



Performance Tuning and Monitoring **Part 3**

Here're some tricks to help you counter disk fragmentation and improve RAID performance on hard drives significantly.

In the October issue of LFY, we discussed the different performance tuning methods related to hard disk drives, using IO elevators. We had also explored the queuing theory and finding hotspots using the strace command. Now, after our Diwali break, it's time we resume fine tuning our hard disks again.

This time we will look at how to counter disk fragmentation and improve RAID performance significantly. We will also learn about cache profiling using Valgrind.

Countering disk fragmentation

Disk fragmentation affects the sequential read access performance. This is because it results in extra head movement on the disk drive. In fact, the Linux filesystem is designed to minimise the effects of fragmentation as much as possible. When a file is created or extended, the filesystem always tries to allocate blocks for the file from the same block group that contains its inode. And to ensure that files are spread evenly in the

partition, when a new file is created, an attempt is made to place it in some block group other than the one containing its parent directory.

To view the fragmentation of a file we can use the following command:

```
$ filefrag -v /path/to/the/filename
```

To check the fragmentation on a mounted filesystem, issue the command shown below:

```
$ dumpe2fs /dev/sdaX
```

...where X is the partition number.

Types of journaling in ext3

One major issue in journaling filesystems is whether they only log changes to the filesystem metadata or log changes to all filesystem data, including changes to the files themselves. The ext3 filesystem supports three different journaling modes. Here's how Wikipedia [<http://en.wikipedia>].

[org/wiki/Ext3](https://www.kernel.org/wiki/Ext3)] explains each:

- **Journal (lowest risk):** Both metadata and file contents are written to the journal before being committed to the main filesystem. Because the journal is relatively continuous on disk, this can improve performance in some circumstances. In other cases, performance gets worse because the data must be written twice—once to the journal, and once to the main part of the filesystem.
- **Ordered (medium risk):** Only metadata is journaled; file contents are not, but it's guaranteed that file contents are written to disk before associated metadata is marked as committed in the journal. This is the default on many Linux distributions.
- **Writeback (highest risk):** Only metadata is journaled; file contents are not. The contents might be written before or after the journal is updated. As a result, files modified right before a crash can become corrupted.

These modes can be made permanent by editing the `/etc/fstab` file and specifying the journaling mode, as shown in Figure 1.

Improving journal performance

By default, the journal is located at an inode within the filesystem. To improve the performance, we should try to keep the journal on a separate device. The journaling filesystem must use the same block size as that of the data filesystem. You should always dedicate the entire partition to the journal and make sure that the external journal is always in a disk drive capable of equal or better performance.

I'm creating a normal RAID-5 device here, and then I will tune the array by using the STRIDE and CHUNK size. Finally, I'll move the journal to an external device to enhance performance.

Creating a normal RAID-5 array

Step 1 Create three partitions of 100 MB (that is, 102400 KB) each

```
sysfs      /sys      sysfs defaults 0 0
proc       /proc     proc  defaults 0 0
/dev/sda5  /data     ext3  defaults,data=ordered 0 0
/dev/sda6  /vbox     ext3  defaults 0 0
```

Figure 1: Changing the journaling mode of an ext3 filesystem by editing `/etc/fstab`

```
[root@server1 ~]# iostat -x -d /dev/sda 1
Linux 2.6.18-92.el5xen (server1.example.com) 09/14/2009

Device:            rrqm/s    wrqm/s     r/s     w/s    rsec/s    wsec/s  avgrq-sz  avgqu-sz   await  svctm    %util
sda                 9.81      11.40   10.40    4.12   1461.35    123.99   109.20     0.06    4.24    2.20    3.20

Device:            rrqm/s    wrqm/s     r/s     w/s    rsec/s    wsec/s  avgrq-sz  avgqu-sz   await  svctm    %util
sda                49.00      0.00   39.00    0.00   7816.00     0.00   200.41     0.10    2.56    2.26    8.80
```

Figure 2: Calculating the chunk size

and change its 'type' to `fd`.

Step 2 Use the `mdadm` command to create the array:

```
# mdadm -C /dev/md0 -l 5 -n 3 /dev/sda(7,8,9)
```

Step 3 Format the array:

```
# mke2fs -b 2048 -j /dev/md0
```

Step 4 The RAID device is now ready and initiated. You can mount it on any location to use it.

Step 5 I will now capture the filesystem layout using the `dumpe2fs` command:

```
# dumpe2fs /dev/md0 > anyfilename
```

Step 6 Now stop the array and its superblock:

```
# mdadm -S /dev/zero --zero-superblock
```

I now want to reduce the disk visit count on this array by using the chunk size and stride values.

Chunk size is actually the amount of data read/written from each device in an array before moving to the next device in a round-robin manner. It is also known as 'granularity of the stripe'. It's always in KB. Chunk size is very important for RAID performance.

If the chunk size is too small, it can result in a file spreading into multiple smaller pieces thus increasing the number of drives it will use. On the other hand, if the chunk size is too big, a situation may develop in which all the IO is handled by one physical device only—a condition

```
Inode count:      25624576
Block count:      25599577
Reserved block count: 1279978
Free blocks:      18711564
Free inodes:      25582961
First block:      0
Block size:       4096
Fragment size:    4096
Reserved GDT blocks: 1037
```

Figure 3: A snippet of `dumpe2fs` output

known as 'hot spot'.

The ideal situation is to spread the IO across all the devices evenly.

To calculate the chunk size, first use the following code:

```
# iostat -x -d /dev/sda 1
```

...to get the value of `avgrq-sz` (average request size). Figure 2 displays the output on my system.

I will consider the value of 200.41 for my calculations. Just note down this value. Now divide it by 2. It's approximately 100.20.

Now divide 100.20 by the number of disk drives in your array. In my case, it was three. So I should be using the figure 100.20/3, which equals 33.4.

Round off this value to the nearest power of 2, which comes to 32. That means my chunk size is 32.

Now, before we move ahead in configuring a tuned RAID, we should also find out the stride value. Stride is the same as chunk size but uses a different value. The aim of the stride is to distribute the block bitmap across RAID member devices.

The formula to calculate the stride is: $\text{STRIDE} = \text{CHUNK SIZE} / \text{BLOCK SIZE}$

We have already retrieved the block size from the `dumpe2fs /dev/md0` command by redirecting the output to the `anyfilename` file. Now, access this file with the `less`

```
[root@legacy alok]# x86info -c
x86info v1.24. Dave Jones 2001-2009
Feedback to <davej@redhat.com>.

Found 2 CPUs
-----
CPU #1
EFamily: 0 EModel: 0 Family: 6 Model: 15 Stepping: 6
CPU Model: Core 2 Duo [B2]
Processor name string: Intel(R) Core(TM)2 CPU          T5500   @ 1.66GHz
Type: 0 (Original OEM) Brand: 0 (Unsupported)
Number of cores per physical package=2
Number of logical processors per socket=2
Number of logical processors per core=1
APIC ID: 0x0 Package: 0 Core: 0 SMT ID 0
Cache info
L1 Instruction cache: 32KB, 8-way associative. 64 byte line size.
L1 Data cache: 32KB, 8-way associative. 64 byte line size.
L2 cache: 2MB, 8-way associative. 64 byte line size.
TLB info
Instruction TLB: 4x 4MB page entries, or 8x 2MB pages entries, 4-way associative
Instruction TLB: 4K pages, 4-way associative, 128 entries.
Data TLB: 4MB pages, 4-way associative, 32 entries
L1 Data TLB: 4KB pages, 4-way set associative, 16 entries
L1 Data TLB: 4MB pages, 4-way set associative, 16 entries
Data TLB: 4K pages, 4-way associative, 256 entries.
64 byte prefetching.
-----
CPU #2
EFamily: 0 EModel: 0 Family: 6 Model: 15 Stepping: 6
CPU Model: Core 2 Duo [B2]
Processor name string: Intel(R) Core(TM)2 CPU          T5500   @ 1.66GHz
Type: 0 (Original OEM) Brand: 0 (Unsupported)
Number of cores per physical package=2
Number of logical processors per socket=2
Number of logical processors per core=1
```

Figure 4: Cache information using the x86info tool

command—it's a huge one—and look for the part shown in Figure 3.

So my stride would be 32/4, or 8.

Now, let's create the tuned array using the chunk size and stride value. We'll use the `mdadm` command as shown:

```
# mdadm -C /dev/md0 -l 5 -n 3 -C 32 /dev/sda7
/dev/sda8 /dev/sda9
```

Now format the device using the stride value. For an ideal filesystem, you can assume a block size of 2 KB (that is, 2048 bytes). But here I am using the actual block size of my filesystem, which is 4 KB or 4096 bytes.

```
# mke2fs -b 4096 -E stride=8 -j /dev/md0
```

That's it! Our job is done. Now just mount the tuned RAID array to some directory (`/raid-location`). You can use the `/etc/fstab` file to make it permanent.

Mounting the journal to the external filesystem

Till now the journal has been on the same filesystem that can lead

to performance degrade. Naturally, it's always better to mount the FS journal to some external or separate filesystem.

Step 1 First *umount* the RAID device:

```
# umount /raid-location
```

Step 2 Check the block size of the RAID filesystem using the `dumpe2fs /dev/md0` command.

Step 3 Create a partition using the `fdisk` command that has to be used for the journal in future.

Step 4 Format the new partition that you had just created to use as the journal device:

```
# mke2fs -O journal_dev -b 4096 -L raid_journal
/dev/sda10
```

Step 5 Before using the new journal filesystem, remove the existing journal from `/dev/md0`:

```
# tune2fs -O ^has_journal /dev/md0
```

Step 6 Now since the old journal has been removed from `/dev/md0`, the new partition `/dev/sda10` can be used to hold the journal for `/dev/md0`:

```
# tune2fs -J device=/dev/sda11 /dev/md0
```

Step 7 Mount the RAID device now:

```
# mount /raid-location
```

Locality of reference

PU cache (L1/L2/L3 cache) is a very important factor in performance. Its speed is normally twice the speed of RAM and is used to hasten the memory retrieval process. Without a cache the life of the CPU will be like that of librarians without a storage-shelf behind them. So every time people request a book or need to store it, they have to climb up the ladder to place the book or get the book from the main shelf (main memory) itself. That will obviously slow down the librarian's performance and lower the customer's satisfaction levels.

The cache is divided into a few lines. Each line is used to cache a specific location in the memory. We have separate cache instructions for the processor: I-cache, data instructions, and D-cache. This classification is also known as the Harvard Memory Architecture.

Every cache has an associated cache-controller. So whenever the CPU asks for any reference from the main memory, the cache-controller first checks if that requested reference is in the cache. If the reference is found there, the request is catered to from there itself, without going to the main memory. So this significantly increases performance. This is known as cache-hit.

However, if the request is not found there (known as cache-miss), its requested location is brought from the main memory to the cache for reference (known as cache-line-fill). On a multi-processor system, if one CPU cache gets a cache-line-fill, then the first CPU must inform the second CPU about this. So the other CPU

need not retrieve it from the main memory. This process is known as cache-snooping.

With the theory out of our way, how can we check the cache? Here's a simple command that does that job:

```
# getconf -a | grep -i cache
```

The following is the output from my test system:

```
LEVEL1_ICACHE_SIZE      32768
LEVEL1_ICACHE_ASSOC      8
LEVEL1_ICACHE_LINESIZE   64
LEVEL1_DCACHE_SIZE      32768
LEVEL1_DCACHE_ASSOC      8
LEVEL1_DCACHE_LINESIZE   64
LEVEL2_CACHE_SIZE       2097152
LEVEL2_CACHE_ASSOC      8
LEVEL2_CACHE_LINESIZE   64
LEVEL3_CACHE_SIZE        0
LEVEL3_CACHE_ASSOC      0
LEVEL3_CACHE_LINESIZE   0
LEVEL4_CACHE_SIZE        0
LEVEL4_CACHE_ASSOC      0
LEVEL4_CACHE_LINESIZE   0
```

You can also use the *x86info* tool (you need to install it first). It gives very detailed information about your CPU and types of cache. Figure 4 shows the output on my system when I execute the *x86info -c* command.

You can also check */var/log/dmesg* for this information.

Locality of reference

Applications normally tend to behave in a particular way when accessing data. An application accessing memory location X is more likely to access memory location X+1 in the next few cycles of execution. This behaviour is known as spatial locality of reference.

Programs that access the memory sequentially will generally get more benefits from the cache. Programs that result in more cache-misses are more expensive, as they increase latency. There is a tool called Valgrind that you can use to profile cache usage. Given a choice, this can help you determine the more efficient program.

```
=6941== Cachegrind, a cache and branch-prediction profiler.
=6941== Copyright (C) 2002-2008, and GNU GPL'd, by Nicholas Nethercote et al.
=6941== Using LibVEX rev 1884, a library for dynamic binary translation.
=6941== Copyright (C) 2004-2008, and GNU GPL'd, by OpenWorks LLP.
=6941== Using valgrind-3.4.1, a dynamic binary instrumentation framework.
=6941== Copyright (C) 2000-2008, and GNU GPL'd, by Julian Seward et al.
=6941== For more details, rerun with: -v
=6941==
Starting
finished
=6941==
=6941== I  refs:      6,188,126,666
=6941== I1 misses:      666
=6941== L2i misses:      664
=6941== I1 miss rate:    0.00%
=6941== L2i miss rate:   0.00%
=6941==
=6941== D  refs:      3,937,849,712 (3,375,259,224 rd + 562,590,488 wr)
=6941== D1 misses:      35,157,103 ( 663 rd + 35,156,440 wr)
=6941== L2d misses:      35,157,056 ( 621 rd + 35,156,435 wr)
=6941== D1 miss rate:    0.8%      0.0%      +      6.2% )
=6941== L2d miss rate:   0.8%      0.0%      +      6.2% )
=6941==
=6941== L2 refs:      35,157,769 ( 1,329 rd + 35,156,440 wr)
=6941== L2 misses:      35,157,720 ( 1,285 rd + 35,156,435 wr)
=6941== L2 miss rate:    0.3%      0.0%      +      6.2% )
root@legacy ~]#
```

Figure 5: Testing application performance using Valgrind

Here are the steps on how to get started with it:

Step 1 Use the *x86info -c* command to get the values of your cache. Valgrind is more interested in: Instruction Cache (I1), Data Level 1 (D1) and Data Level 2 (L2) cache values.

Remember that Valgrind demands the I1 cache value in bytes rounded off to the nearest power of 2, besides the D1 and L2 cache (also in bytes). We also need to jot down the values of associativity and line-size. From my output of *x86info -c*, as shown in Figure 4, I get the following values.

- I1 = 32768,8,64
- D1 = 32768,8,64
- L2 = 2097152,8,64

Step 2 Now, if I have two applications that perform the same task, I need to check which one performs better with respect to cache-hits and cache-misses. I can use Valgrind to find out:


```
# valgrind --tool=cachegrind --I1=32768,8,64 \
-- D1=32768,8,64 \
-- L2=2097152,8,64 sample1
```

Here I'm testing a sample

application called sample1. Figure 5 shows the output (truncated) of the above command.

As you can see, the cache-miss rate for the application sample1 is very low (below 1 per cent). This means that sample1 makes good use of cache-hits, thereby enhancing the performance. It is always recommended to use Valgrind for new applications before putting it in a commercial environment.

As a general rule, the less the miss-rate, the better!

Here I conclude for this month. Next month will be dedicated to how we can tune memory address and allocation using PAM, TLB, pdf flush, etc, as well as tune our network performance. **END** 

By: Alok Srivastava

The author is the founder of Network NUTS and holds MCP, MCSE, MCDBA, MCT, CCNA, CCNP, RHCE and RHCSS certifications. Under his leadership, Network NUTS has been a winner of the "Best Red Hat Training Partner in North India" for the last three years in a row. He has also been a trainer for the Indian Air Force, NIC, LIC, IFFCO, Wartsila India Ltd, the government of Rajasthan, Bajaj Allianz, etc. You can reach him at *alok at networknuts dot net*.