XFS Performance Tuning

Performance Optimisation of the XFS Filesystem

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.1	2014	Initial Version	Dave Chinner

Contents

Abstract

This book documents the knowledge and processes required to optimise the XFS filesystem.

Premature optimization is the root of all evil. - Donald Knuth

So you want to tune your XFS filesystem for a workload? Think hard about it as it’s difficult to do properly and requires significant knowledge and expertise. Reading this guide won’t make you an optimisation expert, though it might just help you solve the problem you are facing.

Part I

Introduction

Chapter 1

Overview

This document describes the theory and practice being tuning XFS for performance. Performance tuning is a mixture of knowledge, process and observation. The Knowledge Section will cover aspects of the XFS configuration space that affect performance and assumes that the reader has some familiarity with the XFS on-disk structures.

The Observation Section will demonstrate various tools that can be used to monitor filesystem performance while workloads are run. Key performance metrics will be discussed and related back to the structure of the filesystem and configuration parameters discussed in the Knowledge Section.

The Process section will cover the typical processes used to optimise a filesystem for a given workload. If the workload measurements are not accurate or reproducible, then no conclusions can be drawn as to whether a configuration changes an improvement or not. Hence without a robust testing process, no amount of knowledge or observation will result in a well optimised filesystem configuration.

While reading this document, it is important to keep in mind the fact that there are relatively few workloads where using non-default mkfs.xfs or mount options make much sense. In general, the default values already used are optimised for best performance in the first place. The most common optimisations are done automatically if possible and hence mostly remove the need for manual optimisation.

Therefore, you should only consider changing the defaults if either:

- you know from experience that your workload causes XFS a specific problem that can be worked around via a configuration change, or
- your workload is demonstrating bad performance when using the default configurations.

In this case, you need to understand why your application is causing bad performance before you start tweaking XFS configurations. This guide is intended to help you identify the cause of perofrmance problems and which knobs you can tweak to avoid or mitigate the problems being observed.

Part II

Knowledge

Chapter 2

Filesystem Tunables

This section covers some of the tuning parameters available to XFS file systems at format and at mount time that are relevant to performance optimisation.

The default formatting and mount settings for XFS are suitable for most workloads. Specific tunings should only be considered if either:

- it is known from experience that the workload causes XFS a specific problem that can be worked around via a configuration change, or
- a workload is demonstrating bad performance when using the default configurations.

2.1 Formatting options

For further details about the syntax and usage any of these formatting options, see the man page:

```
$ man mkfs.xfs
```

Note
Please pay particular attention to the different units used by the different `mkfs.xfs` commands. Also, `mkfs.xfs` and `xfs_info` report the selected configuration in units of filesystem blocks and these are not necessarily same units as used on the `mkfs.xfs` command line to configure these parameters.

Note
The performance examples given in this section are highly dependent on storage, CPU and RAM configuration. They are intended as guidelines to illustrate behavioural differences, not the exact performance any configuration will achieve.

2.1.1 Directory block size

The directory block size affects the amount of directory information that can be retrieved or modified per I/O operation. The minimum value for directory block size is the file system block size (4 KB by default) and the maximum directory block size is 64KB.

Directory block size can only be configured at mkfs time by using the following option:

```
mkfs.xfs -n size=<num>
```

There are various trade-offs that have to be considered when selecting an optimum directory block size. Because directories in XFS are btree structures, performance decrease due to IO overhead is logarithmic with respect to increasing directory size. However, CPU overhead per modification operation increases linearly with directory block size, and hence large directory block sizes will only out-perform small directory block sizes once the IO overhead of manipulating the small directory is greater than the CPU overhead of large directory blocks.

An artificial workload that demonstrated this CPU/IO trade off is creating zero length files in a directory. If the directory is small, the workload will be CPU bound, but if the directory is large it will become IO bound and so the performance profile will change.

Directory block size	**10000 entries**	**100K entries**	**1M entries**
4k	**21,500/s**	**15,500/s**	**6,700/s**
16k	**18,500/s**	**14,400/s**	**11,000/s**
64k	**14,000/s**	**9,500/s**	**7,500/s**

It can be seen from the table that 4k directory block size has the largest modification rate decay as the directory size increases and IO overhead starts to dominate. 64k block sizes have the lowest decay rate because that configuration has the lowest IO overhead and it won't be until the directories contain more than 10 million entries that significant performance degradation will be observed.

Read dominated workloads are more difficult to characterise because best performance comes from keeping a working set of directories and files cached in RAM. Ignoring this aspect of the performance tuning problem, look up performance of directories is really an IO bound workload. hence the fewer IOs that need to be done to find a directory entry, the better the performance will be.

Again, larger directory blocks will consume more CPU per look up than smaller directory blocks, but larger directories blocks can index a far greater number of entries per IO and so the CPU overhead of searching a large block is less than doing an extra IO. Hence a look up dominated workload will have a much lower cross-over point where larger block sizes will perform better.

Table 2.1: Average directory read and inode look up rate when CPU bound

Directory block size	**10000 entries**	**100K entries**	**1M entries**
4k	**32,000/s**	**25,000/s**	**18,500/s**
16k	**42,000/s**	**26,000/s**	**22,500/s**
64k	**42,000/s**	**26,000/s**	**22,000/s**

Table 2.2: Average directory read and inode look up rate when IO bound

Directory block size	**10000 entries**	**100K entries**	**1M entries**
4k	**8,000/s**	**7,000/s**	**6,500/s**
16k	**32,000/s**	**9,500/s**	**8,000/s**
64k	**32,000/s**	**10,000/s**	**11,000/s**

And, finally, removing those files is a combination of look up and modification overhead, so it can be either IO or CPU bound depending on which part of the remove operation is the limiting factor:

Table 2.3: Average file remove rate when CPU bound

Directory block size	**10000 entries**	**100K entries**	**1M entries**
4k	**11,400/s**	**18,000/s**	**3,500/s**

Table 2.3: (continued)

16k	**11,000/s**	**17,000/s**	**9,500/s**
64k	**9,000/s**	**14,000/s**	**7,000/s**

Table 2.4: Average file remove rate when IO bound

Directory block size	10000 entries	100K entries	1M entries
4k	**4,500/s**	**4,000/s**	**2,500/s**
16k	**5,500/s**	**7,000/s**	**4,500/s**
64k	**6,000/s**	**6,500/s**	**5,000/s**

Typical directory block sizes configurations for different workloads and directory sizes are listed in Table 2.5. These are only guidelines - the best size for any given workload will vary. When in doubt, use the mkfs default values.

Table 2.5: Recommended maximum number of directory entries for directory block sizes

Directory block size	Max. entries (read-heavy)	Max. entries (write-heavy)
4 KB	**100000-200000**	**1000000-2000000**
16 KB	**100000-1000000**	**1000000-10000000**
64 KB	**>1000000**	**>10000000**

2.1.2 Allocation groups

An Allocation Group (AG) in XFS in an independent structure that indexes free space and allocated inodes across a section of the filesystem. Each AG can be modified independently and hence allows XFS to be able to perform concurrent space management operations. Space management concurrency is limited by the number of AGs in the filesystem. This concurrency limitation will be seen by workloads that heavily stress the space management routines. Contrary to commen expectations, data intensive workloads don't tend to be space management concurrency limited - concurrent directory and attribute modification workloads tend to stress space management concurrency far more. Hence tuning a filesystem for the correct number of AGs is not as straight forward as it may seem.

The `mkfs.xfs` default configuration is dependent on the configuration of the underlying storage. The underlying storage has the capability to sustain a certain amount IO concurrency and feeding that requires a certain amount of allocation conncurency. Filesystems on a single spindle have extremely limited IO concurrency capability, while a RAID array has far more. As such, `mkfs.xfs` will default to 4 AGs for single spindle storage and default to 32 AGs for RAID based storage.

For most workloads, the defaults will be sufficient. In general, the number of AGs only needs tuning if `mkfs.xfs` cannot detect the type of storage underlying the filesystem or a highly concurrent workload is being used on a filesystem of limited size. In these cases, ensuring that there are at least 32 AGs in the filesystem will ensure that XFS can extract all the concurrency available in the underlying storage.

The number of allocation groups can only be configured at mkfs time by using the following option:

```
mkfs.xfs -d agcount=<num>
```

See the `mkfs.xfs` man page for details.

2.1.2.1 Alignment to storage geometry

TODO: This is extremely complex and requires an entire chapter to itself.

2.1.2.2 Constraints for Growing Filesystems

When a storage system is configured to be grown in future (either by adding more hardware or thin provisioning) special consideration needs to be given to the initial filesystem layout. The size of the allocation group cannot be changed after the initial `mkfs.xfs` execution so this constrains how the filesystem can be grown.

A key concern is that the allocation groups need to be sized according to the eventual capacity of the filesystem, not the initial capacity. Unless the size of the AGs are at their maximum (1TB), the number of AGs in the fully grown filesystem should not exceed more than a few hundred. This means that for typical filesystems, a size growth of roughly 10x over the initial provisioning if the maximum recommended.

Other considerings include the underlying geometry of the storage. If you are going to grow a filesystem on a RAID array, care needs to be taken to align the device size to an exact multiple of the AG size. This means that the last AG in the filesystem does not extend onto the new space when it is grown, and so that new AG headers are correctly aligned on the newly added storage. It is also important that new storage has the same geometry as the existing storage, as the filesystem geometry cannot be changed and so cannot be optimised for storage of different geometries in the one block device.

2.1.3 Inode size and inline attributes

If the inode has sufficient space available, XFS can write attribute names and values directly into the inode. These inline attributes can be retrieved and modified up to an order of magnitude faster than retrieving separate attribute blocks, as additional I/O is not required.

The default inode size is dependent on whether metadata CRCs are enabled or not. CRCs are not enabled by default; the default inode size for this configuration is 256 bytes. When CRCs are enabled, the minimum supported inode size is increased to 512 bytes and this is used as the default.

Inline attributes are stored in the literal area of the inode. The size of this region is dynamic and is shared between data and attribute content. The maximum space available to attributes is determined by the size of the inode core and the the number of data extent pointers stored in the inode. For the default inode sizes, there is a maximum of around 100 bytes for default mkfs configurations and around 300 bytes for CRC enabled filesystems. Increasing inode size when you format the file system can increase the amount of space available for storing attributes.

When attributes are stored in the literal area of the inode, both attribute names and attribute values are limited to a maximum size of 254 bytes. If either name or value exceeds 254 bytes in length, or the total space used by the attributes exceeds the size of the literal area, the entire set of attributes stored on the inode are pushed to a separate attribute block instead of being stored inline.

The inode size can only be configured at mkfs time by using the following option:

```
mkfs.xfs -i size=<num>
```

See the `mkfs.xfs` man page for details.

2.1.4 RAID

If software RAID is in use, `mkfs.xfs` automatically configures itself with an appropriate stripe unit and width for the underlying hardware. Hardware raid may or may not expose the information in the block device to allow `mkfs.xfs` to automatically configure them. Hence stripe unit and width may need to be manually configured if hardware RAID is in use.

To configure stripe unit and width, use one of the following two options:

```
mkfs.xfs -d sunit=<num>b,swidth=<num>b
mkfs.xfs -d su=<size>,sw=<count>
```

See the `mkfs.xfs` man page for details.

2.1.5 Log size

Pending changes are written to the log prior to being written to disk; they are aggregated in memory until a synchronisation event is triggered. When this happens, the aggregated changes are written to the journal. The size of the journal determines maximum amount of change that can be aggregated in memory, as well as the number of concurrent modifications that can be in flight at once.

Therefore, the size of the log determines the concurrency of metadata modification operations the filesystem can sustain, as well as how much and how frequently metadata writeback occurs. A smaller log forces data write-back more frequently than a larger log, but can result in lower synchronisation overhead as there will be fewer changes aggreagted in memory between synchronisation triggers. Memory pressure also generates synchronisatin triggers, so large logs may not benefit systems with limited memory.

To configure the log size, use one the following `mkfs.xfs` option:

```
mkfs.xfs -l size=<num>
```

See the `mkfs.xfs` man page for details.

2.1.6 Log stripe unit

On storage devices using layouts such as RAID5 or RAID6, the log writes may perform better when they are aligned to the underlying stripe unit. That is, they start and end at stripe unit boundaries. When `mkfs.xfs` detects that the underlying storage is a RAID device, it will attempt to set the log stripe unit automatically.

If the workload being optimised triggers log synchronisation events frequently (e.g. fsync() occurs very often), then setting a log stripe unit may reduce performance even though the underlying device is a RAID device. This is caused by the log writes needing to be padded to the log stripe unit size and this causes an increase in log write latency compared to just writing a sector aligned log buffer. Hence for log write latency bound workloads, setting the log stripe unit to 1 block to trigger unaligned log writes as quickly as possible may be optimal.

To configure the log stripe unit, use one of the following two options:

```
mkfs.xfs -l sunit=<num>b
mkfs.xfs -l su=<size>
```

See the `mkfs.xfs` man page for details.

Note

The maximum log stripe unit supported is limited to the maximum log buffer size of 256KB, so the underlying storage may have a larger stripe unit than the log can be configured with. When this happens, `mkfs.xfs` will issue a warning and use 32KB as the default.

2.2 Mount options

2.2.1 Inode allocation

XFS has two different strategies for inode and data allocation for filesystems that are larger than 1 TB. The default allocation policy is `inode64`. This parameter configures XFS to allocate inodes and data across the entire file system whilst maintaining locality between inodes their related data.

The `inode32` option allocates inodes entirely within the first 1TB of filesystem space so that the inode numbers do not exceed 32 bits in size. The related file data is spread evenly across the rest of the filesystem, so there is no locality between related inodes and their data. In most cases, this configuration results is lower performance when compared to the `inode64` configuration, but may be necessary for 32 bit applications to function correctly. This is typically only a problem on 32 bit systems or on NFS exported filesystems that are mounted by 32 bit NFS clients.

2.2.2 Log buffer size and count

The larger the log buffer, the fewer IOs that are required to write the aggregated changes in memory to disk. For storage subsystems that contain non-volaile write caches, this will make little difference to performance, but for other types of storage this reduction in log IOs should help improve performance in IO intensive workloads.

The log buffer size defines the maximum amount of information that can be put in a log buffer; if a log stripe unit is not set then buffer writes can be shorter than the maximum, and hence there is no need to reduce the log buffer size for fsync heavy workloads.

The default size of the log buffer is 32KB. The maximum size is 256KB and other supported sizes are 64KB, 128KB or power of 2 multiples of the log stripe unit between 32KB and 256KB. It can be configured by use of the `logbsize` mount option.

The number of log buffers can also be configured to between 2 and 8. The default is 8 log buffersi and can be configured by the use of the `logbufs` mount option. It is rare that this needs to be configured, and it should only be considered if there is limited memory and lots of XFS filesystems such that the memory allocated to the log buffers would consume a significant amount of memory. Reducing the number of log buffers tends to decrease log performance, especially on log IO latency sensitive workloads, and so tuning this option is typically not required.

Part III

Observation

Chapter 3

TODO

TODO

Part IV

Tuning Processes

Chapter 4

TODO

TODO