Lustre Enhancements - Technical White Paper

Using Interval Tree to scale extent locks

Author	Date	Description of Document Change	Client Approval	Client Approval Date
Jay Xiong	Mar 24, 2008	Create the document		
Bryon Neitzel	Mar 29, 2008	Review and edits		
Nirant Puntambekar	Apr 1, 2008	Review and updates		
Jessica Johnson	Apr 1, 2008	Review.		
Jay Xiong	Apr 7, 2008	Revised per comments		
Jay Xiong	Apr 16, 2008	Clarify and update doc.		

Problem Statement

When a shared file is being accessed by multiple clients, the clients need to grab an extent lock from the OST's (Object Storage Target's), to prevent them from writing to the same portion of the file at the same time, which would lead to data corruption. Also when the shared file is being accessed by thousands of client simultaneously, the time to access it may increase dramatically,since there could potentially be millions of extent locks on the OST side, and the new file access requests first must check for any conflicting locks.

In the original implementation, the OST used a linked list to manage all the extent locks, and traversed it sequentially to check if a conflicting lock overlapped the requested region. The delay in conflict detection could cause an apparent freeze of the OST.

Approach

Based on the above analysis, the conflict detection process requires a more efficient mechanism to accelerate finding the overlapping extent locks.

An interval tree algorithm was chosen to manage the extent locks on the OST side. An interval tree is an enhanced version of a RB-tree. Compared to a RB-tree node, the interval tree node contains a region parameter, to indicate how long this node spans and a number called max high, which indicates the maximum region that the sub-tree of this node may cover.

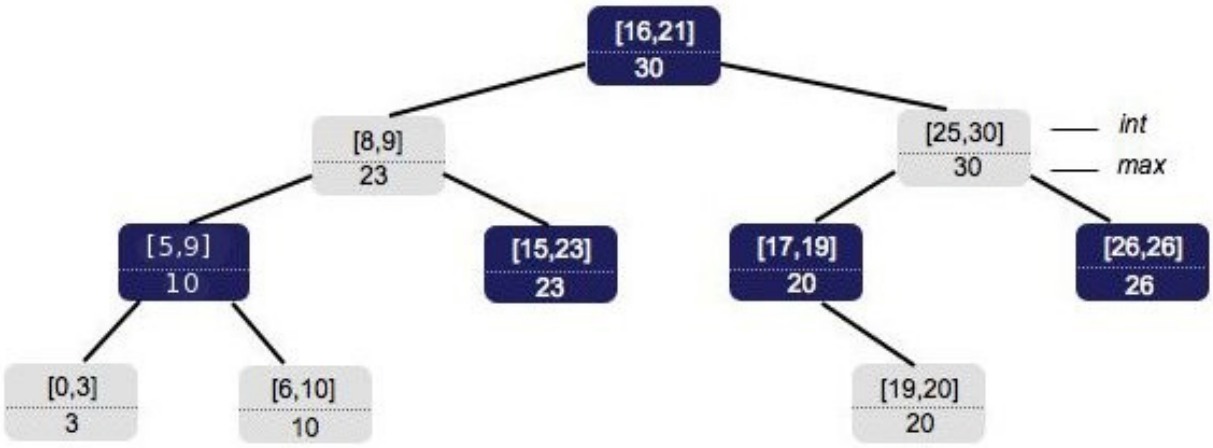

Figure 1: Interval tree

For the above figure, the root node of the interval tree is [16,21]:30, which means the extent lock represented by this node covers the bytes 16 to 21 for the file and the max high is 30, which is the maximum position of its sub-node's reach. When a new extent lock query comes in from the client, it will only recursively check which sub-tree overlaps the requested region of this extent. The search time can hence be reduced to O(logn), where n is the total number of nodes(extent locks).

Test cases and Results

Two types of tests were performed.

The first was a unit test which compared the search performance of the interval tree vs a list implementation. We did this with randomly generated nodes. The test results are as shown in Table 1.

Request size(bytes)	List search(ms)	Interval search(ms)	Contended locks
4K	*166*	*0*	*234*
128K	*168*	*0*	*318*
1M	*167*	*0*	*684*
16M	*164*	*1*	*7,871*
64M	*170*	*6*	*31,259*
256M	*165*	*26*	*125,185*
1G	*171*	*103*	*500,791*
2G	*169*	*57*	*276,642*
3G	*170*	*205*	*952,993*
4G	*166*	*207*	*1,000,000*

Table1: Unit test results

We preset different size regions of 1M in the interval tree, then produced a request region with a specific width. The interval tree was then searched to get all the regions which overlapped the requested region. After the search was completed, the time and number of locks which overlapped were reported.

In most cases (when the request size was less than 2G in a 4G file), the interval tree wins. However, if most of the 1M regions are overlapped by the requested region, the list search wins because of the additional overhead of tree management, which leads to some nodes being accessed multiple times. This might not be a common case in reality, as a 1G bytes write to a file should be relatively rare.

Another test was performed at LLNL (Lawrence Livermore National Laboratory) to take advantage of their large clusters. LLNL ran a test case where 9600 tasks read a shared file, 1K bytes per read, via 400 clients on a Lustre 1.6.2 system. The hardware configuration used was

- 24 servers, 3 OSTs per OSS
- Quad x86_64 CPU 2.80GHz
- 4GB of RAM
- 2 GigE Links
- 1 FC2 connection directly attached to DDN 8500 storage.

The test results are as below.

1.6.2 - *No soft lockup patches (Lustre 1.6.2)*
1.6.2+ IntTree - *With Interval Tree Patch*
1.6.2+ IntTree + LockGrant - *With Interval Tree and Lock Grants Patches*

Rounds	1.6.2	1.6.2+ *IntTree*	1.6.2+ *IntTree + LockGrant*
1	*33.533s*	*17.861s*	*8.598s*
2	*22.370s*	*16.385s*	*10.756s*
3	*34.979s*	*17.295s*	*6.785s*
4	*32.446s*	*18.022s*	*10.759s*
5	*34.530s*	*13.884s*	*7.710s*
6	*32.768s*	*17.058s*	*7.971s*
7	*34.669s*	*17.333s*	*6.777s*
8	*32.337s*	*13.227s*	*7.719s*
9	*31.703s*	*13.244s*	*7.813s*
Average	*31.482s*	*16.155s*	*8.215s*

Table2: System test results

The tasks were run for 9 rounds and the results were averaged to get more accurate performance data. Because the extent locks are all read locks, as part of another bug, a Lustre engineer came up with a solution which accelerated the lock grants. Though this is specialized to read locks, it is listed here because of its efficiency in decreasing the access time.

From the test results, the interval tree decreased the access time by about 50% with further gains from the lock grants patch. Performance should also improve as the number of locks increase.

Conclusions & Future work

Interval tree search is functional and seems to provide the expected performance benefits. However in the current implementation, the search time is not bound to O(logn), additional work could be performed to improve the existing algorithms.