



Lustre™ HPCS Design Overview

June 2009

Table of Contents

Executive Summary.....	3
Lustre and HPCS.....	4
The Lustre File System.....	4
The DARPA HPCS Project.....	4
HPCS Filesystem Goals.....	5
Future Lustre Architecture.....	5
Lustre Metadata Servers.....	7
Lustre Object Storage Servers.....	7
Lustre Networking.....	8
HPCS Architectural Improvements.....	9
ZFS.....	9
End-to-end Data Integrity.....	12
RAID Rebuild Performance Impact.....	17
Lustre-level Rebuild Mitigation.....	18
ZFS Resilvering Improvement.....	19
ZFS Distributed Hot Space.....	20
Filesystem Integrity Checking.....	22
Clustered Metadata.....	25
Imperative Recovery.....	27
Channel Bonding.....	28
Performance Enhancements.....	29
SMP Scalability.....	29
Network Request Scheduler.....	30
Metadata Writeback Cache.....	32
Conclusion.....	33
Appendix A – HPCS Filesystem Example Configuration.....	34

Executive Summary

The DARPA HPCS program sets forth demanding goals for the compute and storage systems of future High Performance Computing (HPC) platforms. The Sun Lustre filesystem is capable of meeting the requirements of HPC systems of today, and is being enhanced to meet the challenges of HPCS in a number of ways, to increase its capacity and scalability for the future. In particular, features related to metadata, and backing filesystem scalability and robustness are being developed to meet specific HPCS goals.

Lustre and HPCS

The Lustre File System

The Lustre filesystem is the leading cluster file system in the HPC market today. Lustre provides POSIX semantics and already effectively scales to support systems with tens of thousands of compute nodes and hundreds of gigabytes per second of aggregate IO throughput.

HPCS requires some scalability and features beyond what Lustre is capable of in certain areas today. This document identifies the gaps between the two, and discusses the high-level approaches to overcoming these obstacles. Both in-progress development efforts and future design and development work will be discussed to illustrate that Lustre can meet the current goals of the HPCS program, and beyond.

The Lustre filesystem segregates the filesystem namespace (metadata) from the file data storage. The metadata will be managed on a cluster of tens to hundreds of Metadata Servers (MDSes) each with its own high-performance redundant filesystem, called a Metadata Target (MDT). The file data is managed by hundreds or thousands of Object Storage Servers (OSSes), to which the majority of drives are attached. The storage attached to the OSS nodes is managed by one or more backing filesystems, called Object Storage Targets (OSTs).

The DARPA HPCS Project

The DARPA HPCS program puts forth demanding capacity and scalability goals on the storage system, as shown below. The HPCS goals can be broken down into two separate classes of usage, when viewed from a file system perspective:

1. the *Parallel Environment*, which requires high aggregate throughput from tens or hundreds of thousands of clients, mostly doing large file read and write operations in a co-ordinated manner on very large files.
2. the *Capture Environment*, which requires high file creation and IO rates from a very small number of clients, and has a very large number of moderately-sized files.

A primary concern for the HPCS project is not simply meeting performance numbers, but also the scalability of the underlying architecture and the efficiency by which the underlying hardware is used.

HPCS Filesystem Goals

The HPCS capacity goals for the entire filesystem are:

- 100PB+ maximum filesystem size
- 1 trillion (10^{12}) files per file system
- > 30K client nodes

Capacity goals for individual files or directories are:

- 10 billion files per directory
- 0 to 1 PB file size range
- 1B to 1 GB I/O request size range
- 100,000 open shared files per process
- Long file names and 0-length file lengths for applications that use file metadata as data records

Performance goals for the HPCS filesystem are:

- 10,000 aggregate metadata operations per second
- 1500 GB/s aggregate, file per process and single shared file
- No impact of RAID rebuilds on file system performance
- POSIX I/O API extensions proposed at the OpenGroup¹

Performance goals for a single client in the capture environment are:

- 40,000 creates per second, with up to 64kB of data
- 30 GB/s full-duplex streaming I/O bandwidth

Reliability goals for the HPCS filesystem are:

- End-to-end resiliency equivalent to T10 DIF or better
- No impact of RAID rebuilds on file system performance
- Uptime of 99.99%
- 100h filesystem integrity check

Note that an uptime of 99.99% is less than 1 hour per year of downtime. This implies that the 100h filesystem integrity check should be performed with the filesystem online in order to meet the 99.99% uptime goal.

Future Lustre Architecture

The compute clusters shown in Figure 1: HPC Center of the Future are connected to a large shared Lustre Storage Cluster through a shared high-performance backbone network. The network is attached to a tape archive, used to back up the Lustre Storage System. It is also connected to external networks (WAN, Internet, etc.) for direct access by remote computing environments or data collection.

1 High End Computing Extensions Working Group <http://www.opengroup.org/platform/hecewg/>

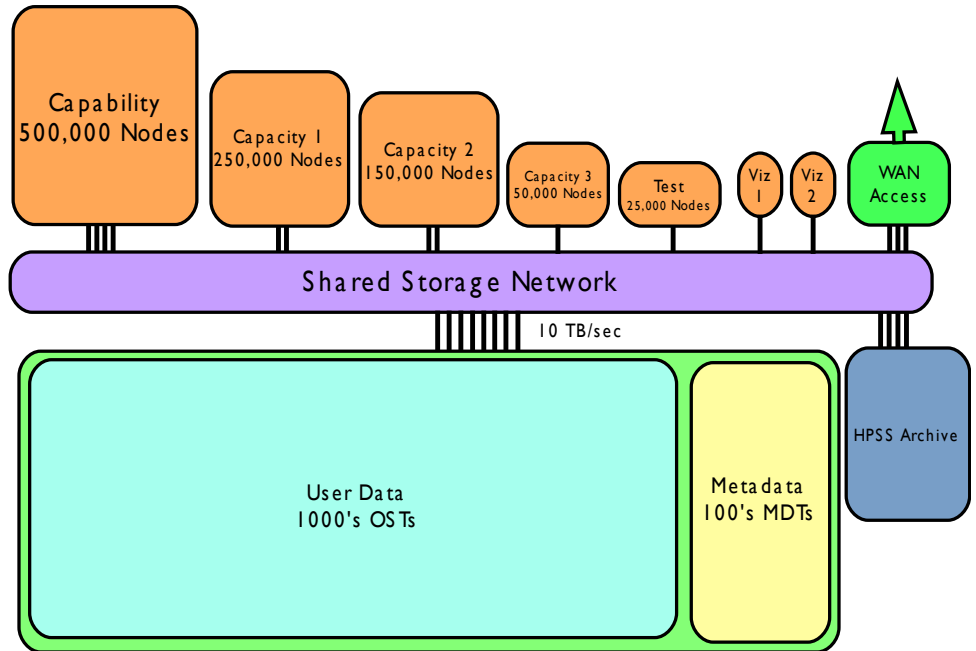


Figure 1: HPC Center of the Future

The Lustre Storage Cluster separates the filesystem management into two different types of servers – one for filesystem metadata, and one for file data. The Storage Cluster will contain tens of thousands of disk drives with a storage capacity of hundreds of petabytes or more. Table 1: HPCS Filesystem Sample Configuration in Appendix A contains an example filesystem configuration that is based on the HPCS project goals and projected hardware capacity and performance in 2011.

Connecting these different computing systems to a shared Storage Cluster via a high-speed network enables data sharing between the systems, makes it easier for users to run their applications on different computing systems, and simplifies the addition or removal of large systems in the HPC center as data management is decoupled from the compute systems.

Sharing the Storage Cluster between computing systems allows the computing resources to be utilized more efficiently. There is no need to copy application

data between filesystems local to each compute system, nor is there a need to pre-stage data to individual compute nodes before a job can be run. This allows users to run their jobs on the best available compute resource for their needs and not be concerned with the location of their data.

Lustre Metadata Servers

The MDS manages filesystem namespace operations (create, unlink, rename, link), and controls access (user, group, ACLs, and extended attributes) to the Metadata Target (MDT). The namespace is stored in a backing disk filesystem such as ext3/ext4, or in the near future ZFS, in order to leverage development efforts made for the disk filesystem.

The MDT is typically stored on high-performance storage such as SAS or SSD in a RAID-1+0 configuration to provide minimum latency and high availability. The storage technology used by the MDTs can be different than that used for the file data, to optimize for the small-block IO performance characteristics needed for the MDT.

The MDS node manages all metadata access and filesystem permissions. The MDS ensures namespace and backing file system integrity by preventing malicious or malfunctioning clients from directly modifying the backing filesystem metadata. The MDT filesystem is never directly modified by clients, only by the MDS.

In current Lustre deployments there is a single MDT per filesystem exported from one MDS at a time. There are typically two MDS nodes that are connected to the shared MDT storage to provide highly available access in an active/standby manner. Future Lustre deployments will have tens of MDSes and MDTs in a single filesystem that are active concurrently, in order to scale up metadata capacity and performance beyond what a single MDS node can manage.

Lustre Object Storage Servers

Object Storage Servers (OSS) manage file data IO operations (read, write, truncate, etc.) to the data objects stored in one or more Object Storage Targets (OSTs). Each OST has a locally-accessible backing filesystem such as ext3/ext4, or ZFS. The OST manages the data block allocation internally within the backing filesystem – the on-disk representation is never exposed to the client. Each OST is typically stored on high-bandwidth disks such as SATA in a RAID-6 configuration, typically handling IO requests of 1MB or larger from the clients.

There are typically tens to thousands of OSTs per Lustre file system, exported from tens to hundreds of OSS nodes. Lustre has excellent IO scalability, since

files can be striped over one or many OSTs, and tens of thousands of clients can communicate directly with the different OSS nodes in parallel without contention. The OSS nodes manage all block allocation and data access locking, allowing scalable IO performance while ensuring backing file system integrity by preventing malicious or malfunctioning clients from directly modifying the backing filesystem metadata.

The MDS provides clients the layout for each file when the clients open the file. The layout maps the filename to one or more data objects, striping each file's data over these objects, each one stored in a different OST. The clients compute the mapping between the file offset and the corresponding object and OST on which that file data resides, so the MDS does not need to be involved in file IO operations.

Lustre Networking

Lustre Networking (LNET) can leverage a wide variety of high performance network interconnect fabrics, utilizing native RDMA capabilities available in most modern networks, such as InfiniBand and Cray XT SeaStar, among many others. Lustre Routers can efficiently bridge heterogeneous RDMA fabrics within the compute cluster and the Shared Storage Network, as well as between the Shared Storage Network and lower-speed/higher-latency networks over a WAN. This allows Lustre to maximize performance of the available networks while allowing different computing resources to share a common storage pool.

HPCS Architectural Improvements

The architectural and functional enhancements needed to improve Lustre scalability to the point where it can meet the HPCS partner goals are described below. Detailed design documents are available separately. Currently, there are two main limitations to Lustre's scalability that prevent it from meeting the HPCS goals today.

Firstly, the ext3/ext4 backing filesystem is nearing the end of its ability to scale to larger capacities needed in the future. The lack of filesystem data management features such as snapshots, data replication, data encryption, and data integrity prevent Lustre from readily providing such functionality itself. Secondly, allowing only a single MDS+MDT to be active in a Lustre file system at a given time places an upper limit on the filesystem metadata performance and size. These limitations are addressed below.

ZFS

Lustre's current disk file system `ldiskfs`, which is currently based on the ext3 filesystem, has scalability limitations that preclude its usage through the HPCS project timeframe. It is currently limited to an 8 TB maximum file system size, and an upper limit of 4 billion inodes that can be stored in a single filesystem. It offers no guarantee of data integrity, nor any mechanism to even detect data corruption. While ext4 will soon be possible to increase the single-filesystem capacity to 16TB and beyond, there are still a number of areas that are lacking. The need to run a full `e2fsck` on such large filesystems is becoming increasingly problematic, even with recent performance improvements, due to the time it takes on such large filesystems, and the requirement that `e2fsck` run with the filesystem offline.

A new generation of file system is needed to address the dramatic increases in storage capacity required by HPCS, as well as provide additional robustness and advanced data management features. This is important for both the Parallel and Capture environments.

To improve the reliability and resilience of the backing file system on the OST and MDT components, Lustre will add support for the existing ZFS disk filesystem. The addition of ZFS support will offer a number of improvements, such as increased single filesystem capacity into the hundreds of Terabytes or more, billions of files in a single filesystem, and dynamic addition of storage capacity. Other capabilities such as transaction-based copy-on-write semantics, improved data integrity with internal data replication, end-to-end checksumming of both data and metadata, and online integrity verification make ZFS a very attractive choice for HPCS-scale filesystems. As well, advanced features such as

filesystem snapshots and hybrid storage pools using a mix of spinning and solid-state disks will allow Lustre to meet future filesystem requirements.

Copy-on-write means that ZFS never overwrites existing data on disk. Modified data and metadata is written to a newly allocated block, and the block pointer to in-use data is updated to reference the new block as part of a single transaction. This mechanism is used for all file system data and metadata and provides an atomic commit mechanism for filesystem transactions. Writing the modified data to a new location also avoids the problem of partially overwriting existing data during system failures and being left without a valid copy of the data.

To avoid silent data corruption caused by disk or controller failures, ZFS checksums all data and metadata blocks on write and verifies the hash value on every read. The per-block checksum is at minimum a robust 64-bit checksum, and optionally a cryptographic-strength SHA-256 hash. The block hash is not stored with the data block, but rather in the pointer to the block, along with a block version number, so that corrupted, misplaced, or missing writes can be automatically detected by the filesystem. The blocks that contain the data and metadata checksums are themselves checksummed in a Merkle tree² that verifies all of the state in each transaction is valid.

ZFS computes block hashes for every read and write, so errors not caught by other file systems can be both detected and corrected by ZFS. Errors that ZFS will detect and correct:

- Phantom writes, where the write is not actually persistent on disk. This would likely be undetectable by T10 Protection Information (PI, formerly called T10 DIF) because there is no version number in the block, and the checksum and data are stored adjacent to the block it protects, so it would also not be overwritten.
- Misdirected reads or writes, where the disk addresses the wrong block and reads the wrong data, or overwrites other data. This would be a possible failure scenario for T10 PI if the sector address has the same low 32 bit value. Single bit flips in the high 32-bits of a sector address can cause such failures undetectably with T10 PI.
- DMA parity errors between the storage array and server memory, or corruption by the driver, since the checksum validates data inside the storage array.
- Driver errors, where data winds up in the wrong buffer inside the kernel.
- Accidental overwrites, such as swapping to a live file system.

2 Hash Tree http://en.wikipedia.org/wiki/Hash_tree

Through Lustre's usage of ZFS for the backing filesystem it will be able to take advantage of the following already available ZFS features:

- **Tunable checksum** algorithms that can be selected for either performance (Fletcher-4) or robustness (SHA-256), and can be updated in the future as needed.
- **Integrated data checksums** and RAID mirroring/parity, so there is never a RAID parity update hole where a RAID stripe undergoes a read-modify-write operation that might fail in the middle or hurt performance. ZFS only ever does full RAID stripe writes.
- **Self-healing data** allows ZFS in a mirrored or RAID configuration to not only detect data corruption, but it automatically corrects the bad data in place if possible, avoiding costly full RAID scanning/rebuild.
- **Verified RAID parity reconstruction** can ensure that the correct data is being rebuilt using the data checksum, unlike configurations that have separate filesystem and RAID subsystems.
- **Simplified storage management** for the unified storage pool allows ZFS to be a single point of control for a large number of disks.
- **Improved administration** due to ZFS's detection and reporting of data corruption on all read and write errors at the block level. This simplifies system administrators' identification of which hardware components are corrupting data.
- **Hybrid storage pools** allows ZFS to combine high-speed I/O devices such as SSDs, and slower SATA disks in a single hybrid storage pool. In many cases this allows fine-tuning of both the storage capacity and performance independently for the desired workload. The Read Cache Pool, or 2-level Adaptive Replacement Cache (L2ARC), acts as a cache layer between memory and the disk. This support can substantially improve the performance of random read operations.
- **Extreme scalability** ensures ZFS will remain viable for future storage needs. ZFS is designed from the ground up as a 128-bit file system. This means that there are virtually no hard limits on file and filesystem size for a single MDT or OST, maximum stripe size, and maximum single file size.

End-to-end Data Integrity

In Lustre today, data checksumming is enabled by default on the Lustre client when the data is sent from the client to the server, and optionally also when the data is first written into the client memory. This allows Lustre to detect data

corruption introduced by the network between the client node and the server, or between the server and the disk drive in the Lustre storage system at write time, and also corruption in the server memory. However, the backing ext3 filesystem on the OSTs today does not persistently store any checksums for the file data, while the Sun ZFS filesystem implements persistent on-disk checksums for both on-disk data and metadata.

The Lustre-level data checksumming has resulted in the discovery and correction of previously undetected data corruption in network cards, despite the presence of network-level checksums. These faulty cards silently introduced data corruption in their internal memory before the network-level checksum was computed by the card and went undetected for months without the use of Lustre-level checksums.

For the HPCS project, Lustre will implement efficient end-to-end data integrity verification by combining the Lustre data checksumming with the ZFS on-disk data checksumming using a scalable hash method known as a hash tree or Merkle hash^{2,3}.

The hash tree shown in Figure 2 will not only allow the data hash to be computed in parallel on the client, leveraging multiple CPU cores if they are otherwise idle, but the interior hash values can also be used on both the client and the server,

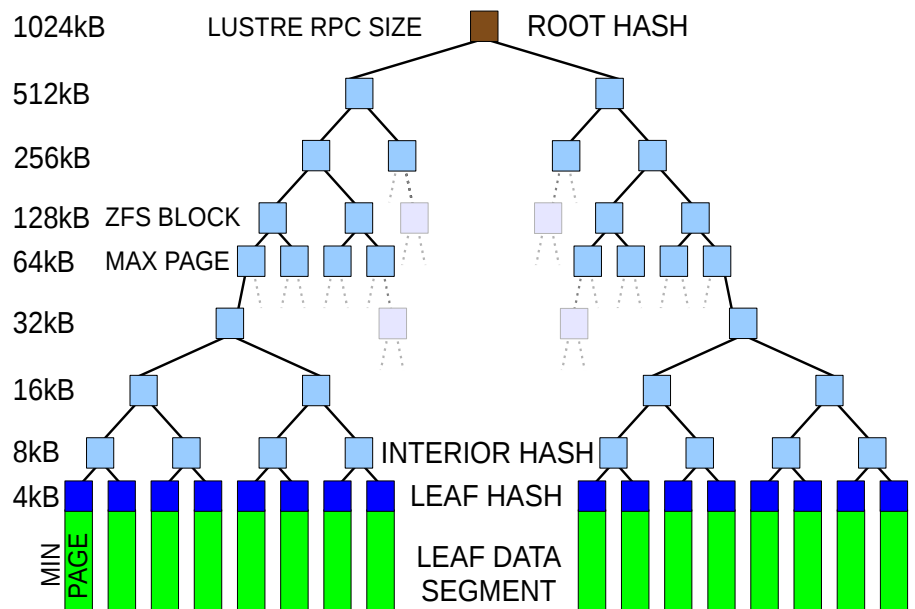


Figure 2: Hash tree calculation with multiple block sizes

independent of the page size on the client and server, and also independent of the backing filesystem block size. This means that the hash tree needs to be computed only once on each node in order to validate both the incoming network data and the persistent on-disk data.

The hash tree begins with a standard hash algorithm such as Fletcher-4, or SHA-256 that is computed independently on the leaf data segments. The size of the leaf data segment is chosen to be some smallest common data size, such as 4kB. Since the cost of this hash calculation is proportional to the total amount of data, this step does not add any overhead compared to a simple full-data hash function.

For each level up the hash tree, starting with the leaf hash, an interior hash function is computed on the concatenated hash digest values of each pair of subsidiary nodes. The interior hash function is often the same as the leaf hash function with a different seed, but need not be.

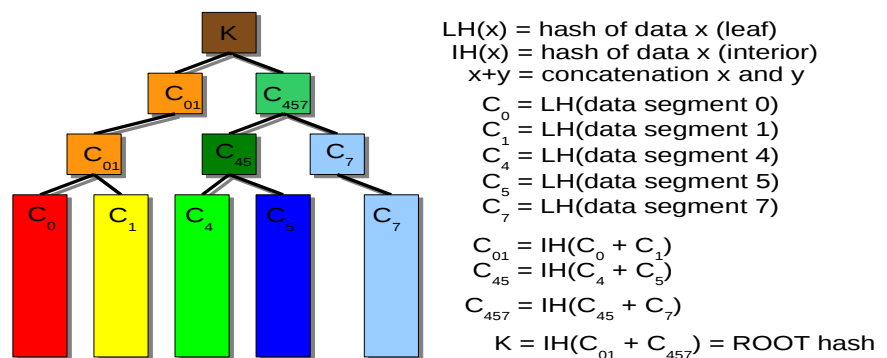


Figure 3: Hash tree for unaligned, discontinuous RPC

If the number of leaf segments is not a power-of-two value, or the leaf segments are not aligned on a power-of-two offset in the file, or are discontinuous, as shown in Figure 3, then the hash value of the unaligned segment is promoted unchanged up the hash tree until it matches a power-of-two alignment and is combined with its sibling in the tree. This increases the chance that the combined hash values can be used unchanged on the peer if the block size or page size is different. The overhead of computing the tree of interior hashes on the leaf hash digests is only a small fraction of computing the leaf hashes, about 1.5% for typical Lustre usage.

The hash tree allows a single hash value to be computed and stored for each page on the client, and a single hash sent over the network. Being able to re-use the leaf hash values avoids a 100% overhead if the leaf hash needs to be

recomputed (e.g. to translate between network and disk block sizes), or to store a digest value for each data segment.

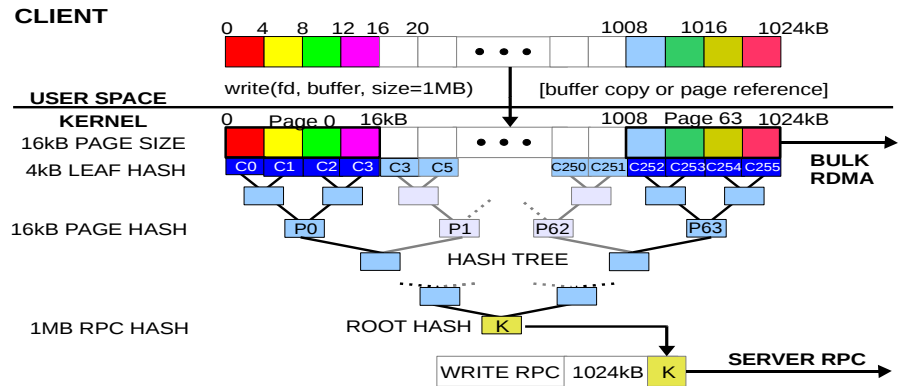


Figure 4: Client write RPC checksum generation

For the Lustre implementation the client will use a base data segment size of 4kB, regardless of the client page size. As shown in Figure 4: Client write RPC checksum generation, the client will compute a set of hash values $\{C_0 \dots C_n\}$ for each segment of the write using the leaf hash function. This hash can be computed while the data is being copied into the kernel buffers and kept with the page until it is sent over the network. If the client page size is larger than the segment size then multiple segment checksum values $C_{j,k}$ will be computed and stored per page as P_n . When the data is being sent from the client to the server the per-page hashes $P_{0..N}$ from the individual pages that compose the write RPC will be combined in the hash tree and the root hash K put into the write RPC sent to the server.

In the case of O_DIRECT writes on the client, no significant difference exists in the end-to-end integrity implementation. The same process to compute the leaf block hash values needs to be done on the client regardless of the buffer origin.

As shown in Figure 5: End-to-end checksum on the server, upon receipt of the RPC and bulk RDMA transfer of the pages, the server repeats the process of hashing segments of data, generating a set of server checksums $S_{0..N}$ and then combining them to find its root hash K' . If $K == K'$ then we know with a high degree of certainty that the set of checksums $\{C_{0..N}\} = \{S_{0..N}\}$, and as a result also that $C_0 = S_0, C_1 = S_1, \dots, C_N = S_N$. In turn we can know with a high degree of

certainty that each segment of data on the client is the same as the segment of data on the server.

The reason for the extra complexity of the tree checksum algorithm becomes

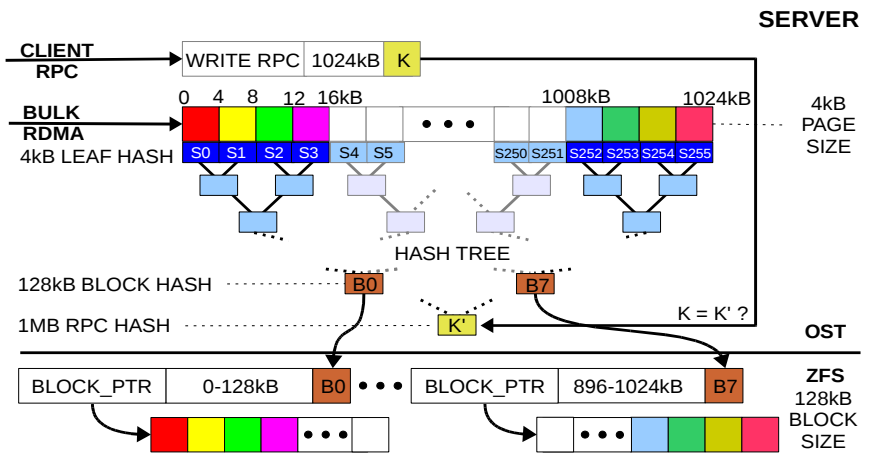


Figure 5: End-to-end checksum on the server

clear when storing the checksum in the backing storage. The block size of each ZFS file is variable (up to 128kB today, possibly larger in the future) and is not known by the client in advance. The ZFS block size will not necessarily align with the page size of either the client or server, and may increase as the file size grows. By computing and tracking the checksums $S_{0..N}$ in 4kB segments on the server pages, we can compute a new hash B_n for each file block using only the per-page checksums, without having to recompute the checksum on the entire block's data.

Additional benefits can be realized when the data is being cached in RAM on the client or server, if the per-leaf-segment hashes are kept in memory with the data. Each time the data is read by a client the network RPC checksum only needs to recompute the hash tree for the segments being read instead of for the entire data, reducing CPU overhead by 98% for each RPC hash, compared to computing the hash each time.

Also, keeping the checksums with the data in cache of the server will allow memory corruption to be detected long after the data was read from disk without

any additional CPU overhead, because the peer would be able to detect a checksum error during RPC processing.

It would even possible to have the server skip any hash verification of the on-disk data on reads without any risk of data corruption, yet significantly reducing the server CPU load. This can be achieved by doing only the hash tree of the per-segment leaf hashes on the server and return the resulting root hash to the client. Since the client will always do a full checksum of the data to regenerate the root hash and verify it against the RPC hash, only in the rare case of the client detecting a data corruption does the client need to request the server to do a full data checksum of the data. The server could then detect if there was data corruption over the network to the client, if the data in cache is bad, or if the data was already bad when read from disk. In the latter case the ZFS filesystem can read a duplicate copy of the data, or regenerate it from the on-disk parity. While this will not be implemented in the initial version of the end-to-end integrity implementation, it would be possible in the future if it is found that the CPU overhead of data reads is problematic.

RAID Rebuild Performance Impact

When disk failures occur in a RAID set, it is important that the RAID set be returned back to full redundancy as quickly as possible. This reduces the risk of a second drive failure occurring in the same RAID set while the rebuild is in progress, as well as avoiding the constant overhead of doing parity reconstruction for the failed disk. The other important aspect of RAID set rebuilding is that the process minimally impact the normal operation of the RAID set. Both of these concerns are being addressed with several new mechanisms being added to Lustre and the backing filesystem.

In the example 115PB filesystem, having 36720 4TB disks in RAID-6 8+2 configuration, and a single disk Mean Time To Failure (MTTF) of 4 years, this gives us an average disk failure rate of about 1 disk/hour. To do the RAID rebuild of a single 4TB disk at 30MB/s (50% of the IO rate of a single disk) would take about 38h, during which that OST is running with severely reduced performance. With the 1 disk/hour failure rate this implies about 38 OSTs are being rebuilt at any time.

Lustre-level Rebuild Mitigation

In order to mitigate performance impact caused by a degraded or rebuilding RAID set on an OST, Lustre can entirely avoid creating new files on the degraded OSTs entirely. While this would reduce the aggregate bandwidth of the filesystem

marginally, it is actually a net performance win for most workloads. In the example 36720-disk configuration, using 30 disks per OST (three RAID-6 8+2 sets each) would result in 1224 OSTs in total. By avoiding new file allocation on the average of 38 OSTs undergoing RAID parity reconstruction would reduce the aggregate write performance by only about 3%. If existing files resident on the degraded OSTs need to be read or modified during the rebuild window this will still be possible at reduced performance, because the OST is still available.

In contrast, if these degraded OSTs would continue to be used for new file allocations and writes, the RAID parity reconstruction could degrade the write performance for those OSTs by 50% or more, assuming a fixed 50% rebuild load needed to finish the rebuild in 38h. If an HPC job is waiting for all of its processes to complete their writes to the filesystem (e.g. checkpoint), and the files are evenly distributed over all OSTs, including the degraded ones, then the files being written to the degraded OST would take 50% longer to complete than files written on other OSTs. This in turn could double the entire job's IO time while it is waiting on a barrier for the IO to complete to the slow OSTs, no better than if all OSTs were undergoing rebuild.

Avoiding the degraded OSTs not only minimizes global IO performance impact, it also minimizes the load on the rebuilding OSTs themselves, allowing the rebuild to run at 100% IO bandwidth for a larger fraction of the time, possibly finishing in half of the time. This in turn reduces the number of degraded OSTs and their performance impact by half.

For Lustre MDTs a similar approach could be used, though with a proportionally larger impact due to fewer MDTs in the filesystem. This would allow new directories and the files created therein to be on non-degraded MDTs. However, it is anticipated that the use of SSD storage for the MDTs will provide sufficient excess IOPS capacity within the MDT that the externally visible performance degradation of RAID rebuild will be significantly smaller than that on spinning media due to the lack of seek cost for reads during rebuild.

ZFS Resilvering Improvement

There is also ongoing work to improve the performance and reduce the impact of ZFS RAID parity rebuild (*resilvering* in ZFS terminology) on the IO performance itself. This is mostly encompassed in two independent and complementary projects.

Due to the ability of ZFS's built-in RAID-Z implementation to avoid all read-modify-write of partial RAID stripes, and the ability to store variable-sized blocks in the filesystem, the way that ZFS rebuilds the disks in its RAID-Z Virtual

Devices (VDEVs) is significantly different than how a standard RAID device would rebuild the parity.

Rebuilding the RAID parity in a ZFS RAID-Z VDEV involves traversing the internal metadata tree in order to determine the checksum of each VDEV block, along with the location and size of each block within the VDEV. At this point the data can be reconstructed on the replacement disk with full confidence that the reconstructed data is valid, because a strong checksum of the block can verify that the remaining data and parity blocks produce the correct data. Also, since the ZFS internal metadata tree only covers allocated blocks, parts of the failed disk that are not currently in use do not need to be reconstructed.

Unfortunately, the current implementation of ZFS parity reconstruction is sub-optimal as it may cause a fair amount of random disk IO during the metadata tree traversal that can negatively impact rebuild performance and normal file IO to the rebuilding VDEV. A major performance improvement to the resilvering algorithm being designed is to do the resilvering in two passes. The first pass traverses the VDEV metadata in tree order, reading the estimated 3% of metadata blocks that encompass the failed drive and the checksum for those blocks. The second pass reads the block data for the parts of the VDEV that are actually in use, reconstructs the parity, verifies the checksum, and writes the rebuilt block to the replacement disk. This can be done in linear order, as the first pass has generated a full list of affected blocks on the failed disk.

While a two-pass VDEV resilvering algorithm has some overhead compared to a normal RAID rebuild operation, there are also advantages with ZFS resilvering in that only the in-use space of the failed disk is rebuilt. As most HPC filesystems are kept in the neighborhood of 80% or less full, there is a significant amount of time that can be allocated to the metadata traversal. If the 100% (60MB/s) bandwidth rebuild time is 19h for a 4TB disk then there is almost 4h that could be spent on the metadata traversal phase before the ZFS resilvering would be slower than a standard RAID rebuild.

ZFS Distributed Hot Space

A second enhancement that is being designed for the ZFS filesystem will be *Distributed Hot Space*. Traditionally, one or more extra disks in the storage enclosure are reserved as hot spares for the increasingly common case that there is a disk failure in the RAID set, in order to allow RAID rebuild to begin immediately after a disk failure is detected so the window of vulnerability is minimized. Unfortunately, these hot spare disks remain completely unused and idle for the majority of time.

With Distributed Hot Space, instead of keeping a whole disk empty for the hot spare, an equal fraction of each disk in the RAID set (including the formerly hot

spare disk) is kept empty, such that the reserved free space is slightly larger than the original hot spare disk. This hot space forms a virtual hot spare disk that can be used for immediate RAID set rebuilding in case of a disk failure, as would a normal hot spare disk. In normal operations, however, the individual disks that make up the virtual hot spare disk are all contributing to improve both normal IO and also to reduce the time taken for parity reconstruction.

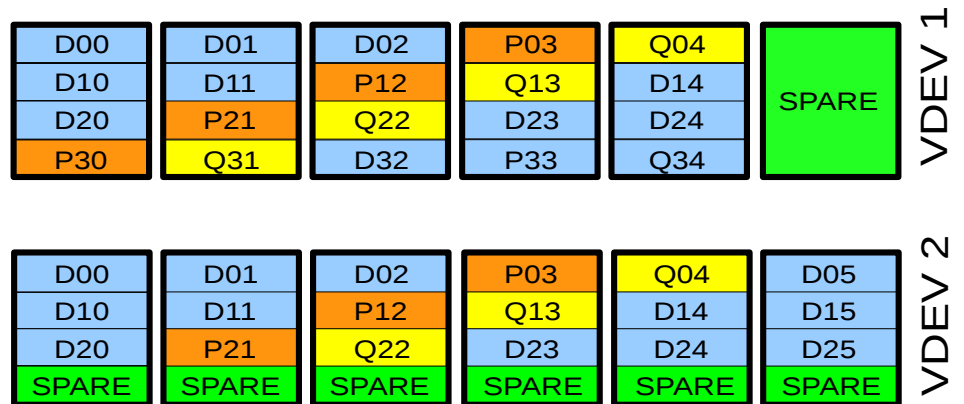


Figure 6: 6-disk double-parity plus spare RAID configurations

Figure 6 shows two different double-parity VDEV configurations using 6 disks per RAID set. VDEV 1 shows a 3+2 RAID set, with a dedicated 6th disk for the hot spare sits idle. VDEV 2 is using distributed hot space, which allows the 6th disk to be put into use as part of a 4+2 RAID set, while 1/4th of the space of each disk is reserved for the virtual hot spare disk. By utilizing the 6th disk inside the 4+2 RAID set, this increases the usable IO bandwidth by 33% in this example compared to the 3+2 RAID set, while keeping the total usable space constant.

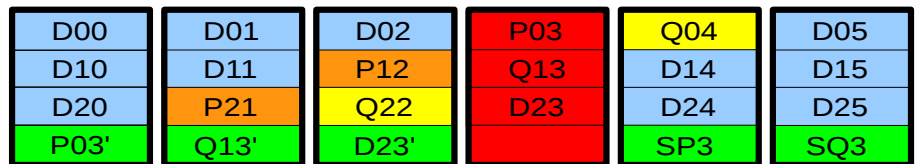


Figure 7: Degraded VDEV with Distributed Hot Space

In Figure 7, disk 3 of the VDEV has failed, and the RAID parity rebuild has placed its data and parity blocks into the hot space of the VDEV. Note that the hot space itself still has double parity RAID so that if any 2 other disks in the VDEV fail then its hot space will maintain data integrity in the same manner that a real hot spare disk would.

In fact, the distributed hot space is expected to provide superior rebuild performance over a hot spare disk. The hot space is spread over all of the disks in the VDEV, so they will all contribute performance for the write operations being done during the rebuild, instead of only being used for reads while the writes all go to the single hot spare disk. Since the capacity of modern disks is growing proportionally faster than their IO performance, using a fraction of the available disk for hot space is a beneficial trade off, as the aggregate IO performance of the whole VDEV can be improved by a similar ratio versus having an idle hot spare disk.

When the failed disk is replaced with a new disk, the data in the hot space will be migrated over to the new disk. This can be done at a rate that does not significantly impact performance because the RAID set is fully redundant during this time and is not in danger of double failures. While the second data copy from the hot space to the replacement disk will also impact performance, if the copy is limited to 25% of the disks' bandwidth this is no worse than the performance would *always* have been with a normal hot spare disk, and once rebuild is complete the performance is again 25% higher than in a standard RAID configuration.

In a typical ZFS configuration there are multiple underlying VDEVs that make up a single OST filesystem. This will allow each VDEV to export a virtual hot space disk within the storage pool for use by any of the other VDEVs in that pool. This will allow each rebuild to utilize the full write bandwidth of all disks, but ensures that multiple failures within the storage pool can immediately begin rebuilding, either concurrently or sequentially, allowing time for the failed disks to be replaced.

Filesystem Integrity Checking

The HPCS goals are to verify the integrity of a Lustre filesystem with 1 trillion files (10^{12} files) in 100 hours. The integrity check should ensure consistency of both the back-end filesystem (ldiskfs, ZFS) and the Lustre filesystem cross-node consistency. A filesystem with this many files will itself take in the neighborhood of 4PB of raw storage just to store the metadata, once overhead such as metadata replication and internal filesystem overhead is taken into account.

It should be possible to do the filesystem integrity checking on a live filesystem (online). It would be acceptable if we need to take individual servers off-line for back-end filesystem checking should they be severely corrupted, but the rest of the filesystem should not be affected by this operation. At the Lustre level there are only a limited number of failure scenarios that can be hit, due to the isolation of the backing on-disk filesystem from the clients.

The ZFS back-end filesystem can verify the data and metadata integrity while the

filesystem is mounted and in use. In normal operation the filesystem idle cycles can be used to opportunistically perform continuous or periodic integrity checking of the local ZFS filesystem. Similarly, the Lustre distributed integrity checking can validate the consistency of any distributed object without doing a filesystem scan and can be run during filesystem idle time. This will provide a high degree of confidence in the on disk data integrity even in the face of ongoing silent data corruption at the device level.

In the case of a catastrophic filesystem failure, a full filesystem integrity check might be needed. Due to the extremely high file verification rate, approaching 3 million files per second, this requires parallel checking of the filesystem using a considerable number MDTs in a CMD configuration, depending on the actual RPC rate that each server can handle. At 25000 RPCs/second per MDT sustained rate, this would require about 128 MDTs on 128 MDS nodes. Each MDT would manage about 32TB of raw space in a RAID-1 configuration.

This quantity of disks will provide the necessary filesystem size, and an IOPS rate matched with the 128 MDS network interfaces to allow a high enough RPC rate to the other MDSes and OSSes to verify the cross-node consistency.

In order to avoid repeat traversal of the MDT backing filesystem metadata, or huge amounts of RAM needed to cache the full state of the filesystem, the Lustre-level coherency checking will be integrated into the ZFS-level online disk scrubbing mechanism via a callback. Immediately after the ZFS filesystem scrub has validated the on-disk data the inode table blocks and directory leaf blocks will be passed on to a Lustre-level callback function where it will do further Lustre-level consistency checks based on the content of the block.

The Lustre-level inconsistencies that can arise between servers are somewhat limited in scope due to the object-oriented nature of Lustre. In order to ensure that distributed consistency checking can be completed in one pass with a minimal amount of saved state, there are back-pointers stored on the OST objects that reference the MDT inodes. For MDT-OST consistency checking the required checks are:

1. Multiple MDT entries do not reference the same object .
2. All objects for each inode on each MDT should exist .
3. Each object on each OST should belong to one file on an MDT.
4. File size on each MDT inode matches the (the sum of?) OST objects' size.

These checks can be done in a single pass during MDT inode traversal using MDT-to-OST RPCs, with the exception that any OST objects not referenced by the MDT need to be tracked and handled at the end of the MDT traversal, using

about 1 bit of memory per existing object. The OST-specific object metadata, such as block allocation, can be checked entirely internally and in parallel on each OST. The OST object back-pointers will ensure that the MDT-OST references are valid, and will allow duplicate object references to be detected.

In a CMD environment there is an additional need to verify MDT-to-MDT consistency. As with MDT-OST consistency checking using reverse object-to-inode back-pointers to allow efficient object checking, there are back-pointers from each MDT inode to its parent directories to allow single-pass consistency checking for the MDT namespace. For MDT-MDT consistency checking the checks that are needed are:

1. Each directory entry references a valid inode which has a corresponding FID-to-path (a FID is a 128-bit File Identifier) reverse mapping entry in that inode for both local and remote entries.
2. The directory entry file type should match the inode file type.
3. For every FID-to-path entry on an inode there is a directory entry that references the inode. An empty FID-to-path table indicates an unreferenced inode.
4. The FID stored on each inode should map to the correct MDT in the FID Location Database.
5. The inode link count for regular files should match the number of entries in the FID-to-path table. Directories should only have a single FID-to-path entry, and the inode link count is equal to the number of subdirectories, as determined by the validated directory entry file type, plus the parent reference.
6. There should be no circular directory loops in the metadata namespace.

While the full cross-MDT coherency checking cannot be guaranteed to complete with only a single read of each block, it can be a single-pass algorithm that opportunistically caches unverified directory and inode blocks up to the amount of available RAM. During MDT filesystem traversal the Lustre MDT code will be passed blocks from the callback and can verify some of the content immediately based on other blocks already in cache, and can defer the checking of the remainder for some time to aggregate the remaining IO and/or wait for the block to be processed under normal filesystem traversal. Since the ZFS filesystem is copy-on-write and related metadata blocks are usually written together on disk this will be fairly common.

If all of the directory entries in a directory block have been verified to reference valid inodes the block can be garbage collected and only a single bit is needed per block to track this state. Likewise, if all of the inodes in an inode block have

been verified to have valid directory entries the inode block can be garbage collected, using a single bit in the same block bitmap. For a single 32TB MDT with this block bitmap would consume at most 1GB of RAM per MDT in an uncompressed form. If the cache of incompletely verified blocks is growing too large, then these blocks can be processed aggressively to remove them from memory by doing the remaining checks out-of-order and freeing the directory or inode block, at the cost of some out-of-order reads from disk. Since there are a number of other unprocessed blocks in memory these extra reads can be batched and ordered to reduce disk seeks, and when read may also serve to further verify other in-cache blocks.

The number of additional blocks that need to be read to process any directory or inode block is bounded, as there is a fixed number of entries in each block, and each entry can be verified with a bounded number of reads. This ensures forward progress can always be made even under very constrained memory. In the very worst case, every directory and inode block would need to be read twice from disk, once to verify the directory-to-inode mapping during directory traversal, and once to verify the inode-to-directory back-pointers during in-use inode traversal.

Clustered Metadata

Allowing only a single Metadata Target (MDT) in a filesystem means that Lustre metadata operations can be processed only as quickly as a single server and its backing filesystem can manage. To date, this has not been a critical limitation and it has in the past been addressed by selecting an MDS node and MDT storage that are capable of handling the required metadata load. However, as a Lustre filesystem grows, a single MDS+MDT becomes a performance bottleneck and constraint on the total number of files in the filesystem. This is important for both Parallel and Capture environments.

It is not possible to cost-effectively scale a single MDS node to meet the demands of the largest compute systems. Increasing the amount of RAM, number of CPUs, number of network and disk interconnects in a single MDS node increases the cost of the system disproportionately to the amount of incremental performance gained. Allowing Lustre to increase the metadata performance by adding lower-cost independent MDS nodes and independent MDT filesystems ensures that system architects can scale the metadata performance in a linear manner, similar to the way Lustre scales the IO performance by adding OSSes.

The metadata performance limit is being addressed by the addition of Clustered Metadata Server (CMD) functionality to Lustre. With CMD functionality, multiple MDSes provide a single filesystem's namespace jointly, storing the directory and

file metadata on a set of MDTs. Each MDT exclusively manages a set of directories and inodes in its backing file system. Each directory inode has a layout attribute that describes whether the directory entries are contained in a single MDT directory or are split over multiple directories (called slave directories), each one on a separate MDT. The directory layout also describes the parameters for the mapping of filenames to the directory object(s), such as hash function, number of slave directories. The layout of each directory is provided to clients during the directory lookup.

The clients can determine which MDS+MDT a given file or subdirectory resides on by hashing the filename and using that as an index into the directory layout to find the directory object that will contain that filename, if present and sending the lookup request to the MDS managing that directory object. Hashing of filenames to map directory entries to a specific MDT is analogous to the offset calculations done on file data that is striped over multiple OSTs.

Normally, a regular file inode will reside in the same MDT as its directory entry in order to allow the name lookup request to also return the inode data for that file. This will not always be possible, in cases such as hard links, rename between directories, or if the directory was split after it was created because it was growing too large. Subdirectories will often be located on an MDT different from its parent, in order to load balance space usage.

Once a lookup operation is completed, the server returns a 128-bit File Identifier (FID) to the client, which allows the client to determine which MDT the given file resides on for fetching the attributes and performing locking. The FID to MDT mapping is handled by the FID Location Database (FLD), which is a table that maps large ranges of FIDs to their corresponding MDT index number. The FIDs are selected such that the FLD will remain very compact and can be replicated across the MDTs, and cached entirely in RAM on the clients. It will be updated only very infrequently, on the order of once per trillion new files created, so the update overhead is extremely low.

The clients merge the multiple MDT filesystems into a single namespace presented to users. The MDS nodes provide distributed locking services to the clients to ensure namespace consistency. MDT storage is internally redundant (typically RAID-1+0) and shared to multiple MDSes in failover pairs to deal with MDS node failures.

While there is some overhead in CMD compared to a single active metadata server, the large majority of metadata operations such as file creates, attribute lookups, and unlinks will be performed by the single MDS that manages a particular inode. Performance for most individual operations is comparable to a system with a single MDS, but the individual operations can be performed in

parallel on multiple MDSes so the aggregate metadata performance is expected to scale in a nearly linear manner with the number of MDSes in the filesystem.

A limited number of less frequently performed operations, such as some sub-directory creation, cross-MDT rename, and hard-link operations may involve multiple metadata servers, and will need multiple RPCs to complete. The number of cross-MDT operations will be actively limited by file creation policy specified by the user and/or administrator in a similar manner to the striping of Lustre files today. Individual directories can be split over multiple MDTs to improve aggregate performance within that directory, but even for split directories the file creation, lookup, and unlink for specific filenames will normally be considered local MDT operations and will be completely asynchronous.

In initial releases of CMD the cross-MDT distributed updates within a single operation will be strictly ordered between the MDTs involved to ensure that arbitrary MDS and client failures will always result in a deterministic state that does not result in loss of namespace coherency or loss of data. In most cases, normal Lustre recovery will handle issues like dropped or resent requests or RPC replay.

Because metadata operations are not idempotent, but rather are order-specific and dependent upon each other, in the event of a total cluster power loss there are some combinations of failures that will at worst result in the temporary unavailability of some small amount of usable filesystem space. This space would be detected and recovered as part of the online integrity checking as described above in *Filesystem Integrity Checking*. The full CMD distributed recovery mechanism, known as *MDS Recovery Epochs*, is being designed and will be available in a later version of Lustre, allowing fully asynchronous metadata operations with guaranteed recovery semantics.

Imperative Recovery

The current Lustre recovery mechanism uses the client RPC timeouts to also trigger client-driven recovery actions such as reconnecting to a server after communication problems or server failover. However, under high load from tens of thousands of clients, a server might take hundreds of seconds to complete a request. As a result, a client cannot itself detect server failure in a shorter interval than the normal RPC service time.

While the *Adaptive Timeouts* feature ensures that the client's expected RPC response time is kept at a minimum during varying server loads, this feature in itself will not eliminate the time that the client spends waiting for an RPC reply when the server has in fact already failed.

In order to reduce or eliminate the time that the client spends waiting for the RPC

timeout before entering recovery in the case of actual server failure, *Imperative Recovery* will have the server notify the clients immediately after the server recovers or the service has failed over to another node.

By notifying the clients directly after the service has been restarted, the clients will no longer depend on the RPC timeout to determine that the service has restarted on the same or a different node. The clients will also not have to try all of the possible service failover servers in order to determine which one is now hosting the service, as the server-driven recovery will also notify the client which server is hosting the service.

Imperative Recovery will eliminate the potentially lengthy client wait for RPC timeout before starting recovery. It will also eliminate the server wait for all of the clients to have connected before it can start and finish the recovery, since it will know very quickly which clients are still available to participate in recovery.

If a cluster already has robust node health monitoring, this can be leveraged to inform Lustre of client or server death quickly. In the absence of existing node health monitoring, Lustre will in the future utilize an internal health monitoring network in order to notify the server of client failure.

Channel Bonding

Channel bonding is an enhancement being added to Lustre to improve network performance and resiliency. While it is possible to use some forms of channel bonding in Lustre today, such as Ethernet channel bonding, this is not always the most desirable configuration. Lustre Channel bonding will allow multiple physical Lustre Network Interfaces (NI) to be combined into a single logical NI. NIs of different types, such as Ethernet and InfiniBand, may be combined into a single bonded NI. Also, the slave NIs of a bonded NI may be of the same type but belong to different networks.

The individual NIs that make up a bonded NI are called slave NIs. Bonding of NIs is single layered; a bonded NI can not itself be a slave NI. While channel bonding is of use in a number of network configurations, it is a mandatory feature for the capture environment due to the very high single-node bandwidth goals.

The bonded NI can operate in one of two modes – either in standby mode or in load balancing mode.

In standby mode, only a single slave NI in the bonded NI is active at a time. The other slave or slaves in the bonded NI are idle unless the primary slave fails and then the designated backup NI becomes active. If the standby fails then it will failover to another standby NI (if available). When the default NI becomes available again, the default NI may become the active NI again immediately or the standby NI may remain active until it fails and the default NI becomes active

again. This will be controlled by standby mode policy options specified by the administrator.

In load balancing mode communication is distributed across all the slave NIs. The traffic is allocated to the slave NIs according to a load balancing policy, e.g. round robin. Since LNet does not guarantee ordered delivery, and Lustre already has to handle out-of-order message delivery, there is no problem with messages being reordered by different links.

Performance Enhancements

In addition to the architectural features discussed above that are targeted specifically at HPCS performance deliverables, Lustre scalability will be enhanced by performance improvements to existing areas of the system.

SMP Scalability

System-wide Lustre scalability can be enhanced by improving the efficiency and scalability of individual Lustre servers. Some examples are:

- **Finer granularity locks.** Currently in Lustre, LNET has one global lock that serializes the initial processing of all network requests. LNET operations that require locking are, therefore, essentially single threaded. The SMP performance of LNET is currently being improved dramatically by replacing a single global lock with multiple locks, to enable greater parallelism and scalability of LNET operations on the increasingly prevalent many-core systems. Similarly, finer granularity locking at the Lustre RPC processing layer and per-cpu request queues will remove a lot of needless lock contention and allow more efficient request processing.
- **CPU affinity.** At the LND level, keeping the processing of given peers on particular CPU cores has resulted in substantial performance improvements due to reduced cache and lock ping-pong. This improvement can be extended to ensure that the requests that arrive into a particular CPU's cache are also processed by RPC service threads on the same CPU as much as possible.
- **Request optimization.** At the RPC level there are a number of optimizations that can be made to the request processing, such as minimizing the number of distributed locks that must be held to process a particular request, or to eliminate extraneous requests entirely. Work is already underway to perform such optimizations, and will continue in the future. As Lustre has a very flexible network protocol compatibility model, changing the way that requests are sent from the client and/or are processed by the server is possible without

loss of compatibility. While older clients may not be able to take full advantage of these optimizations, they allow incremental improvement of the network protocol for new versions of Lustre.

Network Request Scheduler

Lustre IO requests are sent to the OSTs in two parts. First, a small RPC request is sent to the server indicating the operation to be done (read or write), the object number, and a vector of extents to be operated on. The IO requests are currently queued on the OST in (mostly) FIFO order until a service thread is available to process it. The service thread then does the bulk data transfer via RDMA (if available) to get/put the data associated with the IO request from/to the client.

The Network Request Scheduler (NRS) will manage incoming RPC requests on a Lustre server, providing improved and more consistent performance. The NRS re-orders request execution to present a workload to the backing file system that can be optimized more easily while avoiding request starvation. This is analogous to a disk elevator which allows IO requests to sit in a queue, so the IO scheduler can coalesce requests and improve throughput. Unlike a disk elevator which orders the data buffers themselves, NRS does not require the bulk data buffers to be present in memory to schedule and reorder the IO. As a result NRS has much lower memory requirements for the same number of requests, and can therefore, examine and optimize a far larger number of requests at one time compared to a disk elevator.

The currently-implemented NRS prototype implements a request policy that gives clients a fair share of the servers' resources by time slicing the IO request processing between different objects. If clients are accessing different objects, this is essentially time slicing between clients. If clients are writing to a single shared file, this will optimize the IO for the entire object by submitting it to disk in a much more linear order than the current FIFO request processing.

NRS also avoids request starvation by enforcing a deadline for request processing. If a request has not been processed by its deadline then it will become a high priority request that will be serviced promptly. Extending NRS to implement Quality of Service (QoS), allowing specific clients or clusters to force priority request handling, is a natural extension of the current NRS prototype. Conversely, the NRS could also be used to throttle the request load from clients, thereby avoiding server overload and degradations in responsiveness and performance for other clients.

It has also been discussed to have coordinated request gang scheduling across all servers, in order to globally minimize the amount of time spent by applications waiting for IO to complete. In a shared computing environment where two independent jobs of equal priority are competing for the same server bandwidth,

it is probable that both jobs will have IO requests that take a large fraction of the time until both jobs' IO is complete. By scheduling all of the IO for a single batch job first it would allow at least that one job to return to computing after the minimal time. Even though the second job would be delayed in its IO, this would likely not be significantly longer than it would have waited had the two jobs been submitting their IO requests at the same time.

Implementing gang request scheduling would need additional cross-server coordination to manage the scheduling, as well as additional information from the client processes, such as job ID and task ID, to supplement the information the servers already have such as node ID and user/group ID. This would be relatively straightforward to implement in conjunction with higher-level IO APIs such as the Lustre MPI-IO ADIO driver. It could also be possible for existing software that are only using POSIX IO, or software that cannot be modified by informing the kernel of a task's MPI rank via a pre-launch script or environment variables (for MPI implementations that support this).

Metadata Writeback Cache

The Metadata Writeback Cache (WBC) is a proposed mechanism to aggregate metadata operations in a manner similar to existing caches for data operations, such as client data cache that can aggregate multiple small writes into a larger single network RPC and disk IO operation.

While it was originally thought that Metadata Writeback Cache was required to meet the needs of the Capture Environment, it is currently thought that the other improvements to metadata and data performance will be sufficient to meet most of the Capture Environment goals. The work on Metadata Writeback Cache feature has been deferred in order to focus on the other HPCS goals.

Conclusion

In conclusion, the above features will enable Lustre to successfully address all of the HPCS programs IO and storage goals. The table below summarizes which Lustre features address specific HPCS goals.

HPCS Goal	Lustre Feature
Metadata Performance: <ul style="list-style-type: none"> • 40,000 file creates per second from a single client • 10,000 metadata operations per second in aggregate (equal to "1s -1R" from 10 users in separate directories) 	SMP Scalability Clustered Metadata
I/O Performance: <ul style="list-style-type: none"> • 30 GB/s full-duplex streaming I/O bandwidth from a single client • 1500 GB/s in aggregate for both file per process and single shared file access 	Channel Bonding SMP Scalability Network Request Scheduler
Capacity: <ul style="list-style-type: none"> • 1 trillion (10^{12}) files per file system: <ul style="list-style-type: none"> ○ 10 billion files per directory ○ 100 PB maximum file system size ○ 0 to 1 PB file size range ○ 1B to 1 GB I/O request size range * • > 30K client nodes* 	Clustered Metadata ZFS
Reliability: <ul style="list-style-type: none"> • End-to-end resiliency equivalent to T10-DIF or better • Uptime of 99.99% • No impact of rebuilds on file system performance 	ZFS End-to-end Integrity Recovery Rebuild Performance
Client Accessibility: <ul style="list-style-type: none"> • O_DIRECT to maximize large transfer performance* • Open 100,000 shared files per process • Long file names and 0-length file lengths for applications that use file metadata as data records* • POSIX I/O API extensions proposed at the OpenGroup (In progress with Linux developers) 	[In progress with Linux developers]

. *Already supported in Lustre

Appendix A – HPCS Filesystem Example Configuration

In order to have an understanding of the scale of an HPCS filesystem in 2011, a sample Lustre configuration is given in Table 1 below based on projected filesystem sizes and device capacities and performance. This is only a rough estimate based on 100% performance scaling, simply to get an estimate of the component counts involved for the discussion of the filesystem features.

	Data	Metadata
Capacity	115PB	10 ¹² files = 2PB
Drive type	SATA	SSD
Drive capacity	4TB	800GB
Drive throughput	60MB/s	10000 IOPS
Drive configuration	RAID-6 8+2	RAID-1+0
Drive count	36720	5120
Server count	408 OSSes	128 MDSes
Target count	1224 OSTs	128 MDTs
Drives per target	90 disks/OST	40 disks/MDT
Server throughput	4.2 GB/s	25000 ops/sec
Aggregate throughput	1.6 PB/s	3.2M ops/sec

Table 1: HPCS Filesystem Sample Configuration