) requests. The default is **125** ms.

group_idle

When set, CFQ will idle on the last process issuing I/O in a cgroup. This should be set to **1** when using proportional weight I/O cgroups and setting *slice_idle* to **0** (typically done on fast storage).

group_isolation

If group isolation is enabled (set to **1**), it provides a stronger isolation between groups at the expense of throughput. Generally speaking, if group isolation is disabled, fairness is provided for sequential workloads only. Enabling group isolation provides fairness for both sequential and random workloads. The default value is **0** (disabled). Refer to **Documentation/cgroups/blkio-controller.txt** for further information.

low_latency

When low latency is enabled (set to **1**), CFQ attempts to provide a maximum wait time of 300 ms for each process issuing I/O on a device. This favors fairness over throughput. Disabling low latency (setting it to **0**) ignores target latency, allowing each process in the system to get a full time slice. Low latency is enabled by default.

quantum

The quantum controls the number of I/Os that CFQ will send to the storage at a time, essentially limiting the device queue depth. By default, this is set to **8**. The storage may support much deeper queue depths, but increasing *quantum* will also have a negative impact on latency, especially in the presence of large sequential write workloads.

slice_async

This tunable controls the time slice allotted to each process issuing asynchronous (buffered write) I/O. By default it is set to **40** ms.

slice_idle

> This specifies how long CFQ should idle waiting for a process to issue another I/O. The default is **8** ms.

slice_sync

> This tunable dictates the time slice allotted to a process issuing synchronous (read or direct write) I/O. The default is **100** ms.

6.4.2. Deadline I/O Scheduler

The deadline I/O scheduler attempts to provide a guaranteed latency for requests. It is important to note that the latency measurement only starts when the requests gets down to the I/O scheduler (this is an important distinction, as an application may be put to sleep waiting for request descriptors to be freed). By default, reads are given priority over writes, since applications are more likely to block on read I/O.

Deadline dispatches I/Os in batches. A batch is a sequence of either read or write I/Os which are in increasing LBA order (the one-way elevator). After processing each batch, the I/O scheduler checks to see whether write requests have been starved for too long, and then decides whether to start a new batch of reads or writes. The FIFO list of requests is only checked for expired requests at the start of each batch, and then only for the data direction of that batch. So, if a write batch is selected, and there is an expired read request, that read request will not get serviced until the write batch completes.

Tunables

fifo_batch

> This determines the number of reads or writes to issue in a single batch. The default is **16**. Setting this to a higher value may result in better throughput, but will also increase latency.

front_merges

> You can set this tunable to **0** if you know your workload will never generate front merges. Unless you have measured the overhead of this check, it is advisable to leave it at its default setting (**1**).

read_expire

> This tunable allows you to set the number of milliseconds in which a read request should be serviced. By default, this is set to **500** ms (half a second).

write_expire

> This tunable allows you to set the number of milliseconds in which a write request should be serviced. By default, this is set to **5000** ms (five seconds).

writes_starved

> This tunable controls how many read batches can be processed before processing a single write batch. The higher this is set, the more preference is given to reads.

6.4.3. Noop

The Noop I/O scheduler implements a simple first-in first-out (FIFO) scheduling algorithm. Merging of requests happens at the generic block layer, but is a simple last-hit cache. If a system is CPU-bound and the storage is fast, this can be the best I/O scheduler to use.

Following are the tunables available for the block layer.

/sys/block/sdX/queue tunables

add_random

> In some cases, the overhead of I/O events contributing to the entropy pool for **/dev/random** is measurable. In such cases, it may be desirable to set this value to 0.

`max_sectors_kb`

By default, the maximum request size sent to disk is **512** KB. This tunable can be used to either raise or lower that value. The minimum value is limited by the logical block size; the maximum value is limited by `max_hw_sectors_kb`. There are some SSDs which perform worse when I/O sizes exceed the internal erase block size. In such cases, it is recommended to tune `max_hw_sectors_kb` down to the erase block size. You can test for this using an I/O generator such as **iozone** or **aio-stress**, varying the record size from, for example, **512** bytes to **1** MB.

`nomerges`

This tunable is primarily a debugging aid. Most workloads benefit from request merging (even on faster storage such as SSDs). In some cases, however, it is desirable to disable merging, such as when you want to see how many IOPS a storage back-end can process without disabling read-ahead or performing random I/O.

`nr_requests`

Each request queue has a limit on the total number of request descriptors that can be allocated for each of read and write I/Os. By default, the number is **128**, meaning 128 reads and 128 writes can be queued at a time before putting a process to sleep. The process put to sleep is the next to try to allocate a request, not necessarily the process that has allocated all of the available requests.

If you have a latency-sensitive application, then you should consider lowering the value of `nr_requests` in your request queue and limiting the command queue depth on the storage to a low number (even as low as **1**), so that writeback I/O cannot allocate all of the available request descriptors and fill up the device queue with write I/O. Once `nr_requests` have been allocated, all other processes attempting to perform I/O will be put to sleep to wait for requests to become available. This makes things more fair, as the requests are then distributed in a round-robin fashion (instead of letting one process consume them all in rapid succession). Note that this is only a problem when using the deadline or noop schedulers, as the default CFQ configuration protects against this situation.

`optimal_io_size`

In some circumstances, the underlying storage will report an optimal I/O size. This is most common in hardware and software RAID, where the optimal I/O size is the stripe size. If this value is reported, applications should issue I/O aligned to and in multiples of the optimal I/O size whenever possible.

`read_ahead_kb`

The operating system can detect when an application is reading data sequentially from a file or from disk. In such cases, it performs an intelligent read-ahead algorithm, whereby more data than is requested by the user is read from disk. Thus, when the user next attempts to read a block of data, it will already by in the operating system's page cache. The potential down side to this is that the operating system can read more data from disk than necessary, which occupies space in the page cache until it is evicted because of high memory pressure. Having multiple processes doing false read-ahead would increase memory pressure in this circumstance.

For device mapper devices, it is often a good idea to increase the value of `read_ahead_kb` to a large number, such as **8192**. The reason is that a device mapper device is often made up of multiple underlying devices. Setting this value to the default (**128** KB) multiplied by the number of devices you are mapping is a good starting point for tuning.

`rotational`

Traditional hard disks have been rotational (made up of spinning platters). SSDs, however, are not. Most SSDs will advertise this properly. If, however, you come across a device that does not advertise this flag properly, it may be necessary to set rotational to **0** manually; when rotational

is disabled, the I/O elevator does not use logic that is meant to reduce seeks, since there is little penalty for seek operations on non-rotational media.

rq_affinity

I/O completions can be processed on a different CPU from the one that issued the I/O. Setting *rq_affinity* to **1** causes the kernel to deliver completions to the CPU on which the I/O was issued. This can improve CPU data caching effectiveness.

Storage

Read this chapter for an overview of the file systems supported for use with Red Hat Enterprise Linux, and how to optimize their performance.

7.1. Tuning Considerations for File Systems

There are several tuning considerations common to all file systems: formatting and mount options selected on your system, and actions available to applications that may improve their performance on a given system.

7.1.1. Formatting Options

File system block size

Block size can be selected at `mkfs` time. The range of valid sizes depends on the system: the upper limit is the maximum page size of the host system, while the lower limit depends on the file system used. The default block size is appropriate for most use cases.

If you expect to create many files smaller than the default block size, you can set a smaller block size to minimize the amount of space wasted on disk. Note, however, that setting a smaller block size may limit the maximum size of the file system, and can cause additional runtime overhead, particularly for files greater than the selected block size.

File system geometry

If your system uses striped storage such as RAID5, you can improve performance by aligning data and metadata with the underlying storage geometry at `mkfs` time. For software RAID (LVM or MD) and some enterprise hardware storage, this information is queried and set automatically, but in many cases the administrator must specify this geometry manually with `mkfs` at the command line.

Refer to the *Storage Administration Guide* for further information about creating and maintaining these file systems.

External journals

Metadata-intensive workloads mean that the log section of a journaling file system (such as ext4 and XFS) is updated extremely frequently. To minimize seek time from file system to journal, you can place the journal on dedicated storage. Note, however, that placing the journal on external storage that is slower than the primary file system can nullify any potential advantage associated with using external storage.

> ⚠️ **Warning**
>
> Ensure that your external journal is reliable. The loss of an external journal device will cause file system corruption.

External journals are created at `mkfs` time, with journal devices being specified at mount time. Refer to the `mke2fs(8)`, `mkfs.xfs(8)`, and `mount(8)` man pages for further information.

7.1.2. Mount Options

Barriers

A write barrier is a kernel mechanism used to ensure that file system metadata is correctly written and ordered on persistent storage, even when storage devices with volatile write caches lose power. File systems with write barriers enabled also ensure that any data transmitted via `fsync()` persists across a power outage. Red Hat Enterprise Linux enables barriers by default on all hardware that supports them.

However, enabling write barriers slows some applications significantly; specifically, applications that use `fsync()` heavily, or create and delete many small files. For storage with no volatile write cache, or in the rare case where file system inconsistencies and data loss after a power loss is acceptable, barriers can be disabled by using the *nobarrier* mount option. For further information, refer to the *Storage Administration Guide*.

Access Time (noatime)

Historically, when a file is read, the access time (*atime*) for that file must be updated in the inode metadata, which involves additional write I/O. If accurate *atime* metadata is not required, mount the file system with the *noatime* option to eliminate these metadata updates. In most cases, however, *atime* is not a large overhead due to the default relative atime (or *relatime*) behavior in the Red Hat Enterprise Linux 6 kernel. The *relatime* behavior only updates *atime* if the previous *atime* is older than the modification time (*mtime*) or status change time (*ctime*).

> **Note**
>
> Enabling the *noatime* option also enables *nodiratime* behavior; there is no need to set both *noatime* and *nodiratime*.

Increased read-ahead support

Read-ahead speeds up file access by pre-fetching data and loading it into the page cache so that it can be available earlier in memory instead of from disk. Some workloads, such as those involving heavy streaming of sequential I/O, benefit from high read-ahead values.

The **tuned** tool and the use of LVM striping elevate the read-ahead value, but this is not always sufficient for some workloads. Additionally, Red Hat Enterprise Linux is not always able to set an appropriate read-ahead value based on what it can detect of your file system. For example, if a powerful storage array presents itself to Red Hat Enterprise Linux as a single powerful LUN, the operating system will not treat it as a powerful LUN array, and therefore will not by default make full use of the read-ahead advantages potentially available to the storage.

Use the **blockdev** command to view and edit the read-ahead value. To view the current read-ahead value for a particular block device, run:

```
# blockdev --getra device
```

To modify the read-ahead value for that block device, run the following command. *N* represents the number of 512-byte sectors.

```
# blockdev --setra N device
```

Note that the value selected with the **blockdev** command will not persist between boots. We recommend creating a run level **init.d** script to set this value during boot.

7.1.3. File system maintenance

Discard unused blocks

fstrim identifies blocks that are not in use on a mounted and active file system, and discards these unused blocks so that they can be reused by the storage. This tool is supported for use with ext4 and XFS file systems. Depending on workload and how often data changes on the disk in question, it can be useful to run as a **cron** job on a daily or weekly basis. Note, however, that it is only useful for devices that support discard (most modern solid-state drives and thinly provisioned LUNs).

7.1.4. Application Considerations

Pre-allocation

The ext4, XFS, and GFS2 file systems support efficient space pre-allocation via the `fallocate(2)` glibc call. In cases where files may otherwise become badly fragmented due to write patterns, leading to poor read performance, space pre-allocation can be a useful technique. Pre-allocation marks disk space as if it has been allocated to a file, without writing any data into that space. Until real data is written to a pre-allocated block, read operations will return zeroes.

7.2. Profiles for file system performance

The **tuned-adm** tool allows users to easily swap between a number of profiles that have been designed to enhance performance for specific use cases. The profiles that are particularly useful in improving storage performance are:

latency-performance

> A server profile for typical latency performance tuning. It disables **tuned** and **ktune** power saving mechanisms and enables **sysctl** settings that improve the latency performance of your network I/O.

throughput-performance

> A server profile for typical throughput performance tuning. It disables **tuned** and **ktune** power saving mechanisms, enables **sysctl** settings that improve the throughput performance of your disk and network I/O, and switches to the *deadline* scheduler. This profile is recommended if the system does not have enterprise-class storage, including battery-backed controller cache protection and management of on-disk cache.

enterprise-storage

> A server profile to improve throughput performance for enterprise-sized server configurations. This switches to the *deadline* scheduler and disables certain I/O barriers, dramatically improving throughput. This profile is recommended for systems with enterprise-class storage, including battery-backed controller cache protection and management of on-disk cache.

More information about **tuned-adm** is available from the man page (**man tuned-adm**), or from the *Power Management Guide* at *http://docs.redhat.com*.

7.3. File Systems

7.3.1. The Ext4 File System

The ext4 file system is a scalable extension of the default ext3 file system available in Red Hat Enterprise Linux 5. Ext4 is now the default file system for Red Hat Enterprise Linux 6, and is supported for a maximum file system size of 16 TB and a single file maximum size of 16TB. It also removes the 32000 sub-directory limit present in ext3.

> **Note**
>
> For file systems larger than 16TB, we recommend using a scalable high capacity file system such as XFS. For further information, see *Section 7.4, "The XFS File System"*.

The ext4 file system defaults are optimal for most workloads, but if performance analysis shows that file system behavior is impacting performance, several tuning options are available:

Inode table initialization

For very large file systems, the **mkfs.ext4** process can take a very long time to initialize all inode tables in the file system. This process can be deferred with the *-E lazy_itable_init=1* option. If this is used, kernel processes will continue to initialize the file system after it is mounted. The rate at which this initialization occurs can be controlled with the *-o init_itable=n* option for the **mount** command, where the amount of time spent performing this background initialization is roughly 1/n. The default value for **n** is **10**.

Auto-fsync behavior

Because some applications do not always properly fsync() after renaming an existing file, or truncating and rewriting, ext4 defaults to automatic syncing of files after replace-via-rename and replace-via-truncate operations. This behavior is largely consistent with older ext3 file system behavior. However, fsync() operations can be time consuming, so if this automatic behavior is not required, use the *-o noauto_da_alloc* option with the **mount** command to disable it. This will mean that the application must explicitly use fsync() to ensure data persistence.

Journal I/O priority

By default, journal commit I/O is given a slightly higher priority than normal I/O. This priority can be controlled with the *journal_ioprio=n* option of the **mount** command. The default value is **3**. Valid values range from 0 to 7, with 0 being the highest priority I/O.

For other **mkfs** and tuning options, please see the **mkfs.ext4(8)** and **mount(8)** man pages, as well as the **Documentation/filesystems/ext4.txt** file in the *kernel-doc* package.

7.4. The XFS File System

XFS is a robust and highly-scalable single host 64-bit journaling file system. It is entirely extent-based, so it supports very large file and file system sizes. The number of files an XFS system can hold is limited only by the space available in the file system.

XFS supports metadata journaling, which facilitates quicker crash recovery. The XFS file system can also be defragmented and enlarged while mounted and active. In addition, Red Hat Enterprise Linux 6 supports backup and restore utilities specific to XFS.

XFS uses extent-based allocation, and features a number of allocation schemes such as delayed allocation and explicit pre-allocation. Extent-based allocation provides a more compact and efficient method of tracking used space in a file system, and improves large file performance by reducing fragmentation and the space consumed by metadata. Delayed allocation improves the chance that a file will be written in a contiguous group of blocks, reducing fragmentation and improving performance. Pre-allocation can be used to prevent fragmentation entirely in cases where the application knows the amount of data it needs to write ahead of time.

XFS provides excellent I/O scalability by using b-trees to index all user data and metadata. Object counts grow as all operations on indexes inherit the logarithmic scalability characteristics of the underlying b-trees. Some of the tuning options XFS provides at **mkfs** time vary the width of the b-trees, which changes the scalability characteristics of different subsystems.

7.4.1. Basic tuning for XFS

In general, the default XFS format and mount options are optimal for most workloads; Red Hat recommends that the default values are used unless specific configuration changes are expected to benefit the workload of the file system. If software RAID is in use, the **mkfs.xfs** command automatically configures itself with the correct stripe unit and width to align with the hardware. This may need to be manually configured if hardware RAID is in use.

The **inode64** mount option is highly recommended for multi-terabyte file systems, except where the file system is exported via NFS and legacy 32-bit NFS clients require access to the file system.

The **logbsize** mount option is recommended for file systems that are modified frequently, or in bursts. The default value is **MAX** (32 KB, log stripe unit), and the maximum size is 256 KB. A value of 256 KB is recommended for file systems that undergo heavy modifications.

7.4.2. Advanced tuning for XFS

Before changing XFS parameters, you need to understand why the default XFS parameters are causing performance problems. This involves understanding what your application is doing, and how the file system is reacting to those operations.

Observable performance problems that can be corrected or reduced by tuning are generally caused by file fragmentation or resource contention in the file system. There are different ways to address these problems, and in some cases fixing the problem will require that the application, rather than the file system configuration, be modified.

If you have not been through this process previously, it is recommended that you engage your local Red Hat support engineer for advice.

Optimizing for a large number of files

XFS imposes an arbitrary limit on the number of files that a file system can hold. In general, this limit is high enough that it will never be hit. If you know that the default limit will be insufficient ahead of time, you can increase the percentage of file system space allowed for inodes with the **mkfs.xfs** command. If you encounter the file limit after file system creation (usually indicated by ENOSPC errors when attempting to create a file or directory even though free space is available), you can adjust the limit with the **xfs_growfs** command.

Optimizing for a large number of files in a single directory

Directory block size is fixed for the life of a file system, and cannot be changed except upon initial formatting with `mkfs`. The minimum directory block is the file system block size, which defaults to `MAX` (4 KB, file system block size). In general, there is no reason to reduce the directory block size.

Because the directory structure is b-tree based, changing the block size affects the amount of directory information that can be retrieved or modified per physical I/O. The larger the directory becomes, the more I/O each operation requires at a given block size.

However, when larger directory block sizes are in use, more CPU is consumed by each modification operation compared to the same operation on a file system with a smaller directory block size. This means that for small directory sizes, large directory block sizes will result in lower modification performance. When the directory reaches a size where I/O is the performance-limiting factor, large block size directories perform better.

The default configuration of a 4 KB file system block size and a 4 KB directory block size is best for directories with up to 1-2 million entries with a name length of 20-40 bytes per entry. If your file system requires more entries, larger directory block sizes tend to perform better - a 16 KB block size is best for file systems with 1-10 million directory entries, and a 64 KB block size is best for file systems with over 10 million directory entries.

If the workload uses random directory lookups more than modifications (that is, directory reads are much more common or important than directory writes), then the above thresholds for increasing the block size are approximately one order of magnitude lower.

Optimizing for concurrency

Unlike other file systems, XFS can perform many types of allocation and deallocation operations concurrently provided that the operations are occurring on non-shared objects. Allocation or deallocation of extents can occur concurrently provided that the concurrent operations occur in different allocation groups. Similarly, allocation or deallocation of inodes can occur concurrently provided that the concurrent operations affect different allocation groups.

The number of allocation groups becomes important when using machines with a high CPU count and multi-threaded applications that attempt to perform operations concurrently. If only four allocation groups exist, then sustained, parallel metadata operations will only scale as far as those four CPUs (the concurrency limit provided by the system). For small file systems, ensure that the number of allocation groups is supported by the concurrency provided by the system. For large file systems (tens of terabytes and larger) the default formatting options generally create sufficient allocation groups to avoid limiting concurrency.

Applications must be aware of single points of contention in order to use the parallelism inherent in the structure of the XFS file system. It is not possible to modify a directory concurrently, so applications that create and remove large numbers of files should avoid storing all files in a single directory. Each directory created is placed in a different allocation group, so techniques such as hashing files over multiple sub-directories provide a more scalable storage pattern compared to using a single large directory.

Optimizing for applications that use extended attributes

XFS can store small attributes directly in the inode if space is available in the inode. If the attribute fits into the inode, then it can be retrieved and modified without requiring extra I/O to retrieve separate attribute blocks. The performance differential between in-line and out-of-line attributes can easily be an order of magnitude slower for out-of-line attributes.

For the default inode size of 256 bytes, roughly 100 bytes of attribute space is available depending on the number of data extent pointers also stored in the inode. The default inode size is really only useful for storing a small number of small attributes.

Increasing the inode size at mkfs time can increase the amount of space available for storing attributes in-line. A 512 byte inode size increases the space available for attributes to roughly 350 bytes; a 2 KB inode has roughly 1900 bytes of space available.

There is, however, a limit on the size of the individual attributes that can be stored in-line - there is a maximum size limit of 254 bytes for both the attribute name and the value (that is, an attribute with a name length of 254 bytes and a value length of 254 bytes will stay in-line). Exceeding these size limits forces the attributes out of line, even if there would have been enough space to store all the attributes in the inode.

Optimizing for sustained metadata modifications

The size of the log is the main factor in determining the achievable level of sustained metadata modification. The log device is circular, so before the tail can be overwritten all the modifications in the log must be written to the real locations on disk. This can involve a significant amount of seeking to write back all dirty metadata. The default configuration scales the log size in relation to the overall file system size, so in most cases log size will not require tuning.

A small log device will result in very frequent metadata writeback - the log will constantly be pushing on its tail to free up space and so frequently modified metadata will be frequently written to disk, causing operations to be slow.

Increasing the log size increases the time period between tail pushing events. This allows better aggregation of dirty metadata, resulting in better metadata writeback patterns, and less writeback of frequently modified metadata. The trade-off is that larger logs require more memory to track all outstanding changes in memory.

If you have a machine with limited memory, then large logs are not beneficial because memory constraints will cause metadata writeback long before the benefits of a large log can be realized. In these cases, smaller rather than larger logs will often provide better performance because metadata writeback from the log running out of space is more efficient than writeback driven by memory reclamation.

You should always try to align the log to the underlying stripe unit of the device that contains the file system. **mkfs** does this by default for MD and DM devices, but for hardware RAID it may need to be specified. Setting this correctly avoids all possibility of log I/O causing unaligned I/O and subsequent read-modify-write operations when writing modifications to disk.

Log operation can be further improved by editing mount options. Increasing the size of the in-memory log buffers (**logbsize**) increases the speed at which changes can be written to the log. The default log buffer size is **MAX** (32 KB, log stripe unit), and the maximum size is 256 KB. In general, a larger value results in faster performance. However, under fsync-heavy workloads, small log buffers can be noticeably faster than large buffers with a large stripe unit alignment.

The **delaylog** mount option also improves sustained metadata modification performance by reducing the number of changes to the log. It achieves this by aggregating individual changes in memory before writing them to the log: frequently modified metadata is written to the log periodically instead of on every modification. This option increases the memory usage of tracking dirty metadata and increases the potential lost operations when a crash occurs, but can improve metadata modification speed and scalability by an order of magnitude or more. Use of this option does not reduce data or metadata integrity when `fsync`, `fdatasync` or `sync` are used to ensure data and metadata is written to disk.

7.5. Clustering

Clustered storage provides a consistent file system image across all servers in a cluster, allowing servers to read and write to a single, shared file system. This simplifies storage administration by limiting tasks like installing and patching applications to one file system. A cluster-wide file system also eliminates the need for redundant copies of application data, simplifying backup and disaster recovery.

Red Hat's High Availability Add-On provides clustered storage in conjunction with Red Hat Global File System 2 (part of the Resilient Storage Add-On).

7.5.1. Global File System 2

Global File System 2 (GFS2) is a native file system that interfaces directly with the Linux kernel file system. It allows multiple computers (nodes) to simultaneously share the same storage device in a cluster. The GFS2 file system is largely self-tuning, but manual tuning is possible. This section outlines performance considerations when attempting to tune performance manually.

Since GFS2 uses a global locking mechanism that potentially requires communication between nodes of a cluster, the best performance will be achieved when your system is designed to avoid file and directory contention between these nodes. Some methods of avoiding contention are to:

- Pre-allocate files and directories with **fallocate** where possible, to optimize the allocation process and avoid the need to lock source pages.

- Minimize the areas of the file system that are shared between multiple nodes to minimize cross-node cache invalidation and improve performance. For example, if multiple nodes mount the same file system, but access different sub-directories, you will likely achieve better performance by moving one subdirectory to a separate file system.

- Select an optimal resource group size and number. This depends on typical file sizes and available free space on the system, and affects the likelihood that multiple nodes will attempt to use a resource group simultaneously. Too many resource groups can slow block allocation while allocation space is located, while too few resource groups can cause lock contention during deallocation. It is generally best to test multiple configurations to determine which is best for your workload.

However, contention is not the only issue that can affect GFS2 file system performance. Other best practices to improve overall performance are to:

- Select your storage hardware according to the expected I/O patterns from cluster nodes and the performance requirements of the file system.

- Use solid-state storage where possible to lower seek time.

- Where possible, create temporary files on the local file system.

- Create an appropriately-sized file system for your workload, and ensure that the file system is never at more than 80% capacity. Smaller file systems will have proportionally shorter backup times, and require less time and memory for file system checks, but are subject to high fragmentation if they are too small for their workload.

- Set larger journal sizes for metadata-intensive workloads, or when journaled data is in use. Although this uses more memory, it improves performance because more journaling space is available to store data before a write is necessary.

- Ensure that clocks on GFS2 nodes are synchronized to avoid issues with networked applications. We recommend using NTP (Network Time Protocol).

- Unless file or directory access times are critical to the operation of your application, mount the file system with the `noatime` and `nodiratime` mount options.

> **Note**
>
> Red Hat strongly recommends the use of the `noatime` option with GFS2.

- If you need to use quotas, try to reduce the frequency of quota synchronization transactions or use fuzzy quota synchronization to prevent performance issues arising from constant quota file updates.

> **Note**
>
> Fuzzy quota accounting can allow users and groups to slightly exceed their quota limit. To minimize this issue, GFS2 dynamically reduces the synchronization period as a user or group approaches its quota limit.

The guide to *Global File System 2*[1] contains more detailed information about each of these aspects of GFS2 performance tuning.

[1] http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Global_File_System_2/

Networking

Over time, Red Hat Enterprise Linux's network stack has been upgraded with numerous automated optimization features. For most workloads, the auto-configured network settings provide optimized performance.

In most cases, networking performance problems are actually caused by a malfunction in hardware or faulty infrastructure. Such causes are beyond the scope of this document; the performance issues and solutions discussed in this chapter are useful in optimizing perfectly functional systems.

Networking is a delicate subsystem, containing different parts with sensitive connections. This is why the open source community and Red Hat invest much work in implementing ways to automatically optimize network performance. As such, given most workloads, you may never even need to reconfigure networking for performance.

8.1. Network Performance Enhancements

Red Hat Enterprise Linux 6.1 provided the following network performance enhancements:

Receive Packet Steering (RPS)

RPS enables a single NIC **rx** queue to have its receive **softirq** workload distributed among several CPUs. This helps prevent network traffic from being bottlenecked on a single NIC hardware queue.

To enable RPS, specify the target CPU names in **/sys/class/net/*ethX*/queues/*rx-N*/rps_cpus**, replacing ***ethX*** with the NIC's corresponding device name (for example, **eth1**, **eth2**) and ***rx-N*** with the specified NIC receive queue. This will allow the specified CPUs in the file to process data from queue ***rx-N*** on ***ethX***. When specifying CPUs, consider the queue's *cache affinity* [1].

Receive Flow Steering

RFS is an extension of RPS, allowing the administrator to configure a hash table that is populated automatically when applications receive data and are interrogated by the network stack. This determines which applications are receiving each piece of network data (based on source:destination network information).

Using this information, the network stack can schedule the most optimal CPU to receive each packet. To configure RFS, use the following tunables:

/proc/sys/net/core/rps_sock_flow_entries
> This controls the maximum number of sockets/flows that the kernel can steer towards any specified CPU. This is a system-wide, shared limit.

/sys/class/net/*ethX*/queues/*rx-N*/rps_flow_cnt
> This controls the maximum number of sockets/flows that the kernel can steer for a specified receive queue (***rx-N***) on a NIC (***ethX***). Note that sum of all per-queue values for this tunable on all NICs should be equal or less than that of **/proc/sys/net/core/rps_sock_flow_entries**.

---

[1] Ensuring cache affinity between a CPU and a NIC means configuring them to share the same L2 cache. For more information, refer to *Section 8.3, "Overview of Packet Reception"*.

Unlike RPS, RFS allows both the receive queue and the application to share the same CPU when processing packet flows. This can result in improved performance in some cases. However, such improvements are dependent on factors such as cache hierarchy, application load, and the like.

getsockopt support for TCP thin-streams

Thin-stream is a term used to characterize transport protocols wherein applications send data at such a low rate that the protocol's retransmission mechanisms are not fully saturated. Applications that use thin-stream protocols typically transport via reliable protocols like TCP; in most cases, such applications provide very time-sensitive services (for example, stock trading, online gaming, control systems).

For time-sensitive services, packet loss can be devastating to service quality. To help prevent this, the `getsockopt` call has been enhanced to support two extra options:

TCP_THIN_DUPACK
    This Boolean enables dynamic triggering of retransmissions after one dupACK for thin streams.

TCP_THIN_LINEAR_TIMEOUTS
    This Boolean enables dynamic triggering of linear timeouts for thin streams.

Both options are specifically activated by the application. For more information about these options, refer to `file:///usr/share/doc/kernel-doc-version/Documentation/networking/ip-sysctl.txt`. For more information about thin-streams, refer to `file:///usr/share/doc/kernel-doc-version/Documentation/networking/tcp-thin.txt`.

Transparent Proxy (TProxy) support

The kernel can now handle non-locally bound IPv4 TCP and UDP sockets to support transparent proxies. To enable this, you will need to configure iptables accordingly. You will also need to enable and configure policy routing properly.

For more information about transparent proxies, refer to `file:///usr/share/doc/kernel-doc-version/Documentation/networking/tproxy.txt`.

8.2. Optimized Network Settings

Performance tuning is usually done in a pre-emptive fashion. Often, we adjust known variables before running an application or deploying a system. If the adjustment proves to be ineffective, we try adjusting other variables. The logic behind such thinking is that *by default*, the system is not operating at an optimal level of performance; as such, we *think* we need to adjust the system accordingly. In some cases, we do so via calculated guesses.

As mentioned earlier, the network stack is mostly self-optimizing. In addition, effectively tuning the network requires a thorough understanding not just of how the network stack works, but also of the specific system's network resource requirements. Incorrect network performance configuration can actually lead to degraded performance.

For example, consider the *bufferfloat problem*. Increasing buffer queue depths results in TCP connections that have congestion windows larger than the link would otherwise allow (due to deep buffering). However, those connections also have huge RTT values since the frames spend so much time in-queue. This, in turn, actually results in sub-optimal output, as it would become impossible to detect congestion.

When it comes to network performance, it is advisable to keep the default settings *unless* a particular performance issue becomes apparent. Such issues include frame loss, significantly reduced

throughput, and the like. Even then, the best solution is often one that results from meticulous study of the problem, rather than simply tuning settings upward (increasing buffer/queue lengths, reducing interrupt latency, etc).

To properly diagnose a network performance problem, use the following tools:

netstat

> A command-line utility that prints network connections, routing tables, interface statistics, masquerade connections and multicast memberships. It retrieves information about the networking subsystem from the **/proc/net/** file system. These files include:
>
> - **/proc/net/dev** (device information)
>
> - **/proc/net/tcp** (TCP socket information)
>
> - **/proc/net/unix** (Unix domain socket information)
>
> For more information about **netstat** and its referenced files from **/proc/net/**, refer to the **netstat** man page: **man netstat**.

dropwatch

> A monitoring utility that monitors packets dropped by the kernel. For more information, refer to the **dropwatch** man page: **man dropwatch**.

ip

> A utility for managing and monitoring routes, devices, policy routing, and tunnels. For more information, refer to the **ip** man page: **man ip**.

ethtool

> A utility for displaying and changing NIC settings. For more information, refer to the **ethtool** man page: **man ethtool**.

/proc/net/snmp

> A file that displays ASCII data needed for the IP, ICMP, TCP, and UDP management information bases for an **snmp** agent. It also displays real-time UDP-lite statistics.

The *SystemTap Beginners Guide* also contains several sample scripts you can use to profile and monitor network performance. For more information, refer to:

http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/SystemTap_Beginners_Guide/index.html

After collecting relevant data on a network performance problem, you should be able to formulate a theory — and, hopefully, a solution. [2] For example, an increase in UDP input errors in **/proc/net/snmp** indicates that one or more socket receive queues are full when the network stack attempts to queue new frames into an application's socket.

This indicates that packets are bottlenecked at *at least* one socket queue, which means either the socket queue drains packets too slowly, or packet volume is too large for that socket queue. If it is the latter, then verify the logs of any network-intensive application for lost data -- to resolve this, you would need to optimize or reconfigure the offending application.

---

[2] *Section 8.3, "Overview of Packet Reception"* contains an overview of packet travel, which should help you locate and map bottleneck-prone areas in the network stack.

Socket receive buffer size

Socket send and receive sizes are dynamically adjusted, so they rarely need to be manually edited. If further analysis, such as the analysis presented in the SystemTap network example, `sk_stream_wait_memory.stp`, suggests that the socket queue's drain rate is too slow, then you can increase the depth of the application's socket queue. To do so, increase the size of receive buffers used by sockets by configuring either of the following values:

rmem_default

>   A kernel parameter that controls the *default* size of receive buffers used by sockets. To configure this, run the following command:

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

>   Replace *N* with the desired buffer size, in bytes. To determine the value for this kernel parameter, view **/proc/sys/net/core/rmem_default**. Bear in mind that the value of **rmem_default** should be no greater than **rmem_max** (**/proc/sys/net/core/rmem_max**); if need be, increase the value of **rmem_max**.

SO_RCVBUF

>   A socket option that controls the *maximum* size of a socket's receive buffer, in bytes. For more information on **SO_RCVBUF**, refer to the man page for more details: **man 7 socket**.
>
>   To configure **SO_RCVBUF**, use the **setsockopt** utility. You can retrieve the current **SO_RCVBUF** value with **getsockopt**. For more information using both utilities, refer to the **setsockopt** man page: **man setsockopt**.

8.3. Overview of Packet Reception

To better analyze network bottlenecks and performance issues, you need to understand how packet reception works. Packet reception is important in network performance tuning because the receive path is where frames are often lost. Lost frames in the receive path can cause a significant penalty to network performance.
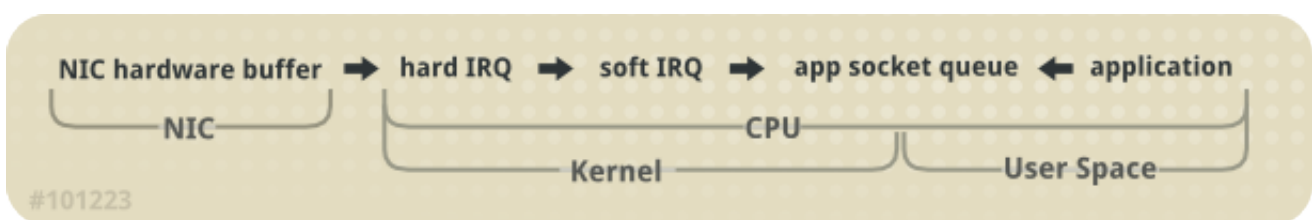

Figure 8.1. Network receive path diagram

The Linux kernel receives each frame and subjects it to a four-step process:

1. *Hardware Reception*: the *network interface card* (NIC) receives the frame on the wire. Depending on its driver configuration, the NIC transfers the frame either to an internal hardware buffer memory or to a specified ring buffer.

2. *Hard IRQ*: the NIC asserts the presence of a net frame by interrupting the CPU. This causes the NIC driver to acknowledge the interrupt and schedule the *soft IRQ operation*.

3. *Soft IRQ*: this stage implements the actual frame-receiving process, and is run in **softirq** context. This means that the stage pre-empts all applications running on the specified CPU, but still allows hard IRQs to be asserted.

In this context (running on the same CPU as hard IRQ, thereby minimizing locking overhead), the kernel actually removes the frame from the NIC hardware buffers and processes it through the network stack. From there, the frame is either forwarded, discarded, or passed to a target listening socket.

When passed to a socket, the frame is appended to the application that owns the socket. This process is done iteratively until the NIC hardware buffer runs out of frames, or until the *device weight* (`dev_weight`). For more information about device weight, refer to *Section 8.4.1, "NIC Hardware Buffer"*

4. *Application receive*: the application receives the frame and dequeues it from any owned sockets via the standard POSIX calls (`read`, `recv`, `recvfrom`). At this point, data received over the network no longer exists on the network stack.

CPU/cache affinity

To maintain high throughput on the receive path, it is recommended that you keep the L2 cache *hot*. As described earlier, network buffers are received on the same CPU as the IRQ that signaled their presence. This means that buffer data will be on the L2 cache of that receiving CPU.

To take advantage of this, place process affinity on applications expected to receive the most data on the NIC that shares the same core as the L2 cache. This will maximize the chances of a cache hit, and thereby improve performance.

8.4. Resolving Common Queuing/Frame Loss Issues

By far, the most common reason for frame loss is a *queue overrun*. The kernel sets a limit to the length of a queue, and in some cases the queue fills faster than it drains. When this occurs for too long, frames start to get dropped.

As illustrated in *Figure 8.1, "Network receive path diagram"*, there are two major queues in the receive path: the NIC hardware buffer and the socket queue. Both queues need to be configured accordingly to protect against queue overruns.

8.4.1. NIC Hardware Buffer

The NIC fills its hardware buffer with frames; the buffer is then drained by the `softirq`, which the NIC asserts via an interrupt. To interrogate the status of this queue, use the following command:

```
ethtool -S ethX
```

Replace `ethX` with the NIC's corresponding device name. This will display how many frames have been dropped within `ethX`. Often, a drop occurs because the queue runs out of buffer space in which to store frames.

There are different ways to address this problem, namely:

Input traffic
> You can help prevent queue overruns by slowing down input traffic. This can be achieved by filtering, reducing the number of joined multicast groups, lowering broadcast traffic, and the like.

Queue length
> Alternatively, you can also increase the queue length. This involves increasing the number of buffers in a specified queue to whatever maximum the driver will allow. To do so, edit the `rx`/`tx` ring parameters of `ethX` using:

```
ethtool --set-ring ethX
```

Append the appropriate **rx** or **tx** values to the aforementioned command. For more information, refer to **man ethtool**.

Device weight

You can also increase the rate at which a queue is drained. To do this, adjust the NIC's *device weight* accordingly. This attribute refers to the maximum number of frames that the NIC can receive before the **softirq** context has to yield the CPU and reschedule itself. It is controlled by the **/proc/sys/net/core/dev_weight** variable.

Most administrators have a tendency to choose the third option. However, keep in mind that there are consequences for doing so. Increasing the number of frames that can be received from a NIC in one iteration implies extra CPU cycles, during which no applications can be scheduled on that CPU.

8.4.2. Socket Queue

Like the NIC hardware queue, the socket queue is filled by the network stack from the **softirq** context. Applications then drain the queues of their corresponding sockets via calls to **read**, **recvfrom**, and the like.

To monitor the status of this queue, use the **netstat** utility; the **Recv-Q** column displays the queue size. Generally speaking, overruns in the socket queue are managed in the same way as NIC hardware buffer overruns (i.e. *Section 8.4.1, "NIC Hardware Buffer"*):

Input traffic

The first option is to slow down input traffic by configuring the rate at which the queue fills. To do so, either filter frames or pre-emptively drop them. You can also slow down input traffic by lowering the NIC's device weight[3].

Queue depth

You can also avoid socket queue overruns by increasing the queue depth. To do so, increase the value of either the **rmem_default** kernel parameter or the **SO_RCVBUF** socket option. For more information on both, refer to *Section 8.2, "Optimized Network Settings"*.

Application call frequency

Whenever possible, optimize the application to perform calls more frequently. This involves modifying or reconfiguring the network application to perform more frequent POSIX calls (such as **recv**, **read**). In turn, this allows an application to drain the queue faster.

For many administrators, increasing the queue depth is the preferable solution. This is the easiest solution, but it may not always work long-term. As networking technologies get faster, socket queues will continue to fill more quickly. Over time, this means having to re-adjust the queue depth accordingly.

The best solution is to enhance or configure the application to drain data from the kernel more quickly, even if it means queuing the data in application space. This lets the data be stored more flexibly, since it can be swapped out and paged back in as needed.

---

[3] Device weight is controlled via **/proc/sys/net/core/dev_weight**. For more information about device weight and the implications of adjusting it, refer to *Section 8.4.1, "NIC Hardware Buffer"*.

8.5. Multicast Considerations

When multiple applications listen to a multicast group, the kernel code that handles multicast frames is required by design to duplicate network data for each individual socket. This duplication is time-consuming and occurs in the `softirq` context.

Adding multiple listeners on a single multicast group therefore has a direct impact on the `softirq` context's execution time. Adding a listener to a multicast group implies that the kernel must create an additional copy for each frame received for that group.

The effect of this is minimal at low traffic volume and small listener numbers. However, when multiple sockets listen to a high-traffic multicast group, the increased execution time of the `softirq` context can lead to frame drops at both the network card and the socket queue. Increased `softirq` runtimes translate to reduced opportunity for applications to run on heavily-loaded systems, so the rate at which multicast frames are lost increases as the number of applications listening to a high-volume multicast group increases.

Resolve this frame loss by optimizing your socket queues and NIC hardware buffers, as described in *Section 8.4.2, "Socket Queue"* or *Section 8.4.1, "NIC Hardware Buffer"*. Alternatively, you can optimize an application's socket use; to do so, configure the application to control a single socket and disseminate the received network data quickly to other user-space processes.

Appendix A. Revision History

Revision 1-0   Friday December 02 2011   Laura Bailey *lbailey@redhat.com*

Release for GA of Red Hat Enterprise Linux 6.2.