Rebuild Performance Design

Author	Date	Description of Document Change	Client Approval By	Client Approval Date
Ricardo M. Correia	05/06/09	Initial draft		
	05/06/09	Edits by John Dawson		
	06/12/09	Edits by John Dawson		

1 Introduction

The goal of the rebuild performance project is to ensure Lustre's ability to remain in operation and continue to provide high performance IO even during disk failures. This is part of ongoing efforts to improve the scalability and reliability of Lustre and to meet the reliability and performance objectives of the HPCS program. The large hardware configurations required to meet the IO requirements of the HPCS program and other large customers will mean that disk failures must be viewed as a normal occurrence rather than an exceptional event.

When disk failures occur, it is important that RAID parity reconstruction (the rebuild of the data on the failed disk) occur as quickly as possible. This reduces the risk of a second or third drive failure occurring in the same group of disks while the rebuild is in progress, which would lead to data loss in single- or double-parity RAID configurations. The other important aspect of RAID rebuilding is that the process minimally impact the normal operation of the disk group. Both of these concerns are being addressed with new mechanisms being added to Lustre and the backing ZFS filesystem it will use for the HPCS deliverables.

In a 115PB filesystem having 36720 4TB disks in RAID-6 8+2 configuration, and a single disk Mean Time To Failure (MTTF) of 4 years, we would expect to see an average disk failure rate of about 1 disk/hour. To do the RAID parity reconstruction 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, and consequently about 38 OSTs are being rebuilt at any one time.

The process of rebuilding a “traditional” RAID array, which is done when replacing a failing disk by a new one, consists of reading all the data on all the disks in the array, reconstructing the original data/parity of the failing device, and writing it to the replacement disk. After this process is complete, the array is restored to its original, full redundancy.

In ZFS, there is a similar and equivalent process, called *resilvering*, which is implemented differently due to the design of ZFS. This process starts by traversing the ZFS metadata tree to discover all the blocks of the ZFS pool that were contained on the failing disk. Upon reaching one of these blocks, it is read, reconstructed, if necessary from the redundant/parity information, its checksum is verified and the missing data or parity from the failing disk is written to the new disk.

One of the advantages of the ZFS resilvering process is that it can use the built-in ZFS checksums to make sure that the correct data is read/written from disk, and that there hasn't been any (possibly transient) silent data corruption. Another advantage compared to a traditional RAID rebuild is that since only blocks that belong to the ZFS tree are traversed, it effectively skips the space that is not currently in use. This means the rebuild time scales in O(used disk space) instead of O(size of the disks).

However, this process may be significantly slower than a typical array rebuild, because the resilvering process is not done linearly in terms of disk offset. Traversing all the blocks of interest in rebuilding involves starting at the top of the ZFS metadata tree (the überblock), down to the object sets (filesystems, snapshots, clones), down to the metadnode (array of dnodes/inodes), indirect blocks and finally data blocks. Since this whole metadata tree is continually being updated with copy-on-write semantics and is not optimized for traversal, the resilvering process will involve lots of disk seeking.

Furthermore, the resilvering process is also sensitive to fragmentation. This is especially a concern for MDTs, due to the way ZAPs (ZFS directories) and the block allocation algorithm work.

2 Requirements

The rebuild performance feature, along with the other Lustre HPCS projects, will contribute to the following HPCS program reliability objectives:

- Maintain T10 DIF or equivalent resiliency.
- Keep 99.97% of the whole Lustre filesystem bandwidth available during a rebuild.
- No impact of rebuilds on file system performance

In addition, the feature will significantly reduce the time and overhead necessary for a rebuild.

3 Functional Specification

3.1 Analysis of current situation

Before describing what needs to be done to improve the rebuild performance, it is important to understand which scenarios are more problematic for rebuilds under the current ZFS implementation.

In a Lustre filesystem, there are two types of targets which present different sets of rebuild performance expectations, MDTs and OSTs.

The majority of the information stored on MDTs are dnodes and ZAPs. Dnodes are the ZFS equivalent of inodes, i.e., they are data structures that describe basic information about an object (file or directory). ZAPs are objects that ZFS-based filesystems can use to map names to values, similar to a hash table. They are used by Lustre to implement directories and other indexes.

Due to the block size and update patterns of these data structures, it is expected that the vast majority (>99%) of the blocks in an MDT to be 16KB or less in size and to be highly fragmented, from the perspective of a thread that is trying to traverse the filesystem (the resilvering thread).

In contrast, OSTs are similar to traditional filesystems, in that they are composed mostly of files with data (the stripes) and all of these files are linked in a directory. A Lustre file system may also maintain additional indexes and some other small objects.

In these targets, it is expected that the amount of metadata in the filesystem to be around 1-3% of the total usage of the filesystem in terms of size. We also expect to be able to find large chunks of data stored sequentially, at least up to 1 MB in size, which is the typical Lustre RPC size. However, fragmentation can also be expected in the directories where the files are linked and in any additional ZAPs/indexes.

So in summary:

Target type	Block size distribution	Long term block layout	Metadata scan I/O pattern	Reconstruction I/O pattern
MDT	>99% of blocks 16KB and smaller	100% randomly distributed	Small random reads, distributed across all devices	Small random I/O, bound to a single top-level device (e.g. a Raid-Z or

				mirrored vdev)
OST	1-3% of blocks 16KB and smaller, 97-99% typically larger (up to 128KB per ZFS block)	Metadata (1-3%) randomly distributed, data (97-99%) usually found in contiguous 1 MB chunks	Small random reads, distributed across all devices	Large I/O (at least up to 1 MB of size per chunk), bound to a single top-level device (e.g. a Raid-Z vdev)

Table 1 – Block size, fragmentation and I/O pattern comparison between MDT and OST

MDTs suffer from two major problems regarding rebuild performance. First, to discover all the block pointers, we need to read almost the entire pool. The only major exception is the data blocks of ZAPs (directories and other indexes), which is considered normal data from the DMU point-of-view. Second, even if we ignore the metadata scan, we will need to effectively do small and completely random I/O to a single top-level device in order to reconstruct a failed disk within it.

For OSTs, the situation is considerably better. The random read workload for the metadata scan is only needed for about 1-3% of the pool and it is distributed across all devices. The reconstruction itself is still bound to the performance of a single top-level device, but it shouldn't be hard to saturate it, considering that we can issue large I/O requests.

Experimental evidence suggests that the rebuild performance for traditional ZFS filesystems (which have similar characteristics to OSTs) can usually saturate the write bandwidth of the replacement disk, even though it can be somewhat sensitive to fragmentation.

3.2 Avoiding unnecessary traversal on MDTs

As mentioned before, the metadata scan phase of the resilvering process needs to traverse the entire metadata tree to discover all the block pointers of the filesystem. This involves reading the entire metadata tree, including all the object sets, indirect blocks of the metadnode, the metadnode itself (which is an object containing a sparse array of the dnodes contained in an object set), and all the indirect blocks of objects.

However, a significant optimization can be made for MDTs. Since most regular

files stored on MDTs don't have any data, they won't also have block pointers in the dnode, and thus we don't actually need to read these dnodes. We still need to read the dnodes of directories and files with extended attributes too large to fit in the dnode, because these objects do contain data in blocks below the dnode. Also, in the future we'd like to store the data of small files in the MDTs as well, so these dnodes must also be read during resilvering.

One minor problem with trying to skip the reading of dnodes is that dnodes are actually stored in groups of 32 on each metadnode block. Thus, if even one directory or file with a large extended attribute happened to be in that metadnode block, we would need to read the entire block.

The latter problem can be solved by biasing the allocation of dnode numbers depending on the object type, so that directories and regular files would be grouped in different parts of the dnode number space. We could also pass a special flag during creation of files that are expected to contain data or large extended attributes, to bias the allocation of these files towards the group that must be scanned during the resilvering.

Another optimization we could apply would be to reallocate a dnode in case a large extended attribute is set, or in case data gets written to the file. Similarly, we could reallocate the dnode in case all large extended attributes are removed or in case the data is truncated. This would allow us to keep these dnodes in the correct dnode number space, such that a single metadnode block containing 32 dnodes always contains dnodes of one type or the other, but never dnodes of both types at the same time.With all of these optimizations in place, we can skip reading a significant percentage of the blocks in an MDT during resilvering (i.e. those containing the dnodes of most regular files).

3.3 New resilvering mechanism

Based on the considerations of section 3.1, it is also necessary to focus on optimizing the reconstruction part of the resilver process. Since the reconstruction is limited to the performance of a single top-level device in the pool, it is very important to make sure that we do as much optimization of the I/O pattern as possible, so that we can saturate the bandwidth of the replacement disk, even on an MDT.

As seen in table 1, currently, for MDTs, this pattern is completely the opposite of what we want – it consists in doing small and completely random I/O.

Therefore, the new resilvering algorithm will be able to transform the random I/Os

of the reconstruction process into a sequential I/O pattern, while still maintaining the same integrity and reliability characteristics of the current ZFS resilvering process.

This should be a significant improvement in performance for an MDT and also an improvement for an OST, especially if there is fragmentation.

3.4 Distributed Hot Space

Another enhancement that is being designed for the ZFS filesystem will be *Distributed Hot Space*. Traditionally, one or more disks extra disks in the storage enclosure are used as hot spares for the increasingly common case that there is a disk failure in the RAID VDEVs, in order to allow parity reconstruction to begin immediately and reduce the window of vulnerability. Unfortunately, these hot spare disks remain completely unused and idle for the majority of time.

With distributed hot space, a fraction of all of the disks in the RAID set making up one (or more) full disk's worth of free space is reserved, and this hot space forms a virtual hot spare disk. This still allows parity reconstruction to begin immediately in case of a disk failure. 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.

4 Use Cases

The following use cases have been considered in developing this design:

1.Rebuild of an MDT disk under normal conditions.

2.Rebuild of an OST disk under normal conditions.

3.Reboot during a rebuild.

4.Additional disk failures during a rebuild.

5 Logic Specification

The following changes will be made to ZFS to fulfill the requirements and improve the performance. These changes are expected to become a standard part of

ZFS.

5.1 Skipping dnodes

In order to skip reading dnodes on MDTs, we will store a bit in the block pointers of the indirect blocks of the metadnode, which will indicate whether that block pointer contains dnodes with data beneath them or not. If this bit is not set, we will know that we won't need to read that part of the metadnode (unless the block itself happens to be stored inside the vdev being resilvered, of course).

5.2 Grouping regular files separately from directories and files with additional data

This can be easily achieved by changing the dnode number allocation procedure to start searching for dnodes in different parts of the dnode number space, depending on the type of the object that we want to allocate.

5.3 New resilvering algorithm

The new resilvering algorithm will consist of three steps:

1.Reserve disk space for the necessary information that we will store. If enough space is not available, we can fall back to the old algorithm.

2.Traversing the metadata tree - For each block pointer that we encounter during the traversal, we will determine if it needs to be reconstructed or not. If so, we will add its offset, size and checksum as a new record to the space map corresponding to the metaslab (a metaslab is a portion of a ZFS vdev) where this block is located. If this particular space map happens to be loaded in memory, we will also keep track of this information in the in-memory AVL tree.

Note that any block that is modified or allocated at any time during the rebuild process will always get written to the new replacement disk, so it does not need to be resilvered. In-place modification doesn't need any special logic to achieve this, because due to COW, ZFS never writes to the same place where a block was – it always allocates a new block. So in fact, we only need to resilver blocks which were allocated prior to the time at which the rebuild started.

Blocks that are freed during the rebuild will simply generate a new record in its corresponding space map, exactly like it's done today, such that when the space map is loaded in memory, we will eliminate the entries for these blocks which don't need to (and shouldn't) be resilvered anymore.

3.Reconstruction - This step consists of iterating through all the metaslabs of the vdev that is being rebuilt, in disk offset order. For each metaslab, we will load its space map into an in-memory AVL tree (if it's not loaded already), ordered by disk offset. The checksums that we discovered in step 2 will also be maintained in the AVL tree. After the space map is loaded, we will iterate through all the blocks in this space map, in disk offset order, and rebuild them.

5.4 Resuming after reboot

Since the threads performing the rebuild will do modifications to the space maps using normal ZFS transactions, and since there is only a very small amount of additional state that we need to keep track of during a rebuild, we can easily resume a resilver after a reboot.

To achieve this, we only need to transactionally store some small amount of state in a special object. This state would include the step that we're on (see the steps in section 5.3), a small zbookmark_t structure describing which block we're currently traversing (in case we're in step 2), and optionally an offset within a vdev, in case we were interrupted during step 3 and we don't want to re-read some of the space maps.

5.5 Handling multiple disk failures

To support multiple simultaneous rebuilds, the rebuild process can be adapted to a producer/consumer model, where the producer is the thread that performs step 2 of the rebuild algorithm, I.e., who produces the checksum records and stores them in the appropriate space maps, and the consumer is the thread that reads these records and reconstructs the vdev.

This allows us to have multiple threads doing different jobs but working in parallel, and also allows us to cleanly and efficiently handle cases where there are multiple vdevs with failing disks.

For the case where a second (or more) disk in the same vdev needs to be rebuilt at the same time (e.g. for RAID-Z2 vdevs, which are the ZFS equivalent to RAID-6), if the rebuild algorithm is currently still in the metadata scan phase, then there are no changes, as the thread doing the rebuild of that vdev will simply rebuild all the disks at the same time once the metadata scan is complete.

However, if the rebuild process is already in the reconstruction phase when another disk needs to be rebuilt, then we have two options:

1) Stop the current reconstruction thread, request another metadata scan and restart the rebuild of that vdev from the beginning. This is the easiest but the less desirable option, because the rebuild of the previous disk(s) will take much longer to finish.

2) Request another metadata scan, but keep rebuilding the previous disk(s).

   In this case, the thread that handles the rebuild of that vdev could keep track of a small persistent record containing a disk offset for each failing disk within that vdev, indicating at which offset the rebuild process started on that particular disk. This would allow the thread to continue to make progress in the rebuild of the previous disk(s) and also partially rebuild the disk that just failed at the same time. When the rebuild of the previous disk(s) is finished, the thread would wait for the requested metadata scan to finish, and then it could simply rebuild the remaining blocks, i.e., those from offset 0 until the offset at which the rebuild process started for that newly-failed disk.

Furthermore, it is also worth noting that given a sufficiently large ZFS pool, with hundreds or even thousands of disks, it wouldn't be inconceivable to need the producer thread running 100% of the time to handle permanently on-going disk rebuilds in different parts of the pool.

5.6 Distributed Hot Space

D00	D01	D02	D03	P04	Q05
D10	D11	D12	P13	Q14	D15
D20	D21	P22	Q23	D24	D25
D30	P31	Q32	D33	D34	D35
SPARE	SPARE	SPARE		SPARE	SPARE

VDEV 1

D0a	D0b	D0c	D0d	P0e	Q0f
D1a	D1b	D1c	P1d	Q1e	D1f
D2a	D2b	P2c	Q2d	D2e	D2f
D3a	P3b	Q3c	D3d	D3e	D3f
D03'	P13'	Q23'	D33'	PS3	QS3

VDEV 2

Figure 1: Hot space for dual parity RAID 4+2 configuration

In this example, disk 3 of VDEV 1 has failed, and the RAID parity reconstruction is done for its data and parity blocks into the hot space of VDEV 2. Note that the hot space of VDEV 2 itself can be configured with double RAID parity so that if VDEV 2 also had a disk failure then its hot space will maintain data integrity. In the case of a concurrent disk failure in VDEV 2 the hot space in VDEV 1 would be used similarly, though with only a single parity. Each VDEV could still survive an additional failure without data loss after RAID reconstruction had completed, and one of the two VDEVs could handle a third failure though it would leave both devices without any parity protection.

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.

In the hypothetical 12-disk configuration from Figure 1, if we instead needed to have 2 hot spares available for immediate reconstruction the VDEVs would have been configured as RAID-6 3+2, reducing the available IO bandwidth by 25% for the majority of the time compared to using 20% of each disk for hot space.

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.

6 State Management

6.1 Scalability & Performance

In theory, the performance of the metadata scan in step 2 of the resilvering algorithm could scale in O(number_of_allocated_blocks / total_available_IOPS_in_the_pool), assuming the I/O load of the metadata scan is sufficiently parallelized.

Therefore, as ZFS pools grow, there should be a relatively linear relationship between amount of information in the pool and the number of IOPS available, in the sense that as new disks are added to achieve the capacity requirements, the IOPS performance of the pool increases by the same factor. In other words, it's not expected for the metadata scan to significantly degrade as ZFS pools grow

larger.

In practice, it should be noted that as of build 102 of the OpenSolaris development code (where ZFS is maintained), the metadata scan is not fully parallelized. However, a limited degree of parallelization of this process is achieved by issuing in advance a number of prefetch requests for blocks that are going to be read soon. In the future, this can be further improved on, if needed.The amount of data needed to be stored in the space maps should be around 42 bytes per block, this is only needed for blocks in the vdev where the rebuild is happening, of course. Furthermore, since the space map update procedure consists solely of appending records, and since these are distributed across all the devices and can be coalesced with other space map updates, using the fully asynchronous ZFS transactional I/O pipeline, with writeback caching safely enabled on all disks, these writes are not expected to have a significant performance impact.

The amount of additional data that needs to be stored in-memory during step 3, when the space maps are loaded and ordered, should be around 32 MB per space map using current disk sizes. We can arbitrarily increase or decrease this by controlling how many metaslabs we split the device into. Since only very few space maps need to be loaded at a time, this is not expected to be a problem.

The performance of the reconstruction phase described in step 3 of the new resilvering algorithm is expected to have an optimal I/O pattern, since we will be able to do fully sequential I/O and we'll be able to coalesce reads and writes in the I/O pipeline. Under these conditions it shouldn't be hard to saturate the write bandwidth of the replacement disk.

It is also important to note that the rebuild process can be done as slowly as needed, in order not to impact normal filesystem performance, as long as we make sure that the resilience expectations are met. Since ZFS has it's own I/O scheduler, the speed of this process can be easily controlled in a dynamic fashion. This may be done either by directly adjusting the IO rates for the pool being rebuilt or by placing rebuild IO at the bottom of the existion ZFS IO priority list. The only overall performance concern is that the rebuild process should not be made so slow as to have a high probability of having sufficient disks fail to cause loss of data.

6.2 Recovery Changes

This task does not affect recovery in any way.

6.3 Locking Changes

Some care needs to be taken that simultaneous updates to the space maps (e.g. by the resilvering thread and by another thread that tries to free the block) do not conflict. There will also need to be care taken that new block allocations by normal filesystem I/O during step 2 or 3 of the resilvering algorithm also append the new checksum record to the space map.

6.4 Disk Format Changes

A bit in the “fill count” field of the block pointer will be reserved for keeping track of whether dnodes with data are stored underneath this block pointer in the metadata tree. This bit will only be valid for indirect blocks of the metadnode.

A new record type will be added to space maps.

For distributed hot space there would need to be a new on-disk layout that included segregating the hot space from the regular VDEV space.

6.5 Wire Format Changes

This project does not introduce any wire format changes.

6.6 Protocol Changes

This project does not introduce any protocol changes.

6.7 API Changes

The project does not introduce any API changes.

6.8 RPC Order Changes

This project does not introduce any RPC order changes.

7 Alternatives

There were quite a few alternative options considered during the design of this task.

It would be too long to describe all of them along with their advantages and disadvantages, but for reference here are a few of them and why they were

eventually discarded.

7.1 Using per-VDEV lists of checksums or even block pointers.

This would be similar to the current proposal, except that ZFS would always keep track of the checksum records in the space maps, even when no rebuild is happening. It would have the advantage of eliminating the metadata scan phase, but according to discussion with the ZFS team, a very similar prototype feature was already tried in the past and it had unacceptable performance.

7.2 Using a disk offset-ordered B-Tree instead of space map records to store the checksums.

This would be doable but considerably more complicated and less efficient.

7.3 RAID-Z + Traditional parity declustering.

This would be a a VDEV that would implement RAID-Z on top of a fixed geometry parity declustering layout. This presented the problem of being unable to guarantee transactional integrity due to partial stripe writes requiring read-modify-write.

7.4 Traditional Parity Declustering + Checksums.

A VDEV that would be almost identical to a traditional software parity declustering VDEV, except that we also store (one or more) sectors with checksums adjacent to the parity block. Has the same read-modify-write problem as above.

7.5 RAID-Z style Parity Declustering.

This would consist in doing a simple modification to make sure that Raid-Z only stripes across a fixed number of disks, thus achieving many of the performance advantages of traditional parity declustering devices. It seems to be unimplementable, due to consequent geometry problems that would arise given the design of ZFS.