Red Hat Enterprise Linux 6

Performance Tuning Guide

Optimizing subsystem throughput in Red Hat Enterprise Linux 6

Russell Doty

Neil Horman

Sanjay Rao

Red Hat Enterprise Linux 6 Performance Tuning Guide
Optimizing subsystem throughput in Red Hat Enterprise Linux 6
Edition 0

Author	Russell Doty	rdoty@redhat.com
Author	Neil Horman	
Author	Sanjay Rao	srao@redhat.com
Editor	Don Domingo	ddomingo@redhat.com

1801 Varsity Drive
Raleigh, NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701

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 on a testing environment before applying it to a production environment. You should also back up all your data and pre-tuning configurations.

Preface

restore conventions and feedback info later


DRAFT

Overview *[Russell Doty]*

main source: rdoty docs

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 planed configurrations in a secure testing environment before applying it to your production servers. You should also back up all data and pre-tuning configurations.

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.

  Features documentation is suitable for readers with 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, i.e. 3-5 years experience in deploying and managing Linux). The *Performance Tuning Guide* aims to provide as much detail on 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. *[tentative]* History of Red Hat Enterprise Linux Performance Enhancements

source: RHEL-History-Perf-key2.odp

1.3. Horizontal Scalability *[BZ#646691]*

from rdoty "Red Hat Enterprise Linux 6 and Horizontal Scalability"
BZ#646691[2]

Red Hat's efforts in improving the performance of Red Hat Enterprise Linux 6 focuses on *scalability*. Performance-boosting features are evaluated primarily on how they affect the platform's performance in different ends of the workload spectrum — i.e. from the lonely webserver 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.

Specifically, Red Hat focuses on improving Red Hat Enterprise Linux to make it suitable for *horizontal scalability*. The basic 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 rack-mounted servers (aka "pizza box servers") and blade servers. Each standard computer may be as small as a simple 2-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 (e.g. databases) and the small ones are dedicated application servers (e.g. web, mail).

This type of scalability simplifies the growth of your IT infrastructure: a medium-sized business with an appropriate load might only need 2 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 sum, 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] Red Hat Certified Engineer. For more information, refer to *http://www.apac.redhat.com/training/certification/rhce/*.
[2] https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=646691

any other points to add RE horizontal scalability?

1.3.1. Parallel Computing

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

Consider this: most 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 normally contain as many as 32 cores total. These low-cost, high-performance systems brought 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 the 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 1.4, “Distributed Systems [BZ#646692] ”* explores the technologies that enable Red Hat Enterprise Linux's horizontal scalability in greater detail.

1.4. Distributed Systems *[BZ#646692]*

BZ#646692[3]

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 (i.e. in parallel); as such, it is necessary for these tasks to 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

russell: does this item relate to Section 1.4.3, “Management (Converged Networks)”?

The most important duty in distributed computing is the *management* layer. An effective distributed system has the proper technologies to coordinate all its hardware components (i.e. computers and appliances within the environment), efficiently managing communication, storage, and the usage of shared resources.

[3] https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=646692

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

1.4.1. Communication

are 10GbE and Infiniband also fully supported in RHEL6?

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.

IMPORTANT! need verification for next para

However, with shared memory, each application assumes full responsibility for managing data sharing and integrity. Shared memory is consolidated inside a single server; this server is typically used as a communication hub for tasks numbering from a few dozen to several hundred.

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 having systems become communication bottlenecked.

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.

can we add some RH links to Infiniband here?

RoCCE

RDMA over Ethernet (RoCCE) implements Infiniband 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.

remove this statement if Infiniband + 10GbE are confirmed to be fully supported as well; use a blanket statement instead

Red Hat Enterprise Linux 6 fully supports RoCCE.

1.4.2. *[link]* Storage

add links to respective books (Storage Admin Guide, GFS2 Guide, and their subsequent places in this doc

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

IMPORTANT: Please check

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

removed description of local storage, since earlier we discussed how local storage was NOT appropriate for distributed computing.

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 get to 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 (i.e. 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 filesystem, 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.

1.4.3. Management (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*[4]. 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; this need is addressed by *Fibre Channel over Ethernet* (FCoE) and **`Internet SCSI`** (iSCSI).

FCoE

With FCoE, standard fibre channel commands and data packets are transported over a 10GbE physical infrastructure via a specialized *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

compact servers = pizza box servers?

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.

[4] *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.

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.


DRAFT

Red Hat Enterprise Linux 6 Performance Features *[BZ#646693]*

source: larry_shak_perf_summit2010_v3.odp
items herein should only present high-level description of features/specs; the main purpose of this chapter is to present a summary that everyone can read in one place
BZ#646693[1]

2.1. System Overview

source: larry_shak_perf_summit2010_v3.odp, slide 4+
supported processors, memory
showcase link to http://www.redhat.com/rhel/compare/

2.2. *[link]* 64-Bit Support

from rdoty: ***`RHEL 6 Scalability 23Aut10.odt`***
add links to respective sections

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 246GB of memory. This value can grow over time, as Red Hat Engineering introduces more features that enable the platform to use larger blocks of 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

The implementation of *huge pages* in Red Hat Enterprise Linux 6 allows the system to manage memory use efficiently across different memory workloads. With huge pages, an application can scale well from processing gigabyes 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 much of the complexities involved in the use of huge pages.

For more information on huge pages and THP, refer to *Section 6.2, “Huge Pages and Transparent Hugepages [BZ#639792]”*.

NUMA improvements

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

[1] https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=646693

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

For more information about NUMA support in Red Hat Enterprise Linux 6, refer to *Section 4.2, “NUMA and Multi-Core Support [BZ#639780] ”* and *Section 4.1, “CPU and NUMA Topology [BZ#641009] ”*.

2.3. *[link]* Ticket Spinlocks

from rdoty: ***RHEL 6 Scalability 23Aut10.odt***
add links to respective sections

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 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 (i.e. "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. As such, processes on the same NUMA node as the lock having 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.

2.4. *[link]* Dynamic List Structure

from rdoty: ***RHEL 6 Scalability 23Aut10.odt***
add links to respective sections

The operating system requires a set of information on each processor in the system. In Red Hat Enterprise Linux 5, this set of information is allocated to a fixed-size array in memory. Information on each individual processor is 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 are created as well.

A dynamic list structure allows a finer granularity of 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 small systems.

2.5. *[link]* Tickless Kernel

from rdoty: ***RHEL 6 Scalability 23Aut10.odt***
add links to respective sections

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*, i.e. 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 ppossible. 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.

add link to more info on tickless kernels

2.6. *[link]* Control Groups

intro from rdoty: ***RHEL 6 Scalability 23Aut10.odt***
add links to respective sections

Red Hat Enterprise Linux provides many useful options for performance tuning that. 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 1.3.1, "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 par 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 (e.g. I/O and network communications) are shared resources, 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
- I/O
- Network (bandwidth)

Within CPUsets, cgroups allow administrators to configure the number of CPUs, affinity for specific CPUs or nodes [2], 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 for 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*[3].

most updated version of guide is available internally at http://documentation-stage.bne.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/index.html

2.7. *[link]* Storage and File System Improvements

from rdoty: ***RHEL 6 Scalability 23Aut10.odt***
add links to respective sections

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

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

comprehensive coverage of performance improvements relating to storage and file systems, refer to *Chapter 7, Storage [Barry Marson]* .

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 maximum file system size of 1 exabyte, and 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 7.1, "The Ext4 File System [BZ#639796] "*.

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
- 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 7.2, "The XFS File System [BZ#640877] "*.

Large Boot Drives

Red Hat Enterprise Linux 6 also supports *Unified Extensible Firmware Interface* (UEFI), which can be used to replace BIOS (still supported). The BIOS was originally created for the IBM PC; while BIOS has evolved considerably to adapt to modern hardware, UEFI is designed to support new and emerging hardware. 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.

link to more info for UEFI?


DRAFT

Monitoring and Analyzing System Performance

source: larry_shak_perf_summit2010_v3.odp, slide 36+
introduction

3.1. The /proc File System

short intro: add reference to Deployment Guide and kernel-docs(?) for more info

3.2. Gnome and KDE System Monitors

short intro: then add reference to tool's built-in Help system

3.3. Built-in Command-line Monitoring Tools

advantages: can run out of runlevel 5, etc
CLI tools that can monitor CPU, Memory, and process
Enumerate and intro:
top
vmstat
ps
sar

3.4. Application Profilers

briefly discuss profiling
useful for developers monitoring running processes
"we don't document them here; read up on each tool's respective docs

3.4.1. SystemTap

introduce
advantages/disadvantages/suitability
mention Eclipse Callgraph

3.4.2. OProfile

introduce
advantages/disadvantages/suitability
mention Eclipse OProfile plug-in

3.4.3. Valgrind

introduce
advantages/disadvantages/suitability
mention Eclipse Valgrind plug-in

3.5. Red Hat Enterprise MRG

high-level; do not go into detail, as this product's components are documented heavily elsewhere

list lbrindle's documentation on MRG
http://www.redhat.com/mrg/
concentrate more on the RHEL/node aspect of an IT infrastructure running MRG; what components are installed on a RHEL machine therein? am i asking the right question here? :-/

CPU *[Jirka Hladky]*

SME is Jirka Hladky[1] *(soon to be hladky.jiri@gmail.com*[2]
present a more detailed description of RHEL6 performance enhancements relating to CPU

4.1. CPU and NUMA Topology *[BZ#641009]*

source: as noted by JHladky
https://bugzilla.redhat.com/show_bug.cgi?id=641009

4.2. NUMA and Multi-Core Support *[BZ#639780]*

source: larry_shak_perf_summit2010_v3.odp, slide 8+, 100+
per-numa node resources (slide 17+)
...BUT MORE DETAILED.
https://bugzilla.redhat.com/show_bug.cgi?id=639780

4.3. The CPU Scheduler *[BZ#639781]*

source: larry_shak_perf_summit2010_v3.odp, slide 98+
https://bugzilla.redhat.com/show_bug.cgi?id=639781

4.4. Tuned IRQs *[BZ#639782]*

source: larry_shak_perf_summit2010_v3.odp, slide 47+
https://bugzilla.redhat.com/show_bug.cgi?id=639782

4.5. Understanding CPU Statistics *[BZ#639783]*

recommended CPU monitoring tools
when is CPU performance crucial? i.e. CPU needs to be optimized for what kinds of services/servers?
tips for reading and understanding CPU stats
caveats and general tips; "when you see this, that means..."
https://bugzilla.redhat.com/show_bug.cgi?id=639783

4.5.1. Analyzing CPU Cycle Statistics With OProfile

source: larry_shak_perf_summit2010_v3.odp, slide 60+

4.6. Tuning CPU Performance *[BZ#639784]*

tools? something with an interface, perhaps?
tunable setting, values, effects, and short description
https://bugzilla.redhat.com/show_bug.cgi?id=639784

[1] mailto:jhladky@redhat.com
[2] mailto:hladky.jiri@gmail.com


DRAFT

Input/Output *[Sanjay Rao]*

SME is Sanjay Rao[1]
have SME update content from http://documentation-stage.bne.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5/html/IO_Tuning_Guide/index.html for RHEL6, then add here content from IO Tuning Guide for intro
are there any new defaults?
START: from old IO Tuning Guide

The I/O subsystem is a series of processes responsible for moving blocks of data between disk and memory. In general, each task performed by either kernel or user consists of a utility performing any of the following (or combination thereof):

- *Reading* a block of data from disk, moving it to memory
- *Writing* a new block of data from memory to disk

Read or write requests are transformed into *block device requests* that go into a queue. The I/O subsystem then batches similar requests that come within a specific time window and processes them all at once. Block device requests are batched together (into an “extended block device request”) when they meet the following criteria:

- They are the same type of operation (read or write).
- They belong to the same block device (i.e. Read from the same block device, or are written to the same block device.
- Each block device has a set maximum number of sectors allowed per request. As such, the extended block device request should not exceed this limit in order for the merge to occur.
- The block device requests to be merged immediately follow or precede each other.

Read requests are crucial to system performance because a process cannot commence unless its read request is serviced. This latency directly affects a user's perception of how fast a process takes to finish.

Write requests, on the other hand, are serviced by batch by **`pdflush`** kernel threads. Since write requests do not block processes (unlike read requests), they are usually given less priority than read requests.

Read/Write requests can be either *sequential* or *random*. The speed of sequential requests is most directly affected by the transfer speed of a disk drive. Random requests, on the other hand, are most directly affected by disk drive seek time.

END: from old IO Tuning Guide

Over time, the amount of data handled by most systems has grown exponentially. With this growth, the speed at which the I/O subsystem reads and writes data becomes more critical. I/O is by far the most time-consuming process in the system, and therefore the most expensive piece of the enterprise platform puzzle. As such, optimizing I/O is crucial to performance tuning.

I/O optimization starts with understanding the I/O needs of applications hosted on the system. This is followed by thoughtful analysis and configuration of the system's hardware, file system, and operating system.

[1] mailto:srao@redhat.com

5.1. High-Level I/O Configuration *[BZ#639786]*

https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=639786

Whenever possible, performance optimization should start at the hardware planning level; i.e. choosing the right hardware for the system. Most applications work quite well with ext3; however, whenever there are critical performance levels (e.g. minimum transaction completion time and recovery time as stated in a service level agreement), you will need to be more conscientious in selecting a file system for your needs. Once both hardware and file system are in place, you will need to tune both operating system and the application in question accordingly.

Always keep in mind that the objective is to optimize the performance of a system *for a specific application*. With I/O tuning, focus on the average size of I/O transactions to be performed by an application. Study the number of I/O transactions and total throughput for all types of I/O. Studying this data will confirm if a configuration delivers the expected performance; in many cases, sub-optimal I/O configuration can bottleneck the overall performance of a system.

Hardware

Every I/O sub-system has limits on how much data it can process per second. If your system hosts an application that needs to perform many light (i.e. 2k-8k) I/O transactions, it is recommended that you choose storage subsystems that have a fair amount of controller cache, high-speed disks (i.e. 15Kbps or higher, typically SAS or solid-state disks), and low-latency connection to storage. Examples of systems that require such hardware are transaction processing systems.

While these hardware recommendations can improve I/O transaction rates, they come at a steep cost. As such, it is imperative that you identify which hardware choices can provide the best performance gains per dollar. High-speed disks are a primary consideration; normally, the cost of such a disk is directly proportional to its speed. Other specific hardware recommendations include:

Controller cache
: The cache on the storage controller. The controller cache is similar physical memory on a server, and is one of the most expensive pieces in the storage-controller infrastructure. In most cases the controller cache is battery-backed; in such cases, once the data is handed off to the controller cache the server considers the I/O transaction as completed.

  The amount of cache on the controller ranges from 1G to 64G (or higher), depending on the storage vendor. The most important factors to consider when choosing controller cache hardware are:

  - The amount of cache and should be the determining factor in deciding which hardware to choose.
  - The rate at which the controllers can process I/O transactions

Low-latency connection
: The most common types of external storage are based on fibre-channel, ethernet, or PCI direct-connect. The rate at which I/O is processed depends on the connectors used. For example, network-based storage can have significant latency overhead as each packet needs to be processed. However, there are special network cards that offload the network layer processing from the operating system to the hardware; this, in turn, reduces latency.

If the system is expected to process larger I/O blocks (typically in data processing environments), it is recommended that you stripe across controllers to take advantage of collective bandwidth. In typical data processing environments, storage tends to be quite large (i.e. 1TB or higher), so low latency is the primary consideration, followed by disk speed. As most similar environments are read-based, the controller cache can be scaled back to the lower end.

File System

Red Hat Enterprise Linux 6 supports the ext3, ext4, and XFS file systems. Ext4 is the default file system of Red Hat Enterprise Linux 6, and is suitable for most types of workstations and servers. The ext4 file system is extent-based (as opposed to block-based), can support files/file systems up to 16 terabytes in size, and allows an unlimited number of sub-directories.

The XFS file system is a high-performance file system designed to support extremely large file systems, i.e. up to 16 exabytes. It can support file sizes up to 8 exabytes in size, and has a large directory structure limit. XFS is also suitable for systems hosting applications that handle large files and require smooth data transfers.

Note that both ext4 and XFS feature *delayed allocation*, which speeds up I/O at the expense of actual, real-time writes. With delayed allocation, block allocation is delayed until the data is immediately going to be written to disk; as such, application writes to the file system are not guaranteed to be on-disk unless the application issues an **`fsync()`** call afterwards.

For more information about XFS, ext4, and other supported file systems in Red Hat Enterprise Linux 6, refer to the *Storage Administration Guide* at:

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

Operating System

Tuning Red Hat Enterprise Linux 6's I/O can be an ongoing process if the application in question evolves over its lifetime. There are many tools that can be used to do low-level I/O subsystem profiling; the best example of this is **`iozone`**. Refer to *Section 5.2, "IOZone Benchmarks [BZ#639788] "* for more information on this tool.

5.2. IOZone Benchmarks *[BZ#639788]*

source: Woodman,Shakshober_Performance Analys.pdf p.70+
http://www.iozone.org
https://bugzilla.redhat.com/show_bug.cgi?id=639788

The **`iozone`** utility is an award-winning open-source tool developed by community developers William Norcott and Don Capps. It profiles a system's overall I/O performance by performing a variety of I/O transactions, such as read, write, re-read, re-write, and the like.

sanjay: for now i'm assuming iozone is not included (i.e. unsupported) in RHEL6; added admonition below.

Important

At present, **`iozone`** is not included (and therefore, currently unsupported) in Red Hat Enterprise Linux 6. To download the **`iozone`** source, package, or documentation, refer to *http://www.iozone.org*.

To perform a simple **`iozone`** I/O test, run:

```
iozone -a
```

This will run **`iozone`** in *full-automatic mode*. Doing so will profile how the system's I/O fares when handling file of different sizes. For more information about **`iozone`** and its output, refer to **`man iozone`**.

The following graph (*Figure 5.1, "RHEL5 in-cache iozone results for ext3, GFS1, and NFS"*) compares the **iozone** test results for file sizes between 1MB to 4MB and transfer sizes between 1k and 1M for an in-cache-run in Red Hat Enterprise Linux 5. These results are from **iozone** runs for three file system types: ext3, GFS1, and NFS.

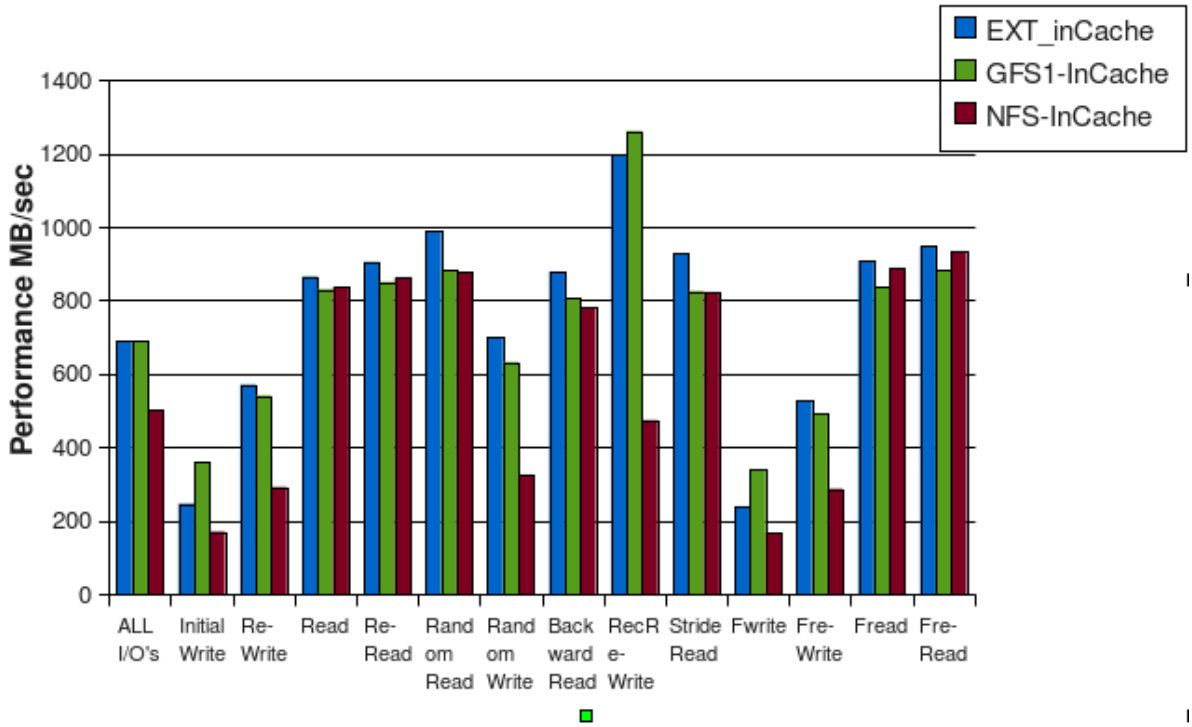

Figure 5.1. RHEL5 in-cache iozone results for ext3, GFS1, and NFS

The *Figure 5.2, "RHEL5 direct I/O in-cache iozone results for ext3, GFS1, and NFS"* graph displays the results for the same test performed direct I/O enabled (for more information about direct I/O, refer to *Section 5.3, "Direct I/O [BZ#639789] "*).

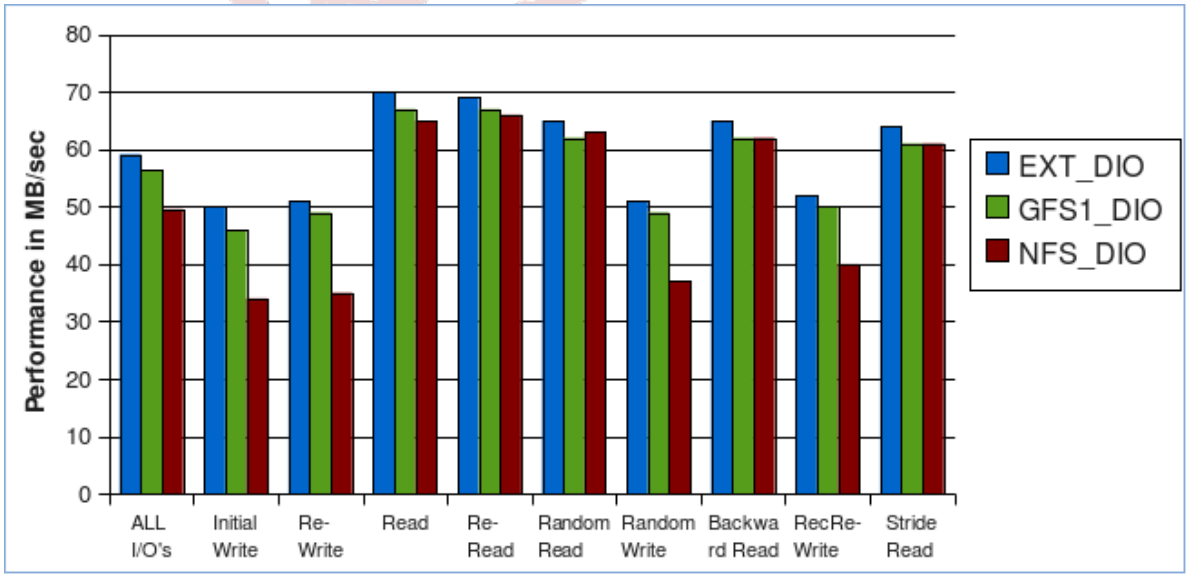

Figure 5.2. RHEL5 direct I/O in-cache iozone results for ext3, GFS1, and NFS

This type of test data can help you profile accurately how the I/O sub-system performs under different conditions, allowing you to make better decisions when configuring I/O.

5.3. Direct I/O [BZ#639789]

source: larry_shak_perf_summit2010_v3.odp, slide 138+
http://www.solarisinternals.com/wiki/index.php/Direct_I/O
https://bugzilla.redhat.com/show_bug.cgi?id=639789

Simply put, direct I/O is a mechanism that allows file systems to perform I/O without going through the file cache. Direct I/O is only used by applications that manage their own caches. Many databases that cache data support direct I/O. In most cases, such databases use direct I/O by default.

Some databases support direct I/O, but do not use it by default. When using such databases, it is recommended that you enable direct I/O and configure the database to use it. This will prevent the database from caching data twice (i.e. in the file cache and database cache).

Enabling direct I/O and configuring databases to support it is especially important in Red Hat Enterprise Linux because all free memory outside the database cache will be used for file caching. This can lead to a lot of page reclamation. Consider the following graph:

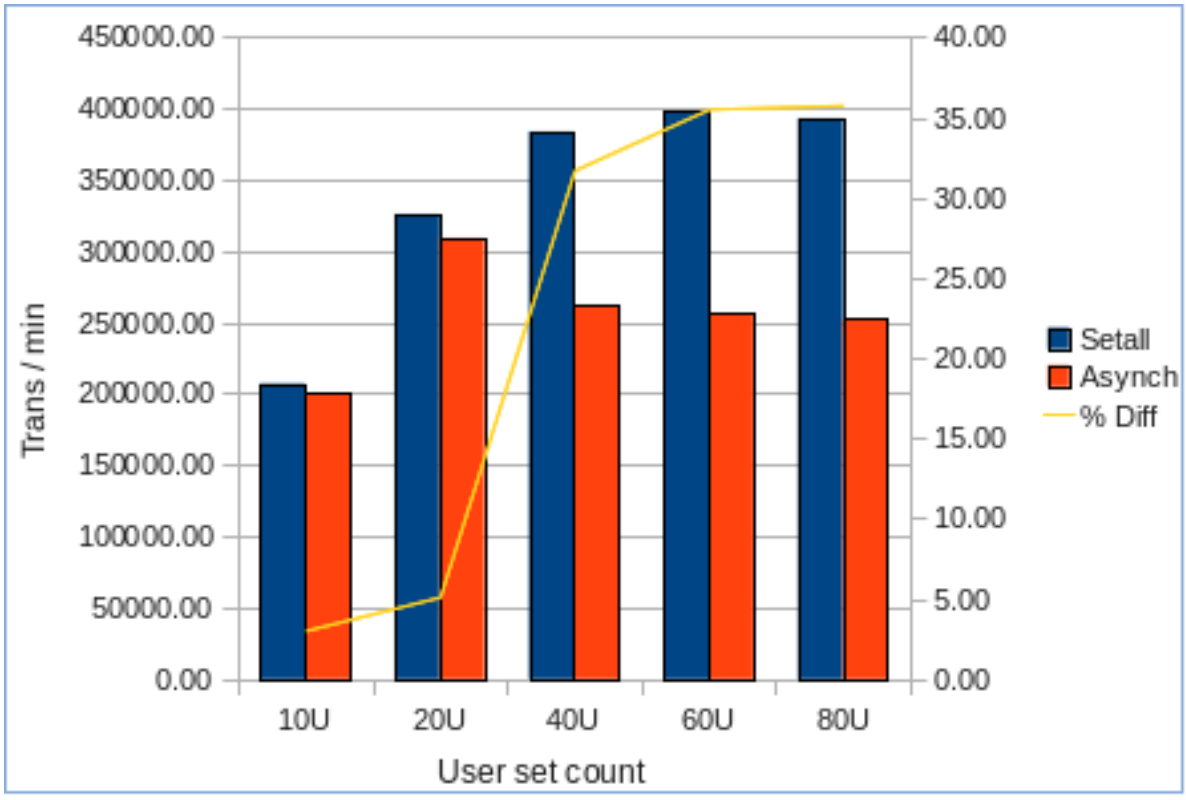

Figure 5.3. Database run without direct I/O

Figure 5.3, "Database run without direct I/O" displays results of an Oracle database I/O test. The blue bars represent the I/O of the system (in terms of transactions per minute) with the database operating in **`setall`** mode; this mode enables both direct I/O and asynchronous I/O. The orange bars represent system I/O with direct I/O disabled and asynchronous I/O enabled.

The results in *Figure 5.3, "Database run without direct I/O"* show that as the number or size of I/O transactions increase, the performance impact of not using direct I/O also increases. This is represented by the yellow line, which shows a significant advantage in I/O performance with direct I/O enabled starting at 40U.

Note

The data collected for *Figure 5.3, "Database run without direct I/O"* was for a single instance. The performance impact of disabling direct I/O can be even greater with more applications running on the same system, as the file cache will be under more pressure then.

5.4. Asynchronous I/O to File Systems *[BZ#639790]*

source: larry_shak_perf_summit2010_v3.odp, slide 138+
https://bugzilla.redhat.com/show_bug.cgi?id=639790
related: http://davmac.org/davpage/linux/async-io.html

When the system is operating in *synchronous mode*, an application waits for an I/O transaction to complete before issuing another. With *asynchronous I/O*, an application no longer needs to wait for an I/O transaction to complete before being able to issue another. This allows an application to issue multiple I/Os to a device and continue with its operation.

Many applications support asynchronous I/O to file systems. Whenever possible, configure your applications to use asynchronous I/O as doing so reduces the I/O waits or stalls commonly seen when running I/O-intensive applications in *synchronous mode*. Consider the followign graph:

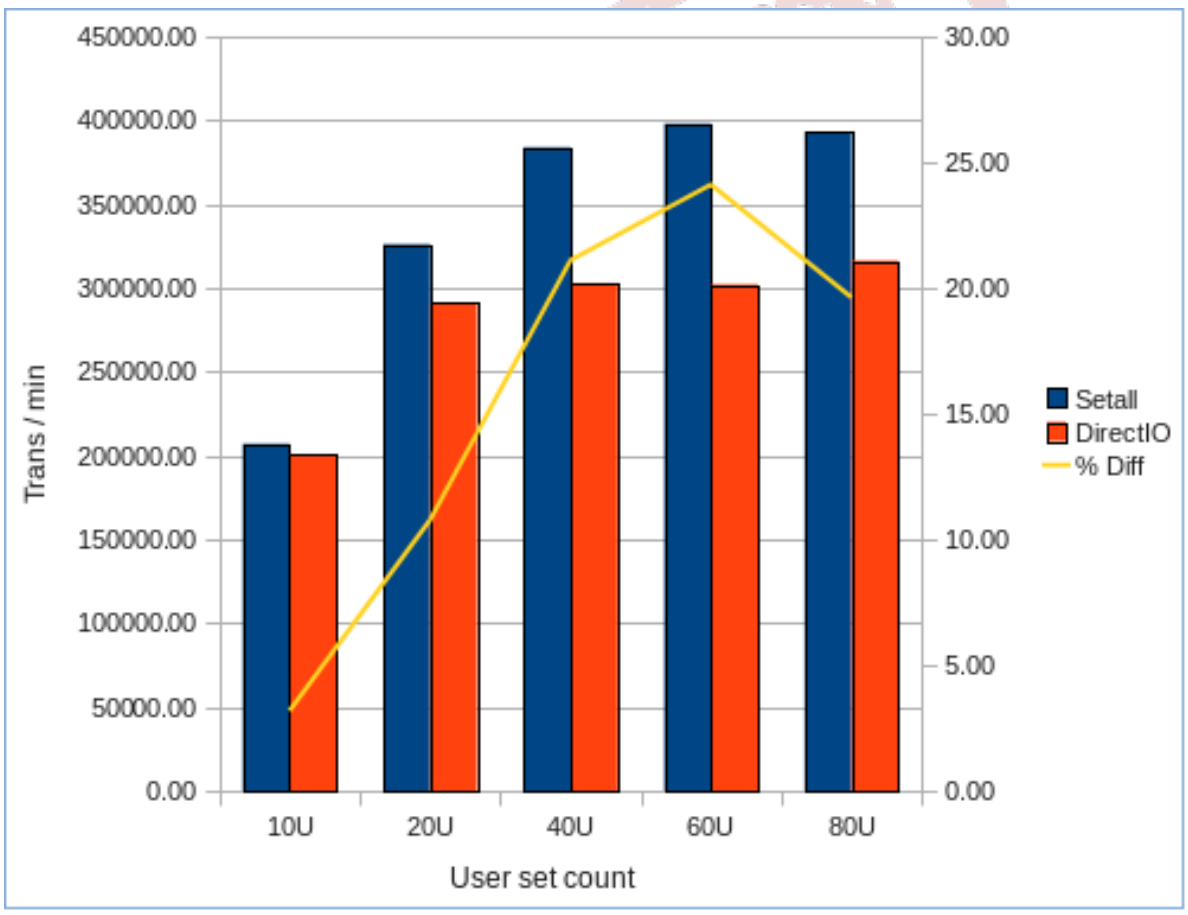

Figure 5.4. Database run with and without asynchronous I/O

Figure 5.4, "Database run with and without asynchronous I/O" displays results of an Oracle database I/O test. The blue bars represent the I/O of the system (in terms of transactions per minute) with the database operating in **`setall`** mode (i.e. both direct I/O and asynchronous I/O enabled). The orange bars represent system I/O with direct I/O enabled in *synchronous mode*.

Like direct I/O, the performance impact of not using asynchronous I/O increases as the volume of I/O also increases. This is represented by the yellow line, which shows a significant advantage in I/O performance with direct I/O enabled starting at 40U.

5.5. I/O Merging *[BZ#640872]*

source: mentioned by Sanjay during chat 2010-10-06_sanjayrao-io_subsystem-intro.html
https://bugzilla.redhat.com/show_bug.cgi?id=640872

Most storage systems can only process a limited number of I/O transactions per second. This puts a limit on the total throughput of each system, depending of the size of each I/O transaction.

For example, a system with a storage device that can process 100 I/O transactions per second has a total throughput of 200 kb/s for a 2KB transfer size. If the transfer size were to be doubled to 4KB, the total throughput of the storage sub-system would also double to 400 kb/s.

This increase is accomplished via *I/O merging*. To do this, the I/O subsystem allows a storage device to merge contiguous I/O streams into larger I/O transactions, thereby resulting in fewer (albeit larger-sized) I/O transactions.

Sanjay: i could not find any other resources/links to help me understand I/O merging more. can you please provide these? i feel

There are three ways to enable I/O merging:

Block device read-ahead settings

Given a heuristic trigger, most block devices can predict which blocks will need to be fetched well before they are due for reading. The *read-ahead* setting allows you to configure how much data (in byte sectors) the block device can fetch and, subsequently, merge during each I/O transfer.

To configure this setting, run the following command:

```
blockdev --setra N
```

Here, ***N*** is the required read-ahead size in 512-byte sectors. For more information about **blockdev**, refer to its **man** page.

Application read-ahead settings

Most databases (and other I/O-intensive applications) allow you to configure transfer size for reads. The parameters for configuring read-ahead varies with each database.

I/O schedulers

I/O schedulers like **cfq** and **deadline** perform I/O merging to reduce the number of I/O transactions. For more information on I/O elevators, refer to *Section 5.7, “Schedulers [BZ#639785]”*.

5.6. I/O Alignment *[BZ#640874]*

https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=640874

Most storage devices and controllers nowadays have a sector size of 4K, as opposed to the traditional 512 bytes. On controllers that support RAID, the 4K sector size allows the controller to align with the RAID chunk/stripe size, which in turn improves performance. In addition, LVM automatically also manages I/O alignment.

However, if a device is partitioned using **fdisk** or **parted**, you may need to manually align sector sizes for older platforms (e.g. Red Hat Enterprise Linux 5). The track size on older platforms was 63 sectors, with which 512-byte sectors can align easily; however, 4K-sector devices on 63-sector track sizes are highly prone to mis-alignment.

Mis-alignment can cause a significant performance penalty for sequential I/O with transfer sizes of 8K and above. To avoid this, choose a 56-sector track during partitioning to align tracks with 4K devices. Consider the following graph:

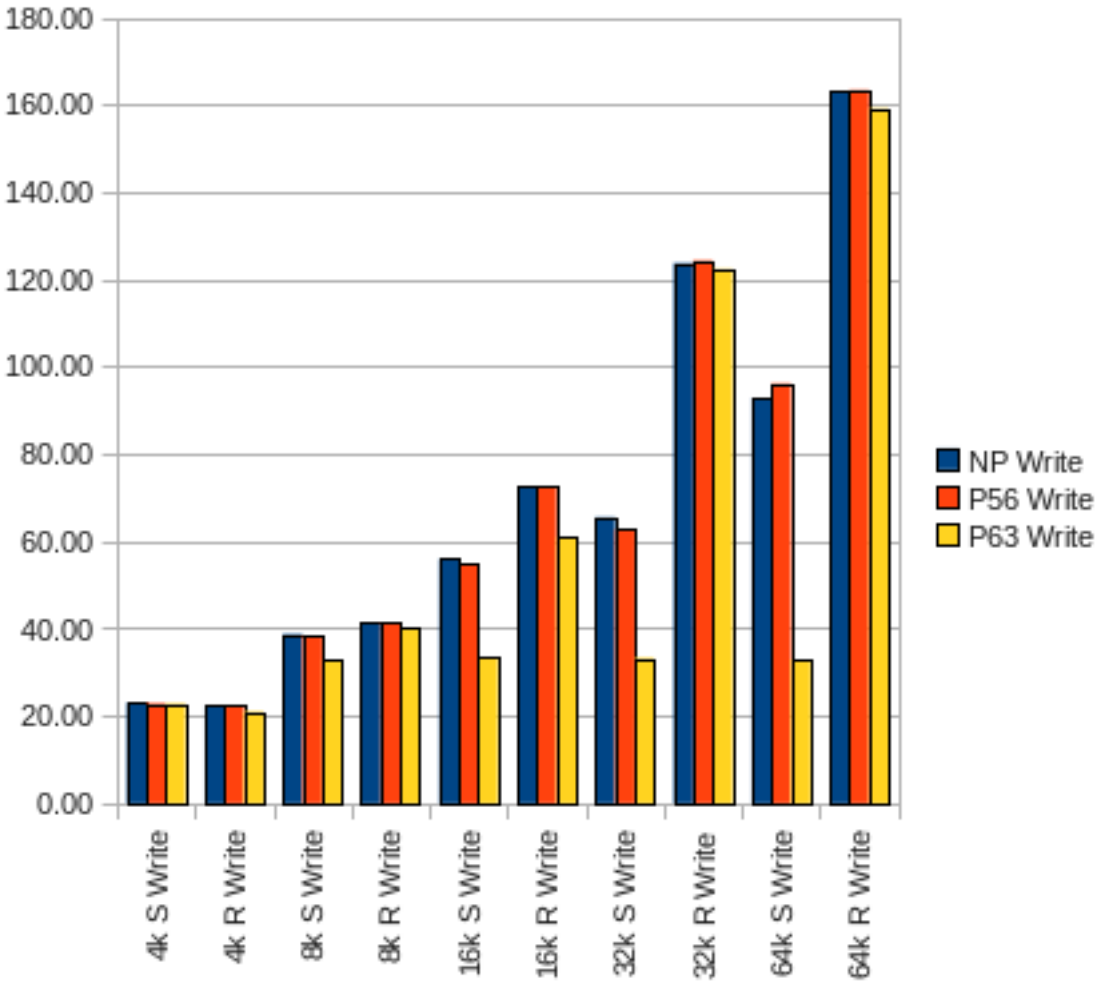

Figure 5.5. Sequential and random writes test for I/O alignment (4K device)

Figure 5.5, "Sequential and random writes test for I/O alignment (4K device)" compares the performance of sequential and random writes on file systems with no partitions (NP, blue bar), partitions aligned to 56 tracks (P56, orange bar), and partitions aligned to 63 tracks (P63, yellow bar). The results show that the 56-track file system outperforms the 63-track file system on transfer sizes 8K, 16K, 32K and 64K during sequential writes.

For more information about I/O alignment, refer to the *Storage I/O Alignment and Size* chapter of the Red Hat Enterprise Linux 6 *Storage Administration Guide*.

5.7. Schedulers *[BZ#639785]*

source: mentioned by Sanjay during chat 2010-10-06_sanjayrao-io_subsystem-intro.html
https://bugzilla.redhat.com/show_bug.cgi?id=639785
placeholder

Generally, the I/O subsystem does not operate in a true FIFO manner. It processes queued read/ write requests depending on selected *scheduler* algorithms.

Scheduler algorithms are sometimes called "elevators" because they operate in the same manner that real-life building elevators do. To be efficient, a real-life elevator does not travel to each floor in the

order they were requested. Rather, it moves in one direction at a time, fulfilling as many requests as it can before reaching the top/lowest floor, then moves again in the opposite direction.

Similarly, scheduler algorithms schedule disk I/O requests according to their target logical block address on disk. The most efficient way to access the disk is to keep the access pattern as sequential (i.e. moving in one direction) as possible. In this case, "sequential" means "by increasing logical block address number".

As such, a disk I/O request targeted for disk block 100 will normally be schedule before a disk I/O request targeted for disk block 200. This is typically the case, even if the disk I/O request for disk block 200 was issued first.

However, the scheduler/elevator also takes into consideration the need for *all* disk I/O requests (except for read-ahead requests) to be processed at some point. This means that the I/O subsystem will not keep putting off a disk I/O request for disk block 200 simply because other requests with lower disk address numbers keep appearing. The conditions which dictate the latency of unconditional disk I/O scheduling is also set by the selected elevator (along with any specified request queue parameters).

5.7.1. Selecting an I/O Scheduler

To specify an I/O scheduler at boot-time, add the following directive to the **kernel** line in **/boot/grub/grub.conf**:

elevator=*scheduler-type*

Replace ***scheduler-type*** with **noop**, **deadline**, **as**, or **cfq**. Each scheduler (except **noop**) has its own set of tunable parameters; refer to their respective sections for more information.

You can also switch I/O schedulers on the fly for specific devices. To do so, run:

echo *scheduler-type* > /sys/block/*dev*/queue/scheduler

Replace ***dev*** with the target device name (e.g. **hda**, **sda**).

For more information about selecting an I/O scheduler, refer to **file:///usr/share/doc/kernel-*version*/Documentation/block/switching-sched.txt**.

5.7.2. Completely Fair Queueing

discuss $TITLE

The *completely fair queueing* (**cfq**) scheduler aims to equally divide all available I/O bandwidth among all processes issuing I/O requests. It is best suited for most medium and large multi-processor systems, as well as systems which required balanced I/O performance over several I/O controllers and LUNs. As such, **cfq** is the default scheduler for Red Hat Enterprise Linux 6.

The **cfq** scheduler maintains a maximum of 64 internal request queues; each process running on the system is assigned is assigned to any of these queues. Each time a process submits a synchronous I/O request, it is moved to the assigned internal queue. Asynchronous requests from all processes are batched together according to their process's I/O priority; for example, all asynchronous requests from processes with a scheduling priority of "idle" (3) are put into one queue.

During each cycle, requests are moved from each non-empty internal request queue into one dispatch queue. in a round-robin fashion. Once in the dispatch queue, requests are ordered to minimize disk seeks and serviced accordingly.

> Example 5.1. How the **cfq** scheduler works
>
> To illustrate: let's say that the 64 internal queues contain 10 I/O request seach, and **quantum** is set to 8. In the first cycle, the **cfq** scheduler will take one request from each of the first 8 internal

> queues. Those 8 requests are moved to the dispatch queue. In the next cycle (given that there are 8 free slots in the dispatch queue) the **cfq** scheduler will take one request from each of the next batches of 8 internal queues.

The tunable variables for the **cfq** scheduler are set in files found under **/sys/block/<*device*>/queue/iosched/**. These files are:

quantum
: Total number of requests to be moved from internal queues to the dispatch queue in each cycle.

queued
: Maximum number of requests allowed per internal queue.

Prioritizing I/O Bandwidth for Specific Processes

When the **cfq** scheduler is used, you can adjust the I/O throughput for a specific process using **ionice**. **ionice** allows you to assign any of the following scheduling classes to a program:

- idle (lowest priority)
- best effort (default priority)
- real-time (highest priority)

For more information about **ionice**, scheduling classes, and scheduling priorities, refer to **man ionice**.

5.7.3. Anticipatory Scheduler

An application that issues a read request for a specific disk block may also issue a request for the next disk block after a certain think time. However, in most cases, by the time the request for the next disk block is issued, the disk head may have already moved further past. This results in additional latency for the application.

To address this, the *anticipatory* (**as**) scheduler enforces a delay after servicing an I/O requests before moving to the next request. This gives an application a window within which to submit another I/O request. If the next I/O request was for the next disk block (as anticipated), the **anticipatory** scheduler helps ensure that it is serviced before the disk head has a chance to move past the targeted disk block.

Read and write requests are dispatched and serviced in batches. The **anticipatory** scheduler alternates between dispatching/servicing batches of read and write requests. The frequency, amount of time and priority given to each batch type depends on the settings configured in **/sys/block/<*device*>/queue/iosched/**.

The cost of using the **anticipatory** scheduler is the overall latency caused by numerous enforced delays. You should consider this trade-off when assessing the suitability of the **anticipatory** scheduler for your system. In most small systems that use applications with many dependent reads, the improvement in throughput from using the **anticipatory** scheduler significantly outweighs the overall latency.

The **anticipatory** scheduler tends to be recommended for servers running data processing applications that are not regularly interrupted by external requests. Examples of these are servers dedicated to compiling software. For the most part, the **anticipatory** scheduler performs well on most personal workstations, but very poorly for server-type workloads.

The tunable variables for the **anticipatory** scheduler are set in files found under **/sys/block/<*device*>/queue/iosched/**. These files are:

read_expire
: The amount of time (in milliseconds) before each read I/O request expires. Once a read or write request expires, it is serviced immediately, regardless of its targeted block device. This tuning option is similar to the **read_expire** option of the **deadline** scheduler (for more information, refer to *Section 5.7.4, "**deadline** Scheduler"*.

  Read requests are generally more important than write requests; as such, it is advisable to issue a faster expiration time to **read_expire**. In most cases, this is half of **write_expire**.

  For example, if **write_expire** is set at 248, it is advisable to set **read_expire** to 124.

write_expire
: The amount of time (in milliseconds) before each write I/O request expires.

read_batch_expire
: The amount of time (in milliseconds) that the I/O subsystem should spend servicing a batch of read requests before servicing pending write batches (if there are any). . Also, **read_batch_expire** is typically set as a multiple of **read_expire**.

write_batch_expire
: The amount of time (in milliseconds) that the I/O subsystem should spend servicing a batch of write requests before servicing pending write batches.

antic_expire
: The amount of time (in milliseconds) to wait for an application to issue another I/O request before moving on to a new request.

5.7.4. deadline Scheduler

The **deadline** scheduler assigns an expiration time or "deadline" to each block device request. Once a request reaches its expiration time, it is serviced immediately, regardless of its targeted block device. To maintain efficiency, any other similar requests targeted at nearby locations on disk will also be serviced.

The main objective of the **deadline** scheduler is to guarantee a response time for each request. This lessens the likelihood of a request getting moved to the tail end of the request queue because its location on disk is too far off.

In some cases, however, this comes at the cost of disk efficiency. For example, a large number of read requests targeted at locations on disk far apart from each other can result in excess read latency.

The **deadline** scheduler aims to keep latency low, which is ideal for real-time workloads. On servers that receive numerous small requests, the **deadline** scheduler can help by reducing resource management overhead. This is achieved by ensuring that an application has a relatively low number of outstanding requests at any one time.

The tunable variables for the **deadline** scheduler are set in files found under **/sys/block/<*device*>/queue/iosched/**. These files are:

read_expire
: The amount of time (in milliseconds) before each read I/O request expires. Since read requests are generally more important than write requests, this is the primary tunable option for the **deadline** scheduler.

`write_expire`
: The amount of time (in milliseconds) before each write I/O request expires.

`fifo_batch`
: When a request expires, it is moved to a "dispatch" queue for immediate servicing. These expired requests are moved by batch. **`fifo_batch`** specifies how many requests are included in each batch.

`writes_starved`
: Determines the priority of reads over writes. **`writes_starved`** specifies how many read requests should be moved to the dispatch queue before any write requests are moved.

`front_merges`
: In some instances, a request that enters the **`deadline`** scheduler may be contiguous to another request in that queue. When this occurs, the new request is normally merged to the back of the queue.

  `front_merges` controls whether such requests should be merged to the front of the queue instead. To enable this, set **`front_merges`** to **`1`**. **`front_merges`** is disabled by default (i.e. set to **`0`**).

5.7.5. Noop Scheduler

Among all I/O scheduler types, the **noop** scheduler is the simplest. While it still implements request merging, it moves all requests into a simple unordered queue, where they are processed by the disk in a simple FIFO order. The **noop** scheduler has no tunable options

The **noop** scheduler is suitable for devices where there are no performance penalties for seeks. Examples of such devices are ones that use flash memory. **noop** can also be suitable on some system setups where I/O performance is optimized at the block device level, with either an intelligent host bus adapter, or a controller attached externally.

Memory *[Larry Woodman]*

source: larry_shak_perf_summit2010_v3.odp, slide 114+
Larry Woodman[1]
The assumption is that users already know about memory, we just need to document RHEL-specific memory management features

6.1. Huge Translation Lookaside Buffers *[BZ#639791]*

source: larry_shak_perf_summit2010_v3.odp, slide 104+
https://bugzilla.redhat.com/show_bug.cgi?id=639791

6.2. Huge Pages and Transparent Hugepages *[BZ#639792]*

source: larry_shak_perf_summit2010_v3.odp, slide 119+
https://bugzilla.redhat.com/show_bug.cgi?id=639792
intro from rdoty: ***`RHEL 6 Scalability 23Aut10.odt`***

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

Hardware and memory management algorithms that work well with thousands of pages (i.e. megabytes of memory) have difficulty performing well with millions (or even billions) of pages. This is especially critical since the hardware memory management unit in modern processors only support hundreds or thousands of page table entries – when an application needs to use more memory pages, the system falls back to slower, software-based memory management.

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 and could result in other performance issues. Red Hat Enterprise Linux 6 implemented the second method via the use of *huge pages*.

can we elaborate on what "performance issues" can arise? link?

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 the 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.

[1] mailto:lwoodman@redhat.com

6.3. Using Valgrind to Profile Memory Usage *[BZ#639793]*

suitability for memory profiling
one or two use cases
references to built-in documentation
https://bugzilla.redhat.com/show_bug.cgi?id=639793

6.4. Capacity Tuning *[BZ#639794]*

source: larry_shak_perf_summit2010_v3.odp, slide 78+
https://bugzilla.redhat.com/show_bug.cgi?id=639794

6.5. Tuning Virtual Memory *[BZ#639795]*

source: larry_shak_perf_summit2010_v3.odp, slide 87+
swappiness, 87
min_free_kbytes, 88
dirty_ratio
dirty_background_ratio
pagecache
slabcache
https://bugzilla.redhat.com/show_bug.cgi?id=639795

Storage *[Barry Marson]*

SME is Barry Marson[1]
use as main reference: RH442-RHEL5u1-en-4-20090715-ddomingo.pdf. any other docs??

7.1. The Ext4 File System *[BZ#639796]*

provide performance-related stats
https://bugzilla.redhat.com/show_bug.cgi?id=639796

7.1.1. Useful Journaling and Mount Options

RH442-RHEL5u1-en-4-20090715-ddomingo.pdf p.172+

7.2. The XFS File System *[BZ#640877]*

source: mentioned by Barry during talk
https://bugzilla.redhat.com/show_bug.cgi?id=640877

7.3. Clustering *[BZ#639797]*

provide intro only; refer to existing pkennedy docs
https://bugzilla.redhat.com/show_bug.cgi?id=639797

7.4. Global File System 2 *[BZ#639798]*

provide intro only; refer to http://documentation-stage.bne.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Global_File_System_2/index.html
https://bugzilla.redhat.com/show_bug.cgi?id=639798
https://access.redhat.com/kb/docs/DOC-35662

[1] mailto:bmarson@redhat.com


DRAFT

Networking *[Neil Horman] [BZ#639799]*

SME is Neil Horman[1]
REVISIT: take most important content from ***RH442-RHEL5u1-en-4-20090715-ddomingo.pdf*** *- Unit 13*
IMPORTANT: all the kernel docs references herein are from Fedora 14, which i assume will be identical to RHEL6. please advise.

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.

overview: https://bugzilla.redhat.com/show_bug.cgi?id=639799

8.1. Network Performance Enhancements *[BZ#639799]*

https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=639799

The following network performance enhancements are available as of Red Hat Enterprise Linux 6.1:

omitted SO_REUSEPORT for now, since we don't have a target for that yet.

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.

Neil: do we need to define ***rx*** *for our readers? do system admins typically know what it is? (answer: no need to define!)*

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 (e.g. **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*[2] .

Neil, Linda: where can users get more info on RPS and RFS (e.g. kernel docs file, man page)? also, please verify definition of ***rx-N*** *in previous paragraph*
from: http://lwn.net/Articles/378617/

Receive Flow Steering

RFS is an extension of RPS, allowing the administrator to configure a hash table that gets populated automatically when applications receive data and get interrogated by the network stack. This

[1] mailto:nhorman@redhat.com
[2] 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 [BZ#639800] "*. *Neil: correct rephrasing?*

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.

http://permalink.gmane.org/gmane.linux.network/179976

`/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`**.

Neil, Linda: ditto on ***`rx-N`***

Unlike RPS, RFS allows both receive queue and 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 heirarchy, application load, and the like.

http://lwn.net/Articles/381955/

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 (e.g. 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`**.

http://www.mjmwired.net/kernel/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 *[BZ#639801]*

https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=639801
"reasons to not bother with adjusting your 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 the specific system's network resource requirements. Incorrect configurations to network performance settings can actually lead to degraded performance.

For example, consider the *bufferfloat problem*. Increasing buffer queue depths result 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 perfomance 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 (e.g. 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 **`man netstat`**.

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

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

ethtool
: A utility for displaying and changing NIC settings. For more information, refer to **`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. [3] 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 applications 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.

Socket receive buffer size

If further analysis proves that the socket queue's drain rate is too slow, then you can increase the depth of the applications socket queue. To do so, increase the size of receive buffers used by sockets. This is achieved 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`**.

  Neil: please verify, got this from kernel docs

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 **`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 **`man setsockopt`**.

Neil: are there any other popular examples to add here?

8.3. Overview of Packet Reception *[BZ#639800]*

https://bugzilla.redhat.com/show_bug.cgi?id=639800

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 often get lost. Lost frames in the receive path often cause a significant penalty to network performance.

[3] *Section 8.3, "Overview of Packet Reception [BZ#639800] "* contains an overview of packet travel, which should help you locate and map bottleneck-prone areas in the network stack.

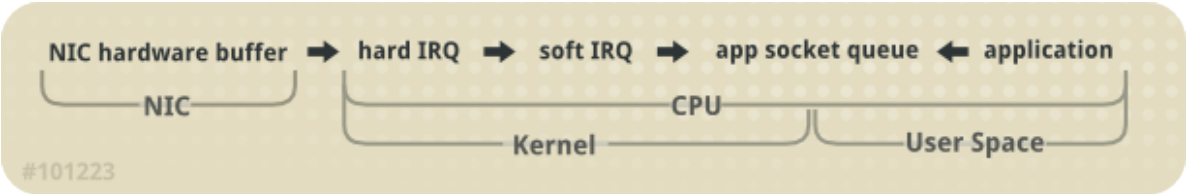

Figure 8.1. Network receive path diagram

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.

   Neil: i substituted "network data" with "frame" in the following parts, for consistency. correct?
   In this context (i.e. 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 (e.g. **`read`**, **`recv`**, **`recvfrom`**). At this point, data received over the network no longer exist 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 Queueing/Frame Loss Issues *[BZ#639802]*

https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=639802

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 to do 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[4] .

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 [BZ#639801] "*.

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 (e.g. **`recv`**, **`read`**). In turn, this allows application to drain the queue faster.

For many administrators, increasing the queue depth is the preferable solution. This is the easiest solution, and often works in the long term. However, this is only a short-term solution; 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 quicker, even if it means queueing the data in application space. At least then, the data can be stored more flexibly (e.g. swapped out and paged back in as needed).

8.5. Multicast Considerations *[BZ#639803]*

https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=639803

Multicast traffic users report that in high volumes, the rate at which they lose frames increases as they add instances of an application to the same system that listens on a multicast socket. This is caused by a design requirement on the kernel code that handles multicast frames. If multiple applications are listening to a multicast group, the kernel must make a copy of the network data for each individual socket. This duplication must occur in the **`softirq`** context, and is time-consuming.

This means that adding multiple listeners on a single multicast group has a direct impact on the **`softirq`** context's execution time. As such, adding one more listener to a given multicast group implies that for each frame received destined for that group, the kernel needs to create an additional copy of the frame.

The effect of this is minimal in low traffic and small numbers of listeners. However, when several sockets begin listening on a single, high-volume traffic multicast group, the significantly increased execution time of the **`softirq`** context can lead to frame drops, both at the network card *and* at the socket queue. This implies that increased **`softirq`** runtimes translate to reduced opportunity for applications to run on heavily-loaded systems.

To resolve this, optimize the socket queues and NIC hardware buffers. To do so, employ any of the techniques enumerated in *Section 8.4.2, "Socket Queue"* or *Section 8.4.1, "NIC Hardware Buffer"*. Alternatively, you can try optimizing 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.

[4] 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"*.


DRAFT

Tuned Profiles *[SME TBA]*

https://engineering.redhat.com/rt3/Ticket/Display.html?id=72381
Q: can we also squeeze in special tuning considerations here for specific services? i.e. the current nested sections

9.1. Oracle

RHELTuningandOptimizationforOracleV11.pdf
Woodman,Shakshober_Performance Analys.pdf, p.80+

9.2. Samba Server

from old PTG

9.3. Web Servers

from old PTG


DRAFT

Appendix A. Revision History

Revision 0-1 **Mon Sep 6 2010** **Dude McPants** *Dude.McPants@example.com*

Initial creation of book by publican


DRAFT

Index

S

T

W