Quantum®

File System Tuning Guide

StorNext 4.2

6-67367-01 Rev A

StorNext 4.2 File System Tuning Guide, 6-67367-01, Rev A, September 2011, Product of USA.

Contents

Chapter 1
StorNext File System Tuning

The StorNext File System (SNFS) provides extremely high performance for widely varying scenarios. Many factors determine the level of performance you will realize. In particular, the performance characteristics of the underlying storage system are the most critical factors. However, other components such as the Metadata Network and MDC systems also have a significant effect on performance.

Furthermore, file size mix and application I/O characteristics may also present specific performance requirements, so SNFS provides a wide variety of tunable settings to achieve optimal performance. It is usually best to use the default SNFS settings, because these are designed to provide optimal performance under most scenarios. However, this guide discusses circumstances in which special settings may offer a performance benefit.

Note: The configuration file examples in this guide show both the .cfgx (XML) format used by StorNext for Linux and the .cfg format used by Windows.

For information about locating sample configuration files, see Example FSM Configuration File on page 37.

The Underlying Storage System

The performance characteristics of the underlying storage system are the most critical factors for file system performance. Typically, RAID storage systems provide many tuning options for cache settings, RAID level, segment size, stripe size, and so on.

RAID Cache Configuration

The single most important RAID tuning component is the cache configuration. This is particularly true for small I/O operations. Contemporary RAID systems such as the EMC CX series and the various Engenio systems provide excellent small I/O performance with properly tuned caching. So, for the best general purpose performance characteristics, it is crucial to utilize the RAID system caching as fully as possible.

For example, write-back caching is absolutely essential for metadata stripe groups to achieve high metadata operations throughput.

However, there are a few drawbacks to consider as well. For example, read-ahead caching improves sequential read performance but might reduce random performance. Write-back caching is critical for small write performance but may limit peak large I/O throughput.

Caution: Some RAID systems cannot safely support write-back caching without risk of data loss, which is not suitable for critical data such as file system metadata.

Consequently, this is an area that requires an understanding of application I/O requirements. As a general rule, RAID system caching is critically important for most applications, so it is the first place to focus tuning attention.

RAID Write-Back Caching

Write-back caching dramatically reduces latency in small write operations. This is accomplished by returning a successful reply as soon as data is written into cache, and then deferring the operation of actually writing the data to the physical disks. This results in a great performance improvement for small I/O operations.

Many contemporary RAID systems protect against write-back cache data loss due to power or component failure. This is accomplished through various techniques including redundancy, battery backup, battery-backed memory, and controller mirroring. To prevent data corruption, it is important to ensure that these systems are working properly. It is particularly catastrophic if file system metadata is corrupted, because complete file system loss could result. Check with your RAID vendor to make sure that write-back caching is safe to use.

Minimal I/O latency is critically important for metadata stripe groups to achieve high metadata operations throughput. This is because metadata operations involve a very high rate of small writes to the metadata disk, so disk latency is the critical performance factor. Write-back caching can be an effective approach to minimizing I/O latency and optimizing metadata operations throughput. This is easily observed in the hourly File System Manager (FSM) statistics reports in the **cvlog** file. For example, here is a message line from the **cvlog** file:

```
PIO HiPriWr SUMMARY SnmsMetaDisk0 sysavg/350 sysmin/333
sysmax/367
```

This statistics message reports average, minimum, and maximum write latency (in microseconds) for the reporting period. If the observed average latency exceeds 500 microseconds, peak metadata operation throughput will be degraded. For example, create operations may be around 2000 per second when metadata disk latency is below 500 microseconds. However, if metadata disk latency is around 5 milliseconds, create operations per second may be degraded to 200 or worse.

Another typical write caching approach is a “write-through.” This approach involves synchronous writes to the physical disk before returning a successful reply for the I/O operation. The write-through approach exhibits much worse latency than write-back caching; therefore, small I/O performance (such as metadata operations) is severely impacted. It is important to determine which write caching approach is employed, because the performance observed will differ greatly for small write I/O operations.

In some cases, large write I/O operations can also benefit from caching. However, some SNFS customers observe maximum large I/O throughput by disabling caching. While this may be beneficial for special large I/O scenarios, it severely degrades small I/O performance; therefore, it is suboptimal for general-purpose file system performance.

RAID Read-Ahead Caching

RAID read-ahead caching is a very effective way to improve sequential read performance for both small (buffered) and large (DMA) I/O operations. When this setting is utilized, the RAID controller pre-fetches disk blocks for sequential read operations. Therefore, subsequent application read operations benefit from cache speed throughput, which is faster than the physical disk throughput.

This is particularly important for concurrent file streams and mixed I/O streams, because read-ahead significantly reduces disk head movement that otherwise severely impacts performance.

While read-ahead caching improves sequential read performance, it does not help highly transactional performance. Furthermore, some SNFS customers actually observe maximum large sequential read throughput by disabling caching. While disabling read-ahead is beneficial in these unusual cases, it severely degrades typical scenarios. Therefore, it is unsuitable for most environments.

RAID Level, Segment Size, and Stripe Size

Configuration settings such as RAID level, segment size, and stripe size are very important and *cannot be changed after put into production*, so it is critical to determine appropriate settings during initial configuration.

The best RAID level to use for high I/O throughput is usually RAID 5. The stripe size is determined by the product of the number of disks in the RAID group and the segment size. For example, a 4+1 RAID 5 group with 64K segment size results in a 256K stripe size. The stripe size is a very critical factor for write performance because I/Os smaller than the stripe size may incur a read/modify/write penalty. It is best to configure RAID 5 settings with no more than 512K stripe size to avoid the read/modify/write penalty. The read/modify/write penalty is most noticeable in the absence of "write-back" caching being performed by the RAID controller.

The RAID stripe size configuration should typically match the SNFS `StripeBreadth` configuration setting when multiple LUNs are utilized in a stripe group. However, in some cases it might be optimal to configure the SNFS `StripeBreadth` as a multiple of the RAID stripe size, such as when the RAID stripe size is small but the user's I/O sizes are very large. However, this will be suboptimal for small I/O performance, so may not be suitable for general purpose usage.

Note: If available, RAID 3 can offer better sequential performance compared to RAID 5 for streaming-intense production. RAID 5 works well for general purpose file systems but suffers compared to RAID 3 in raw streaming throughput. RAID 3 and RAID 5 offer the same redundancy. The difference is that RAID 3 dedicates a drive to parity and RAID 5 spreads it out across all drives. If RAID 3 is supported on a RAID, it should be considered, especially for customers that do a lot of media playback from their RAID.

RAID 1 mirroring is the best RAID level for metadata and journal storage because it is most optimal for very small I/O sizes. Quantum recommends using fibre channel or SAS disks (as opposed to SATA) for metadata and journal due to the higher IOPS performance and reliability. It is also very important to allocate entire physical disks for the Metadata and Journal LUNs in ordep to avoid bandwidth contention with other I/O traffic. Metadata and Journal storage requires very high IOPS rates (low latency) for optimal performance, so contention can severely impact IOPS (and latency) and thus overall performance. If Journal I/O exceeds 1ms average latency, you will observe significant performance degradation.

Note: For Metadata, RAID 1 works well, but RAID 10 (a stripe of mirrors) offers advantages. If IOpS is the primary need of the file system, RAID 10 supports additional performance by adding additional mirror pairs to the stripe. (The minimum is 4 disks, but 6 or 8 are possible). While RAID 1 has the performance of one drive (or slightly better than one drive), RAID 10 offers the performance of RAID 0 and the security of RAID 5. This suits the small and highly random nature of metadata. There are few RAIDs that support RAID 10, but if IOpS is the goal, they should be seriously considered.

It can be useful to use a tool such as **lmdd** to help determine the storage system performance characteristics and choose optimal settings. For example, varying the stripe size and running lmdd with a range of I/O sizes might be useful to determine an optimal stripe size multiple to configure the SNFS **StripeBreadth**.

Some storage vendors now provide RAID 6 capability for improved reliability over RAID 5. This may be particularly valuable for SATA disks where bit error rates can lead to disk problems. However, RAID 6 typically incurs a performance penalty compared to RAID 5, particularly for writes. Check with your storage vendor for RAID 5 versus RAID 6 recommendations.

File Size Mix and Application I/O Characteristics

It is always valuable to understand the file size mix of the target dataset as well as the application I/O characteristics. This includes the number of concurrent streams, proportion of read versus write streams, I/O size, sequential versus random, Network File System (NFS) or Common Internet File System (CIFS) access, and so on.

For example, if the dataset is dominated by small or large files, various settings can be optimized for the target size range.

Similarly, it might be beneficial to optimize for particular application I/O characteristics. For example, to optimize for sequential 1MB I/O size it would be beneficial to configure a stripe group with four 4+1 RAID 5 LUNs with 256K stripe size.

However, optimizing for random I/O performance can incur a performance trade-off with sequential I/O.

Furthermore, NFS and CIFS access have special requirements to consider as described in the Direct Memory Access (DMA) I/O Transfer section.

Direct Memory Access (DMA) I/O Transfer

To achieve the highest possible large sequential I/O transfer throughput, SNFS provides DMA-based I/O. To utilize DMA I/O, the application must issue its reads and writes of sufficient size and alignment. This is called well-formed I/O. See the **mount command** settings **auto_dma_read_length** and **auto_dma_write_length**, described in the Mount Command Options on page 25.

Buffer Cache

Reads and writes that aren't well-formed utilize the SNFS buffer cache. This also includes NFS or CIFS-based traffic because the NFS and CIFS daemons defeat well-formed I/Os issued by the application.

There are several configuration parameters that affect buffer cache performance. The most critical is the RAID cache configuration because buffered I/O is usually smaller than the RAID stripe size, and therefore incurs a read/modify/write penalty. It might also be possible to match the RAID stripe size to the buffer cache I/O size. However, it is typically most important to optimize the RAID cache configuration settings described earlier in this document.

It is usually best to configure the RAID stripe size no greater than 256K for optimal small file buffer cache performance.

For more buffer cache configuration settings, see Mount Command Options on page 25.

NFS / CIFS

It is best to isolate NFS and/or CIFS traffic off of the metadata network to eliminate contention that will impact performance. For optimal performance it is necessary to use 1000BaseT instead of 100BaseT. On NFS clients, use the `vers=3`, `rsize=262144` and `wsize=262144` mount options, and use TCP mounts instead of UDP. When possible, it is also best to utilize TCP Offload capabilities as well as jumbo frames.

It is best practice to have clients directly attached to the same network switch as the NFS or CIFS server. Any routing required for NFS or CIFS traffic incurs additional latency that impacts performance.

It is critical to make sure the **speed/duplex** settings are correct, because this severely impacts performance. Most of the time **auto-detect** is the correct setting. Some managed switches allow setting **speed/duplex** (for example `1000Mb/full`,) which disables **auto-detect** and requires the host to be set exactly the same. However, if the settings do not match between switch and host, it severely impacts performance. For example, if the switch is set to `auto-detect` but the host is set to `1000Mb/full`, you will observe a high error rate along with extremely poor performance. On Linux, the `ethtool` tool can be very useful to investigate and adjust **speed/duplex** settings.

If performance requirements cannot be achieved with NFS or CIFS, consider using a StorNext Distributed LAN client or fibre-channel attached client.

It can be useful to use a tool such as **netperf** to help verify network performance characteristics.

The NFS subtree_check Option

Although supported in previous StorNext releases, the `subtree_check` option (which controls NFS checks on a file handle being within an exported subdirectory of a file system) **is no longer supported** as of StorNext 4.0.

Reverse Path Lookup (RPL)

Beginning with release 4.0, StorNext includes a new feature called Reverse Path Lookup (RPL). When enabled, RPL provides the following benefits:

- StorNext replication reports containing lists of files show full pathnames instead of inode numbers.
- Operations involving reverse path lookup on managed file systems containing directories with very large file counts (>50,000) perform significantly better.

RPL is automatically enabled for file systems created using StorNext 4.0 and later. File systems created with StorNext releases prior to 4.0 do not have RPL enabled.

RPL can be turned on for these file systems by running the command `cvupdatefs -L`. However, there are possible side effects to dynamically enabling RPL, including the following:

- Extensive downtime to populate existing inodes with RPL information
- Increased metadata space usage (running `cvupdatefs -L` may result in as much as double the amount used)
- Decreased performance for certain inode-related operations

Therefore, consider carefully when deciding whether to enable RPL for file systems created with StorNext releases prior to 4.0.

SNFS and Virus Checking

Virus-checking software can severely degrade the performance of any file system, including SNFS. If you have anti-virus software running on a Windows Server 2003 or Windows XP machine, Quantum recommends configuring the software so that it does NOT check SNFS.

The Metadata Network

As with any client/server protocol, SNFS performance is subject to the limitations of the underlying network. Therefore, it is recommended that you use a dedicated Metadata Network to avoid contention with other network traffic. Either 100BaseT or 1000BaseT is required, but for a dedicated Metadata Network there is usually no benefit from using 1000BaseT over 100BaseT. Neither TCP offload nor are jumbo frames required.

It is best practice to have all SNFS clients directly attached to the same network switch as the MDC systems. Any routing required for metadata traffic will incur additional latency that impacts performance.

It is critical to ensure that **speed/duplex** settings are correct, as this will severely impact performance. Most of the time **auto-detect** is the correct setting. Some managed switches allow setting **speed/duplex**, such as **100Mb/full**, which disables **auto-detect** and requires the host to be set exactly the same. However, performance is severely impacted if the settings do not match between switch and host. For example, if the switch is set to **auto-detect** but the host is set to **100Mb/full**, you will observe a high error rate and extremely poor performance. On Linux the **ethtool** tool can be very useful to investigate and adjust **speed/duplex** settings.

It can be useful to use a tool like **netperf** to help verify the Metadata Network performance characteristics. For example, if **netperf -t TCP_RR -H <host>** reports less than 4,000 transactions per second capacity, a performance penalty may be incurred. You can also use the **netstat** tool to identify tcp retransmissions impacting performance. The cvadmin "latency-test" tool is also useful for measuring network latency.

Note the following configuration requirements for the metadata network:

- In cases where gigabit networking hardware is used and maximum StorNext performance is required, a separate, dedicated switched Ethernet LAN is recommended for the StorNext metadata network. If maximum StorNext performance is not required, shared gigabit networking is acceptable.
- A separate, dedicated switched Ethernet LAN is mandatory for the metadata network if 100 Mbit/s or slower networking hardware is used.
- StorNext does not support file system metadata on the same network as iSCSI, NFS, CIFS, or VLAN data when 100 Mbit/s or slower networking hardware is used.

The Metadata Controller System

The CPU power and memory capacity of the MDC System are important performance factors, as well as the number of file systems hosted per system. In order to ensure fast response time it is necessary to use dedicated systems, limit the number of file systems hosted per system (maximum 8), and have an adequate CPU and memory.

Some metadata operations such as file creation can be CPU intensive, and benefit from increased CPU power. The MDC platform is important in these scenarios because lower clock- speed CPUs such as Sparc degrade performance.

Other operations can benefit greatly from increased memory, such as directory traversal. SNFS provides three config file settings that can be used to realize performance gains from increased memory: `BufferCacheSize`, `InodeCacheSize`, and `ThreadPoolSize`.

However, it is critical that the MDC system have enough physical memory available to ensure that the FSM process doesn’t get swapped out. Otherwise, severe performance degradation and system instability can result.

The operating system on the metadata controller must always be run in U.S. English.

FSM Configuration File Settings

The following FSM configuration file settings are explained in greater detail in the `cvfs_config` man page. For a sample FSM configuration file, see Example FSM Configuration File on page 37.

The examples in the following sections are excerpted from the sample configuration file from Example FSM Configuration File on page 37.

Stripe Groups

Splitting apart data, metadata, and journal into separate stripe groups is usually the most important performance tactic. The **create**, **remove**, and **allocate** (e.g., write) operations are very sensitive to I/O latency of the journal stripe group. Configuring a separate stripe group for journal greatly benefits the speed of these operations because disk seek latency is minimized. However, if **create**, **remove**, and **allocate** performance aren't critical, it is okay to share a stripe group for both metadata and journal, but be sure to set the exclusive property on the stripe group so it doesn't get allocated for data as well.

Note: It is recommended that you have only a single metadata stripe group. For increased performance, use multiple LUNs (2 or 4) for the stripe group.

RAID 1 mirroring is optimal for metadata and journal storage. Utilizing the write-back caching feature of the RAID system (as described previously) is critical to optimizing performance of the journal and metadata stripe groups.

Example (Linux)

```
<stripeGroup index="0" name="MetaFiles" status="up"
stripeBreadth="262144" read="true" write="true"
metadata="true" journal="false" userdata="false"
realTimeIOs="200" realTimeIOsReserve="1"
realTimeMB="200" realTimeMBReserve="1"
realTimeTokenTimeout="0" multipathMethod="rotate">
   <disk index="0" diskLabel="CvfsDisk0"
   diskType="MetaDrive"/>
</stripeGroup>
<stripeGroup index="1" name="JournFiles" status="up"
stripeBreadth="262144" read="true" write="true"
metadata="false" journal="true" userdata="false"
```

```
realTimeIOs="0" realTimeIOsReserve="0" realTimeMB="0"
realTimeMBReserve="0" realTimeTokenTimeout="0"
multipathMethod="rotate">
   <disk index="0" diskLabel="CvfsDisk1"
   diskType="JournalDrive"/>
</stripeGroup>
<stripeGroup index="4" name="RegularFiles" status="up"
stripeBreadth="262144" read="true" write="true"
metadata="false" journal="false" userdata="true"
realTimeIOs="0" realTimeIOsReserve="0" realTimeMB="0"
realTimeMBReserve="0" realTimeTokenTimeout="0"
multipathMethod="rotate">
   <disk index="0" diskLabel="CvfsDisk14"
   diskType="DataDrive"/>
   <disk index="1" diskLabel="CvfsDisk15"
   diskType="DataDrive"/>
   <disk index="2" diskLabel="CvfsDisk16"
   diskType="DataDrive"/>
   <disk index="3" diskLabel="CvfsDisk17"
   diskType="DataDrive"/>
</stripeGroup>
```

Example (Windows)

```
[StripeGroup MetaFiles]
Status Up
StripeBreadth 256K
Metadata Yes
Journal No
Exclusive Yes
Read Enabled
Write Enabled
Rtmb 200
Rtios 200
RtmbReserve 1
RtiosReserve 1
RtTokenTimeout 0
MultiPathMethod Rotate
Node CvfsDisk0 0
```

```
[StripeGroup JournFiles]
Status Up
StripeBreadth 256K
Metadata No
Journal Yes
Exclusive Yes
Read Enabled
Write Enabled
Rtmb 0
Rtios 0
RtmbReserve 0
RtiosReserve 0
RtTokenTimeout 0
MultiPathMethod Rotate
Node CvfsDisk1 0

[StripeGroup RegularFiles]
Status Up
StripeBreadth 256K
Metadata No
Journal No
Exclusive No
Read Enabled
Write Enabled
Rtmb 0
Rtios 0
RtmbReserve 0
RtiosReserve 0
RtTokenTimeout 0
MultiPathMethod Rotate
Node CvfsDisk14 0
Node CvfsDisk15 1
Node CvfsDisk16 2
Node CvfsDisk17 3
```

Affinities

Affinities are another stripe group feature that can be very beneficial. Affinities can direct file allocation to appropriate stripe groups according to performance requirements. For example, stripe groups can be set up with unique hardware characteristics such as fast disk versus slow disk, or wide stripe versus narrow stripe. Affinities can then be employed to steer files to the appropriate stripe group.

For optimal performance, files that are accessed using large DMA-based I/O could be steered to wide-stripe stripe groups. Less performance-critical files could be steered to slow disk stripe groups. Small files could be steered clear of large files, or to narrow-stripe stripe groups.

Example (Linux)

```
<stripeGroup index="3" name="AudioFiles" status="up"
stripeBreadth="1048576" read="true" write="true"
metadata="false" journal="false" userdata="true"
realTimeIOs="0" realTimeIOsReserve="0" realTimeMB="0"
realTimeMBReserve="0" realTimeTokenTimeout="0"
multipathMethod="rotate">
   <affinities exclusive="true">
      <affinity>Audio</affinity>
   </affinities>
   <disk index="0" diskLabel="CvfsDisk10"
   diskType="AudioDrive"/>
   <disk index="1" diskLabel="CvfsDisk11"
   diskType="AudioDrive"/>
   <disk index="2" diskLabel="CvfsDisk12"
   diskType="AudioDrive"/>
   <disk index="3" diskLabel="CvfsDisk13"
   diskType="AudioDrive"/>
</stripeGroup>
```

Example (Windows)

```
[StripeGroup AudioFiles]
Status Up
StripeBreadth 1M
Metadata No
Journal No
Exclusive Yes
Read Enabled
Write Enabled
Rtmb 0
```

```
Rtios 0
RtmbReserve 0
RtiosReserve 0
RtTokenTimeout 0
MultiPathMethod Rotate
Node CvfsDisk10 0
Node CvfsDisk11 1
Node CvfsDisk12 2
Node CvfsDisk13 3
Affinity Audio
```

Note: Affinity names cannot be longer than eight characters.

StripeBreadth

This setting should match the RAID stripe size or be a multiple of the RAID stripe size. Matching the RAID stripe size is usually the most optimal setting. However, depending on the RAID performance characteristics and application I/O size, it might be beneficial to use a multiple or integer fraction of the RAID stripe size. For example, if the RAID stripe size is 256K, the stripe group contains 4 LUNs, and the application to be optimized uses DMA I/O with 8MB block size, a `StripeBreadth` setting of 2MB might be optimal. In this example the 8MB application I/O is issued as 4 concurrent 2MB I/Os to the RAID. This concurrency can provide up to a 4X performance increase. This typically requires some experimentation to determine the RAID characteristics. The `lmdd` utility can be very helpful. Note that this setting is not adjustable after initial file system creation.

Optimal range for the `StripeBreadth` setting is 128K to multiple megabytes, but this varies widely. *This setting cannot be changed after being put into production*, so its important to choose the setting carefully during initial configuration.

Example (Linux)

```
<stripeGroup index="2" name="VideoFiles" status="up"
stripeBreadth="4194304" read="true" write="true"
metadata="false" journal="false" userdata="true"
realTimeIOs="0" realTimeIOsReserve="0" realTimeMB="0"
```

```
realTimeMBReserve="0" realTimeTokenTimeout="0"
multipathMethod="rotate">
   <affinities exclusive="true">
      <affinity>Video</affinity>
   </affinities>
   <disk index="0" diskLabel="CvfsDisk2"
   diskType="VideoDrive"/>
   <disk index="1" diskLabel="CvfsDisk3"
   diskType="VideoDrive"/>
   <disk index="2" diskLabel="CvfsDisk4"
   diskType="VideoDrive"/>
   <disk index="3" diskLabel="CvfsDisk5"
   diskType="VideoDrive"/>
   <disk index="4" diskLabel="CvfsDisk6"
   diskType="VideoDrive"/>
   <disk index="5" diskLabel="CvfsDisk7"
   diskType="VideoDrive"/>
   <disk index="6" diskLabel="CvfsDisk8"
   diskType="VideoDrive"/>
   <disk index="7" diskLabel="CvfsDisk9"
   diskType="VideoDrive"/>
</stripeGroup>
```

Example (Windows)

```
[StripeGroup VideoFiles]
Status Up
StripeBreadth 4M
Metadata No
Journal No
Exclusive Yes
Read Enabled
Write Enabled
Rtmb 0
Rtios 0
RtmbReserve 0
RtiosReserve 0
RtTokenTimeout 0
MultiPathMethod Rotate
Node CvfsDisk2 0
Node CvfsDisk3 1
Node CvfsDisk4 2
Node CvfsDisk5 3
Node CvfsDisk6 4
Node CvfsDisk7 5
Node CvfsDisk8 6
Node CvfsDisk9 7
Affinity Video
```

BufferCacheSize

This setting consumes up to 2X bytes of memory times the number specified. Increasing this value can reduce latency of any metadata operation by performing a **hot cache** access to directory blocks, inode information, and other metadata info. This is about 10 to 1000 times faster than I/O. It is especially important to increase this setting if metadata I/O latency is high, (for example, more than 2ms average latency). Quantum recommends sizing this according to how much memory is available; more is better. Optimal settings for `BufferCacheSize` range from 16MB to 128MB for a new file system and can be increased to 256MB or 512MB as a file system grows. A higher setting is more effective if the CPU is not heavily loaded.

Example (Linux)

```
<bufferCacheSize>33554432</bufferCacheSize>
```

Example (Windows)

```
BufferCacheSize 32M
```

InodeCacheSize

This setting consumes about 800 to 1000 bytes of memory times the number specified. Increasing this value can reduce latency of any metadata operation by performing a hot cache access to inode information instead of an I/O to get inode info from disk, about 100 to 1000 times faster. It is especially important to increase this setting if metadata I/O latency is high, (for example, more than 2ms average latency). You should try to size this according to the sum number of working set files for all clients. Optimal settings for `InodeCacheSize` range from 16K to 128K for a new file system and can be increased to

256K or 512K as a file system grows. A higher setting is more effective if the CPU is not heavily loaded.

Example (Linux)

```
<inodeCacheSize>32768</inodeCacheSize>
```

Example (Windows)

```
InodeCacheSize 32K
```

ThreadPoolSize

This setting consumes up to 512 KB memory times the number specified. Increasing this value can improve concurrency of metadata operations. For example, if many client processes are executing concurrently, the thread pool can become exhausted by I/O wait time. Increasing the thread pool size permits hot cache operations to be processed that would otherwise be backed up behind the I/O-bound operations. There are various O/S limits to the number of threads that can cause fatal problems for the FSM daemon, so it's not a good idea to set this setting too high. A range from 32 to 128 is recommended, depending on the amount of available memory. It is recommended to size it according to the **max threads** FSM hourly statistic reported in the `cvlog` file. Optimal settings for `ThreadPoolSize` range from 32K to 128K.

Note: **ThreadPoolSize** should be adjusted until the Max Threads hourly statistic no longer tops out at one half of the value of **ThreadPoolSize**. Therefore, a setting of 32 is too small when 16 is seen in the hourly logs. This value does not have to be a power of 2, but it should be even.

Example (Linux)

```
<threadPoolSize>32</threadPoolSize>
```

Example (Windows)

```
ThreadPoolSize 32
```

FsBlockSize

The `FsBlockSize` (FSB), metadata disk size, and `JournalSize` settings all work together. For example, the `FsBlockSize` must be set correctly in order for the metadata sizing to be correct. `JournalSize` is also dependent on the `FsBlockSize`.

For `FsBlockSize` the optimal settings for both performance and space utilization are in the range of 16K or 64K.Settings greater than 64K are not recommended because performance will be adversely impacted due to inefficient metadata I/O operations. Values less than 16K are not recommended in most scenarios because startup and failover time may be adversely impacted. Setting `FsBlockSize` to higher values is important for multiterabyte file systems for optimal startup and failover time.

Note: This is particularly true for slow CPU clock speed metadata servers such as Sparc. However, values greater than 16K can severely consume metadata space in cases where the file-to-directory ratio is low (e.g., less than 100 to 1).

For metadata disk size, you must have a *minimum* of 25 GB, with more space allocated depending on the number of files per directory and the size of your file system.

The following table shows suggested `FsBlockSize` (FSB) settings and metadata disk space based on the average number of files per directory and file system size. The amount of disk space listed for metadata is *in addition* to the 25 GB minimum amount. Use this table to determine the setting for your configuration.

Average No. of Files Per Directory	File System SIze: Less Than 10TB	File System Size: 10TB or Larger
Less than 10	FSB: 16KB Metadata: 32 GB per 1M files	FSB: 64KB Metadata: 128 GB per 1M files
10-100	FSB: 16KB Metadata: 8 GB per 1M files	FSB: 64KB Metadata: 32 GB per 1M files

Average No. of Files Per Directory	File System SIze: Less Than 10TB	File System Size: 10TB or Larger
100-1000	FSB: 64KB Metadata: 8 GB per 1M files	FSB: 64KB Metadata: 8 GB per 1M files
1000 +	FSB: 64KB Metadata: 4 GB per 1M files	FSB: 64KB Metadata: 4 GB per 1M files

This setting is not adjustable after initial file system creation, so it is very important to give it careful consideration during initial configuration.

Example (Linux)

```
<config configVersion="0" name="example"
fsBlockSize="16384" journalSize="16777216">
```

Example (Windows)

```
FsBlockSize 16K
```

JournalSize

The optimal settings for `JournalSize` are in the range between 16M and 64M, depending on the `FsBlockSize`. Avoid values greater than 64M due to potentially severe impacts on startup and failover times. Values at the higher end of the 16M-64M range may improve performance of metadata operations in some cases, although at the cost of slower startup and failover time.

The following table shows recommended settings. Choose the setting that corresponds to your configuration.

FsBlockSize	JournalSize
16KB	16MB
64KB	64MB

This setting is adjustable using the **cvupdatefs** utility. For more information, see the **cvupdatefs** man page.

Note: **JournalSize** should be evaluated after a few months of use by viewing the hourly statistics and looking for any journal waits. If there are many in a single hour, consider increasing the journal size and then reexamine the hourly statistics to see if the bottleneck has moved to some other part of the file system (like **ThreadPoolSize** or cache misses) or the hardware (high sysmax and sysavg times).

Example (Linux)

```
<config configVersion="0" name="example"
fsBlockSize="16384" journalSize="16777216">
```

Example (Windows)

```
JournalSize 16M
```

SNFS Tools

The **snfsdefrag** tool is very useful to identify and correct file extent fragmentation. Reducing extent fragmentation can be very beneficial for performance. You can use this utility to determine whether files are fragmented, and if so, fix them.

The global configuration settings **InodeExpandMin**, **InodeExpandInc**, and **InodeExpandMax** have been deprecated and settings are instead calculated on a file-by-file basis as allocations are performed. This results in better allocations for more files as the values are no longer a compromise if there are widely varying file types on the file system. However, if a majority of the files are still fragmented, then these values can be adjusted and will override the default behavior.

Note: Beginning with StorNext 4.0, the InodeExpand parameters have been replaced by a new method called Optimistic Allocation. Although the InodeExpand parameters can still be entered and used in StorNext 4.0 and later, Quantum recommends using Optimistic Allocation instead.

For a comparison between InodeExpand and Optimistic Allocation, see Optimistic Allocation on page 27.

Another way to combat fragmentation is with the **cachebufsize** mount option (increasing it from the default of 64k to something larger, such as 256K or 512K) on the clients that are creating the fragmented files, or by altering the way the application writes data to the SAN. The **InodeExpand** parameters are file system wide and can be adjusted after the file system has been created. The **cachebufsize** parameter is a mount option and can be unique for every client that mounts the file system.

FSM hourly statistics reporting is another very useful tool. This can show you the mix of metadata operations being invoked by client processes, as well as latency information for metadata operations and metadata and journal I/O. This information is easily accessed in the cvlog log files. All of the latency oriented stats are reported in microsecond units.

It also possible to trigger an instant FSM statistics report by setting the **Once Only** debug flag using **cvadmin**. For example:

```
cvadmin -F snfs1 -e ‘debug 0x01000000’ ; tail -100 /usr/
cvfs/data/snfs1/log/cvlog
```

Keep in mind the following when running **cvadmin**:

- Quantum recommended that you have only a single metadata stripe group. For increased performance, use multiple LUNs (2 or 4) for the stripe group.
- A large value for **FSM threads SUMMARY max busy** indicates the FSM configuration setting **ThreadPoolSize** is insufficient.
- Extremely high values for **FSM cache SUMMARY inode lookups**, **TKN SUMMARY TokenRequestV3**, or **TKN SUMMARY TokenReqAlloc** might indicate excessive file fragmentation. If so, the **snfsdefrag** utility can be used to fix the fragmented files.
- The **VOP** and **TKN** summary statistics of the form count avg/q+e min/q+e max/q+e show microsecond queue and execution latency for the various metadata operations. This shows what type of metadata operations are most prevalent and most costly. These are also broken out per client, which can be useful to identify a client that is disproportionately loading the FSM.

SNFS supports the Windows **Perfmon** utility. This provides many useful statistics counters for the SNFS client component. Run **rmperfreg.exe** and **instperfreg.exe** to set up the required registry settings. Next, call **cvdb -P**. After these steps, the SNFS counters should be visible to the

Windows Perfmon utility. If not, check the Windows Application Event log for errors.

The **cvcp** utility is a higher performance alternative to commands such as **cp** and **tar**. The **cvcp** utility achieves high performance by using threads, large I/O buffers, preallocation, stripe alignment, DMA I/O transfer, and Bulk Create. Also, the **cvcp** utility uses the SNFS External API for preallocation and stripe alignment. In the directory-to-directory copy mode (for example, **cvcp source_dir destination_dir**,) **cvcp** conditionally uses the **Bulk Create** API to provide a dramatic small file copy performance boost. However, it will not use **Bulk Create** in some scenarios, such as non-root invocation, managed file systems, quotas, or Windows security. When **Bulk Create** is utilized, it significantly boosts performance by reducing the number of metadata operations issued. For example, up to 20 files can be created all with a single metadata operation. For more information, see the **cvcp** man page.

The **cvmkfile** utility provides a command line tool to utilize valuable SNFS performance features. These features include preallocation, stripe alignment, and affinities. See the **cvmkfile** man page.

The **Lmdd** utility is very useful to measure raw LUN performance as well as varied I/O transfer sizes. It is part of the **lmbench** package and is available from http://sourceforge.net.

The **cvdbset** utility has a special “Perf” trace flag that is very useful to analyze I/O performance. For example: `cvdbset perf`

Then, you can use **cvdb -g** to collect trace information such as this:

```
PERF: Device Write 41 MB/s IOs 2 exts 1 offs 0x0 len
0x400000 mics 95589 ino 0x5
```

```
PERF: VFS Write EofDmaAlgn 41 MB/s offs 0x0 len 0x400000
mics 95618 ino 0x5
```

The “PERF: Device” trace shows throughput measured for the device I/O. It also shows the number of I/Os into which it was broken, and the number of extents (sequence of consecutive filesystem blocks).

The “PERF: VFS” trace shows throughput measured for the read or write system call and significant aspects of the I/O, including:

- Dma: DMA
- Buf: Buffered
- Eof: File extended

- Algn: Well-formed DMA I/O
- Shr: File is shared by another client
- Rt: File is real time
- Zr: Hole in file was zeroed

Both traces also report file offset, I/O size, latency (mics), and inode number.

Sample use cases:

- Verify that I/O properties are as expected.

  You can use the VFS trace to ensure that the displayed properties are consistent with expectations, such as being well formed; buffered versus DMA; shared/non-shared; or I/O size. If a small I/O is being performed DMA, performance will be poor. If DMA I/O is not well formed, it requires an extra data copy and may even be broken into small chunks. Zeroing holes in files has a performance impact.
- Determine if metadata operations are impacting performance.

  If VFS throughput is inconsistent or significantly less than Device throughput, it might be caused by metadata operations. In that case, it would be useful to display “fsmtoken,” “fsmvnops,” and “fsmdmig” traces in addition to “perf.”
- Identify disk performance issues.

  If Device throughput is inconsistent or less than expected, it might indicate a slow disk in a stripe group, or that RAID tuning is necessary.
- Identify file fragmentation.

  If the extent count “exts” is high, it might indicate a fragmentation problem.This causes the device I/Os to be broken into smaller chunks, which can significantly impact throughput.
- Identify read/modify/write condition.

  If buffered VFS writes are causing Device reads, it might be beneficial to match I/O request size to a multiple of the “cachebufsize” (default 64KB; see **mount_cvfs** man page). Another way to avoid this is by truncating the file before writing.

The **cvadmin** command includes a **latency-test** utility for measuring the latency between an FSM and one or more SNFS clients. This utility causes small messages to be exchanged between the FSM and clients as quickly as possible for a brief period of time, and reports the average time it took for each message to receive a response.

The **latency-test** command has the following syntax:

latency-test <index-number> [<seconds>]

latency-test all [<seconds>]

If an **index-number** is specified, the test is run between the currently-selected FSM and the specified client. (Client index numbers are displayed by the **cvadmin who** command). If **all** is specified, the test is run against each client in turn.

The test is run for 2 seconds, unless a value for seconds is specified.

Here is a sample run:

```
snadmin (lsi) > latency-test

Test started on client 1 (bigsky-node2)... latency
55us

Test started on client 2 (k4)... latency 163us
```

There is no rule-of-thumb for “good” or “bad” latency values. Latency can be affected by CPU load or SNFS load on either system, by unrelated Ethernet traffic, or other factors. However, for otherwise idle systems, differences in latency between different systems can indicate differences in hardware performance. (In the example above, the difference is a Gigabit Ethernet and faster CPU versus a 100BaseT Ethernet and a slower CPU.) Differences in latency over time for the same system can indicate new hardware problems, such as a network interface going bad.

If a latency test has been run for a particular client, the **cvadmin who long** command includes the test results in its output, along with information about when the test was last run.

Mount Command Options

The following SNFS mount command settings are explained in greater detail in the **mount_cvfs** man page.

The default size of the buffer cache varies by platform and main memory size, and ranges between 32MB and 256MB. And, by default, each buffer is 64K so the cache contains between 512 and 4096 buffers. In general, increasing the size of the buffer cache will not improve performance for streaming reads and writes. However, a large cache helps greatly in cases of multiple concurrent streams, and where files are being written and subsequently read. Buffer cache size is adjusted with the buffercachecap setting.

The buffer cache I/O size is adjusted using the **cachebufsize** setting. The default setting is usually optimal; however, sometimes performance can be improved by increasing this setting to match the RAID 5 stripe size.

Using a large **cachebufsize** setting decreases random I/O performance when the amount of data being read is smaller than the cache buffer size.

You can combat fragmentation with the **cachebufsize** mount option (increasing it from the default of 64k to something larger, such as 256K or 512K) on the clients that are creating the fragmented files, or by altering the way the application writes data to the SAN. The **InodeExpand** parameters are file system wide and can be adjusted after the file system has been created. The **cachebufsize** parameter is a mount option and can be unique for every client that mounts the file system.

Buffer cache read-ahead can be adjusted with the **buffercache_readahead** setting. When the system detects that a file is being read in its entirety, several buffer cache I/O daemons pre-fetch data from the file in the background for improved performance. The default setting is optimal in most scenarios.

The **auto_dma_read_length** and **auto_dma_write_length** settings determine the minimum transfer size where direct DMA I/O is performed instead of using the buffer cache for well-formed I/O. These settings can be useful when performance degradation is observed for small DMA I/O sizes compared to buffer cache. The **auto_dma_write_length** setting should be tuned with InodeExpand to get optimal allocations.

For example, if buffer cache I/O throughput is 200 MB/sec but 512K DMA I/O size observes only 100MB/sec, it would be useful to determine which DMA I/O size matches the buffer cache performance and adjust **auto_dma_read_length** and **auto_dma_write_length** accordingly. The **lmdd** utility is handy here.

The **dircachesize** option sets the size of the directory information cache on the client. This cache can dramatically improve the speed of readdir operations by reducing metadata network message traffic between the SNFS client and FSM. Increasing this value improves performance in scenarios where very large directories are not observing the benefit of the client directory cache.

SNFS External API

The SNFS External API might be useful in some scenarios because it offers programmatic use of special SNFS performance capabilities such as affinities, preallocation, and quality of service. For more information, see the “Quality of Service” chapter of the *StorNext User’s Guide API Guide*.

Optimistic Allocation

Starting with StorNext 4.0, the `InodeExpand` parameters (`InodeExpandMin`, `InodeExpandInc`, and `InodeExpandMax`) in the file system configuration file have been deprecated and replaced by a simple formula that should work better in most cases, especially with very large files.

The `InodeExpand` values are still honored if they are in the .cfgx file, but the StorNext GUI no longer lets you set these values. Furthermore, when converting to StorNext 4.0 and later, during the .cfg to .cfgx conversion process, if the `InodeExpand` values in the .cfg file are found to be the default example values, these values are not set in the new .cfgx. Instead, the new formula is used.

The original `InodeExpand` configuration was difficult to explain, which could lead to misconfigurations that caused either over or under allocations (resulting in wasted space or fragmentation,) which is why the new formula seeks to use allocations that are a percentage of the existing file's size to minimize wasted space and fragmentation.

How Optimistic Allocation Works

The `InodeExpand` values come into play whenever a write to disk is done, and works as an "optimistic allocator." It is referred to as “optimistic” because it works under the assumption that where there is one allocation, there will be another, so it allocates more than you asked for believing that you'll use the over-allocated space soon.

There are three ways to do a DMA I/O:

- By having an I/O larger than `auto_dma_write_length` (or `auto_dma_read_length`, but that doesn't cause an allocation so it will be ignored for this case)

- Doing a write to a file that was opened with `O_DIRECT`
- Opening a file for writes that's already open for writes by another client (commonly referred to as "shared write mode" which requires all I/Os go straight to disk to maintain coherency between the clients)

The first allocation is the larger of the `InodeExpandMin` or the actual IO size. For example, if the `InodeExpandMin` is 2MB and the initial IO is 1MB, the file gets a 2MB allocation. However, if the initial IO was 3MB and the `InodeExpandMin` is 2MB, the file gets only a 3MB allocation.

In both cases, the `InodeExpandMin` value is saved in an internal data structure in the file's inode, to be used with subsequent allocations. Subsequent DMA IOs that require more space to be allocated for the file add to the `InodeExpandInc` value saved in the inode, and the allocation is the larger of this value or the IO size.

For example, if `InodeExpandMin` is 2MB and `InodeExpandInc` is 4MB and the first I/O is 1MB, then the file is initially 2MB in size. On the third 1MB I/O the file is extended by 6MB (2MB + 4MB) and is now 8MB though it only has 3MB of data in it. However, that 6MB allocation is likely contiguous and therefore the file has at most 2 fragments which is better than 8 fragments it would have had otherwise.

Assuming there are more 1MB I/Os to the file, it will continue to expand in this manner. The next DMA I/O requiring an allocation over the 8MB mark will extend the file by 10MB (2MB + 4MB + 4MB). This pattern repeats until the file's allocation value is equal to or larger than `InodeExpandMax`, at which point it's capped at `InodeExpandMax`.

This formula generally works well when it's tuned for the specific I/O pattern. If it's not tuned, with certain I/O patterns it can cause suboptimal allocations resulting in excess fragmentation or wasted space from files being over allocated.

This is especially true if there are small files created with `O_DIRECT`, or small files that are simultaneously opened by multiple clients which cause them to use an `InodeExpandMin` that's too large for them. Another possible problem is an `InodeExpandMax` that's too small, causing the file to be composed of fragments smaller than it otherwise could have been created with.

With very large files, without increasing `InodeExpandMax`, it can create fragmented files due to the relatively small size of the allocations and the large number that are needed to create a large file.

Another possible problem is an `InodeExpandInc` that's not aggressive enough, again causing a file to be created with more fragments than it could be created with, or to never reach `InodeExpandMax` because writes stop before it can be incremented to that value.

Note: Although the preceding example uses DMA I/O, the `InodeExpand` parameters apply to both DMA and non-DMA allocations.

Optimistic Allocation Formula

The following table shows the new formula (beginning with StorNext 4.x):

File Size (in bytes)	Optimistic Allocation
<= 16MB	1MB
16MB to 64MB + 4 bytes	4MB
64MB + 4 bytes to 256MB + 16 bytes	16MB
256MBs + 16 bytes to 1 GB + 64 bytes	64MB
1GB + 64 bytes to 4GB + 256 bytes	256MB
4GB + 256 bytes to 16GB + 1k bytes	1GB
16GB + 1k bytes to 64GB + 4k bytes	4GB
64GB + 4k bytes to 256GB + 16k bytes	16GB
256GB + 16k bytes to 1TB + 64k bytes	64GB
1TB + 64k bytes or larger	256GB

To examine how well these allocation strategies work in your specific environment, use the `snfsdefrag` utility with the `-e` option to display the individual extents (allocations) in a file.

Here is an example output from `snfsdefrag -e testvideo2.mov`:

```
testvideo2.mov:
#      group  frbase         fsbase       fsend        kbytes     depth
0      7      0x0            0xa86df6     0xa86df6     16         4
1      7      0x4000         0x1fb79b0    0x1fb79e1    800        4
HOLE @ frbase 0xcc000 for 41 blocks (656 kbytes)
2      7      0x170000       0x57ca034    0x57ca03f    192        4
3      7      0x1a0000       0x3788860    0x3788867    128        4
4      7      0x1c0000       0x68f6cb4    0x68f6cff    1216       4
5      7      0x2f0000       0x70839dd    0x70839df    48         4
```

Here is an explanation of the column headings:

- `#`: This is the extent index.
- `group`: The group column tells you which stripe group on which the extent resides. Usually it's all on the same stripe group, but not always.
- `frbase`: This is the file's logical offset
- `fsbase` and `fsend`: These are the StorNext logical start and end addresses and should be ignored.
- `kbytes`: This is the size of the extent (fragment)
- `depth`: This tells you the number of LUNs that existed in the stripe group when the file was written. If you perform bandwidth expansion, this number is the old number of LUNs before bandwidth expansion, and signifies that those files aren't taking advantage of the bandwidth expansion.

If the file is sparse, you will see "`HOLE`" displayed. Having holes in a file isn't necessarily a problem, but it does create extra fragments (one for each side of the hole). Tuning to eliminate holes can reduce fragmentation, although it does that by using more disk space.

The Distributed LAN (Disk Proxy) Networks

As with any client/server protocol, SNFS Distributed LAN performance is subject to the limitations of the underlying network. Therefore, it is strongly recommended that you use Gigabit (1000BaseT) for Distributed LAN traffic. Neither TCP offload nor jumbo frames are required.

Hardware Configuration

SNFS Distributed LAN can easily fill several Gigabit Ethernets with data, so take special care when selecting and configuring the switches used to interconnect SNFS Distributed LAN clients and servers. Ensure that your network switches have enough internal bandwidth to handle all of the anticipated traffic between all Distributed LAN clients and servers connected to them.

A network switch that is dropping packets will cause TCP retransmissions. This can be easily observed on both Linux and Windows platforms by using the **netstat -s** command while Distributed LAN is in progress. Reducing the TCP window size used by Distributed LAN might also help with an oversubscribed network switch. The Windows client **Distributed LAN** tab and the Linux **dpserver** file contain the tuning parameter for the TCP window size. Note that Distributed LAN server remounts are required after changing this parameter.

It is best practice to have all SNFS Distributed LAN clients and servers directly attached to the same network switch. A router between a Distributed LAN client and server could be easily overwhelmed by the data rates required.

It is critical to ensure that **speed/duplex** settings are correct, as this will severely impact performance. Most of the time **auto-detect** is the correct setting. Some managed switches allow setting **speed/duplex**, such as **1000Mb/full**, which disables **auto-detect** and requires the host to be set exactly the same. However, performance is severely impacted if the settings do not match between switch and host. For example, if the switch is set to **auto-detect** but the host is set to **1000Mb/full**, you will observe a high error rate and extremely poor performance. On Linux the **ethtool** command can be very useful to investigate and adjust **speed/ duplex** settings.

In some cases, TCP offload seems to cause problems with Distributed LAN by miscalculating checksums under heavy loads. This is indicated by

bad segments indicated in the output of **netstat -s**. On Linux, the TCP offload state can be queried by running **ethtool -k**, and modified by running **ethtool -K**. On Windows it is configured through the **Advanced** tab of the configuration properties for a network interface.

The internal bus bandwidth of a Distributed LAN client or server can also place a limit on performance. A basic PCI- or PCI-X-based workstation might not have enough bus bandwidth to run multiple Gigabit Ethernet NICs at full speed; PCI Express is recommended but not required.

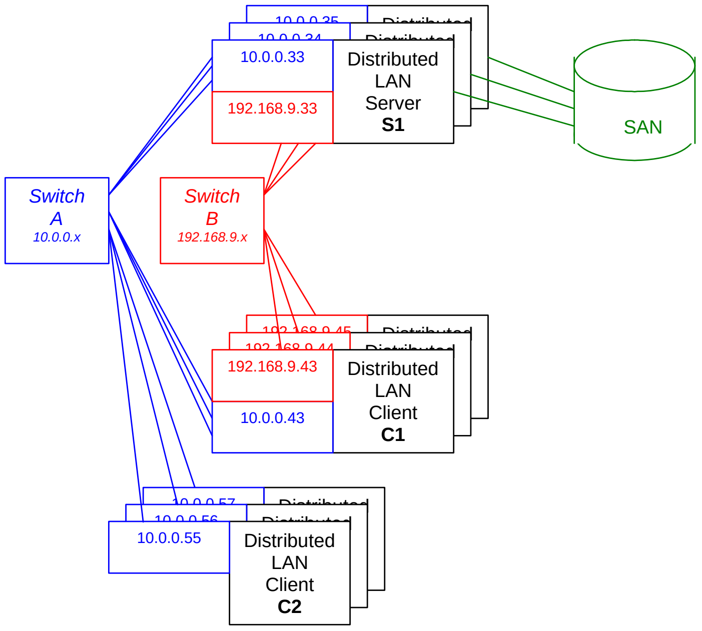
Similarly, the performance characteristics of NICs can vary widely and ultimately limit the performance of Distributed LAN. For example, some NICs might be able to transmit or receive each packet at Gigabit speeds, but not be able to sustain the maximum needed packet rate. An inexpensive 32-bit NIC plugged into a 64-bit PCI-X slot is incapable of fully utilizing the host's bus bandwidth.

It can be useful to use a tool like **netperf** to help verify the performance characteristics of each Distributed LAN network. (When using **netperf**, on a system with multiple NICs, take care to specify the right IP addresses in order to ensure the network being tested is the one you will be running Distributed LAN over. For example, if **netperf -t TCP_RR -H <host>** reports less than 4,000 transactions per second capacity, a performance penalty might be incurred. Multiple copies of **netperf** can also be run in parallel to determine the performance characteristics of multiple NICs.

Network Configuration and Topology

For maximum throughput, SNFS distributed LAN can utilize multiple NICs on both clients and servers. In order to take advantage of this feature, each of the NICs on a given host must be on a different IP subnetwork. (This is a requirement of TCP/IP routing, not of SNFS - TCP/IP can't utilize multiple NICs on the same subnetwork.) An example of this is shown in the following illustration.

Figure 1 Multi-NIC Hardware and IP Configuration Diagram



In the diagram there are two subnetworks: the blue subnetwork (10.0.0.x) and the red subnetwork (192.168.9.x). Servers such as S1 are connected to both the blue and red subnetworks, and can each provide up to 2 GByte/s of throughput to clients. (The three servers shown would thus provide an aggregate of 6 GByte/s.)

Clients such as C1 are also connected to both the blue and red subnetworks, and can each get up to 2 GByte/s of throughput. Clients such as C2 are connected only to the blue subnetwork, and thus get a maximum of 1 GByte/s of throughput. SNFS automatically load-balances among NICs and servers to maximize throughput for all clients.

Note: The diagram shows separate physical switches used for the two subnetworks. They can, in fact, be the same switch, provided it has sufficient internal bandwidth to handle the aggregate traffic.

Distributed LAN Servers

Distributed LAN Servers must have sufficient memory. When a Distributed LAN Server does not have sufficient memory, its performance in servicing Distributed LAN I/O requests might suffer. In some cases (particularly on Windows,) it might hang.

Refer to the StorNext Release Notes for this release's memory requirements.

Distributed LAN Servers must also have sufficient bus bandwidth. As discussed above, a Distributed LAN Server must have sufficient bus bandwidth to operate the NICs used for Distributed LAN I/O at full speed, while at the same time operating their Fibre Channel HBAs. Thus, Quantum strongly recommends using PCI Express for Distributed LAN Servers.

Distributed LAN Client Vs. Legacy Network Attached Storage

StorNext provides support for legacy Network Attached Storage (NAS) protocols, including Network File System (NFS) and Common Internet File System (CIFS).

However, using Distributed LAN Client (DLC) for NAS connectivity provides several compelling advantages in the following areas:

- Performance
- Fault Tolerance
- Load Balancing
- Client Scalability
- Robustness and Stability
- Security Model Consistency

Performance

DLC outperforms NFS and CIFS for single-stream I/O and provides higher aggregate bandwidth. For inferior NFS client implementations, the difference can be more than a factor of two. DLC also makes extremely efficient use of multiple NICs (even for single streams), whereas legacy NAS protocols allow only a single NIC to be used. In addition, DLC clients communicate directly with StorNext metadata controllers instead of going through an intermediate server, thereby lowering IOP latency.

Fault Tolerance

DLC handles faults transparently, where possible. If an I/O is in progress and a NIC fails, the I/O is retried on another NIC (if one is available). If a Distributed LAN Server fails while an I/O is in flight, the I/O is retried on another server (if one is running). When faults occur, applications performing I/O will experience a delay but not an error, and no administrative intervention is required to continue operation. These fault tolerance features are automatic and require no configuration.

Load Balancing

DLC automatically makes use of all available Distributed LAN Servers in an active/active fashion, and evenly spreads I/O across them. If a server goes down or one is added, the load balancing system automatically adjusts to support the new configuration.

Client Scalability

The following table shows testing results using NFS, CIFS and DLC.

Largest Tested Configuration			
Number of Clients Tested (via simulation)	**NFS**	**CIFS**	**DLC**
	4	4	1000

Robustness and Stability

The code path for DLC is simpler, involves fewer file system stacks, and is not integrated with kernel components that constantly change with every operating system release (for example, the Linux NFS code). Therefore, DLC provides increased stability that is comparable to the StorNext SAN Client.

Consistent Security Model

DLC clients have the same security model as StorNext SAN clients. When CIFS and NFS are used, some security models aren't supported. (For example, Windows ACLs are not accessible when running UNIX Samba servers.)

Windows Memory Requirements

Beginning in version 2.6.1, StorNext includes a number of performance enhancements that enable it to better react to changing customer load. However, these enhancements come with a price: memory requirement.

When running on a 32-bit Windows system that is experiencing memory pressure, the tuning parameters might need adjusting to avoid running the system out of non-paged memory. To determine current operation, open the Task Manager and watch the **Nonpaged** tag in the **Kernel Memory** pane in the lower right hand corner. This value should be kept under 200MB. If the non-paged pool approaches this size on a 32-bit system, instability might occur.

The problem will manifest itself by commands failing, messages being sent to the   system log about insufficient memory, the **fsmpm** mysteriously dying, repeated FSM reconnect attempts, and messages being sent to the application log and **cvlog.txt** about socket failures with the status code (10555) which is ENOBUFS.

The solution is to adjust a few parameters on the **Cache Parameters** tab in the SNFS control panel (**cvntclnt**). These parameters control how much memory is consumed by the directory cache, the buffer cache, and the local file cache.

As always, an understanding of the customers' workload aids in determining the correct values. Tuning is not an exact science, and requires some trial-and-error (and the unfortunate reboots) to come up with values that work best in the customer's environment.

The first is the **Directory Cache Size**. The default is 10 (MB). If you do not have large directories, or do not perform lots of directory scans, this number can be reduced to 1 or 2 MB. The impact will be slightly slower directory lookups in directories that are frequently accessed.

Also, in the **Mount Option** panel, you should set the **Paged DirCache** option.

The next parameters control how many file structures are cached on the client. These are controlled by the **Meta-data Cache low water mark**, **Meta-data Cache high water mark** and **Meta-data Cache Max water mark**. Each file structure is represented internally by a data structure called the “cvnode.” The cvnode represents all the state about a file or directory. The more cvnodes that there are encached on the client, the fewer trips the client has to make over the wire to contact the FSM.

Each cvnode is approximately 1462 bytes in size and is allocated from the non-paged pool. The **cvnode** cache is periodically purged so that unused entries are freed. The decision to purge the cache is made based on the **Low**, **High**, and **Max** water mark values. The 'Low' default is 1024, the 'High' default is 3072, and the 'Max' default is 4096.

These values should be adjusted so that the cache does not bloat and consume more memory than it should. These values are highly dependent on the customers work load and access patterns. Values of 512 for the **High** water mark will cause the **cvnode** cache to be purged when more than 512 entries are present. The cache will be purged until the low water mark is reached, for example 128. The **Max** water mark is for situations where memory is very tight. The normal purge algorithms takes access time into account when determining a candidate to evict from the cache; in tight memory situations (when there are more than 'max' entries in the cache), these constraints are relaxed so that memory can be released. A value of 1024 in a tight memory situation should work.

Example FSM Configuration File

On Linux, the StorNext configuration file uses an XML format (.cfgx). On Windows, the configuration file uses a text format (.cfg). However, the values contained in both files are similar.

You can locate an example StorNext configuration file in the following directory:

- Linux — **/usr/cvfs/examples/example.cfgx**
- Windows — **C:\Program Files\Stornext\config\example.cfg**

If you installed StorNext in a location other than the default installation directory, the example configuration file is located in **C:\<install_directory>\config\example.cfg**

Linux Example Configuration File

Below are the contents of the StorNext example configuration file for Linux (**example.cfgx**):

```
<?xml version="1.0" encoding="UTF-8"?>
<configDoc xmlns="http://www.quantum.com/snfs" version="1.0">
   <config configVersion="0" name="example" fsBlockSize="16384"
   journalSize="16777216">
      <globals>
         <abmFreeLimit>false</abmFreeLimit>
         <allocationStrategy>round</allocationStrategy>
         <haFsType>HaUnmonitored</haFsType>
         <bufferCacheSize>33554432</bufferCacheSize>
         <cvRootDir>/</cvRootDir>
         <storageManager>false</storageManager>
         <dataMigrationThreadPoolSize>128</dataMigrationThreadPoolSize>
         <debug>00000000</debug>
         <dirWarp>true</dirWarp>
         <extentCountThreshold>49152</extentCountThreshold>
         <enableSpotlight>false</enableSpotlight>
         <enforceAcls>false</enforceAcls>
         <fileLocks>false</fileLocks>
         <fileLockResyncTimeOut>20</fileLockResyncTimeOut>
         <forcePerfectFit>false</forcePerfectFit>
         <fsCapacityThreshold>0</fsCapacityThreshold>
         <globalSuperUser>true</globalSuperUser>
         <inodeCacheSize>32768</inodeCacheSize>
         <inodeExpandMin>0</inodeExpandMin>
         <inodeExpandInc>0</inodeExpandInc>
         <inodeExpandMax>0</inodeExpandMax>
         <inodeDeleteMax>0</inodeDeleteMax>
         <inodeStripeWidth>0</inodeStripeWidth>
         <maxConnections>32</maxConnections>
         <maxLogs>4</maxLogs>
         <remoteNotification>false</remoteNotification>
```

```
        <reservedSpace>true</reservedSpace>
        <fsmRealTime>false</fsmRealTime>
        <fsmMemLocked>false</fsmMemLocked>
        <opHangLimitSecs>180</opHangLimitSecs>
        <perfectFitSize>131072</perfectFitSize>
        <quotas>false</quotas>
        <restoreJournal>false</restoreJournal>
        <restoreJournalDir/>
        <restoreJournalMaxHours>0</restoreJournalMaxHours>
        <restoreJournalMaxMb>0</restoreJournalMaxMb>
        <stripeAlignSize>0</stripeAlignSize>
        <trimOnClose>0</trimOnClose>
        <threadPoolSize>32</threadPoolSize>
        <unixDirectoryCreationModeOnWindows>644</
        unixDirectoryCreationModeOnWindows>
        <unixIdFabricationOnWindows>false</unixIdFabricationOnWindows>
        <unixFileCreationModeOnWindows>755</unixFileCreationModeOnWindows>
        <unixNobodyUidOnWindows>60001</unixNobodyUidOnWindows>
        <unixNobodyGidOnWindows>60001</unixNobodyGidOnWindows>
        <windowsSecurity>true</windowsSecurity>
        <eventFiles>true</eventFiles>
        <eventFileDir/>
        <allocSessionReservation>false</allocSessionReservation>
    </globals>
    <diskTypes>
        <diskType typeName="MetaDrive" sectors="99999999" sectorSize="512"/>
        <diskType typeName="JournalDrive" sectors="99999999" sectorSize="512"/>
        <diskType typeName="VideoDrive" sectors="99999999" sectorSize="512"/>
        <diskType typeName="AudioDrive" sectors="99999999" sectorSize="512"/>
        <diskType typeName="DataDrive" sectors="99999999" sectorSize="512"/>
    </diskTypes>
    <stripeGroups>
        <stripeGroup index="0" name="MetaFiles" status="up"
        stripeBreadth="262144" read="true" write="true" metadata="true"
        journal="false" userdata="false" realTimeIOs="200"
        realTimeIOsReserve="1" realTimeMB="200" realTimeMBReserve="1"
        realTimeTokenTimeout="0" multipathMethod="rotate">
            <disk index="0" diskLabel="CvfsDisk0" diskType="MetaDrive"/>
        </stripeGroup>
```

```
<stripeGroup index="1" name="JournFiles" status="up"
stripeBreadth="262144" read="true" write="true" metadata="false"
journal="true" userdata="false" realTimeIOs="0" realTimeIOsReserve="0"
realTimeMB="0" realTimeMBReserve="0" realTimeTokenTimeout="0"
multipathMethod="rotate">
   <disk index="0" diskLabel="CvfsDisk1" diskType="JournalDrive"/>
</stripeGroup>
<stripeGroup index="2" name="VideoFiles" status="up"
stripeBreadth="4194304" read="true" write="true" metadata="false"
journal="false" userdata="true" realTimeIOs="0" realTimeIOsReserve="0"
realTimeMB="0" realTimeMBReserve="0" realTimeTokenTimeout="0"
multipathMethod="rotate">
   <affinities exclusive="true">
      <affinity>Video</affinity>
   </affinities>
   <disk index="0" diskLabel="CvfsDisk2" diskType="VideoDrive"/>
   <disk index="1" diskLabel="CvfsDisk3" diskType="VideoDrive"/>
   <disk index="2" diskLabel="CvfsDisk4" diskType="VideoDrive"/>
   <disk index="3" diskLabel="CvfsDisk5" diskType="VideoDrive"/>
   <disk index="4" diskLabel="CvfsDisk6" diskType="VideoDrive"/>
   <disk index="5" diskLabel="CvfsDisk7" diskType="VideoDrive"/>
   <disk index="6" diskLabel="CvfsDisk8" diskType="VideoDrive"/>
   <disk index="7" diskLabel="CvfsDisk9" diskType="VideoDrive"/>
</stripeGroup>
<stripeGroup index="3" name="AudioFiles" status="up"
stripeBreadth="1048576" read="true" write="true" metadata="false"
journal="false" userdata="true" realTimeIOs="0" realTimeIOsReserve="0"
realTimeMB="0" realTimeMBReserve="0" realTimeTokenTimeout="0"
multipathMethod="rotate">
   <affinities exclusive="true">
      <affinity>Audio</affinity>
   </affinities>
   <disk index="0" diskLabel="CvfsDisk10" diskType="AudioDrive"/>
   <disk index="1" diskLabel="CvfsDisk11" diskType="AudioDrive"/>
   <disk index="2" diskLabel="CvfsDisk12" diskType="AudioDrive"/>
   <disk index="3" diskLabel="CvfsDisk13" diskType="AudioDrive"/>
</stripeGroup>
<stripeGroup index="4" name="RegularFiles" status="up"
stripeBreadth="262144" read="true" write="true" metadata="false"
```

```
         journal="false" userdata="true" realTimeIOs="0" realTimeIOsReserve="0"
         realTimeMB="0" realTimeMBReserve="0" realTimeTokenTimeout="0"
         multipathMethod="rotate">
            <disk index="0" diskLabel="CvfsDisk14" diskType="DataDrive"/>
            <disk index="1" diskLabel="CvfsDisk15" diskType="DataDrive"/>
            <disk index="2" diskLabel="CvfsDisk16" diskType="DataDrive"/>
            <disk index="3" diskLabel="CvfsDisk17" diskType="DataDrive"/>
         </stripeGroup>
      </stripeGroups>
   </config>
</configDoc>
```

Windows Example Configuration File

Below are the contents of the StorNext example configuration file for Windows (**example.cfg**):

```
# Globals

ABMFreeLimit no
AllocationStrategy Round
HaFsType HaUnmonitored
FileLocks No
BrlResyncTimeout 20
BufferCacheSize 32M
CvRootDir /
DataMigration No
DataMigrationThreadPoolSize 128
Debug 0x0
DirWarp Yes
ExtentCountThreshold 48K
EnableSpotlight No
ForcePerfectFit No
FsBlockSize 16K
GlobalSuperUser Yes
InodeCacheSize 32K
InodeExpandMin 0
InodeExpandInc 0
InodeExpandMax 0
InodeDeleteMax 0
```

```
InodeStripeWidth 0
JournalSize 16M
MaxConnections 32
MaxLogs 4
PerfectFitSize 128K
RemoteNotification No
ReservedSpace Yes
FSMRealtime No
FSMMemlock No
OpHangLimitSecs 180
Quotas No
RestoreJournal No
RestoreJournalMaxHours 0
RestoreJournalMaxMB 0
StripeAlignSize 0
TrimOnClose 0
ThreadPoolSize 32
UnixDirectoryCreationModeOnWindows 0644
UnixIdFabricationOnWindows No
UnixFileCreationModeOnWindows 0755
UnixNobodyUidOnWindows 60001
UnixNobodyGidOnWindows 60001
WindowsSecurity Yes
EventFiles Yes
AllocSessionReservation No
# Disk Types

[DiskType MetaDrive]
Sectors 99999999
SectorSize 512
[DiskType JournalDrive]
Sectors 99999999
SectorSize 512
[DiskType VideoDrive]
Sectors 99999999
SectorSize 512
[DiskType AudioDrive]
Sectors 99999999
```

```
SectorSize 512
[DiskType DataDrive]
Sectors 99999999
SectorSize 512
# Disks

[Disk CvfsDisk0]
Type MetaDrive
Status UP
[Disk CvfsDisk1]
Type JournalDrive
Status UP
[Disk CvfsDisk2]
Type VideoDrive
Status UP
[Disk CvfsDisk3]
Type VideoDrive
Status UP
[Disk CvfsDisk4]
Type VideoDrive
Status UP
[Disk CvfsDisk5]
Type VideoDrive
Status UP
[Disk CvfsDisk6]
Type VideoDrive
Status UP
[Disk CvfsDisk7]
Type VideoDrive
Status UP
[Disk CvfsDisk8]
Type VideoDrive
Status UP
[Disk CvfsDisk9]
Type VideoDrive
Status UP
[Disk CvfsDisk10]
Type AudioDrive
```

```
Status UP
[Disk CvfsDisk11]
Type AudioDrive
Status UP
[Disk CvfsDisk12]
Type AudioDrive
Status UP
[Disk CvfsDisk13]
Type AudioDrive
Status UP
[Disk CvfsDisk14]
Type DataDrive
Status UP
[Disk CvfsDisk15]
Type DataDrive
Status UP
[Disk CvfsDisk16]
Type DataDrive
Status UP
[Disk CvfsDisk17]
Type DataDrive
Status UP
# Stripe Groups

[StripeGroup MetaFiles]
Status Up
StripeBreadth 256K
Metadata Yes
Journal No
Exclusive Yes
Read Enabled
Write Enabled
Rtmb 200
Rtios 200
RtmbReserve 1
RtiosReserve 1
RtTokenTimeout 0
MultiPathMethod Rotate
```

```
Node CvfsDisk0 0

[StripeGroup JournFiles]
Status Up
StripeBreadth 256K
Metadata No
Journal Yes
Exclusive Yes
Read Enabled
Write Enabled
Rtmb 0
Rtios 0
RtmbReserve 0
RtiosReserve 0
RtTokenTimeout 0
MultiPathMethod Rotate
Node CvfsDisk1 0

[StripeGroup VideoFiles]
Status Up
StripeBreadth 4M
Metadata No
Journal No
Exclusive No
Read Enabled
Write Enabled
Rtmb 0
Rtios 0
RtmbReserve 0
RtiosReserve 0
RtTokenTimeout 0
MultiPathMethod Rotate
Node CvfsDisk2 0
Node CvfsDisk3 1
Node CvfsDisk4 2
Node CvfsDisk5 3
Node CvfsDisk6 4
Node CvfsDisk7 5
Node CvfsDisk8 6
Node CvfsDisk9 7
Affinity Video

[StripeGroup AudioFiles]
Status Up
StripeBreadth 1M
Metadata No
Journal No
Exclusive No
Read Enabled
Write Enabled
Rtmb 0
Rtios 0
RtmbReserve 0
RtiosReserve 0
RtTokenTimeout 0
MultiPathMethod Rotate
Node CvfsDisk10 0
Node CvfsDisk11 1
Node CvfsDisk12 2
Node CvfsDisk13 3
Affinity Audio

[StripeGroup RegularFiles]
Status Up
StripeBreadth 256K
Metadata No
Journal No
Exclusive No
Read Enabled
Write Enabled
Rtmb 0
Rtios 0
RtmbReserve 0
RtiosReserve 0
RtTokenTimeout 0
MultiPathMethod Rotate
Node CvfsDisk14 0
Node CvfsDisk15 1
Node CvfsDisk16 2
Node CvfsDisk17 3
```

Ports Used By StorNext

The following table lists ports that are used by StorNext and its ancillary components.

For additional information about ports used by StorNext, see the man page for `fsports(4)`.

Port	StorNext Use	Notes
81	GUI (Java)	User starts at port 81, redirected to 443
443	GUI (Java)	
1527	GUI (Java connection to derby db)	
1070	GUI (Java connection to Linter)	
1062, 1063	Blockpool	Both ports if HA primary
14500	snpolicyd	
5164	fsmpm	(See CR 30846)
5189	HA Manager	Symbol `HAMGR_DEFAULT_PORT`
Various	fsm, fsmpm	Change ports with fsports file
20566	Linter	Only used internally on an MDC.

Port	StorNext Use	Notes
60001, 60002 …	ACSLS Tape Libraries	Not used by StorNext, but related

Chapter 2

Allocation Session Reservation (ASR)

Debuting in StorNext 4.2, the Allocation Session Reservation (ASR) feature provides another method of allocating space to regular files. ASR optimizes on-disk allocation behavior in workflows (such as some rich media streaming applications) which write and read sequences of files of certain sizes in specific directories.

With ASR, file sequences in a directory are usually placed on disk based on the order in which they are written. ASR keeps these files together even if other applications are writing at the same time in different directories or from different StorNext clients. The feature also has the ability to reduce file system free space fragmentation since collections of files which are written together are then typically removed together.

The workflows which see reduced free space fragmentation are those which have concurrent applications each writing files in their own directories and using files mostly larger than 1MB each. With this kind of workflow, when a collection of files from one application is removed, the space is freed up in big chunks which are independent of other application runs.

Some workflows will see worse performance and may also see more free space fragmentation with ASR. These workflows are those which have concurrent applications all using the same directory on the same client, or all writing the same file on different clients. Additionally, performance may be adversely affected when stripe groups are configured and used to distribute applications. (See Hotspots and Locality on page 53.)

Some applications depend on stripe alignment for performance. Stripe alignment can cause the allocator to chop an allocation request to make its head and tail land on a stripe boundary. The ASR feature disables stripe alignment since the chopping can lead to even more free space fragmentation since the chopping is within ASR chunks.

Customers should run with ASR and see if performance is adversely affected. You can do this by turning On or Off by setting the size in the configuration file or via the StorNext GUI and then restarting the FSM. Then, run your application and measure performance.

The fact that files are kept together on a stripe group for the ASR chunk size may improve performance and make stripe alignment unnecessary.

The ideal situation is for a system administrator to watch the system both with and without ASR enabled. First, performance should be monitored. Second, fragmentation can be checked. There are two kinds of fragmentation:

1. Fragmentation within files.
2. Free space fragmentation.

Fragments within a collection of files can be counted using snfsdefrag(1), e.g., `snfsdefrag -t -r -c <directory>`. This command lists all the files and the number of extents in each file, and then the total of all regular files, extents, and extents per file.

The command, `cvfsck -a -f <file system>` lists free space fragments on each stripe group by chunk size, the total number of free space fragments for each stripe group, and then the total number of stripe groups and free space fragments for the entire file system. With this tool, free space fragments can be counted before and after a workflow is run. (“Workflows” should include normal administrative cleanup and modifications which occur over time.)

Administrators are encouraged to monitor their system to see how fragmentation is occurring.

The `snfsdefrag(1)` command can be run periodically to defragment files, reducing the number of fragments in those files. This usually helps reduce free space fragmentation, too.

How ASR Works

For details on how to set the “size” and enable this feature, refer to the snfs_config(5) man page and the StorNext GUI’s online help. The man page snfs_config(5) also contains an overview of how the ASR feature works.

Because this “How ASR Works” section provides more detail, before reading this section you should already be familiar with the man page contents.

Allocation Sessions

Allocation requests (which occur whenever a file is written to an area that has no actual disk space allocated,) are grouped into sessions. A chunk of space is reserved for a session. The size of the chunk is determined using the configured size and the size of the allocation request. If the allocation size is bigger than 1MB and smaller than 1/8th the configured ASR chunk size, the ASR chunk size is rounded up to be a multiple of the initial allocation request size.

There are three session types: **small**, **medium** (directory), and **large** (file). The session type is determined by the file offset and requested allocation size on a given allocation request.

- Small sessions are for sizes (offset + allocation size) smaller than 1MB.
- Medium sessions are for sizes 1MB through 1/10th of the configured ASR size.
- Large sessions are sizes bigger than medium.

Here is another way to think of these three types: small sessions collect or organize all small files into small session chunks; medium sessions collect medium-sized files by chunks using their parent directory; and large file allocations are collected into their own chunks and are allocated independently of other files.

All sessions are client specific. Multiple writers to the same directory or large file on different clients will use different sessions. Small files from different clients use different chunks by client.

Small sessions use a smaller chunk size than the configured size. The small chunk size is determined by dividing the configured size by 32.

For example, for 128 MB the small chunk size is 4 MB, and for 1 GB the small chunk size is 32 MB. Small sessions do not round the chunk size. A file can get an allocation from a small session only if the allocation request (offset + size) is less than 1MB. When users do small I/O sizes into a file, the client buffer cache coalesces these and minimizes allocation requests. If a file is larger than 1MB and is being written through the buffer cache, it will most likely have allocation on the order of 16MB or so requests (depending on the size of the buffer cache on the client and the number of concurrent users of that buffer cache).

With NFS I/O into a StorNext client, the StorNext buffer cache is used. NFS on some operating systems breaks I/O into multiple streams per file. These will arrive on the StorNext client as disjointed random writes. These are typically allocated from the same session with ASR and are not impacted if multiple streams (other files) allocate from the same stripe group. ASR can help reduce fragmentation due to these separate NFS generated streams.

Files can start using one session type and then move to another session type. A file can start with a very small allocation (small session), become larger (medium session), and end up *reserving* the session for the file. If a file has more than 10% of a medium sized chunk, it “reserves” the remainder of the session chunk it was using for itself. After a session is reserved for a file, a new session segment will be allocated for any other medium files in that directory.

Small chunks are never reserved.

When allocating subsequent pieces for a session, they are rotated around to other stripe groups that can hold user data. This is done the same was as InodeStripeWidth (ISW). (For more information about ISW, refer to the snfs_config man page.)

The direction of rotation is determined by a combination of the session key and the index of the client in the client table. The session key is based on the inode number, so odd inodes will rotate in a different direction from even inodes. Directory session keys are based on the parent directory’s inode number.

Video Frame Per File Formats

Video applications typically write one frame per file and place them in their own unique directory, and then write them from the same StorNext client. The file sizes are all greater than 1MB and smaller than 50 MB each and written/allocated in one I/O operation. Each file and write land in “medium/directory” sessions.

For this kind of workflow, ASR is the ideal method to keep “streams” (a related collection of frames in one directory) together on disk, thereby preventing checker boarding between multiple concurrent streams. In addition, when a stream is removed, the space can be returned to the free space pool in big ASR pieces, reducing free space fragmentation when compared to the default allocator.

Hotspots and Locality

Suppose a file system has four data stripe groups and an ASR size of 1 GB. If four concurrent applications writing medium-sized files in four separate directories are started, they will each start with their own 1 GB piece and most likely be on different stripe groups.

Without ASR

Without ASR, the files from the four separate applications are intermingled on disk with the files from the other applications. The default allocator does not consider the directory or application in any way when carving out space. All allocation requests are treated equally. With ASR turned off and all the applications running together, any hotspot is very short lived: the size of one allocation/file. (See the following section for more information about hotspots.)

With ASR

Now consider the 4 GB chunks for the four separate directories. As the chunks are used up, ASR allocates chunks on a new SG using rotation. Given this rotation and the timings of each application, there are times when multiple writers/segments will be on a particular stripe group together. This is considered a “hotspot,” and if the application expects more throughput than the stripe group can provide, performance will be sub par.

At read time, the checker boarding on disk from the writes (when ASR is off) can cause disk head movement, and then later the removal of one application run can also cause free space fragmentation. Since ASR collects the files together for one application, the read performance of one application's data can be significantly better since there will be little to no disk head movement.

Small Session Rationale

Small files (those less than 1 MB) are placed together in small file chunks and grouped by StorNext client ID. This was done to help use the leftover pieces from the ASR size chunks and to keep the small files away from medium files. This reduces free space fragmentation over time that would be caused by the leftover pieces. Leftover pieces occur in some rare cases, such as when there are many concurrent sessions exceeding 500 sessions.

Large File Sessions and Medium Session Reservation

When an application starts writing a very large file, it typically starts writing in some units and extending the file size. For this scenario, assume the following:

- ASR is turned on, and the configured size is 1 GB.
- The application is writing in 2 MB chunks and writing a 10 GB file.
- ISW is set to 1 GB.

On the first I/O (allocation), an ASR session is created for the directory (if one doesn't already exist,) and space is either stolen from an expired session or a new 1 GB piece is allocated on some stripe group.

When the file size plus the request allocation size passes 100 MB, the session will be converted from a directory session to a file-specific session and reserved for this file. When the file size surpasses the ASR size, chunks are reserved using the ISW configured size.

Returning to our example, the extents for the 10 GB file should start with a 1 GB extent (assuming the first chunk wasn't stolen and a partial,) and the remaining extents except the last one should all be 1 GB.

Following is an example of extent layout from one process actively writing in it's own directory as described above:

```
root@per2:() -> snfsdefrag -e 10g.lmdd
10g.lmdd:
#   group  frbase        fsbase      fsend       kbytes     depth
0   3   0x0           0xdd4028    0xde4027    1048576    1
1   4   0x40000000    0xdd488a    0xde4889    1048576    1
2   1   0x80000000    0x10f4422   0x1104421   1048576    1
3   2   0xc0000000    0x20000     0x2ffff     1048576    1
4   3   0x100000000   0xd34028    0xd44027    1048576    1
5   4   0x140000000   0xd9488a    0xda4889    1048576    1
```

```
6    1    0x180000000    0x10c4422    0x10d4421    1048576    1
7    2    0x1c0000000    0x30000      0x3ffff      1048576    1
8    3    0x200000000    0x102c028    0x103c027    1048576    1
9    4    0x240000000    0xd6c88a     0xd7c889     1048576    1
```

Here are the extent layouts of two processes writing concurrently but in their own directory:

```
root@per2:() -> lmdd of=1d/10g bs=2m move=10g & lmdd of=2d/10g bs=2m move=10g &
[1] 27866
[2] 27867
root@per2:() -> wait
snfsdefrag -e 1d/* 2d/*
10240.00 MB in 31.30 secs, 327.14 MB/sec
[1]-  Done                  lmdd of=1d/10g bs=2m move=10g
10240.00 MB in 31.34 secs, 326.74 MB/sec
[2]+  Done                  lmdd of=2d/10g bs=2m move=10g
root@per2:() ->
root@per2:() -> snfsdefrag -e 1d/* 2d/*
1d/10g:
#    group  frbase        fsbase       fsend        kbytes     depth
0    1    0x0           0xf3c422     0xf4c421     1048576    1
1    4    0x40000000    0xd2c88a     0xd3c889     1048576    1
2    3    0x80000000    0xfcc028     0xfdc027     1048576    1
3    2    0xc0000000    0x50000      0x5ffff      1048576    1
4    1    0x100000000   0x7a0472     0x7b0471     1048576    1
5    4    0x140000000   0xc6488a     0xc74889     1048576    1
6    3    0x180000000   0xcd4028     0xce4027     1048576    1
7    2    0x1c0000000   0x70000      0x7ffff      1048576    1
8    1    0x200000000   0x75ef02     0x76ef01     1048576    1
9    4    0x240000000   0xb9488a     0xba4889     1048576    1

2d/10g:
#    group  frbase        fsbase       fsend        kbytes     depth
0    2    0x0           0x40000      0x4ffff      1048576    1
1    3    0x40000000    0xffc028     0x100c027    1048576    1
2    4    0x80000000    0xca488a     0xcb4889     1048576    1
3    1    0xc0000000    0xedc422     0xeec421     1048576    1
4    2    0x100000000   0x60000      0x6ffff      1048576    1
5    3    0x140000000   0xea4028     0xeb4027     1048576    1
6    4    0x180000000   0xc2c88a     0xc3c889     1048576    1
7    1    0x1c0000000   0x77f9ba     0x78f9b9     1048576    1
8    2    0x200000000   0x80000      0x8ffff      1048576    1
```

```
9    3    0x240000000    0xbe4028    0xbf4027    1048576    1
```

Finally, consider two concurrent writers in the same directory on the same client writing 10 GB files. The files will checker board until they reach 100 MBs. After that, each file will have its own large session and the checker boarding will cease.

Here is an example of two 5 GB files written in the same directory at the same time with 2MB I/Os. The output is from the `snfsdefrag -e <file>` command.

```
One:
#    group  frbase        fsbase       fsend        kbytes     depth
0    1    0x0           0x18d140     0x18d23f     4096       1
1    1    0x400000      0x18d2c0     0x18d33f     2048       1
2    1    0x600000      0x18d3c0     0x18d43f     2048       1
3    1    0x800000      0x18d4c0     0x18d53f     2048       1
4    1    0xa00000      0x18d5c0     0x18d73f     6144       1
5    1    0x1000000     0x18d7c0     0x18d83f     2048       1
6    1    0x1200000     0x18d8c0     0x18d9bf     4096       1
7    1    0x1600000     0x18dbc0     0x18dcbf     4096       1
8    1    0x1a00000     0x18dfc0     0x18e4bf     20480      1
9    1    0x2e00000     0x18e8c0     0x18e9bf     4096       1
10   1    0x3200000     0x18eac0     0x18ebbf     4096       1
11   1    0x3600000     0x18ecc0     0x18f3bf     28672      1
12   1    0x5200000     0x18f9c0     0x18fdbf     16384      1
13   1    0x6200000     0x1901c0     0x19849f     536064     1
14   3    0x26d80000    0x1414028    0x1424027    1048576    1
15   4    0x66d80000    0x150f092    0x151f091    1048576    1
16   1    0xa6d80000    0x10dc6e     0x11dc6d     1048576    1
17   3    0xe6d80000    0x1334028    0x1344027    1048576    1
18   4    0x126d80000   0x8f74fe     0x8fd99d     412160     1

Two:
#    group  frbase        fsbase       fsend        kbytes     depth
0    1    0x0           0x18d0c0     0x18d13f     2048       1
1    1    0x200000      0x18d240     0x18d2bf     2048       1
2    1    0x400000      0x18d340     0x18d3bf     2048       1
3    1    0x600000      0x18d440     0x18d4bf     2048       1
4    1    0x800000      0x18d540     0x18d5bf     2048       1
5    1    0xa00000      0x18d740     0x18d7bf     2048       1
6    1    0xc00000      0x18d840     0x18d8bf     2048       1
7    1    0xe00000      0x18d9c0     0x18dbbf     8192       1
```

```
8    1    0x1600000      0x18dcc0     0x18dfbf     12288      1
9    1    0x2200000      0x18e4c0     0x18e8bf      16384      1
10    1    0x3200000      0x18e9c0     0x18eabf      4096       1
11    1    0x3600000      0x18ebc0     0x18ecbf      4096       1
12    1    0x3a00000      0x18f3c0     0x18f9bf     24576      1
13    1    0x5200000      0x18fdc0     0x1901bf     16384      1
14    4    0x6200000      0x1530772     0x1540771     1048576     1
15    3    0x46200000     0x1354028     0x1364027     1048576     1
16    1    0x86200000     0x12e726      0x13e725      1048576     1
17    4    0xc6200000     0x14ed9b2     0x14fd9b1     1048576     1
18    3    0x106200000     0x1304028     0x13127a7     948224      1
```

Without ASR and with concurrent writers of big files, each file typically starts on its own stripe group. The checker boarding doesn't occur until there are more writers than the number of data stripe groups. However, once the checker boarding starts, it will exist all the way through the file. For example, if we have two data stripe groups and four writers, all four files would checker board until the number of writers is reduced back to two or less.

Appendix A

StorNext File System Stripe Group Affinity

This appendix describes the behavior of the stripe group affinity feature in the StorNext file system, and it discusses some common use cases.

Note: This section does not discuss file systems managed by StorNext Storage Manager. There are additional restrictions on using affinities for these managed file systems.

Definitions

Following are definitions for terms used in this appendix:

Stripe Group

A *stripe group* is collection of LUNs (typically disks or arrays,) across which data is striped. Each stripe group also has a number of associated attributes, including affinity and exclusivity.

Affinity

An *affinity* is used to steer the allocation of a file’s data onto a set of stripe groups. Affinities are referenced by their name, which may be up to eight characters long. An affinity may be assigned to a set of stripe groups, representing a named pool of space, and to a file or directory, representing the space from which space should be allocated for that file (or files created within the directory).

Exclusivity

A stripe group which has both an affinity and the exclusive attribute can have its space allocated only by files with that affinity. Files without a matching affinity cannot allocate space from an exclusive stripe group.

Setting Affinities

Affinities for stripe groups are defined in the file system configuration file. They can be created through the StorNext GUI or by adding one or more `Affinity` lines to a `StripeGroup` section in the configuration file. A stripe group may have multiple affinities, and an affinity may be assigned to multiple stripe groups.

Affinities for files are defined in the following ways:

- Using the `cvmkfile` command with the '`-k`' option
- Using the `snfsdefrag` command with the '`-k`' option
- Using the `cvaffinity` command with the '`-s`' option
- Through inheritance from the directory in which they are created

Through the `CvApi_SetAffinity()` function, which sets affinities programmatically

Allocation Strategy

StorNext has multiple allocation strategies which can be set at the file system level. These strategies control where a new file's first blocks will be allocated. Affinities modify this behavior in two ways:

- A file with an affinity is allocated only on a stripe group with matching affinity.
- A stripe group with an affinity and the exclusive attribute is used only for allocations by files with matching affinity.

Once a file has been created, StorNext attempts to keep all of its data on the same stripe group. If there is no more space on that stripe group, data may be allocated from another stripe group.

If the file has an affinity, only stripe groups with that affinity are considered. If all stripe groups with that affinity are full, new space may not be allocated for the file, even if other stripe groups are available.

Common Use Cases

Here are some sample use cases in which affinities are used to maximize efficiency and operation.

Segregating Audio and Video Files Onto Their Own Stripe Groups

To segregate audio and video files onto their own stripe groups:

One common use case is to segregate audio and video files onto their own stripe groups. Here are the steps involved in this scenario:

- Create one or more stripe groups with an AUDIO affinity and the exclusive attribute.
- Create one or more stripe groups with a VIDEO affinity and the exclusive attribute.
- Create one or more stripe groups with no affinity (for non-audio, non-video files).
- Create a directory for audio using ‘`cvmkdir -k AUDIO audio`’.
- Create a directory for video using ‘`cvmkdir -k VIDEO video`’.

Files created within the audio directory will reside only on the AUDIO stripe group. (If this stripe group fills, no more audio files can be created.)

Files created within the video directory will reside only on the VIDEO stripe group. (If this stripe group fills, no more video files can be created.)

Reserving High-Speed Disk For Critical Files

In this use case, high-speed disk usage is reserved for and limited to only critical files. Here are the steps for this scenario:

- Create a stripe group with a FAST affinity and the exclusive attribute.
- Label the critical files or directories with the FAST affinity.

The disadvantage here is that the critical files are restricted to using only the fast disk. If the fast disk fills up, the files will not have space allocated on slow disks.

To work around this limitation, you could reserve high-speed disk for critical files but also allow them to grow onto slow disks. Here are the steps for this scenario:

- Create a stripe group with a FAST affinity and the exclusive attribute.
- Create all of the critical files, pre allocating at least one block of space, with the FAST affinity. (Or move them using `snfsdefrag` after ensuring the files are not empty.)
- Remove the FAST affinity from the critical files.

Because files allocate from their existing stripe group even if they no longer have a matching affinity, the critical files will continue to grow on the FAST stripe group. Once this stripe group is full, they can allocate space from other stripe groups since they do not have an affinity.

This scenario will not work if new critical files can be created later, unless there is a process to move them to the FAST stripe group, or an affinity is set on the critical files by inheritance but removed after their first allocation (to allow them to grow onto non-FAST groups).

Appendix B
Best Practice Recommendations

This appendix contains some best practice recommendations for various StorNext features which you can implement to ensure optimal performance and efficiency.

Replication Best Practices

This section describes some best practices related to using the StorNext replication feature.

Replication Copies

The replication target can keep one or more copies of data. Each copy is presented as a complete directory tree for the policy. The number of copies and placement of this directory are ultimately controlled by the replication target. However, if the target does not implement policy here, the source system may request how many copies are kept and how the directories are named.

When multiple copies are kept, the older copies and current copy share files where there are no changes. This shows up as extra hard links to the files. If a file is changed on the target, it affects all copies sharing the file. If a file is changed on the replication source, older copies on the target are not affected.

The best means to list which replication copies exist on a file system is running `snpolicy -listrepcopies` command. The `rmrepcopy`, `mvrepcopy` and `exportrepcopy` options should be used to manage the copies.

Replication and Deduplication

Replication can be performed on deduplicated or non-deduplicated data. Even if the source system is running deduplication, you can still replicate non-deduplicated data to the target using the `rep_dedup=off` policy parameter.

A good example of when this makes sense is replicating into a TSM relation point which is storing to tape. If deduplicated replication is used, the store to tape requires retrieving files from the blockpool. This is much more likely to stall tape drives than streaming raw file content to tape.

The tradeoff here is that all file data will be sent over the network even if the target system has already seen it. So if the limiting resource is network bandwidth and the data is amenable to deduplication, then deduplication-enabled replication into TSM may perform better.

With deduplicated replication, the file contents are deduplicated prior to replication. There is no concept of replication using deduplicated data without deduplicating the data on the source system.

Replication data is moved via a pull model, in which the target of replication asks the source system to send it data it does not yet have. For non-deduplicated replication, this will be performed over the network UNLESS the source file system is cross mounted on the target, in which case the target will use local I/O to copy the data. The number of files actively being replicated at the same time, and the size of the buffer used for I/O in the non-deduplicated data case are controlled by the `replicate_threads` and `data_buffer_size` parameters on the target system. The default buffer size is 4 Mbytes, and the default stream count is 4.

Replication and Distributed LAN Client Servers

It might seem obvious, but it’s worth mentioning that if your configuration includes Distributed LAN Clients (DLC), the machines you use for your DLC servers should not also be metadata controllers. Doing so may not only cause performance degradation, but also expose the virtual IPs to additional vulnerability. For best performance, machines used as DLC servers should always be dedicated machines.

Replication with Multiple Physical Network Interfaces

If you want to use replication with multiple physical network interfaces, you must arrange for traffic on each interface to be routed appropriately.

In cases where both the replication source and target are plugged into the same physical Ethernet switch, you can accomplish this with VLANs.

In cases where replication is over multiple WAN links, the addresses used on the source and target replication systems must route over the appropriate WAN links in order for replication to use all the links.

Deduplication Best Practices

This section describes some best practices related to using the StorNext deduplication feature.

Deduplication and File Size

Deduplication will not be beneficial on small files, nor will it provide any benefit on files using compression techniques on the content data (such as mpeg format video). In general, deduplication is maximized for files that are 64MB and larger. Deduplication performed on files below 64MB may result in sub-optimal results.

You can filter out specific files to bypass by using the `dedup_skip` policy parameter. This parameter works the same as filename expansion in a UNIX shell.

You can also skip files according to size by using the `dedup_min_size` parameter.

Deduplication and Backups

Backup streams such as tar and netbackup can be recognized by the deduplication algorithm if the `dedup_filter` parameter on the policy is set to `true`.

In this configuration the content of the backup image is interpreted to find the content files, and these are deduplicated individually. When this this flag is not set to `true`, the backup image is treated as raw data and the backup metadata in the file will interfere with the reduction potential of the deduplication algorithm. Recognition of a backup stream is according to its contents, not the file name.

Deduplication and File Inactivity

Deduplication is performed on a file after a period of inactivity after the file is last closed, as controlled by the dedup_age policy parameter. It is worth tuning this parameter if your workload has regular periods of inactivity on files before they are modified again.

Note: Making the age too small can lead to the same file being deduplicated more than once.

Deduplication and System Resources

Running deduplication is a CPU and memory-intensive operation, and the backing store for deduplicated data can see a lot of random I/O, especially when retrieving truncated files.

Consequently, plan accordingly, and do not under-resource the blockpool file system or metadata system if you are striving for optimal performance.

Deduplication Parallel Streams

The number of deduplication parallel streams running is controlled by the `ingest_thread` parameter in `/usr/cvfs/config/snpolicyd.conf`.

If you are not I/O limited and have more CPU power available, increasing the stream count from the default value of 4 streams can improve throughput.

Truncation Best Practices

This section describes some best practices related to using the StorNext truncation feature.

Deduplication and Truncation

If deduplication is run without StorNext Storage Manager also storing the file contents, then `snpolicyd` can manage file truncation. If Storage Manager is also running on a directory, it becomes the engine which removes the online copy of files.

Note: Storage Manager can retrieve deleted files from tape. With deduplication, if the primary file is removed from a directory, the deduplicated copy is no longer accessible. This is a fundamental difference between the two mechanisms (truncation and deduplication) which must be understood.

If a policy is configured not to deduplicate small files, it will automatically not truncate them. It is also possible to set an independent minimum size for files to truncate, and a stub length to leave behind when a file is truncated.

Once a file is truncated by the policy daemon, the contents must be retrieved from the deduplicated storage. This can be done by reading the file, or via the `snpolicy -retrieve` command.

Note: When using the command line to run commands, the truncation policy can potentially remove the contents again before they are used, depending on how aggressive the policy is. Unlike TSM, the whole file does not have to be retrieved before I/O can proceed. The number of parallel retrieves is governed by the `event_threads` parameter in `/usr/cvfs/config/snpolicyd.conf`.

In the case where both deduplication and tape copies of data are being made, TSM is the service which performs truncation.