Minimizing Lustre Ping Effects at Scale on Cray Systems

Cory Spitz,
I/O File Systems
Cray Inc.
St. Paul, MN
spitzcor@cray.com

Nic Henke, Doug Petesch, and Joe Glenski
Cray Inc.
St. Paul, MN
dpetesch@cray.com, glenski@cray.com, nic@cray.com

***Abstract*—Cray is committed to pushing the boundaries of scale of its deployed Lustre file systems, in terms of both client count and the number of Lustre server targets. However, scaling Lustre to such great heights presents a particular problem with the Lustre pinger, especially with routed LNET configurations used on so-called external Lustre file systems. There is an even greater concern for LNETs with finely grained routing. The routing of small messages must be improved otherwise Lustre pings have the potential to 'choke out' real bulk I/O, an effect we call 'dead time'. Pings also contribute to OS jitter so it is important to minimize their impact even if a scale threshold has not been met that disrupts real I/O. Moreover, the Lustre idle pings are an issue even for very busy systems because each client must ping every target. This paper will discuss the techniques used to illustrate the problem and best practices for avoiding the effects of Lustre pings.**

Keywords-Lustre; LNET; FGR; jitter; noise; I/O; scaling

I. INTRODUCTION

This paper assumes that the reader is familiar with basic Lustre concepts. However, enough background should be given such that the reader can follow the scaling issues. For more background on Lustre, please refer to the Lustre Operations Manual.

Over the course of a performance investigation, we determined that Lustre pings were the root cause of a serious I/O throughput issue, especially and primarily for routed LNETs.

For a variety of reasons, it is not easy to change the basic Lustre ping implementation, so we focused our efforts on tuning. It was soon clear that there was much ground to be gained by improving LNET performance. We determined that many changes were needed to accommodate efficient routing of small messages on Infiniband fabrics. Because of the peer credit concept in LNET routing, special tuning effort is required for configurations that use Fine Grained Routing (FGR) [1]. However, we discovered that due to limitations of the Infiniband LNET Network Driver (LND), o2iblnd, there is a limit to the tuning one can make.

Once we made improvements to the LNET, our focus turned again to changing the ping behavior. However, since every client node sets timeouts for ping sends we must be careful so that we do not inject additional OS noise and contribute to jitter.

II. BACKGROUND ON LUSTRE PINGS

Lustre uses an 'obd_ping' and each ping targets an 'obd_import', but essentially clients ping all metadata and object targets in the filesystem. The pings serve three purposes. First, a ping is used to detect server health (and that has a side effect of triggering clients to reinitiate connections to targets). Second, the server uses the lack of pings (or other traffic) to infer client health. Finally, servers add the *last_committed* value to ping responses, which is needed for clients using the asynchronous journal commit feature.

Unfortunately, Lustre pings do not scale well. Because each client pings each target, we have to deal with O(n*m) scaling. It is not smart enough to consolidate pings when there are multiple targets per server, which is very typical. For current petascale sized systems with component counts on the order of 25,000 nodes and hundreds of OSTs there can be tens of millions of pings per ping interval. With FGR configurations that Cray is exploring and deploying, the scale we are seeing is about 100,000 pings per OSS and 75,000 pings per router.

Therefore, for even moderately sized systems, there will be an enormous number of pings sent every ping interval. The scheduling of those pings is aligned to minimize the effect of OS jitter. That in turn creates a ping 'flood' since the pings are transmitted simultaneously, which leads to an acute issue for bulk data transmission.

The ping flood can lead to so-called 'dead time' where bulk I/O processing essentially stops because the system is choked up with a high volume of concurrent ping messages. The 'dead time' occurs every *PING_INTERVAL* seconds, which is the default idle time allotted before pings are transmitted, which has been configured to 75 seconds on all Cray systems. In very badly configured system, the 'dead time' can account for up to ten percent of the potential I/O bandwidth.

Unfortunately, Lustre pings cannot easily be removed. Not only would it be difficult to replace the functionality, but also Cray would have to find a general solution in Lustre for the community. Since the ping serves purposes that are inherently associated with health data, it can be tempting to use Cray's proprietary HSS network and its RAS facilities. However, this cannot extend to external components as are the bulk of Cray's future Lustre product offerings.

The pings are also a source of noise that leads to OS jitter. Aligning the pings in time has the effect of ganging up on routers and servers, so we could spread them out over time. However, that would naturally have consequences for jitter. Jitter is notoriously tough to measure, so careful consideration should be made for any changes that impact OS noise.

Note that the Lustre 'obd_ping' should not be confused with an LNET ping. LNET pings are LNET/LND level pings that are used to check LNET connectivity.

III. Discovering Dead Time

When benchmarking distributed filesystems such as Lustre, IOR is a natural benchmark to use because it is an industry standard and it can drive I/O exactly as many applications do. For example, it can be configured to read/write single shared files, use a file per process, use direct I/O, etc.

Whether IOR is used in a fixed data or fixed time format, a single numerical result is output: the average throughput for the run. For fixed data runs (the default), the run cannot complete until the slowest component has finished. Sometimes there are head-of-line blocking issues or disk block layout issues that impede progress of some processes. If those issues or other transient behavior were to occur during a run, the result would simply be lower average throughput measured. In other words, IOR can only be as fast as the slowest element and often the sustained rate is faster than the reported rate.

Cray benchmarking was investigating just such a case on a large-scale system where the measured IOR rates were not meeting expectations. Individual components were benchmarked and performance was measured. Results seemed good. Clearly, there was some effect occurring during runtime. We knew that the performance was often good, but we were missing why the rated IOR performance was not closer in value to those that we expected based on the component testing. At this point, using IOR alone is not useful for further debugging because we cannot expect rates to be as simple as in Fig. 1 and IOR does not give you the real-time view of progress. In addition, backend-monitoring tools like collectl or LMT cannot tie the data that they collect to individual jobs or even processes. Therefore, the

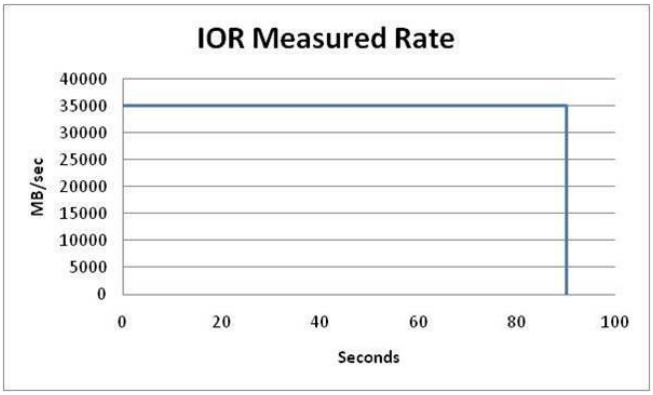

Fig. 1 Idealized IOR rate

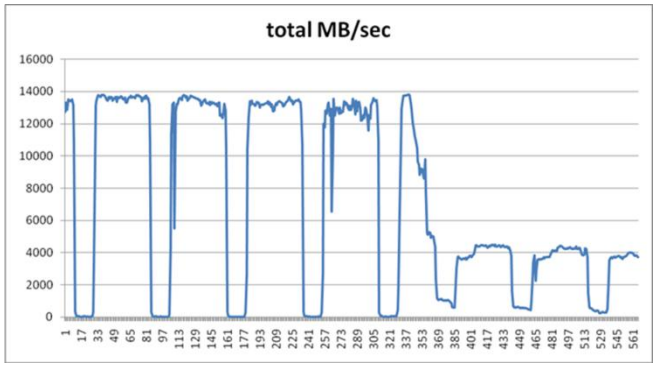

Fig 2. Instrumented IOR shows 'dead time' occurs every 75s

performance benchmarking team at Cray instrumented IOR to sample each processes' I/O progress.

With IOR instrumented to collect rates at sub-second intervals on a per process basis, we can begin to investigate the runtime fluctuations. The insight that we can draw here is obvious. Fig. 2 shows that something is clearly wrong.

It is now apparent that something is causing all outstanding I/O to stop on a regular interval. By measuring the period, we see that it is always 75 seconds. Experienced Lustre users could immediately recognize the cause as the Lustre pinger because the period was exactly the PING_INTERVAL[1]. However, even if we had a strong hunch as to the cause, we must verify it.

IV. Investigating Deadtime

To help us investigate the dynamics of the dead time, we found it helpful to emplace some telemetry to pull LNET stats for visualization. For this, we added an OSS and LNET router plug-in to collectl because collectl data can be easily visualized using Graphite[2]. The following charts of various LNET and LND stats shown here were generated from Graphite and all show a seven-second window into a dead time interval during an IOR write.

A. Determining the cause of deadtime

At this point, we used the Lustre llstat facility to gather server side stats. Upon examining the information, it is clear that the OSSes nearly stop all ost_{write,read} activity in the ost_io threads. We note the time of a 'dead time' incident and look at the remaining Lustre and LNET stats to see what other activity occurs then. Now we see that ost threads begin to process a very large number of pings. Fig. 3 very clearly shows the relationship.

Digging further into the data, we see that the number of I/O requests completed by the servers slows to a trickle (as also seen in Fig. 3). The waittime for ost_io I/O operations is constant. Further, we see that neither the OSS nor LNET

[1] The default Lustre PING_INTERVAL is obd_timeout/4 or 25 seconds since obd_timeout defaults to 100 seconds. However, Cray Lustre modifies obd_timeout to 300 seconds, so the PING_INTERVAL becomes 75 seconds.

[2] Cray will review the OSS and LNET collect plug-ins for possible distribution.

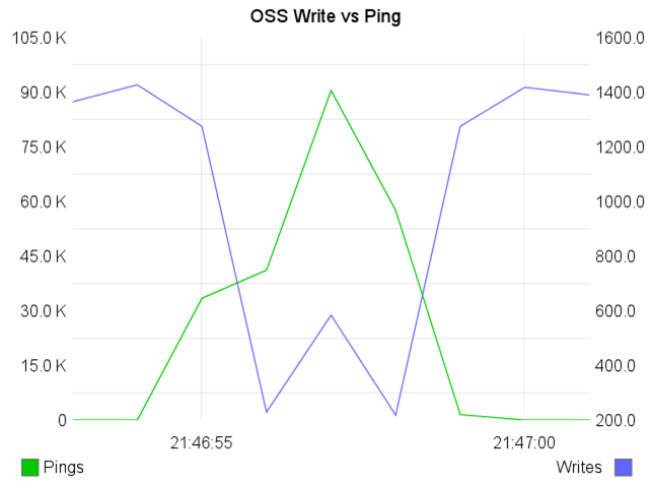

Fig 3. OSS writes and pings during dead time

routers are CPU bound. The huge volume of pings means that with fair distribution, bulk messages will make up only about one percent of the message traffic during the ping floods. Therefore, it appears that we cannot saturate the available disk bandwidth because the pings choke out the bulk data requests. It is simply the case that we cannot deliver enough messages to the OSS for processing.

B. Why do pings choke regular I/O?

We quickly turn to investigating the LNET routers to see why we cannot get the ping messages through efficiently. Is the problem with either the gnilnd, o2iblnd, or the router functionality itself?

This is actually a hard question to answer because the available LNET stats data cannot necessarily paint a complete picture. For example, there is lack of directional information for use of the router buffers and there are not waitqueue times available to indicate how long messages are sitting in the buffers.

However, LNET does provide information about credit usage including whether they are oversubscribed and by how much. At this point, we determine that we are lacking in peer_credits on the o2iblnd side of the routers. The 'min' column in /proc/sys/lnet/peers will show the high water mark for credit usage. The absolute value of negative values are by how many credits are oversubscribed. Fig. 4 shows that we require well above the maximum of 255 peer_credits that the o2iblnd wire protocol can provide. In some configurations, we see that we could use an additional 10K credits! Clearly, we will need to investigate tuning up peer_credits. Nevertheless, even after allowing more traffic to flow to a specific host, we still see that we become dramatically oversubscribed on peer_credits. In fact, the peer_credit usage in Fig 4 is with peer_credits configured well beyond the default of eight to 126.

C. Understanding LNET message flow through routers

Since we do not know how long messages are waiting for buffers or how long they sit in buffers we had to rely on our collectl telemetry from the existing LNET stats to understand the message queuing model. We needed to be sure that the o2iblnd peer_credit consumption was more than an instantaneous artifact and if there were other problems with routing. The high and low water marks available from LNET stats cannot provide that kind of information.

From the charts generated from the data, we can begin to see how traffic is delayed. For example, Fig. 5 shows the small_router_buffer usage and the o2iblnd peer credit usage during a ping storm (since pings are ~200 bytes they use the small buffers, which are sized at four KiB). The curves appear to be highly correlated, but we should first understand the client to server traffic flow to assign meaning.

On Cray clients, messages originate from the gnilnd. The gnilnd will acquire an interface (called ni, for network interface) tx credit and peer credit and send directly to a router. The Gemini hardware allows the gnilnd to release the credits as soon as the message is in remote memory (without software intervention.

On the routers, there is a lot of activity as LNET shuttles messages from one interface to another. First, gnilnd will try to acquire a router buffer and a peer router buffer credit. If either fail, it will do an 'eager receive' and perform a memory copy to free up the Gemini HW mailbox resource. Once it can get all the credits and buffers it needs, it will copy the message into a router buffer. At this point, the o2iblnd takes responsibility for moving it out the IB interface, but the router buffer and peer router buffer credits are still used. The o2iblnd has to acquire two more credits, peer and ni tx credits just as the gnilnd did, before it can transmit the message further. If credit gathering fails, the transmission is queued until the needed credits are available. The router buffer and peer router buffer credit cannot be freed until the tx credit is returned from the next peer (server). Moreover, there is a cost for memory registration for RDMA that is avoided with a memory copy. The o2iblnd developers had determined that nothing under four KiB was worth incurring that registration cost. So small messages such as obd_pings are copied instead of RDMA'd.

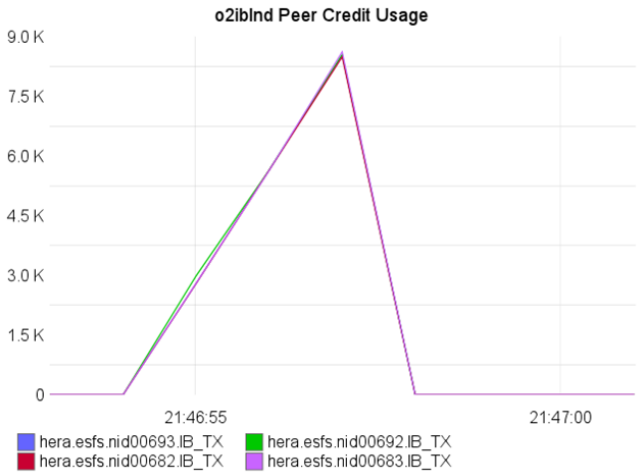

Fig 4. Oversubscribed o2iblnd peer_credits

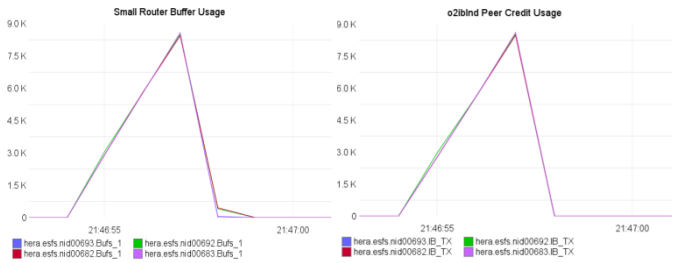

Fig 5. Router buffers (L) and o2iblnd peer_credit (R)

That memory copy has a cost of its own, which can add additional latency for transmission.

Finally, on servers, LNET will not receive a message until it finds a service buffer in the ptlrpc layer to copy the message into. Otherwise, it must queue up the message and wait for buffers to be posted. Only after this copy can o2iblnd tx credits be returned, which happens via an explicit SW message. This explicit message means that resource consumption on the server will cause backups and queuing into the routers.

Now we can understand that moving messages through the routers means that we may not only need more peer_credits on o2iblnd, but also enough router buffers and peer router buffer credits because they can be consumed for a relatively long time. Fig. 6 shows the Gemini receive side of the routers. We can see that there are many router buffers consumed on the gnilnd side during the dead time. The fact that these queue quickly indicates to us that clients do not have difficulty injecting messages into the routers. However, we do have a problem freeing them since the number of consumed credits continues to climb during the dead time. We know that these cannot be freed until the message has been completely forwarded to the server and the o2iblnd tx credit has been returned. Then in considering Fig. 5 as well, in this case where the router buffer usage practically mirrors the o2iblnd peer credit usage, it indicates that the gnilnd is

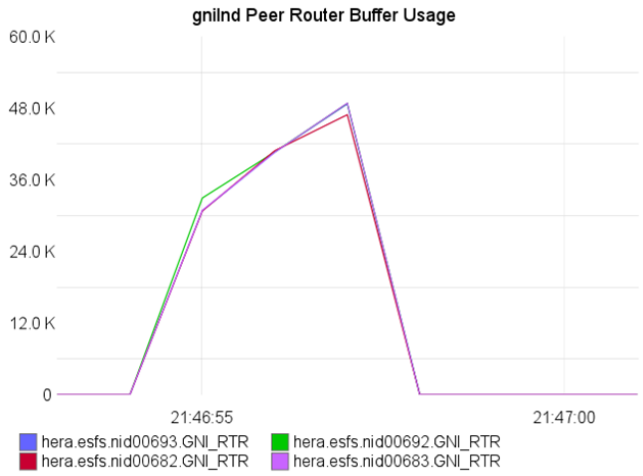

Fig. 6 gnilnd peer router buffer consumption (receive)

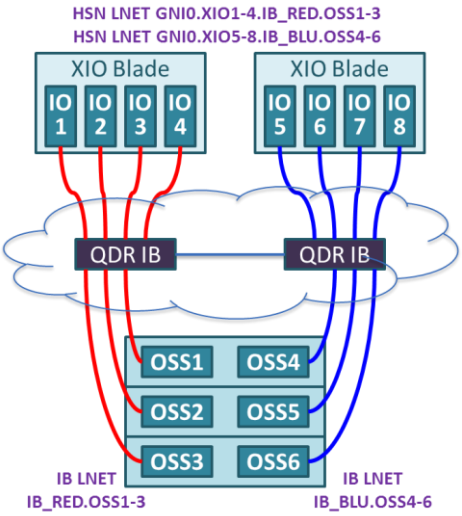

Fig. 7 Example of FGR routes

not a bottleneck for message flow through routers and that messages are queuing on o2iblnd peer_credits. The peer credits are oversubscribed and only as we free tx and peers for o2iblnd can we begin to achieve higher message throughput through LNET routers from Cray clients.

V. FGR WITH INFINIBAND

Fig. 7 shows an example of an LNET layer that is finely routed. If this were a flat network, Cray compute nodes would round robin messages among all eight routers, which will create crosstalk in both the HSN and IB fabrics. Cross talk in the IB fabric can be very expensive if inter-switch links (ISLs) become over utilized. In reality, using flat LNET networks means that most traffic travels suboptimal paths. Use of FGR is designed to use the optimal paths at all times. With FGR in our example, a compute node will use only the routers on the red network to communicate with OSSes 1, 3, and 5 and the blue network to communicate with OSSes 2, 4, and 6, avoiding the inter-switch link.

FGR presents a special problem for LNET because it was designed mostly with flat networks in mind. As discussed, there are per-interface and per-peer credits consumed for message transmission (and credits and per-peer credits for router buffers too). Per-peer credits exist so that no one peer can monopolize all of the resources [2]. Its drawback is that an interface can consume all of the available credits for a given endpoint without fully utilizing all of the network resources. That is, without saturating the network. Yet this has purpose because the credit mechanism can be a tool that administrators use to keep Lustre clients from overwhelming a server. In other words, sometimes throttling network performance is desirable.

It is rarely a problem in flat LNET networks to under-saturate the network because there are many possible endpoints or destinations. With FGR, that set of endpoints is dramatically reduced and the default peer_credits setting will limit the overall number of messages that can be inserted onto the wire. Although there is little to no contention or crosstalk on the fabric, one must pay careful attention to tuning up the per peer credit values or the performance will

be less than what can be achieved with a flat network. This is particularly important when routing to Infiniband networks because the wire protocol is limited to very few credits. This is not a problem at all for large writes (Lustre typically transmits data in one MiB chunks) because a few number of messages can consume all of the available bandwidth.

VI. Tuning LNET

A. o2iblnd tuning hints

Our main problem is that routers are simply not able to get enough small messages in flight to either keep up with the demand or saturate the IB link and that is mainly due to the lack of peer credits. As already stated, the drawback with the o2iblnd LND is the small limit for peer_credits and that they are consumed until there is an explicit SW release message. The maximum value that can be achieved without a wire protocol change is 255. There are some other rules and guides to follow that will further limit the maximum value.

First, peer_credits must be less than or equal to twice the value of the concurrent_sends tunable. Unfortunately, another esoteric feature, map_on_demand, needs to be configured to allow concurrent_sends to be greater than 63. The map_on_demand tunable has side effects for bulk RDMA transmission, so it is best left un-configured.

Therefore, the best and highest value we can reach is to set concurrent_sends to 63 and peer_credits to 126, which is done by adding the following to modprobe.conf:

```
options ko2iblnd peer_credits=126 concurrent_sends=63
```

Since we are not able to push peer_credits as high as we would like, these recommendations hold for both flat and FGR LNETs.

Note that peer_credits must agree across all o2iblnd peers! That means that any changes made must be instantiated simultaneously across the common fabric peers.

B. Router tuning hints

For routers, we need to ensure that we have reserved enough router buffers and peer router buffer credits for each interface. Fortunately, router buffers just consume memory, so we are free to add many more. Since LNET routers run on nodes that do not provide other services we should feel free to consume a lot of memory.

Especially for the four KiB small_router_buffers, the default number of credits is massively undersized. For large Cray systems, the client side should dominate since there are many more clients than servers. The gnilnd defaults to 16 peer router buffer credits.[3] Therefore, the maximum number of small router buffers that could be consumed by the gnilnd side by default is (num_clients * 16). We are recommending 16K buffers, which at four KiB per buffer only consumes 64 MiB. LNET stats make it easy to see if that is enough. After some production workload, check /proc/sys/lnet/buffers. A negative 'min' value gives the high water mark for how many credits the router was oversubscribed. If so, simply increase the number of router buffers since only the amount of memory consumed will be limiting.

For the IB side of the router, the number of o2iblnd peer router buffer credits can be tuned by configuring peer_buffer_credits. Since we recommend tuning o2iblnd peer_credits to 126 servers cannot ever land more messages than that into the router buffers. Therefore, we should choose to set peer_buffer_credits to 126 as well because with the client side dominating the number of router buffers, there is no reason to limit the number consumed on the IB side of the router.

Therefore, the best recommendations to be included into the router modprobe.conf would be:

```
options ko2iblnd peer_credits=126 concurrent_sends=63
options ko2iblnd peer_buffer_credits=126
options lnet small_router_buffers=16384
```

Remember that peer_credits must agree across all o2iblnd peers! That means that any changes made must be instantiated simultaneously across the common fabric peers.

C. Results with LNET tuning

Tuning credits does reduce the measured dead time. However, we conducted some scaling tests by leveraging 'fake scaling' where we simply mount file systems multiple times on each client. Because the ping algorithm is not very smart, it will generate additional ping load to each target. After tuning, we generated about 45 million pings per PING_INTERVAL. Fig. 8 shows that the dead time is still present, but it is drastically reduced.

VII. Pings and OS Jitter

It seems obvious that we could also eliminate the dead time by spreading the Lustre pings out in time so that their effect was not felt so acutely. However, pings are aligned in the first place to minimize jitter. We must be careful not to trade one problem for a more severe one.

We had an opportunity to explore the ping effects on noise when ORNL reported that global barrier times had increased immensely after upgrading to CLE 3.1 from CLE

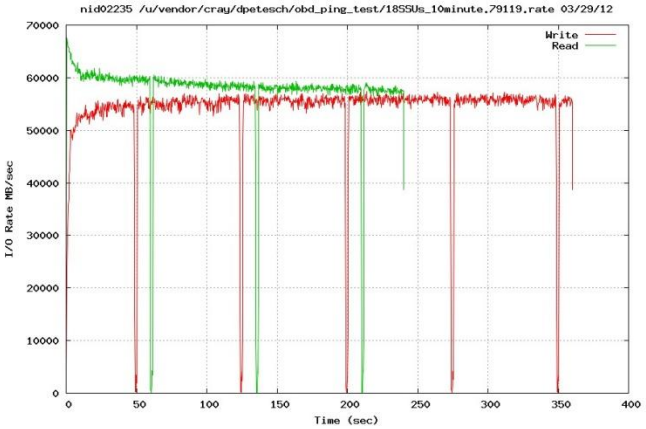

Fig. 8 Dead time is reduced but still present at scale

[3] The gnilnd does not tune peer router buffers separately as the o2iblnd can. The number of peer router buffers configured is based on the peer_credits, which defaults to 16. Otherwise, it can be configured by enabling the lnet module parameter, peer_buffer_credits.

2.2. CLE 2.2 uses Lustre 1.6.5 and CLE 3.1 used 1.8.2. See Bug 771347 for details. It was also determined that the problem did not manifest unless the Spider file system was mounted. Moreover, the problem was experienced when the file system was not in use. Between 1.6.x and 1.8.x, the obd_ping algorithm changed to more strongly align pings to the current time modulo PING_INTERVAL boundary. In 1.6.x, pings were coalesced into a single interrupt, but the pings could offset based on the idle period. In addition, the 1.8.x algorithm added a callback facility to run functions when the ping timer expired. Currently the only use of the callback facility is the async journal commit feature. If these functions were long running, then perhaps they were the source of the new jitter. Since the async journal commit feature could be avoided by avoiding buffered writes, we could try undoing the change.

We created three experimental Lustre clients to determine if we could isolate one of the changes as the source of the degradation. The combinations were aligned pings with callbacks (default 1.8.x behavior), aligned pings without callbacks, unaligned pings with callbacks, and unaligned pings without callbacks (the default 1.6.x behavior).

Unfortunately, OS jitter is a hard thing to measure and even harder to correct. If we improved intra-node jitter, then we do not necessarily improve inter-node jitter as all the interrupts on CPUs within a node could occur at the same time, but at different times compared to other nodes. However, it does seem clear that if we degrade intra-node jitter, then inter-node jitter would also have to degrade. We are interested in inter-node jitter because that is what leads to large barrier times.

Jitter can be counter intuitive though. It is not necessarily a win to strongly align interrupts within nodes if those interrupts are going to take a long or variable amount of time (compared to more frequent, shorter, deterministic interrupts). Further, some applications are sensitive to inter-node jitter, while others are sensitive to intra-node jitter.

Our experiments showed that the journal commit callbacks were not a significant source of intra-node noise. Therefore, we must conclude that the strong alignment of pings was the source of the degradation in system wide barrier performance. However, Cray's limited scale testing (after all Cray does not own a Jaguar sized system) was inconclusive. We saw that there was a very small difference in barrier performance between the options. We are looking to get access to a larger system for further investigation. In the meantime, we will not be making changes to ping timing despite the reports of degradation because of the perceived value of aligning pings.

Clearly, it would be simplest to remove pings altogether so as to solve the jitter problem and dead time problem at once.

VIII. Conclusion

The Lustre ping problem is quite complex and the problem is compounded on modern Cray system with routed Lustre file systems with FGR. LNET tuning in IB is needed to reduce the dead time. The recommendations provided here for LNET tuning will reduce the dead time spent handling idle Lustre pings.

IX. Future Work

Besides further investigation of the 1.6.x style ping timer, there are other ideas that the authors would like to explore. First, removing the extra memory copies on routers is worth examining because it could have benefits for other small message traffic like metadata loads.

There are some stopgap measures to explore before outright removal of the pings. We will continue to explore the issues surrounding ping timers and jitter. Next, we could look into reducing the number of pings, both in scale and in frequency. If we lower the frequency, we still suffer from dead time, just not as often, but that has side effects for inferring server and client health.

However, clearly the biggest payoff would be to eliminate pings altogether. RIKEN and Fujitsu have removed some pings on the K computer through a custom solution detecting node health out of band from Lustre [3]. They plan to release these changes to the Lustre community. But, current and future Cray Lustre file system deployments will be loosely coupled routed environments where it won't be possible to tightly couple client and server (as Cray did with a custom imperative recovery feature beginning in CLE 3.1).

With the advent of imperative recovery supported by the MGS (as of Lustre 2.2) it may be easier to remove the reliance on RPC timeouts and pings for node health, which should make it easier to remove the ping functionality.

Acknowledgment

The authors would like to thank ORNL, John Carrier, and especially Dave Dillow for their work in LNET routing; Isaac Huang for helping us to understand LNET and o2iblnd tuning; Whamcloud for their work on imperative recovery; Jeff Garlough for driving test runs; and finally Dave Hensler for his perspectives on OS noise and jitter.

References

[1] D. Dillow, G. Shipman, S. Oral, Z. Zhang, "I/O congestion avoidance via routing and object placement," Proc. Cray User Group, 2011

[2] Lustre 2.x Filesystem: Operations Manual, Section 31.1.4.

[3] Current Status of FEFS for the K Computer, LUG 2012, Shinji Sumimoto,

http://www.opensfs.org/wp-content/uploads/2011/11/LUG2012-FJ-20120426.pdf