Whitepapers 1.0

Red Hat Enterprise Linux 5 IO Tuning Guide

Performance Tuning Whitepaper for Red Hat Enterprise Linux 5.2

Red Hat Inc.
Don Domingo

Abstract

The Red Hat Enterprise Linux 5 I/O Tuning Guide presents the basic principles of performance analysis and tuning for the I/O subsystem. This document also provides techniques for troubleshooting performance issues for the I/O subsystem.

1. Preface 2
    1.1. Audience 2
    1.2. Document Conventions 3
    1.3. Feedback 4
2. The I/O Subsystem 4
3. Schedulers / Elevators 5
4. Selecting a Scheduler 6
5. Tuning a Scheduler and Device Request Queue Parameters 6
    5.1. Request Queue Parameters 7
6. Scheduler Types 7
    6.1. **cfq** Scheduler 7
    6.2. **deadline** Scheduler 8
    6.3. **anticipatory** Scheduler 9
    6.4. **noop** Scheduler 10

Index **10**

A. Revision History **11**

1. Preface

This guide describes how to analyze and appropriately tune the I/O performance of your Red Hat Enterprise Linux 5 system.

Caution

While this guide contains information that is field-tested and proven, it is recommended that you properly test everything you learn on a testing environment before you apply anything to a production environment.

In addition to this, be sure to back up all your data and pre-tuning configurations. It is also prudent to plan for an implementation reversal.

Scope

This guide discusses the following major topics:

- Investigating system performance
- Analyzing system performance
- Red Hat Enterprise Linux 5 performance tuning
- Optimizing applications for Red Hat Enterprise Linux 5

The scope of this document does not extend to the investigation and administration of faulty system components. Faulty system components account for many percieved performance issues; however, this document only discusses performance tuning for fully functional systems.

1.1. Audience

Due to the deeply technical nature of this guide, it is intended primarily for the following audiences.

Senior System Administrators

This refers to administrators who have completed the following courses / certifications:

- RH401 - *Red Hat Enterprise Deployment, Virtualization and Systems Management*; for more information, refer to *https://1www.redhat.com/1training/1rhce/1courses/1rh401.html*
- RH442 - *Red Hat Enterprise System Monitoring and Performance Tuning*; for more information, refer to *https://1www.redhat.com/1training/1architect/1courses/1rh442.html*
- RHCE - *Red Hat Certified Engineers*, or administrators who have completed RH300 (*Red Hat Rapid Track Course*); for more information, refer to *https://1www.redhat.com/1training/1rhce/1courses/1rh300.html*

Application Developers

This guide also contains several sections on how to properly tune applications to make them more resource-efficient.

1.2. Document Conventions

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

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

When shown as below, it indicates computer output:

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

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

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

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

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

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

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

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

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

Warning

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

1.3. Feedback

If you have thought of a way to make this manual better, submit a bug report through the following Bugzilla link: *File a bug against this book through Bugzilla*[1]

File the bug against **Product**: Red Hat Enterprise Linux, **Version**: rhel5-rc1. The **Component** should be *Performance_Tuning_Guide*.

Be as specific as possible when describing the location of any revision you feel is warranted. If you have located an error, please include the section number and some of the surrounding text so we can find it easily.

2. The I/O Subsystem

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.

[1] https://bugzilla.redhat.com/enter_bug.cgi?product=Red%20Hat%20Enterprise%20Linux%205&bug_status=NEW&version=5.2&component=Performance_Tuning_Guide&rep_platform=All&op_sys=Linux&priority=low&bug_severity=low&assig%3A%2F%2F&short_desc=&comment=&status_whiteboard=&qa_whiteboard=&devel_whiteboard=&keywords=&issuetrackers=&dependson=&blocked=&ext_bz_i%2Fplain&contenttypeentry=&maketemplate=Remember%20values%20as%20bookmarkable%20template&form_name=enter_bug

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.

Sequential read requests can take advantage of *read-aheads*. Read-ahead assumes that an application reading from disk block X will also next ask to read from disk block X+1, X+2, etc. When the system detects a sequential read, it caches the following disk block ahead in memory, then repeats once the cached disk block is read. This strategy decreases seek time, which ultimately improves application response time. The read-ahead mechanism is turned off once the system detects a non-sequential file access.

3. Schedulers / Elevators

Generally, the I/O subsystem does not operate in a true FIFO manner. It processes queued read/write requests depending on the selected scheduler algorithms. These scheduler algorithms are called *elevators*. Elevators were introduced in the 2.6 kernel.

Scheduler algorithms are sometimes called “elevators” because they operate in the same manner that real-life building elevators do. The algorithms used to operate real-life building elevators make sure that it services requests per floor efficiently. To be efficient, the elevator does not travel to each floor depending on which one issued a request to go up or down first. Instead, it moves in one direction at a time, taking as many requests as it can until it reaches the highest or lowest floor, then does the same in the opposite direction.

Simply put, these algorithms schedule disk I/O requests according to which logical block address on disk they are targeted to. This is because the most efficient way to access the disk is to keep the access pattern as sequential (i.e. moving in one direction) as possible. Sequential, in this case, means “by increasing logical block address number”.

As such, a disk I/O request targeted for disk block 100 will normally be scheduled 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).

There are several types of schedulers:

- **`deadline`**
- **`as`**
- **`cfq`**
- **`noop`**

These scheduler types are discussed individually in the following sections.

4. Selecting a Scheduler

To specify a scheduler to be selected at boot time, add the following directive to the kernel line in **/boot/grub/grub.conf**:

elevator=<*elevator type*>

For example, to specify that the **noop** scheduler should be selected at boot time, use:

elevator=noop

You can also select a scheduler during runtime. To do so, use this command:

echo <*elevator type*> > /sys/block/<*device*>/queue/scheduler

For example, to set the **noop** scheduler to be used on **hda**, use:

echo noop > /sys/block/hda/queue/scheduler

At any given time, you can view **/sys/block/<*device*>/queue/scheduler** (using **cat**, for example) to verify which scheduler is being used by **<*device*>**. For example, if **hda** is using the **noop** scheduler, then **cat /sys/block/hda/queue/scheduler** should return:

```
[noop] anticipatory deadline cfq
```

Note that selecting a scheduler in this manner is *not* persistent throughout system reboots. Unlike the **/proc/sys/** file system, the **/sys/** file system does not have a utility similar to **sysctl** that can make such changes persistent throughout system reboots.

To make your scheduler selection persistent throughout system reboots, edit **/boot/grub/grub.conf** accordingly. Do this by appending **elevator=<*scheduler*>** to the the **kernel** line. **<*scheduler*>** can be either **noop**, **cfq**, **as** (for anticipatory), or **deadline**.

For example, to ensure that the system selects the **noop** scheduler at boot-time:

```
title Red Hat Enterprise Linux Server (2.6.18-32.el5)
 root (hd0,4)
 kernel /boot/vmlinuz-2.6.18-32.el5 ro root=LABEL=/1 rhgb quiet
 elevator=noop
 initrd /boot/initrd-2.6.18-32.el5.img
```

5. Tuning a Scheduler and Device Request Queue Parameters

Once you have selected a scheduler, you can also further tune its behavior through several request queue parameters. Every I/O scheduler has its set of tunable options. These options are located (and tuned) in **/sys/block/<*device*>/queue/iosched/**.

In addition to these, each device also has tunable request queue parameters located in **/sys/block/<*device*>/queue/**.

Scheduler options and device request queue parameters are set in the same fashion. To set these tuning options, echo the specified value to the specified tuning option, i.e.:

echo <*value*> > /sys/block/<*device*>/queue/iosched/<*option*>

For example: the system is currently using the **anticipatory** scheduler for device **hda**. To change **/sys/block/hda/queue/iosched/read_expire** to 80 milliseconds, use:

echo 80 > /sys/block/hda/queue/iosched/read_expire

However, as mentioned in *Section 4, "Selecting a Scheduler"*, any tuning made though **echo** commands to the **/sys/** file system is not persistent thoughout system reboots. As such, to make any scheduler selection/request queue settings persistent throughout system reboots, use **/etc/rc.d/rc.local**.

5.1. Request Queue Parameters

Block devices have the following tunable parameters:

nr_requests

This file sets the depth of the request queue. **nr_requests** sets the maximum number of disk I/O requests that can be queued up. The default value for this is dependent on the selected scheduler.

If **nr_requests** is set higher, then generally the I/O subsystem will have a larger threshold at which it will keep scheduling requests by order of increasing disk block number. This keeps the I/O subsystem moving in one direction longer, which in most cases is the most efficient way of handling disk I/O.

read_ahead_kb

This file sets the size of read-aheads, in kilobytes. As discussed in *Section 2, "The I/O Subsystem"*, the I/O subsystem will enable read-aheads once it detects a sequential disk block access. This file sets the amount of data to be "pre-fetched" for an application and cached in memory to improve read response time.

6. Scheduler Types

This section describes the different behaviors of each type of scheduler. For instructions on how to select a scheduler, refer to *Section 4, "Selecting a Scheduler"*. For instructions on how to tune scheduler options, refer to *Section 5, "Tuning a Scheduler and Device Request Queue Parameters"*.

6.1. cfq Scheduler

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 5.

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.

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.

Example 1. How the **cfq** scheduler works

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**.

6.2. 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**).

6.3. 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** 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 6.2, "**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.

6.4. 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.

Index

A

Audience
Introduction, 2

I

Introduction
Audience, 2

A. Revision History

Revision History
Revision 1.0 Tue Sep 23 2008 DonDomingo*ddomingo@redhat.com*
updated to build on Publican 0.36
under product heading Whitepapers now